Operating Systems

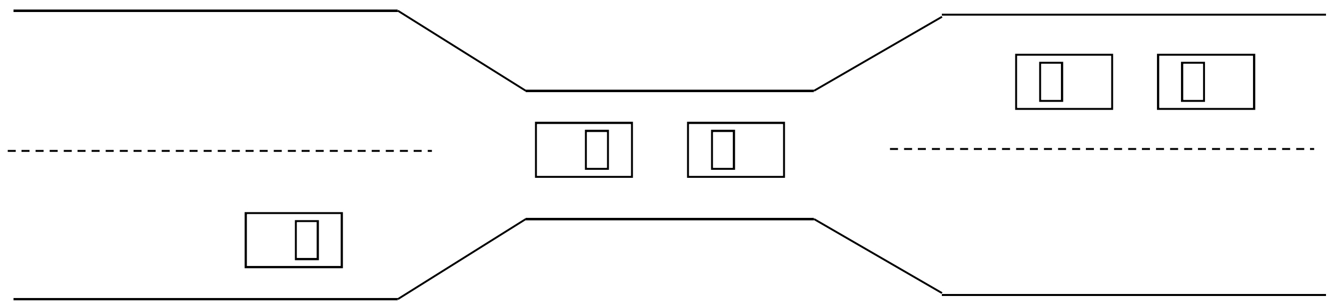
6. Deadlocks

Deadlock

- A set of blocked processes each waiting for an event that only another process in the set can cause

- Example of a possible event
  - Resource to become available

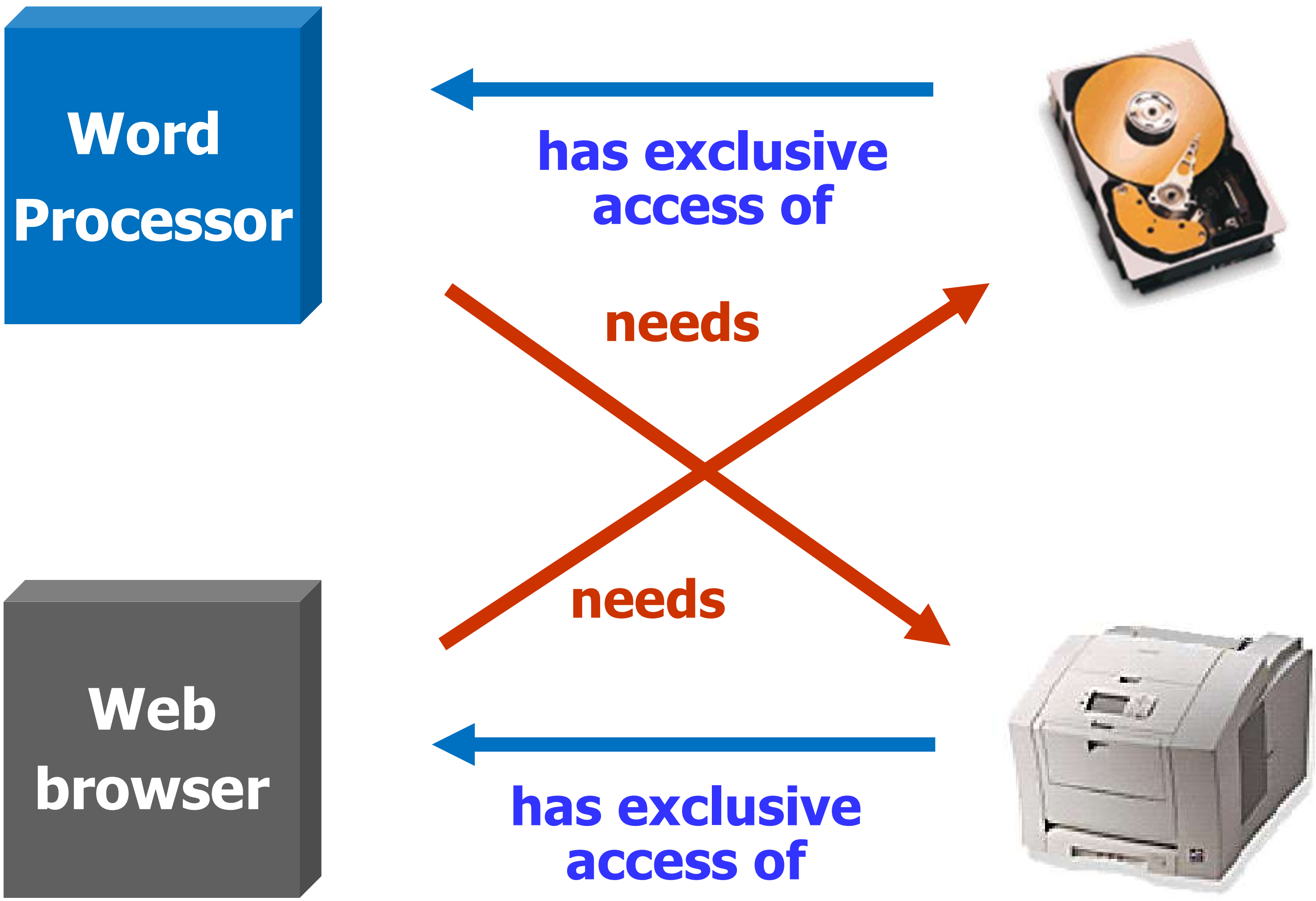
Bridge Crossing Example



- Traffic only in one direction
- Each "half" of the bridge can be viewed as a resource

- If a deadlock occurs, it can be resolved if cars back up (preempt resources and rollback)
  - Several cars may have to be backed up
  - Starvation is possible

Deadlocks in the Computer World



Deadlocks in Resource Allocation – Examples

- Semaphores A and B, initialized to 1 (or: system has 2 tape drives; P0 and P1 each hold one tape drive and each needs another one)

P0	P1
wait (A);	wait(B);
wait (B);	wait(A);

- 200Kbytes memory-space is available

P0	P1
request (80Kbytes);	request (80Kbytes);
…	…
request (70Kbytes);	request (70Kbytes);

  - Deadlock might occur if both processes proceed to the second request

Deadlocks in Resource Allocation – Examples

- Message-passing with blocking receive

P0	P1
receive(P1);	receive(P0)
send(P0, M1);	send(P1 , M0);

Conditions for Deadlock

Four conditions must hold simultaneously for a deadlock to occur [Coffman-et al 1971]

1. Mutual exclusion: Only one process at a time can use a resource
2. Hold and wait: A process holding some resource can request additional resources and wait for them if they are held by other processes
3. No preemption: A resource can only be released voluntarily by the process holding it, after that process has completed its task
4. Circular wait: There exists a circular chain of 2 or more blocked processes, each waiting for a resource held by the next process in the chain

Resource Allocation & Handling of Deadlocks

- **Many possible options dealing with deadlocks in an OS!**
- Deadlock Prevention: Structurally restrict the way in which processes request resources (restriction on Conditions 1-3)
- Deadlock Avoidance: Processes are required to specify info in advance about their resource usage
  - Info: (max) resources required in running state
  - OS schedules processes in a way that deadlock is avoided
    - **No restrictions** on Conditions 1-3 !
- Deadlock Detection: Deadlock state allowed followed by recovery
- Deadlock Ignorance: Pretend that deadlocks never occur in the system (can be a “solution” sometimes?!...)

Resource Allocation with Deadlock Prevention

- Restrain the ways requests can be made
  – Attack at least one of the 4 conditions, so that deadlocks are not possible to happen

1. Mutual Exclusion
   – Cannot do much here …

2. Hold and Wait
   – Must guarantee that when a process requests a resource, it does not hold any other resources
   – Require process to request and be allocated all its resources at once
   – Or allow process to request resources only when the process has none
   – Low resource utilization; starvation possible

Resource Allocation with Deadlock Prevention

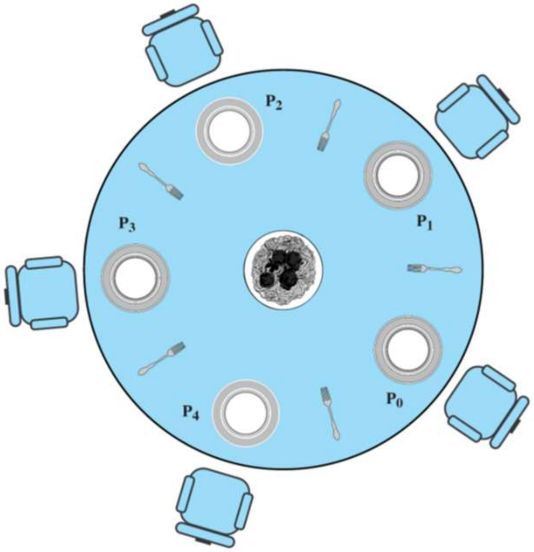
3. No Preemption
   - If a process holds some resources, requests another resource that cannot be immediately allocated
   - It release all held resources and request them again
     - ➢ Risk for starvation!

4. Circular Wait
   - Impose total ordering of all resource types
   - Require that each process requests resources in an increasing order of enumeration
     - ➢ For example, e.g., first the tape, then the disk

Resource Allocation: System Model

- Resource types $R_1, R_2, \ldots, R_m$
  - For example, CPU, memory space, I/O devices, files
  - Each resource type R_i has W_i instances

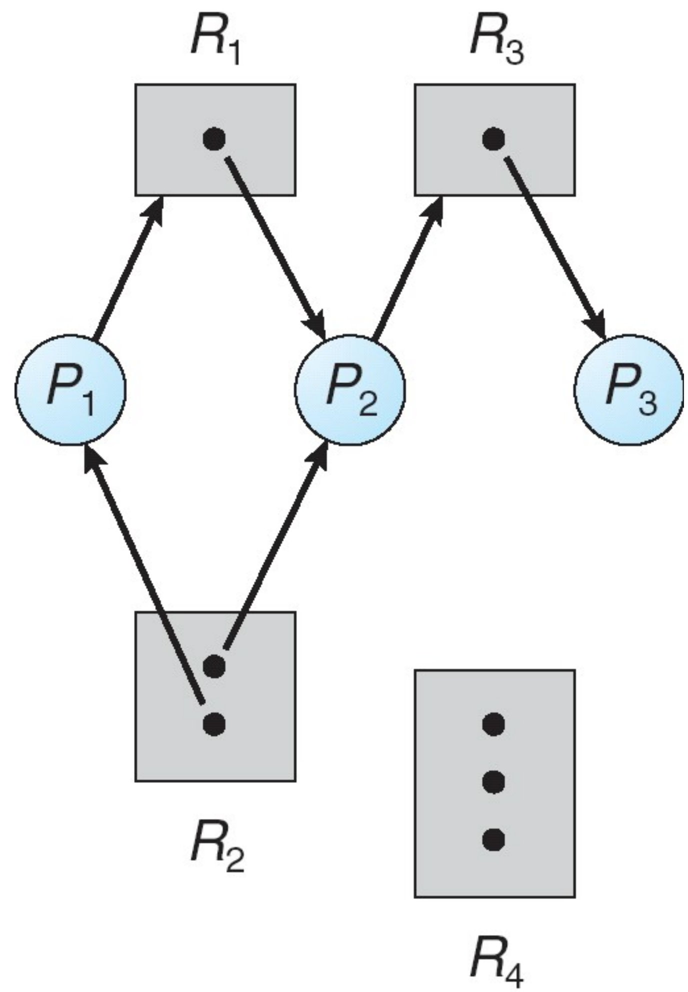
- Each process utilizes a resource as follows (as discussed for the dining philosophers):
  - Request (hungry)
  - Use (eat)
  - Release



Resource Allocation Graph

A set of vertices V and a set of edges E

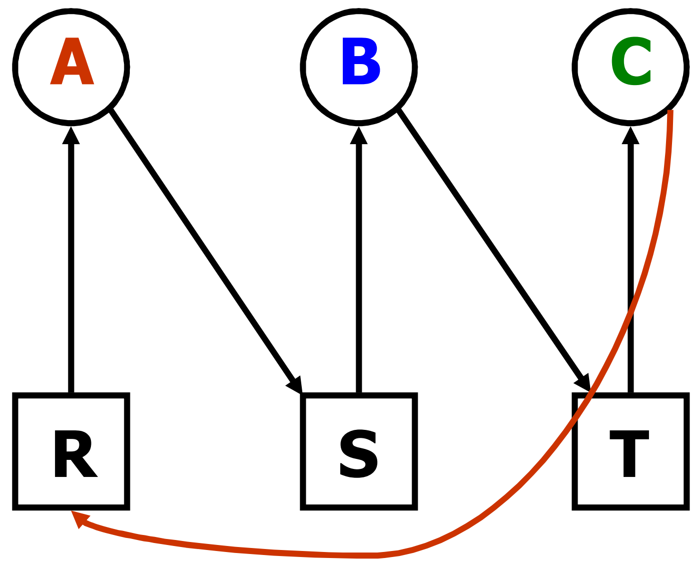
- V is partitioned into two sets
  - $P = \{P_1, P_2, \ldots, P_n\}$ the set of processes
  - $R = \{R_1, R_2, \ldots, R_m\}$ the set of resource types
- Request edge: $P_i \rightarrow R_j$
- Assignment edge: $R_j \rightarrow P_i$



Resource Allocation Graph – Example

A	**B**	**C**
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R — *assigned*
2. B requests S — *assigned*
3. C requests T — *assigned*
4. A requests S — *waiting*
5. B requests T — *waiting*
6. C requests R — *waiting*

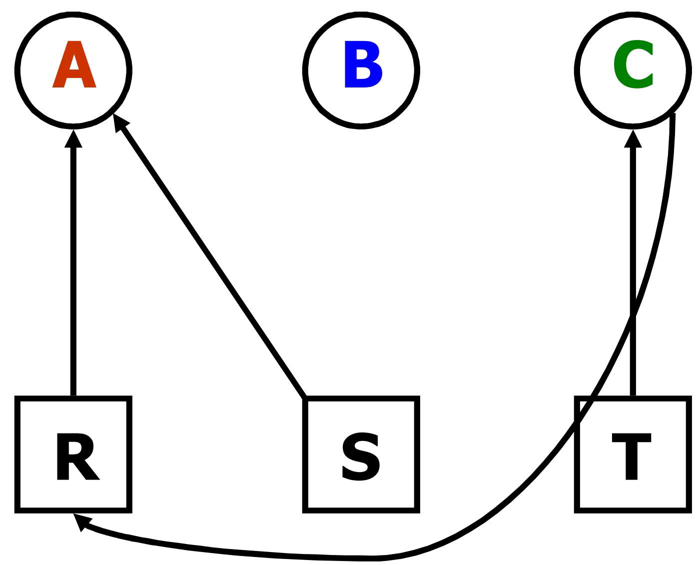


DEADLOCK

Resource Allocation Graph – Example

A	B	C
Request R Request S Release R Release S	Request S Request T Release S Release T	Request T Request R Release T Release R

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

Basic Facts

- Graph contains no cycles $\Rightarrow$ no deadlock
  (i.e., cycle is always a necessary condition for deadlock)

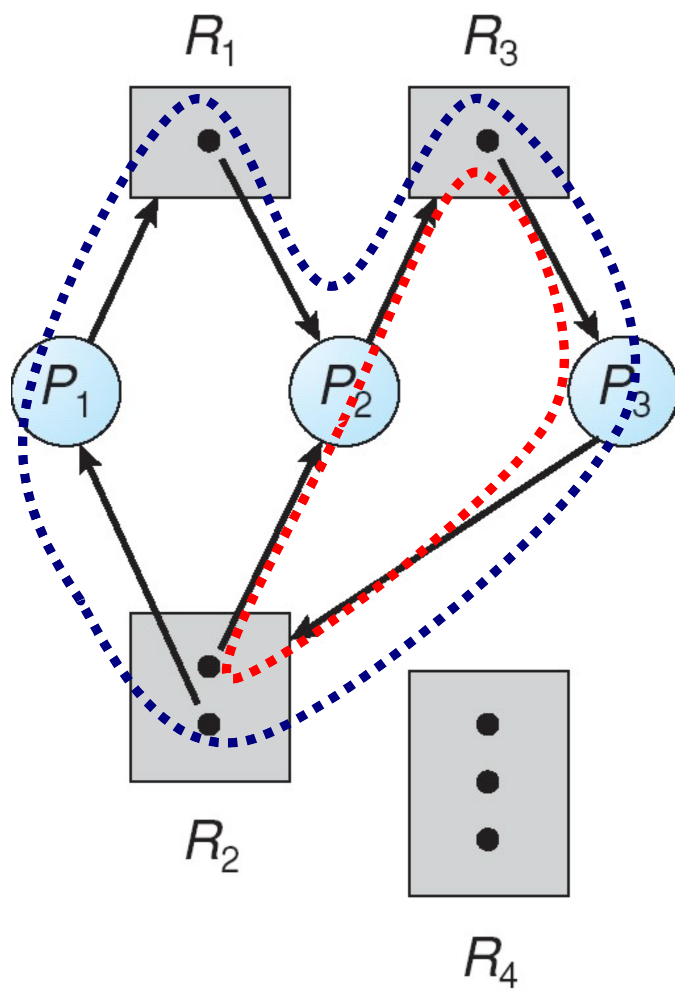
- If graph contains a cycle $\Rightarrow$
  - if $\exists$ one instance per resource type, then deadlock
  - if $\exists$ several instances per resource type, then possibility of deadlock

Theorem:

Assume we apply the immediate-allocation-method

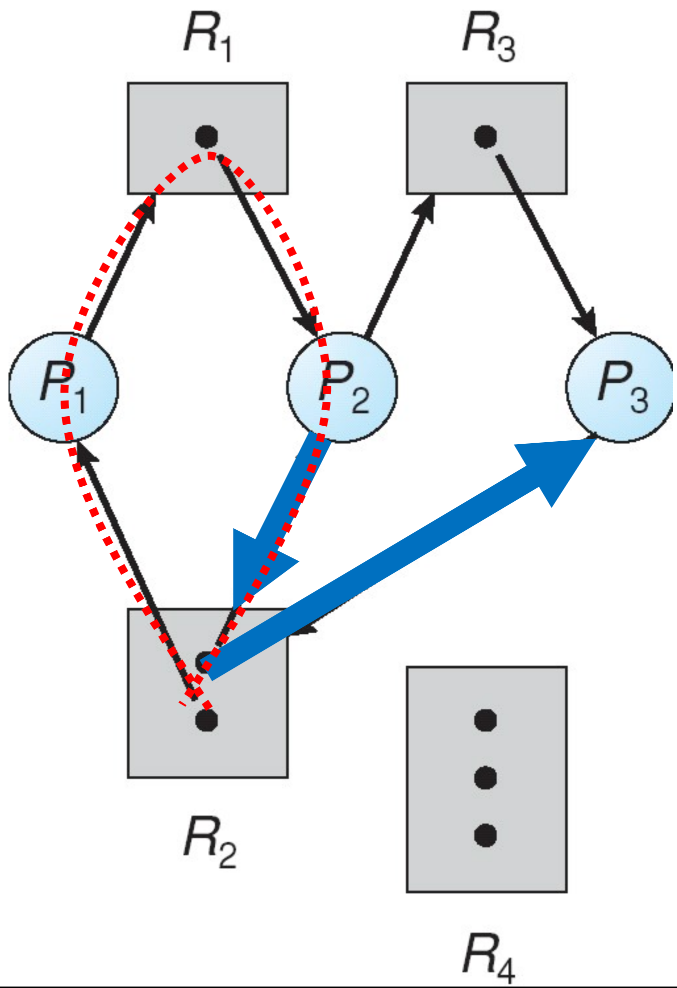
If the corresponding graph contains a knot, i.e. a strongly connected component and no outgoing edges (sinks), we encounter a deadlock

Resource Allocation Graph: Deadlock



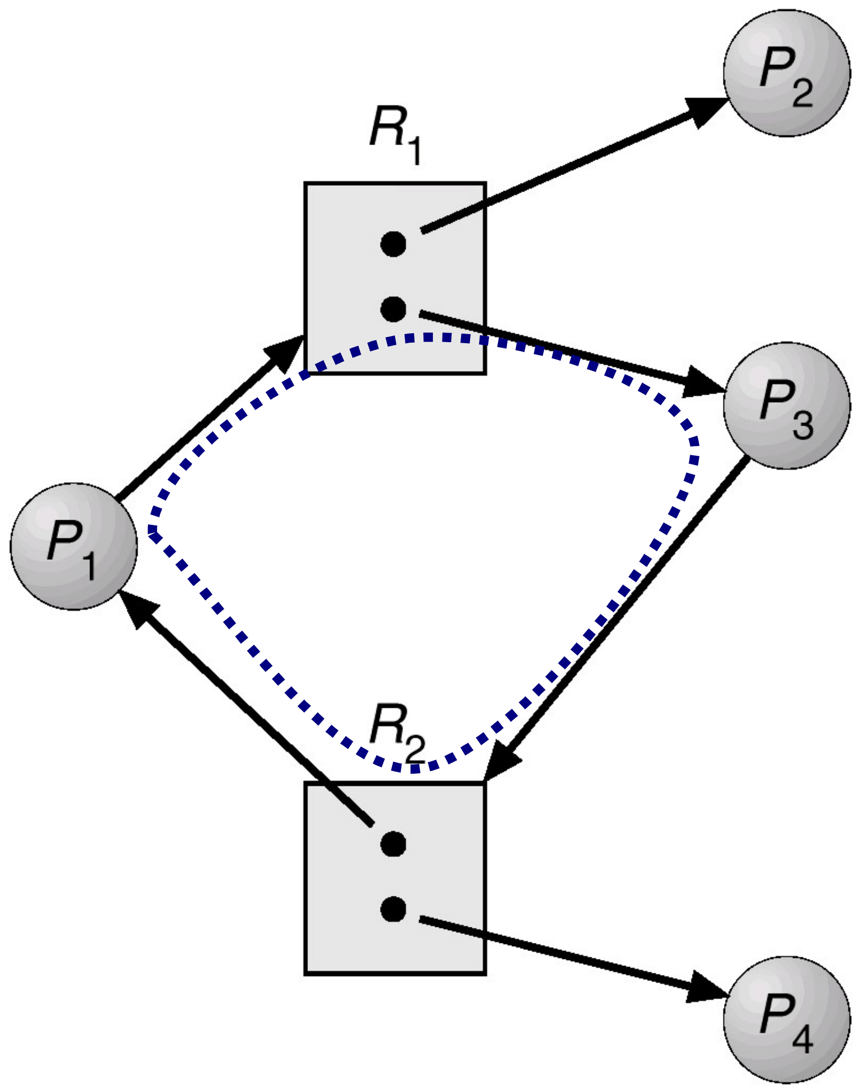
Strongly connected component and no outgoing edge

Resource Allocation Graph: Cycle But No Deadlock



Strongly connected component with an outgoing edge

Resource Allocation Graph: Cycle But No Deadlock

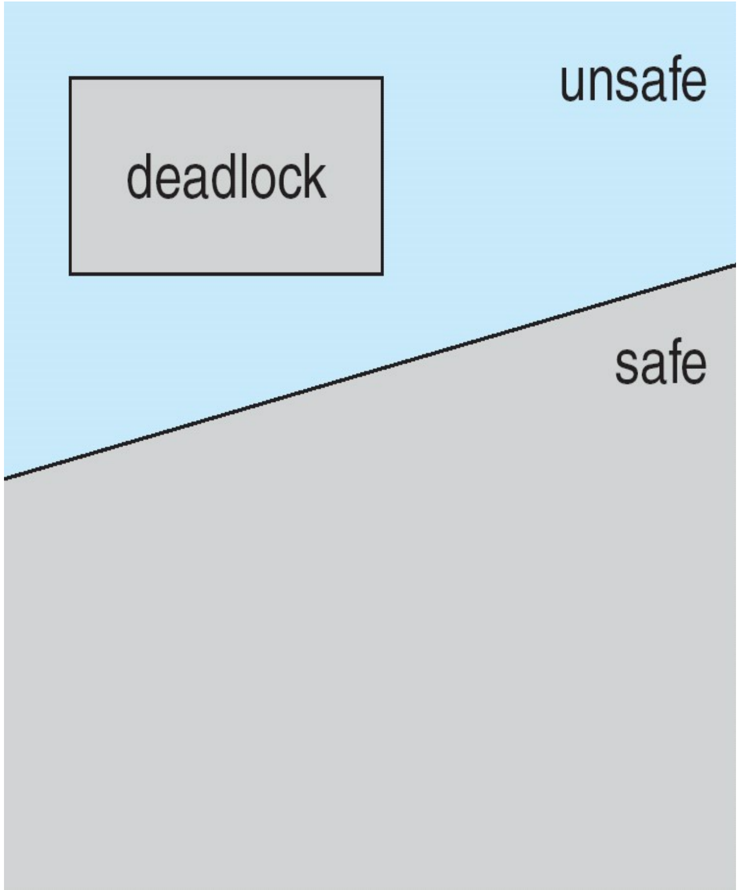


Save, Unsafe, and Deadlock States

Possible global OS states depending on the current resource allocation of processes

- Safe state
  - No deadlocks in the horizon
  - OS can find a schedule which avoids a deadlock
- Unsafe state
  - Deadlock might occur (later…)
  - For example, future resource requests

Note: Safe states require information on future behavior of processes!



Resource Allocation with Deadlock Avoidance

- **Requires** a priori information available
  - For instance, each process declares maximum number of resources of each type that it may need (e.g., memory/disk pages)

Deadlock-avoidance Algorithm

- Examines the resource-allocation state…
  - Available and allocated resources
  - Maximum possible demands of the processes

- Avoidance = Ensure that system will not enter an unsafe state

Idea: If satisfying a request will result in an unsafe state, requesting process is suspended until enough resources are freed by processes that will terminate in the meanwhile

Safety Checking: Determining a Safe State

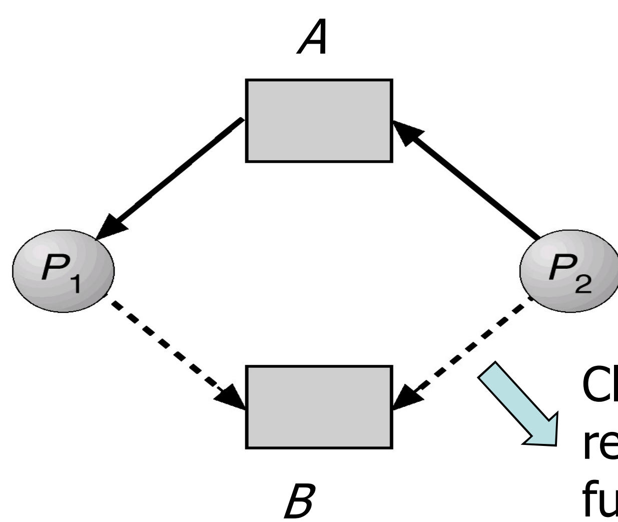
- **Safe state** = ∃ a safe sequence $<P_1, P_2, \ldots, P_n>$ of terminating all processes
  - For each P_i, the requests can still be granted by
    - Currently available resources
    - Resources held by $P_1, P_2, \ldots, P_{i-1}$

- The system can schedule the processes as follows
  - If P_i's need for resources cannot be satisfied immediately, then it can
    - Wait until all $P_1, P_2, \ldots, P_{i-1}$ have finished
    - Obtain needed resources, execute, release resources, terminate
  - Then the next process can obtain its needed resources, and so on

Simple Example (Snapshot 1)

	Allocation	Max	Need	Available
	A B	A B	A B	A B
P1	1 0	1 1	0 1	0 1
P2	0 0	1 1	1 1	

- P2 currently requests resource A (hold by P1)
  - The system is in a safe state since the sequence < P1, P2> satisfies safety criteria



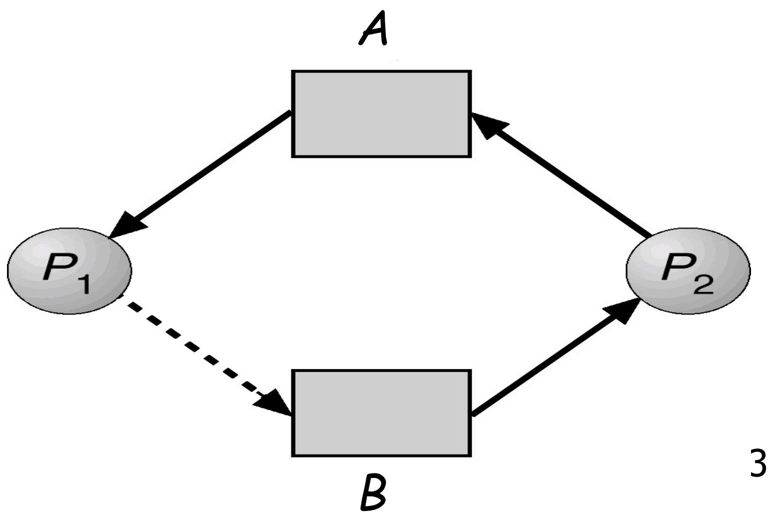
Claim edge: May request resource at some time in future

Simple Example (Snapshot 2)

- Consider P_2 also requests B

	Allocation	Max	Need	Available
	A B	A B	A B	A B
P1	1 0	1 1	0 1	0 0
P2	0 1	1 1	1 0	

- Allocating B to P2 leaves the system in an unsafe state since there is no sequence that satisfies safety criteria (Available vector is 0 !)



Banker's Algorithm for Deadlock Avoidance

Data Structures

- Max: n x m matrix (n =no. of processes, m= no. of resource types)
  - Max [i,j] = k : P_i may request max k instances of resource type R_j
- Allocation: n x m matrix
  - Allocation[i,j] = k : P_i is currently allocated k instances of R_j
- Available: length m vector
  - Available [j] = k : k instances of resource type R_j available
- Need: n x m matrix
  - Need [i,j] = Max[i,j] – Allocation[i,j]: potential max request by P_i for resource type R_j

RECALL: Avoidance = Ensure that system will not enter unsafe state

Banker's Algorithm – Example

- Processes P0 through P4
- Resource types A (10 instances), B (5 instances), C (7 instances)
- Snapshot at time T0

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- How to check whether the system is in safe state?

Banker's Algorithm: Safety Check

- Work and Finish: Auxiliary vectors of length m and n, respectively

```
init:
    Work := Available
    Finish[i] = false for i = 1,2, …, n.

while (∃i s.t. (Finish [i] = false) && (Need_i ≤ Work )) do
    Work := Work + Allocation_i
    Finish[i] := true

if (∀i Finish[i] = true) then
    system is in a safe state
else
    state is unsafe
```

Banker's Algorithm – Example

- Processes P0 through P4
- Resource types A (10 instances), B (5 instances), C (7 instances)
- Snapshot at time T0

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

Banker's Algorithm: Resource Allocation

```
For each new Request_i  do
    Assert Request_i ≤ Need_i
    if Request_i > Available ⇒ P_i must wait
    else
        remember the current resource-allocation state S;
        Available := Available - Request_i;
        Allocation_i := Allocation_i + Request_i;
        Need_i := Need_i – Request_i;
        if (safety-check is OK) ⇒
            commit allocation of resources to P_i
        else ( unsafe ) ⇒
            P_i must wait
            restore resource-allocation state S;
```

Banker's Algorithm – Example

- Processes P1 request (1,0,2)

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- Check $REQUEST_1 \leq Need_1$ (that is $(1,0,2) \leq (1,2,2)$)
- Check that $Request_1 \leq Available$ (that is, $(1,0,2) \leq (3,3,2)$) $\Rightarrow$ true

Banker's Algorithm – Example

- Processes P1 request (1,0,2)

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	2 3 0
P1	3 0 2	3 2 2	0 2 0	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement

Banker's Algorithm – Example

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	2 3 0
P1	3 0 2	3 2 2	0 2 0	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detection & Recovery

- Allow system to enter a deadlock state
- Detection algorithm
  - Using Banker's algorithm idea
    - Resources can have multiple instances
  - Using resource allocation graph
    - Resources have single instance
- Recovery scheme

Deadlock Detection Using Banker's Algorithm

- Similar as detecting unsafe states using Banker's algorithm
  - How is similarity explained?
  - If they cost the same why not use avoidance instead of detection & recovery?

Data structures

- Available: Vector of length m: number of available resources of each type
- Allocation: n x m matrix: number of resources of each type currently allocated to each process
- Request: n x m matrix: current request of each process
  - Request [i j] = k: P_i is requesting k more instances of resource type R_j

Deadlock Detection Using Banker's Algorithm

- Work and Finish: Auxiliary vectors of length m and n, respectively

```
init:
    Work := Available
    Finish[i] = false for i = 1,2, …, n.

while (∃i s.t. (Finish [i] = false) && (Request_i ≤ Work )) do
    Work := Work + Allocation_i
    Finish[i] := true

if (∃i Finish[i] = false) then
    system is in a deadlock state
```

Example of Detection Algorithm

- Five processes P_0 through P_4
- Three resource types
  - A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in Finish[i] = true for all i

Example of Detection Algorithm

- P_2 requests an additional instance of type C
- Snapshot at time T_0

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

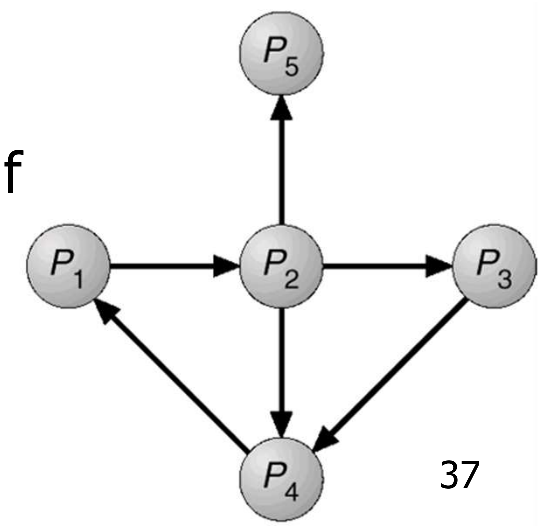
Deadlock Detection Using Graphs

- Deadlock detection using wait-for graph
  - Variant of resource allocation graph

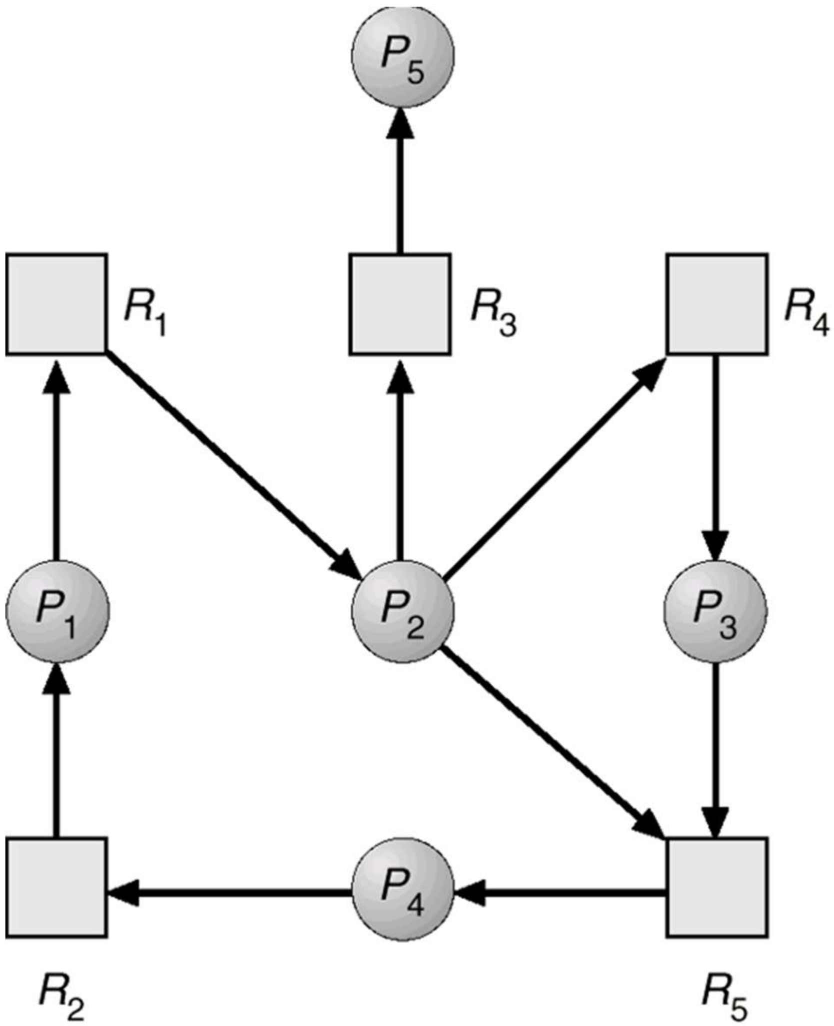
- Wait-for graph only contains edges between processes
  - Edge $P_i \rightarrow P_j$ in wait-for graph implies
  - Process P_i is waiting for P_j to release a resource that P_i needs

- Wait-for graph is obtained from resource allocation graph
  - Removing the resource nodes
  - Collapsing the appropriate edges

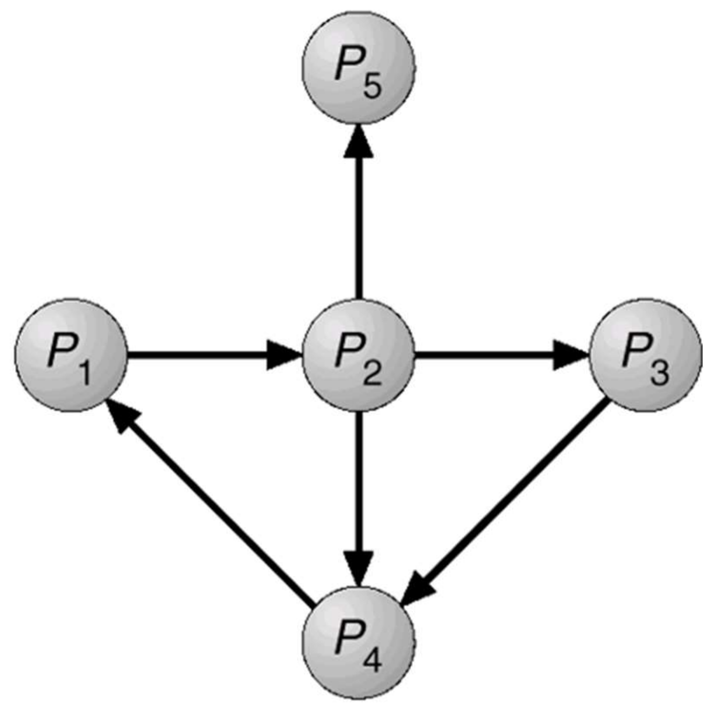
- Edge $P_i \rightarrow P_j$ exists in wait-for graph if and only if
  - Resource allocation graph contains two edges
  - $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q



Deadlock Detection Using Graphs



Resource-Allocation Graph



Wait-for graph

Deadlock Detection Using Graphs

- Deadlock exists if and only if wait-for graph contains cycle
- Periodically invoke an algorithm that searches for a cycle in graph
  - Which algorithm can be used for deadlock detection?
- An algorithm to detect a cycle in a graph requires an order of n^2 operations
  - Where n is the number of vertices in the graph

Detection Algorithm Usage

- Note: cost for detection algorithm is the same as that for the Bankers algorithm!

- When, and how often, to invoke depends on
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?

- If algorithm is invoked arbitrarily, there may be many cycles in the resource graph
  - We would not be able to tell which of the many deadlocked processes "caused" the deadlock

How to Recover From Deadlocks

- Once there is a deadlock situation
  - No way that all processes can proceed with the current allocation state

- Two main techniques to recover
  - Process Termination
  - Resource Preemption

Recovery From Deadlock: Process Termination

- Abort all deadlocked processes
  - Most common
  - Long computations may be lost
- Abort one process at a time until deadlock is eliminated
  - Need to run after every abort another instance of the deadlock detection algorithm

- In which order should we choose to abort? Criteria?
  - Least amount of processor time consumed so far
  - Least amount of output produced so far
  - Most estimated time remaining
  - Least total resources allocated so far
  - Lowest priority

Recovery From Deadlock: Resource Preemption

- Select a victim
  - Which resources to preempt from which processes
  - As in process termination, determine order of preemption to minimize cost
- Rollback
  - Return to some safe state, restart process from that state
  - Checkpointing necessary
- Watch for starvation
  - Resources may always be preempted from same process
  - A process must only be picked as victim only a (small) finite no. of times
    - Number of rollbacks are a cost factor

Combined Approach to Deadlock Handling

Combine the three basic approaches

- Prevention, avoidance, detection
- Can use the optimal approach for each type of resources in the system

Approach

- Group resources into number of different resource classes
- Use linear ordering to prevent deadlocks between resource classes (i.e., prevent circular wait)
- Use most appropriate technique for handling deadlocks within each class, e.g.,
  - Swap space (e.g., blocks in disk): Prevention by preallocation
  - Process resources (e.g., files): Avoidance by knowing max needs
  - Main memory: Prevention by preemption
  - Internal (e.g., I/O channels): Prevention by ordering

Summary

- Studied conditions when deadlocks occur

- Discussed options to deal with deadlocks

- Most operating systems completely ignore the problem
  - Unix, Windows, …

- None of the approaches suitable for all resources
  - Possibly need to select optimal approach for each class of resource

Any Question So Far?