CMPT 300

Introduction to Operating Systems

Introduction to Deadlocks

Preemptible resources

- Resources that can be taken away from a process without adversely affecting outcome
- Example: memory (swapping)
  - Can make a copy of memory on disk, suspend proc.
  - Give memory to another proc.
  - When other process is done replace information from disk to memory and proceed with initial task
- Can be thought of as reusable resources
- These are generally not involved in deadlock

Non-Preemptable resources

- Cannot be taken away from the present user without causing the process to fail
- Eg., Output to CD/printer. Taking the resource away and giving it to another proc. produces unusable outputs
- Must protect the resource (mutual exclusion)
- Requesting process may be forced to wait or re-request the resource later
- Sharing may lead to deadlock

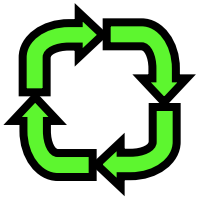
What is a deadlock

- Permanent blocking of a set of processes that share multiple resources.
  - Each process is waiting for resource that,
  - only another proc. in the can make available.

4

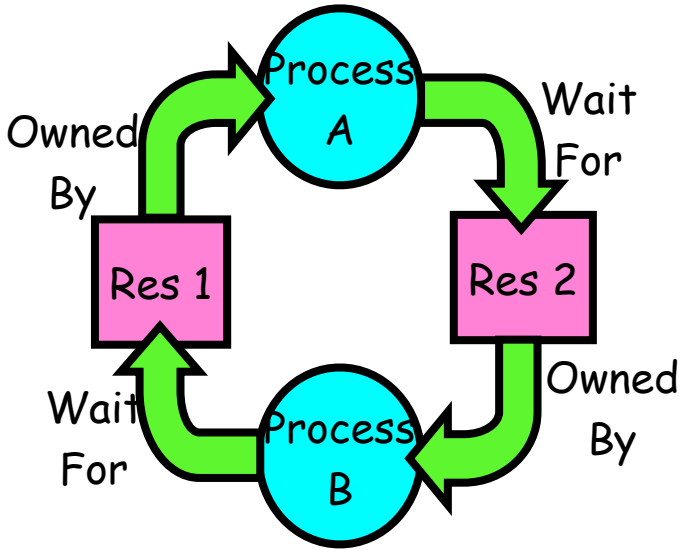
A typical deadlock

- Arises when sharing multiple resources
- Process A is using resource 1 and waiting for resource 2
- Process B is using resource 2 and waiting for resource 1
- Neither process can proceed



Starvation vs Deadlock

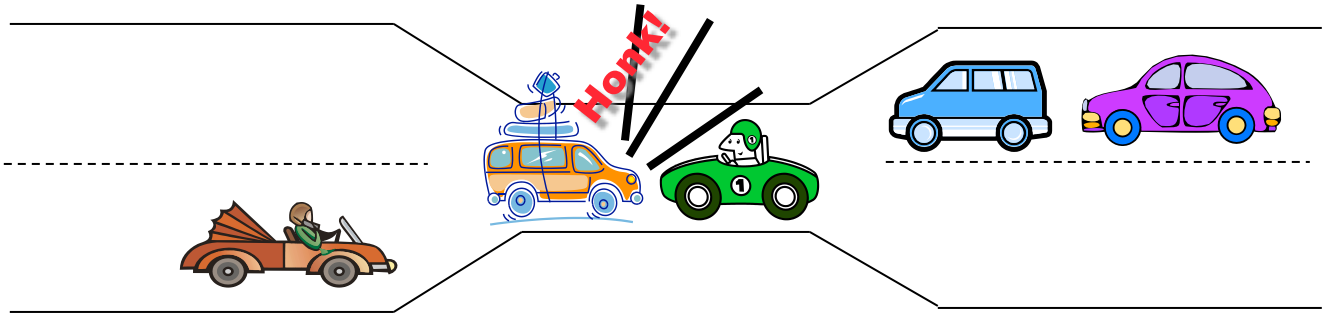
- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
    Thread B owns Res 2 and is waiting for Res 1
- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock needs external intervention



Bridge Crossing Example

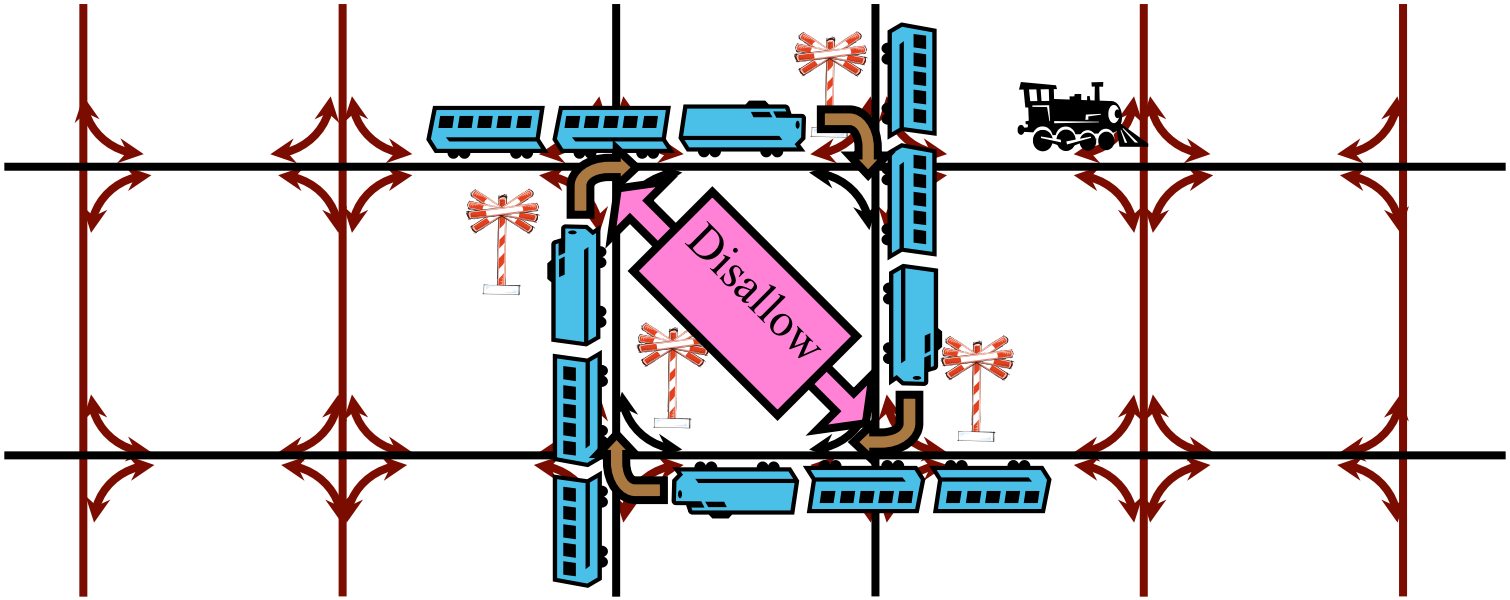
- Each segment of road can be viewed as a resource
  - Car must own the segment under them ; acquire segment before moving
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge:

- If a deadlock occurs, it can be resolved if one car backs up
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west



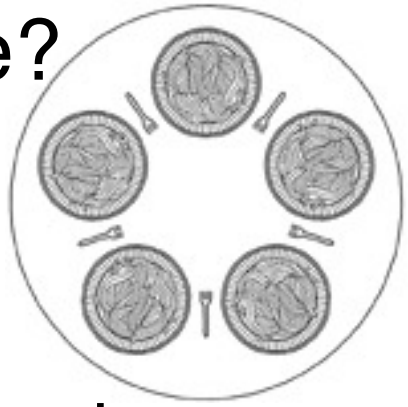
Trains (Wormhole routing)

- Circular dependency (Deadlock!)
  - Each train wants to turn right; Blocked by other trains
- Imagine grid extends in all four directions
  - Force ordering of channels (fixed global order on resource requests)
    - Protocol: Always go horizontal (east-west) first, then vertical (north-south)
  -



Dining Philosopher Problem

- 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



Necessary conditions for deadlock

- Mutual exclusion
  - Only one process at a time can use a resource
- Hold and wait
  - A process may hold resources while waiting for other resources
- No preemption
  - Resource cannot be forcibly removed from a process
- Circular wait
  - A circle of processes exists such that each process holds at least one resource needed by the previous process in the circle

10

Deadlock handing

- **Prevention**
  - Make sure one of the conditions necessary to create a deadlock cannot occur
- **Detection**
  - Resource Allocation Graph (multi-resource ?)
  - Banker's algorithm for detecting (potential) deadlocks
- **Recovery**
  - Let deadlock happen
  - Monitor the system state periodically to detect
  - Break the deadlock

11

Deadlock Prevention Techniques

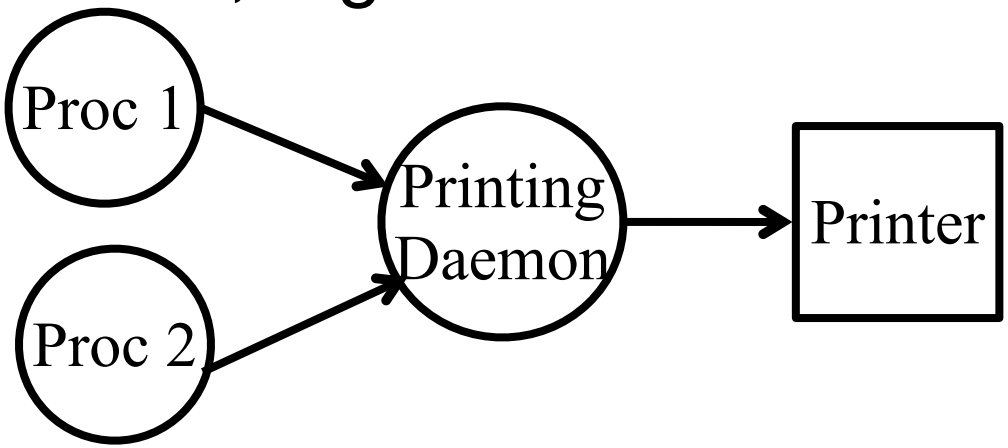
Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 6-14. Summary of approaches to deadlock prevention.

12

Spooling

- A single daemon process directly uses the resource; other processes send their requests to the daemon, e.g.:



- Resource is no longer shared by multiple processes

13

Request all resources initially

- Disallow hold and wait
  - Make each process request all resources at the same time; block until all resources are available
- May be inefficient
  - Process may have to wait a long time instead of working with the resources that were available
  - 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.

14

Order resources numerically

- Prevent circular wait
  - Define a total order of resource type; (# id)
  - If a process holds certain resources,
  - it can request only resources that follow the types of held resources in the total order.

- Prevents a process from requesting a resource that might cause a circular wait.
  - Two chefs need salt, then each chef has to wait for long time.
  - Can deny resources unnecessarily.
  - May be impossible to re-request the same resource

Example from text: deadlock

```
semaphore res1, res2;

 void procA( ) {
     semWait(&res1);
     semWait(&res2);
     useBothRes( );
     semSignal(&res2);
     semSignal(&res1);
}
```

```
 void procB( ) {
     semWait(&res2);
     semWait(&res1);
     useBothRes( );
     semSignal(&res1);
     semSignal(&res2);
}
```

Example from text: no deadlock

```
semaphore res1, res2;

 void procA( ) {
     semWait(&res1);
     semWait(&res2);
     useBothRes( );
     semSignal(&res2);
     semSignal(&res1);
}
```

```
 void procB( ) {
     semWait(&res1);
     semWait(&res2);
     useBothRes( );
     semSignal(&res2);
     semSignal(&res1);
}
```

Take resources away

- Allow preemption
  - If a process holding a resource is denied another resource must relinquish the resource it is holding
  - lower prio. process yields resource to higher prio. process.
  - Requires additional OS and/or program complexity
- Really used for deadlock recovery, not prevention (more later)

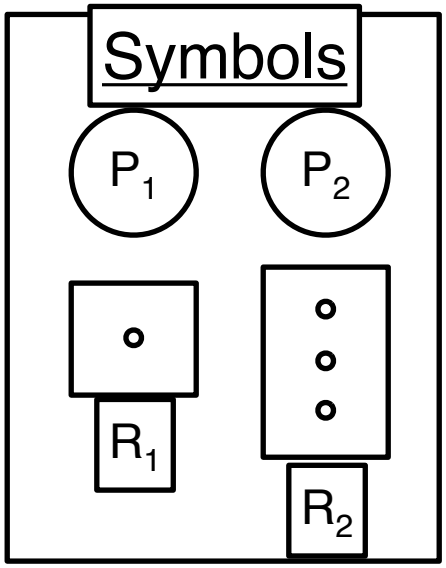
Ostrich algorithm

- 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.
  - Events will occur randomly, we don't know that the 1 in 100 years will not occur in 1 second. In mission critical applications?

Resource-allocation graph

- System Model
  - A set of Processes $P_1, P_2, \ldots, P_n$
  - Resource types $R_1, R_2, \ldots, R_m$

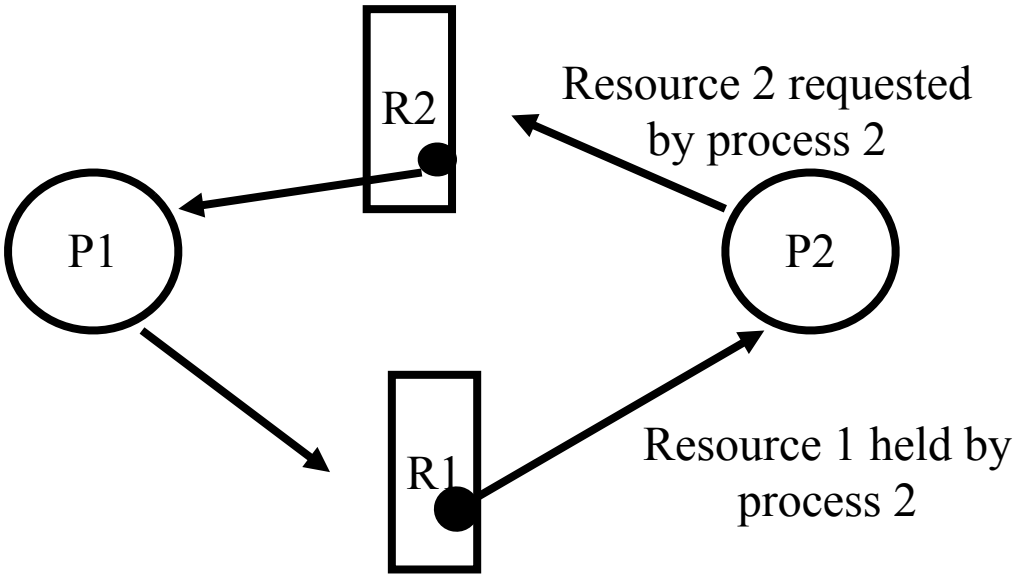
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, set of processes in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, set of resource types in system
  - request edge – directed edge $P_1 \rightarrow R_j$



RAG for deadlock detection

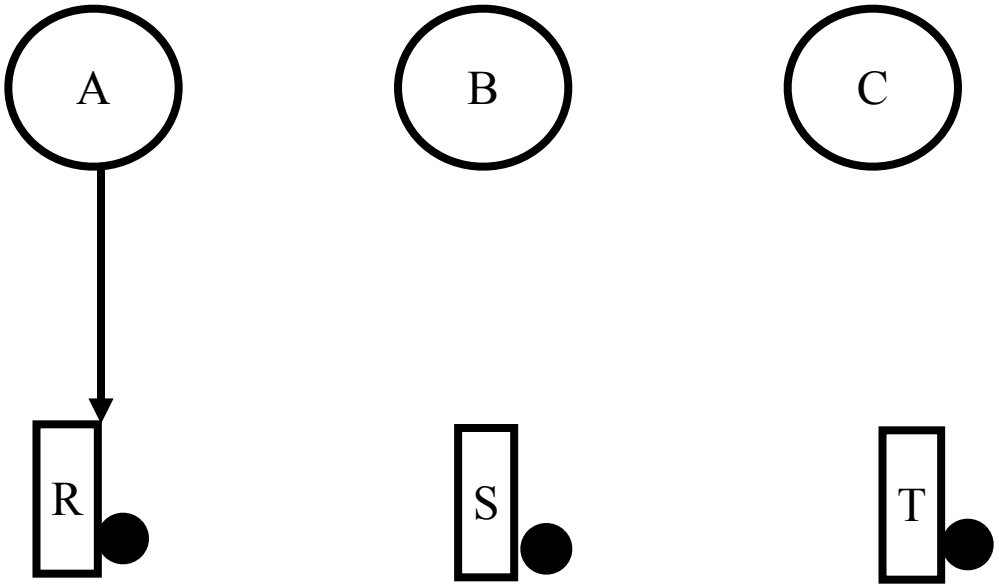
- For any given sequence of requests for and releases of resources a Resource Allocation Graph can be constructed.
- We check the graph
  - no cycle → no deadlock
  - Each resource has a single instance AND cycle → deadlock (necessary and sufficient)
  - Each resource has multiple instances AND cycle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

A RAG with a deadlock



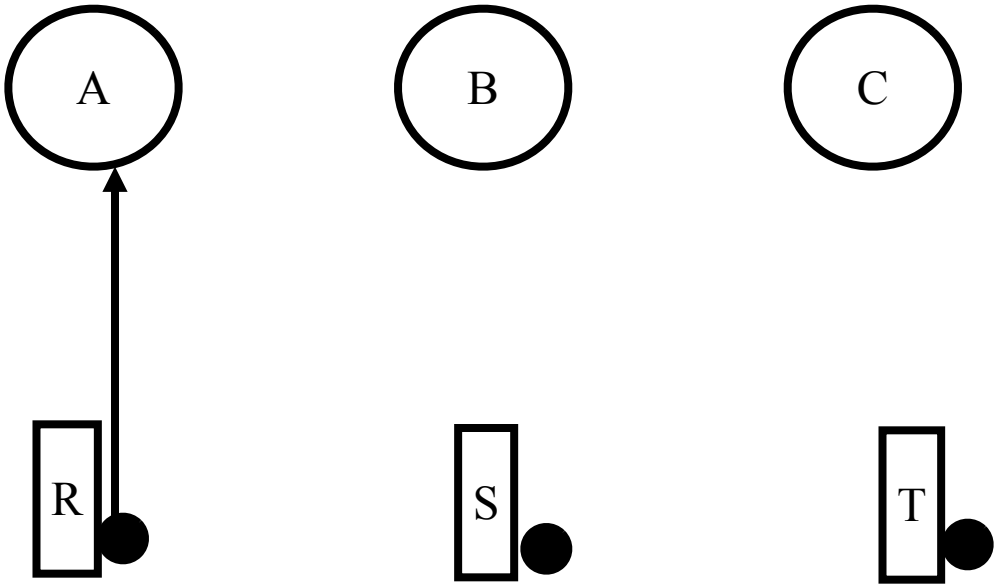
Example

- Consider 3 processes A, B and C scheduled using round-robin algorithm
- A requests R

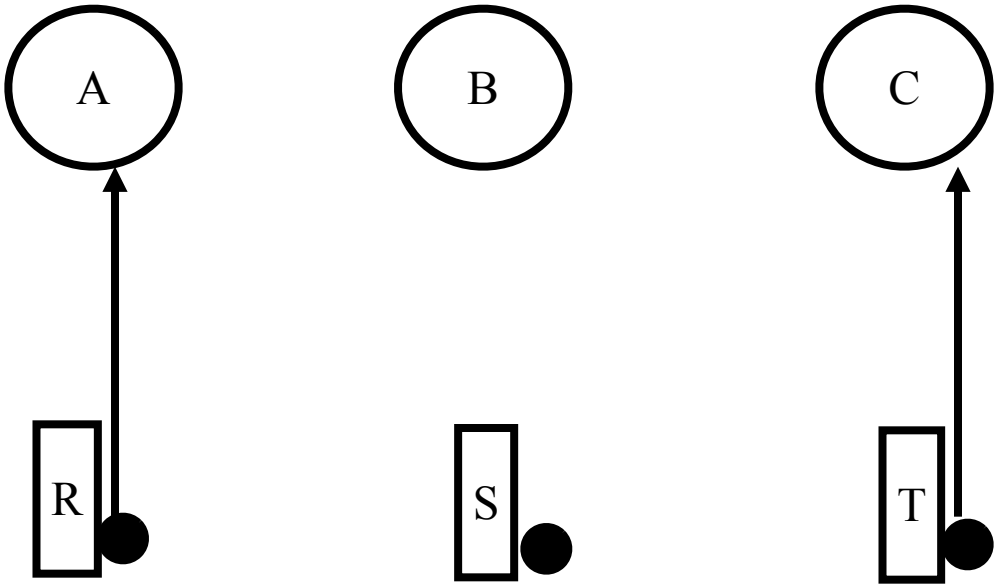


Example

- A requests R
- A holds R

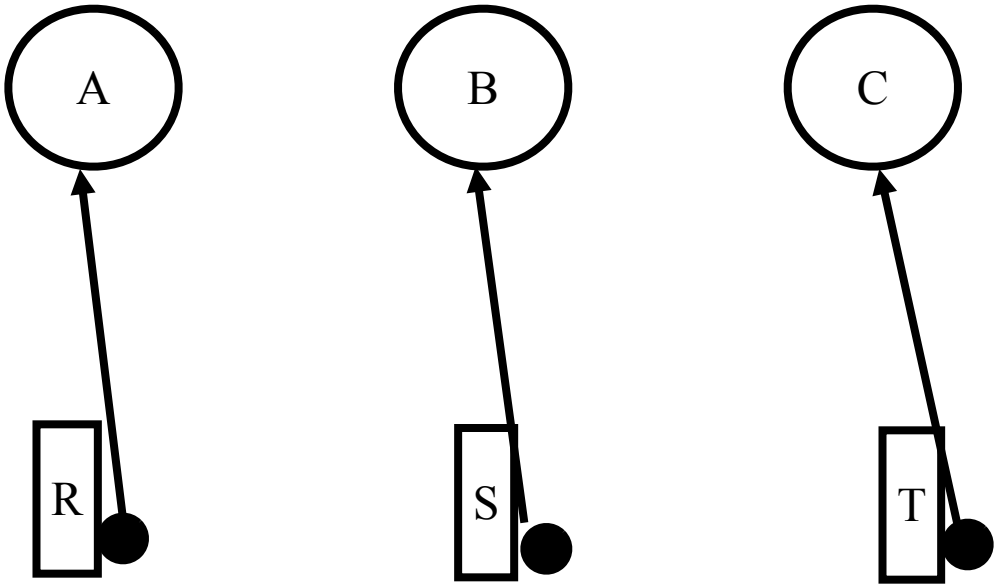


Example



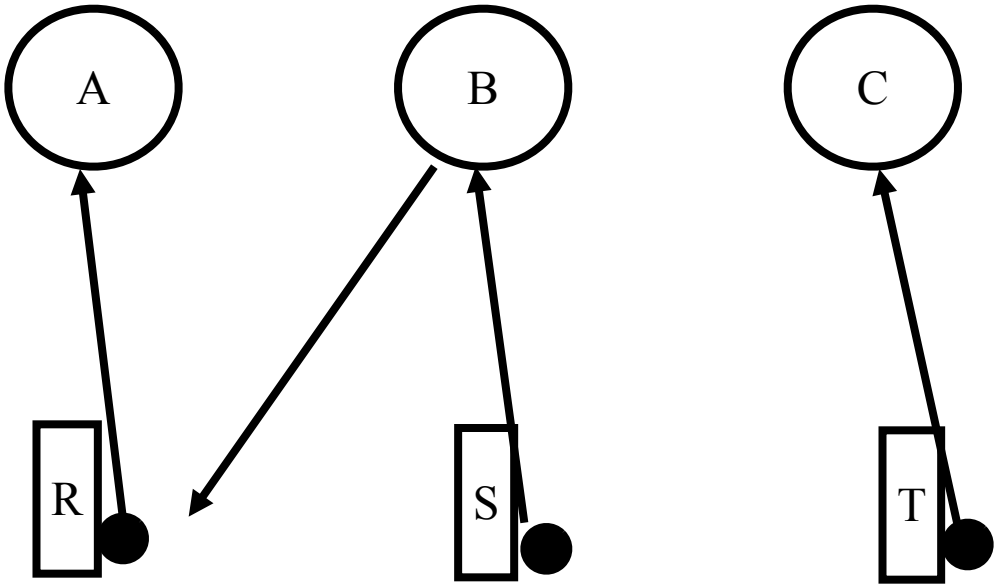
- A requests R
- A holds R
- C requests T
- C holds T

Example



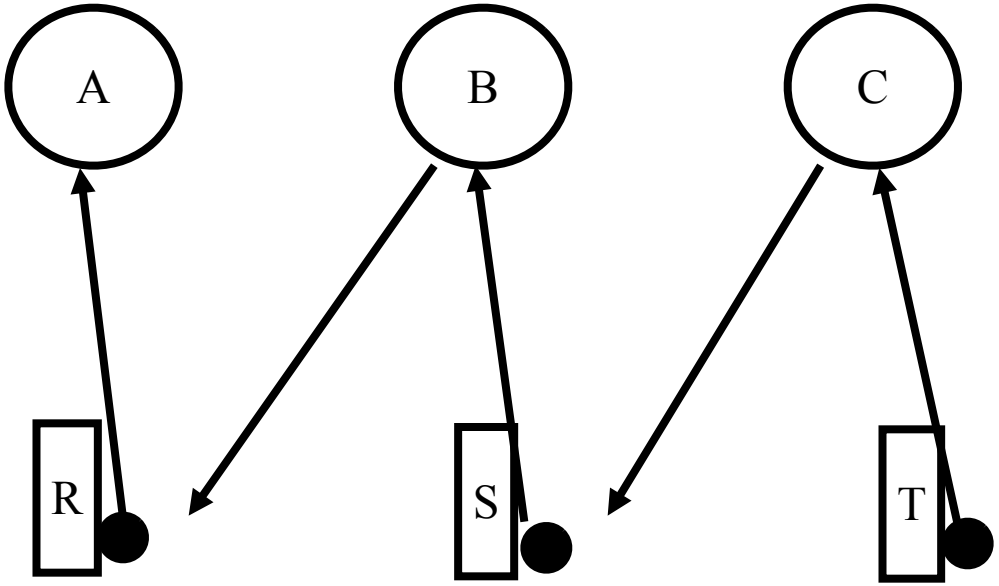
- A requests R
- A holds R
- C requests T
- C holds T
- B requests S
- B holds S

Example



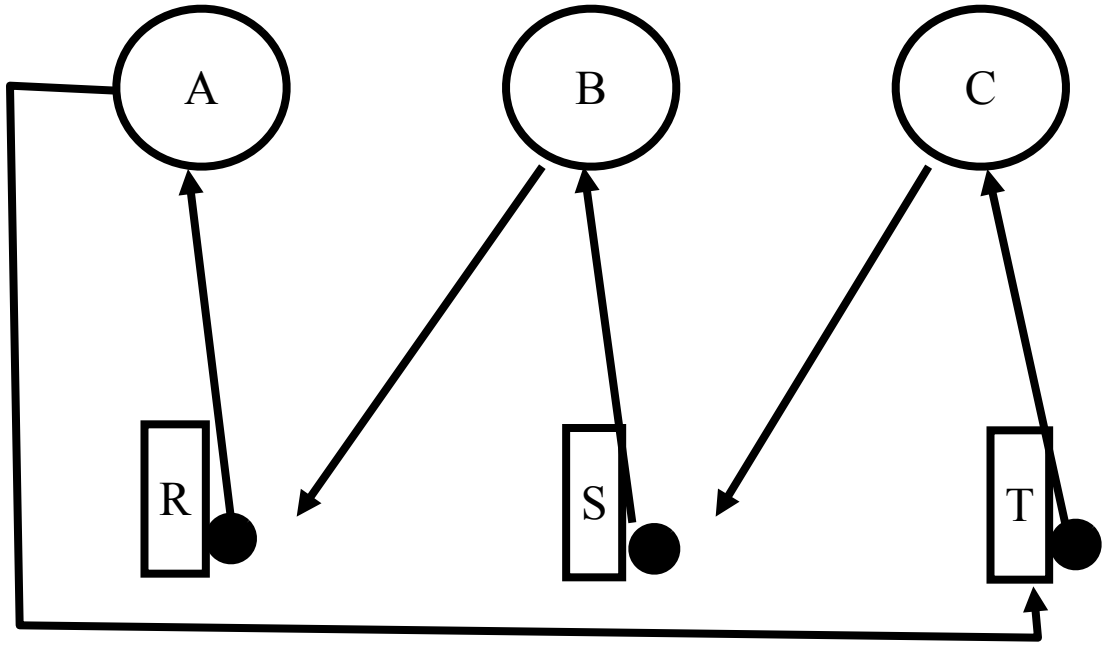
- A requests R
- A holds R
- C requests T
- C holds T
- B requests S
- B holds S
- B requests R

Example



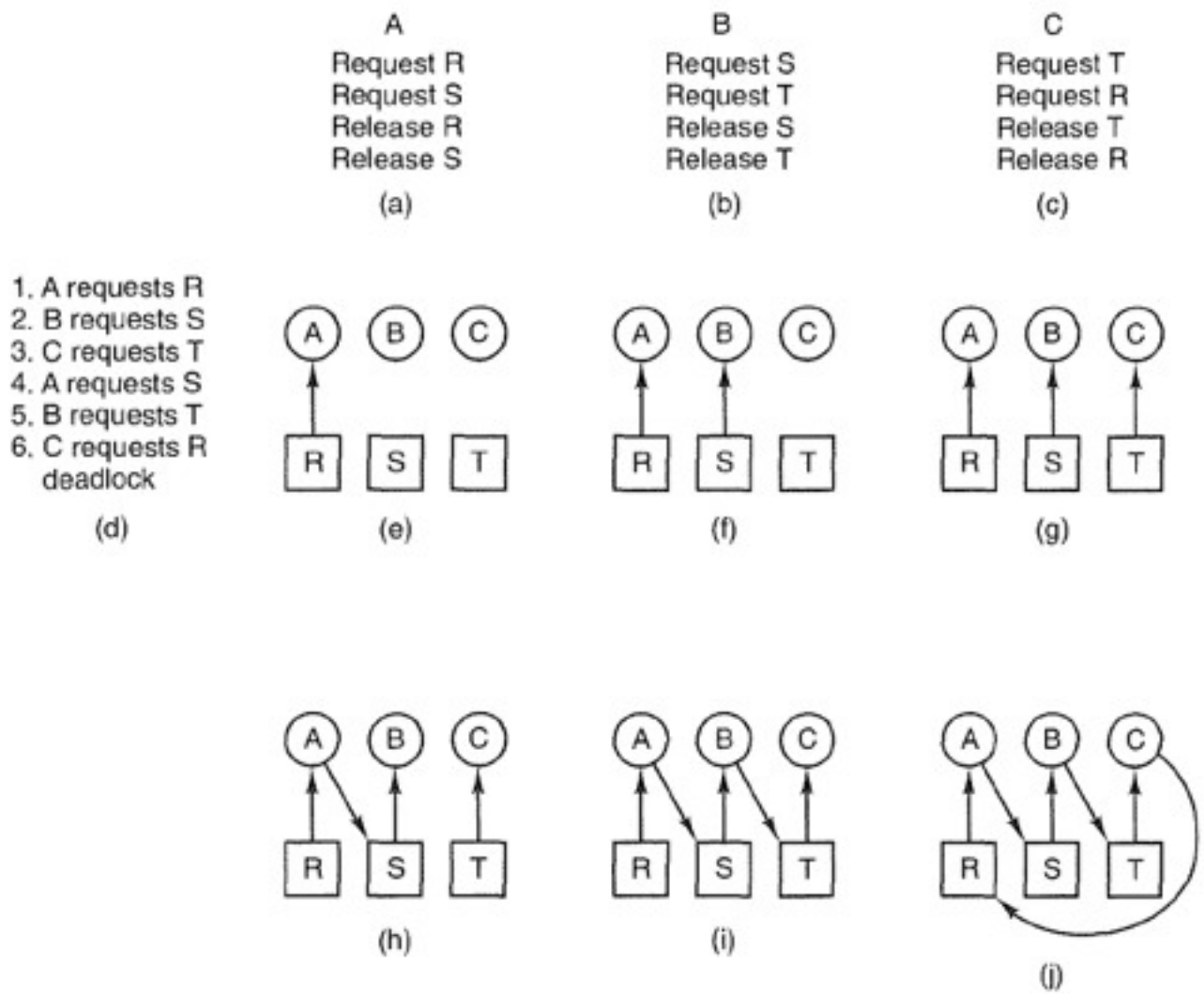
- A requests R
- A holds R
- C requests T
- C holds T
- B requests S
- B holds S
- B requests R
- C requests S

Example: cycle (deadlock)



- A requests R
- A holds R
- C requests T
- C holds T
- B requests S
- B holds S
- B requests R
- C requests S
- A requests T

Another example with deadlock



Deadlock is avoided by delaying B's request

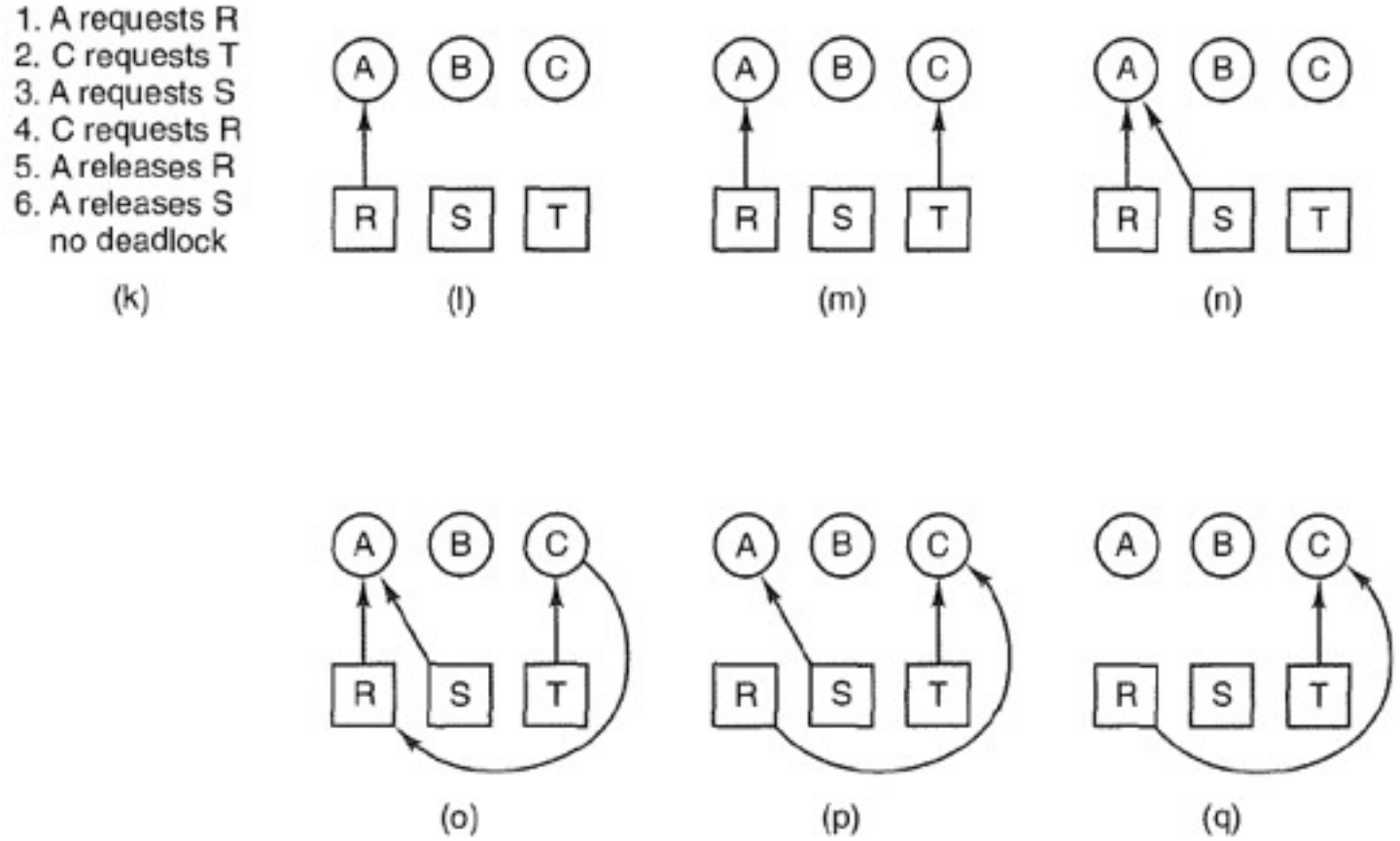
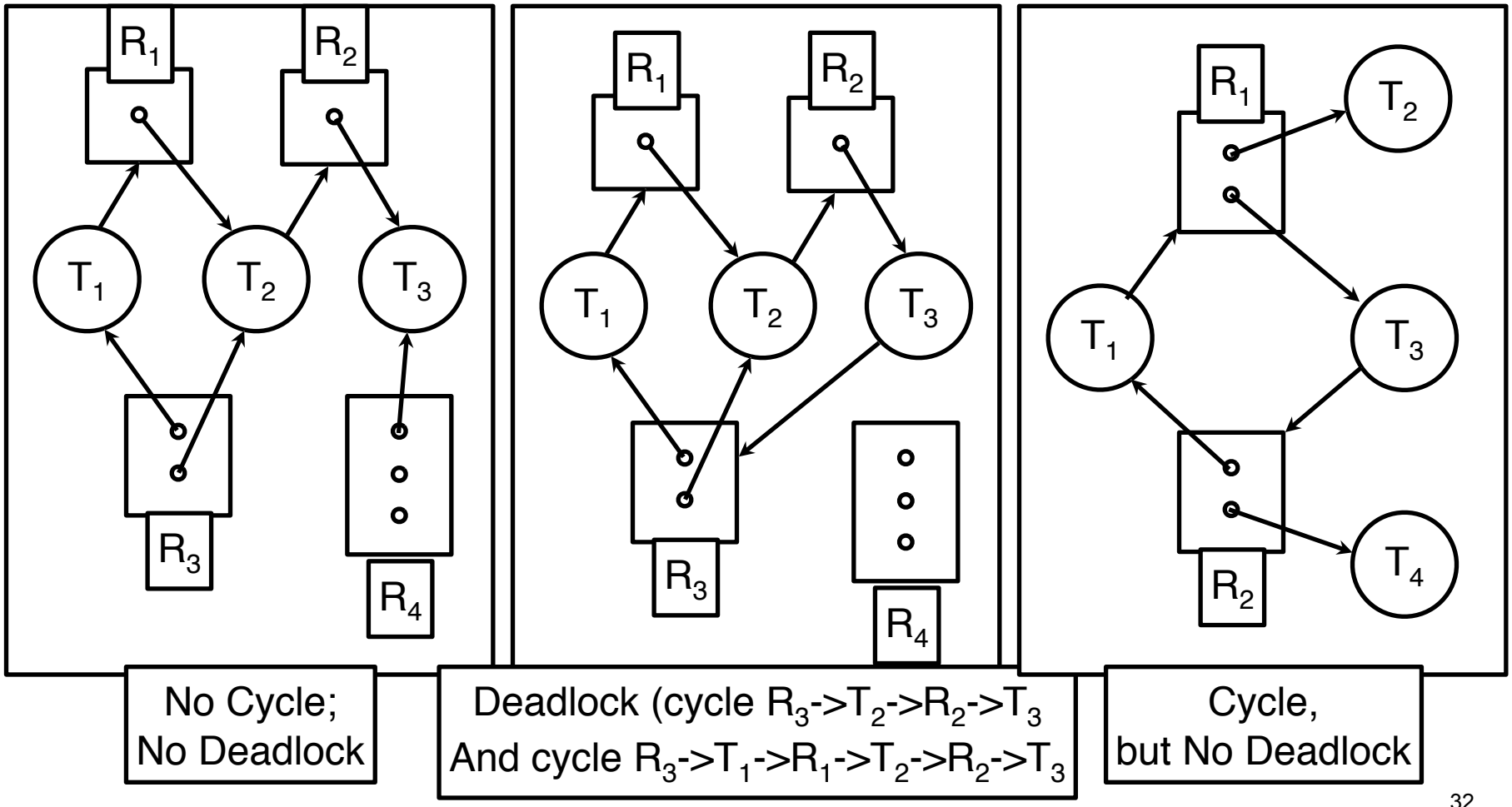


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

Resource Allocation Graph Examples



Animation: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/deadlock.htm

Deadlock detection

- In order 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.

Banker's algorithm for deadlock detection

- Banker's algorithm is used to recognize when it is safe to allocate resources
  - If unsafe, break actual or potential deadlocks
  - Do not grant additional resources to a process if this allocation *might* lead to a deadlock

Defining our problem: 1

- *n* processes and *m* different types of resources. *E* is the *Existing resource vector*

$$E_1, E_2, ..., E_m$$

  - 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

- track # of instances of each resource available (not in-use) with A, the Available resource vector

$$A_1, A_2, ..., A_m$$

Defining our problem: 2

- Which processes are using which of the resources that are in use? *C* is the *Current allocation matrix*:

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

- If process *i* is using 2 resources of type *j* then $C_{ij} = 2$. Process *i* has claimed these 2 resources.

Defining our problem: 3

- We also want to know which processes will need which of the resources during their execution. *R* is the *Total Request matrix*:

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

- If process *i* is will need 4 resources of type *j* then R_{ij} = *4.* Process *i* will need these 4 resources.

Defining our problem: 4

- No process can claim more than the total amount of resources in the system

  $R_{ij} \leq E_j$ For all i and j

- No process is allocated more resources of any type than the process originally claimed to need

  $C_{ij} \leq R_{ij}$ For all i and j

State of the system

- # of resources of each available type presently allocated to it, and the MAX# of resources of each type needed.
- Union of the states of all the procs. along with the # of available and allocated resources of each type.
- Given by the information in the matrices
  - E (Existing resource vector)
  - A (Available resource vector)
  - R (Total Request matrix)
  - C (Current allocation matrix)

Recap of the 4 data structures

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

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
  - Safe state: There exists at least one allocation that will allow all processes to complete without deadlock
  - Unsafe state : there exists no series of resource allocation that will complete all processes in the system
- Each time a resource is requested,
  - determine the state of the system after the resource is allocated, check state.
  - If unsafe we block the process and do not allocate

An example system: starting state

$$R = \begin{bmatrix} 3 & 2 & 2 & 1 \\ 7 & 1 & 3 & 1 \\ 3 & 1 & 4 & 0 \\ 4 & 2 & 2 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

$$E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix} \qquad A = \begin{bmatrix} 2 & 3 & 2 & 4 \end{bmatrix}$$

Request to check for safety

- Assume the starting state is a safe state (left to student to demonstrate)
- Process 2 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

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

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

Bankers algorithm: preliminaries

- To determine if the new state is a safe state need to compare two vectors.
- One vector will be row L in the matrix (R-C) of unmet resource needs
- The other vector will be the available resources vector A
- $(R\text{-}C)_L <= A$

  IF $(R_{Lk} - C_{Lk}) <= A_k$ for all k

Banker's algorithm

1. Look for an unmarked process, P_i , for which the i-th row of $R =< A$.
2. Add the *i-th* row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

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
    - If so grant the request for resources
    - Otherwise block the process until it is safe to grant the request

An example: is new state safe

- Check row L=1

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

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

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

An example: is new state safe

- Check row L=2

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

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

- $(R\text{-}C)_2 = [0, 0, 1, 0]$ $(R\text{-}C)_2 <= A$
- Allocate resources and run process 2 to completion;
- Reallocate all freed resources from process 2 to available (A):
  - A=[0 3 1 4]+[7 1 2 1] = [7 4 3 5]

An example: is new state safe

- Check row L=1 again

Row (process marked as completed)

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

$$E = \begin{bmatrix} 10 & 5 & 6 & 5 \end{bmatrix} \qquad A = \begin{bmatrix} 7 & 4 & 3 & 5 \end{bmatrix}$$

- $(R\text{-}C)_1 = [2, 2, 2, 1]$ $(R\text{-}C)_1 <= A$
- Allocate resources and run process 1 to completion;
- Reallocate resources from process 1 to available (A):
  - A = [7 4 3 5] + [1 0 0 0]=[8 4 3 5]

An example: is new state safe

- Check row L=3

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

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

- $(R\text{-}C)_3 = [1, 0, 3, 0]$ $(R\text{-}C)_3 <= A$
- Allocate resources and run process 3 to completion;
- Reallocate resources from process 3 to available (A)
  - A = [8 4 3 5] + [2 1 1 0] = [10 5 4 5]

An example: is new state safe

- Check row L=4

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

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

- $(R\text{-}C)_4 = [4, 2, 0, 1]$ $(R\text{-}C)_4 <= A$
- Allocate resources and run process 4 to completion
- Reallocate resources from process 4 to available (A)
  - A = [10 5 4 5] + [0 0 2 0] = [10 5 6 5]

An example: is new state safe

- Now:

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

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

- All process can complete successfully
- Therefore, this is a safe state
- Allow the resources to be allocated and proceed with execution of all processes

New starting state: next request

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

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

Next Request to Check for Safety

- Start from safe state and consider the next request for resources
- Process 1 is now requesting 1 more of resource 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

New starting state: next request Is this state safe?

- Check row L=1, L=2, L=3

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

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

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

UNSAFE!!!, no row marked as complete!

Processes may be deadlocked

Unsafe state vs. deadlock

- unsafe state: indicates 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.
- ‘Detection’ of a deadlock on worst case assumptions
  - The process 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
  - i.e., 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 **holding LEFT** fork, then the **N^{th} philosopher** will be **prevented** from **taking the last fork.**
  - If a philosopher is **holding LEFT** fork, he **can safely pick** up his **RIGHT** fork if available.

  - If one or more philosophers are holding 2 forks and eating, then any remaining forks can be picked up safely another philosopher.

Banker's algo applied to Dinning Philosophers cont'

- Note: you need to model each fork as a separate resource.

- 5 philosophers numbered 1-5, and 5 forks numbered 1-5;
- left fork : i
- right fork : (i+1)%5.

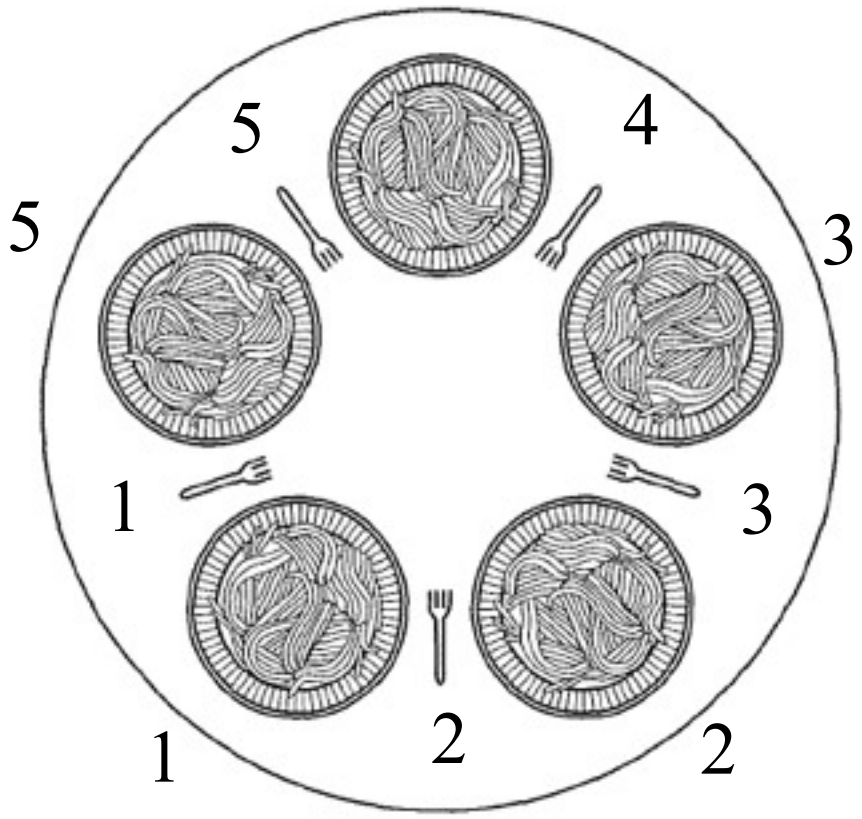


Figure 2-44. Lunch time in the Philosophy Department.

When 4 philosophers each holds his left fork

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

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

- If the 5th philosopher makes a request for his left fork, should we grant it?

The deadlocked state when each holds his left fork

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

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

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

Multi-Armed Lawyers

- Consider a large table with IDENTICAL multi-armed alien lawyers. One chopstick/hand.
- Center is a pile of chopsticks.
- The lawyers are so busy talking that they can only grab one chopstick at a time.
- Assume total number of chopsticks >= number of hands of each lawyer.

```
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 */
  this.NumArms = NumArms;  /* Remember # arms/Lawyer */
  this.NumChopsticks = NumChopsticks; /* Remember # Chopsticks/table */
}
```

- Allows a lawyer to grab one chopstick.
- Puts a lawyer to sleep if he cannot be granted a chopstick w/o deadlock.

```
public void ReleaseAll(int Lawyer) {

  lock.Acquire();

  NumChopsticks += AllocTable[Lawyer]; /* Put chopsticks back */
  AllocTable[Lawyer]=0;

  Lock.Release();

}
```

- ReleaseAll() : Lawyer releases all chopsticks
- Wakes up others

```
public void GrabOne(int Lawyer) {
   lock.Acquire();

   /* Use bankers algorithm to check if alloc isC ok! */
   while(!BankerCheck(Lawyer))
      CV.wait();

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

   Lock.Release();
}
```

- BankerCheck() : #Lawyer id,
  - checks resources,
  - **True: one new chopstick**
- Assume Mesa-style monitor, hence while loop is used in GrabOne().

```
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 */

}
}
```

- State is safe if when a lawyer tries to take a chopstick, either
  - It is the last chopstick, but someone will have *NumArms* chopsticks afterwards
  - Or it is the 2nd to last chopstick, but someone will have *NumArms-1* chopsticks afterwards
  - Or…

Dining Lawyers Questions I

- 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*

Dining Lawyers Questions II

- Q: Is it a generalization of the 2-armed Dining Philosophers problem?
- A: Not exactly.
- We should model them as a single resource with multiple instances,
- Dining Philosophers: Multiple resources for the Dining Philosophers.
- Hence the *R* and *C* matrices have a single column.

Review: Dining Lawyers Questions III

- Q: In its general form, the Banker's algorithm makes multiple passes through the set of resource takers, finishing one at a time until all resource takers have finished. Explain why this particular application allows the *BankerCheck* method to implement the Banker's algorithm by taking a single pass (until any one lawyer can get NumArms chopsticks).
- A: Since every Lawyer has the same maximum allocation, and all chopsticks are equivalent. 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 specification of the Banker's algorithm).

Dining Lawyers Variation I

- Q: Each lawyer has 2 arms,
- Pile of knives and forks at center of the table.
- lawyer steps:
  - (1) Pick up a knife
  - (2) Pick up a fork
  - (3) Eat
  - (4) Return the knife and fork to the pile

- Can the system be deadlocked?
- A: No, since it's not possible to have circular waiting.

Dining Lawyers Variation II

- Q: Each lawyer has 4 arms,
- Assume there are at least 2 knives and 2 forks, lawyer steps:
  - (1) Pick up 2 knives at the same time
  - (2) Pick up 2 forks at the same time
  - (3) Eat
  - (4) Return the knives and forks to the pile

- Can the system be deadlocked?
- A: No, since it's not possible to have circular waiting.

Dining Lawyers Variation III

- Q: Lawyer 4 arms. 2knives and fork per lawyer
- Each lawyer follows the following steps:
  - (1) Pick up a knife
  - (2) Pick up another knife
  - (3) Pick up a fork
  - (4) Pick up another fork
  - (5) Eat
  - (6) Return the knife and fork to the pile

- Can the system be deadlocked?
- A: Yes, since requests for each resource type (knife or fork) are not granted atomically. Need Banker's algorithm to detect (potential) deadlocks.
- Consider 2 lawyers, and a total of 2 knives and 2 forks available. If each lawyer picks up a knife, the system is deadlocked.
- No total order on knives or forks.

Dining Lawyers Variation III

- Q: What if each lawyer has a different # of arms, req. a different ratio of knives vs. forks?

- A: e.g., have an array of variables *NumArms[]* instead of a single variable *NumArms*, and so on.

Minimum Resource Constraint

- Minimum number of resources to allow at least one process to finish.
- 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.
- Driven by a timer. Longer interval
  - Higher efficiency due to less checking
  - Undetected deadlocks can persist for longer times
- What to do if an actual deadlock is detected?

Deadlock recovery

1. Abort all deadlocked processes:
   - most common solution implemented in OSs
2. Rollback:

   - Back up each process periodically
   - Roll back to the previous backup (checkpoint).

   - Nondeterministic nature of the execution of concurrent processes (different interleaving)
   - It is possible the deadlock may reoccur.

Deadlock recovery: 2

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

Choosing processes/resources

- For options 3 and 4 it is necessary to choose which of the possibly deadlocked processes to abort or which resource to preempt (and the possibly deadlocked process to preempt it from)
- Can base this decision on a number of 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 processes (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 processes cannot be rolled back
    - How do you roll back shared variables that have been successively updated by multiple processes
  - Process rollback is expensive
    - Successive checkpoints must save both image and state
    - Multiple checkpoints need to be saved for a process

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

Livelock

```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}
```

```
void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
}
```

- Both processes use the polling primitive enter_region() to try to acquire locks via busy-waiting; process_A gets resource_1 and process_B gets resource_2.
- Livelock, not deadlock, since no process is blocked.

Summary

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one process at a time can use a resource
  - Hold and wait
    - Process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption
    - Resources are released only voluntarily by the processes
  - Circular wait
    - Cyclic waiting pattern

Summary (2)

- 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.
    - There exists a sequence of process executions $\{P_1, P_2, \ldots P_n\}$ with P_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..that can finish successfully.

- Algorithm allocates resources dynamically, and allows the sum of maximum resource needs of processes > total.