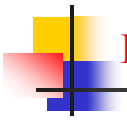
Chapter 5

DEADLOCK

Contents

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling



Learning Outcomes

- Understand what deadlock is and how it can occur when giving mutually exclusive access to multiple resources.
- Understand several approaches to mitigating the issue of deadlock in operating systems.
  - Including deadlock *prevention, detection and recovery*, and deadlock *avoidance*

Resources

- Examples of computer resources
  - Printers
  - tape drives
  - Tables in a database
- Processes need access to resources in reasonable order
- Preemptable resources
  - can be taken away from a process with no ill effects
- Nonpreemptable resources
  - will cause the process to fail if taken away

Resources & Deadlocks

- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so – *Deadlocked*
- Deadlocks occur when …
  - processes are granted exclusive access to devices, locks, tables,etc..
  - we refer to these entities generally as resources

Resource Access

- Sequence of events required to use a resource
  - 1. request the resource
  - 2. use the resource
  - 3. release the resource
- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

Introduction to Deadlocks

- Formal definition :
  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is release of a currently held resource
- None of the processes can …
  - ✓ Run
  - ✓ release resources
  - ✓ be awakened

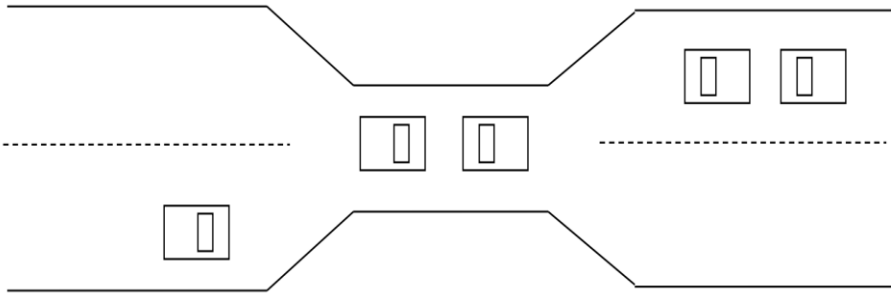
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - ✓ System has 2 tape drives.
  - ✓ P1 and P2 each hold one tape drive and each needs another one.
- Example
  - ✓ semaphores A and B, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- 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 if a deadlock occurs.
- Starvation is possible



Four Conditions for Deadlock

- Deadlock can arise if four conditions hold simultaneously
- 1. Mutual exclusion condition
  - ✓ each resource assigned to 1 process or is available
- 2. Hold and wait condition
  - ✓ process holding resources can request additional
- 3. No preemption condition
  - ✓ previously granted resources cannot be forcibly taken away
- 4. Circular wait condition
  - ✓ must be a circular chain of 2 or more processes
  - ✓ each is waiting for resource held by next member of the chain

System Model

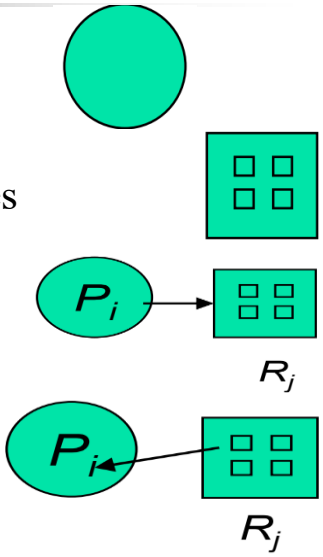
- Resource types R1, R2, . . ., Rm
  CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances.
- Each process utilizes a resource as follows:
  - Request
  - Use
  - release

Resource Allocation Graph

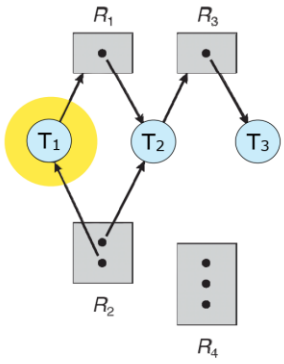
- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
- Request edge – directed edge P1 → Rj
- Assignment edge – directed edge Rj → Pi

Resource Allocation Graph (Cont)

- Process
- Resource Type with 4 instances
- Pi requests instance of Rj
- Pi is holding an instance of Rj



Example of a resource allocation graph (RAG)



T_1
Holds one instance of the **R_2** resource. **Waits** for the **R_1** resource.

T_2
Holds the **R_1** resource and one instance of the **R_2** resource. **Waits** for the **R_3** resource.

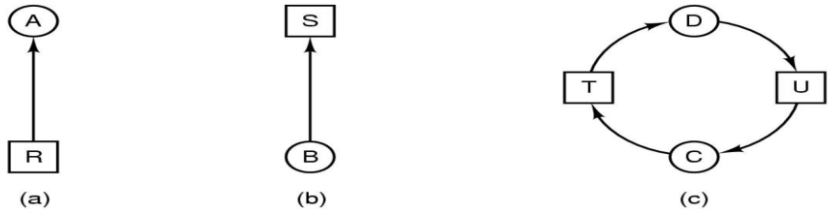
T_3
Holds the **R_3** resource.

Deadlock? **No deadlock!**

When P_3 releases the R_3 resource, T_2 will obtain R_3. Eventually T_2 will release R_1 and T_1 will be able to get access to R_1.

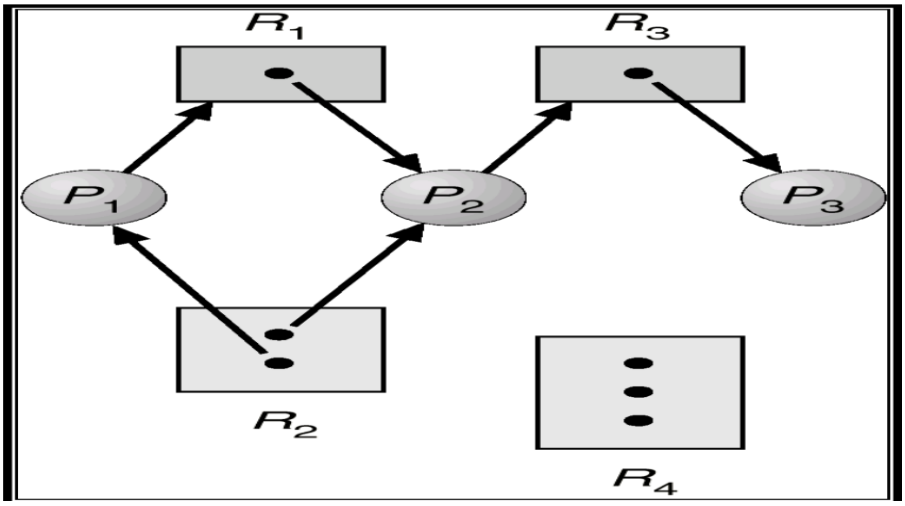
Deadlock Modeling - **Example**

- Modeled with directed graphs

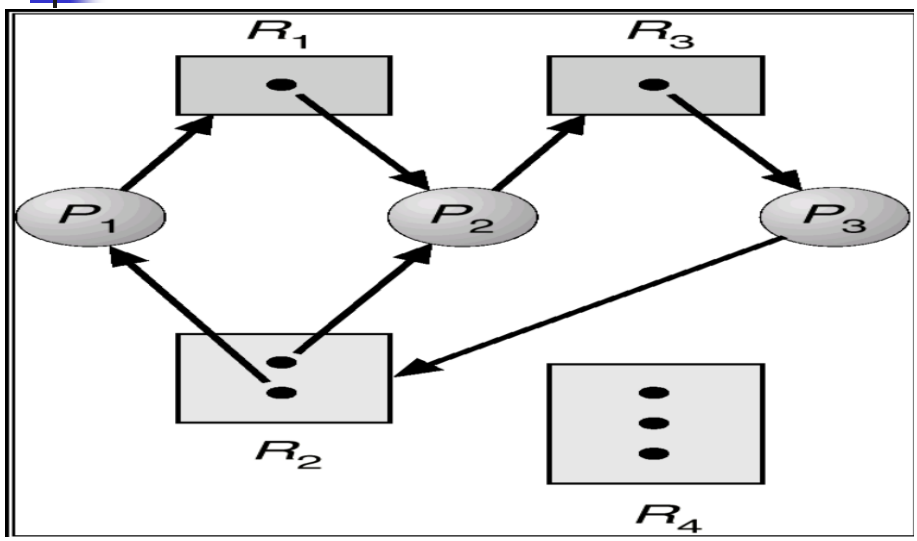


- Resource R assigned to process A
- Process B is requesting/waiting for resource S
- Process C and D are in deadlock over resources T and U

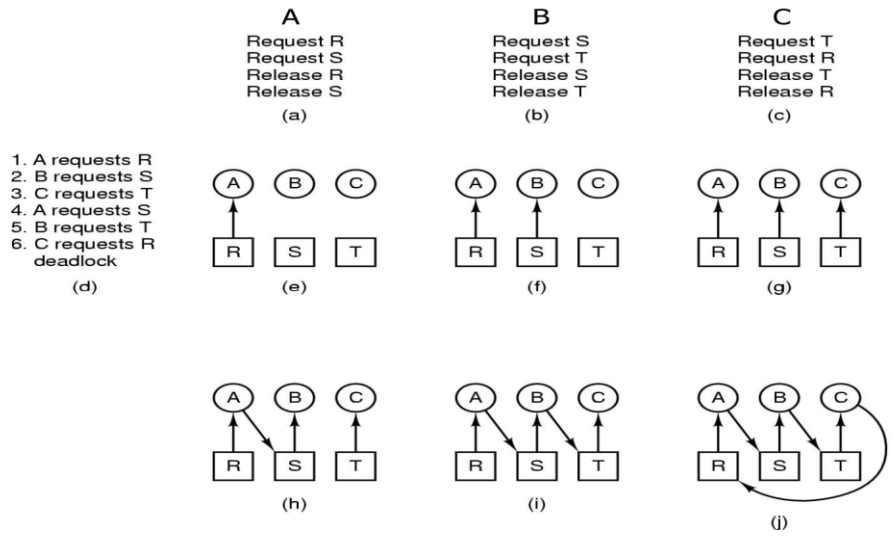
Example of Resource Allocation graph

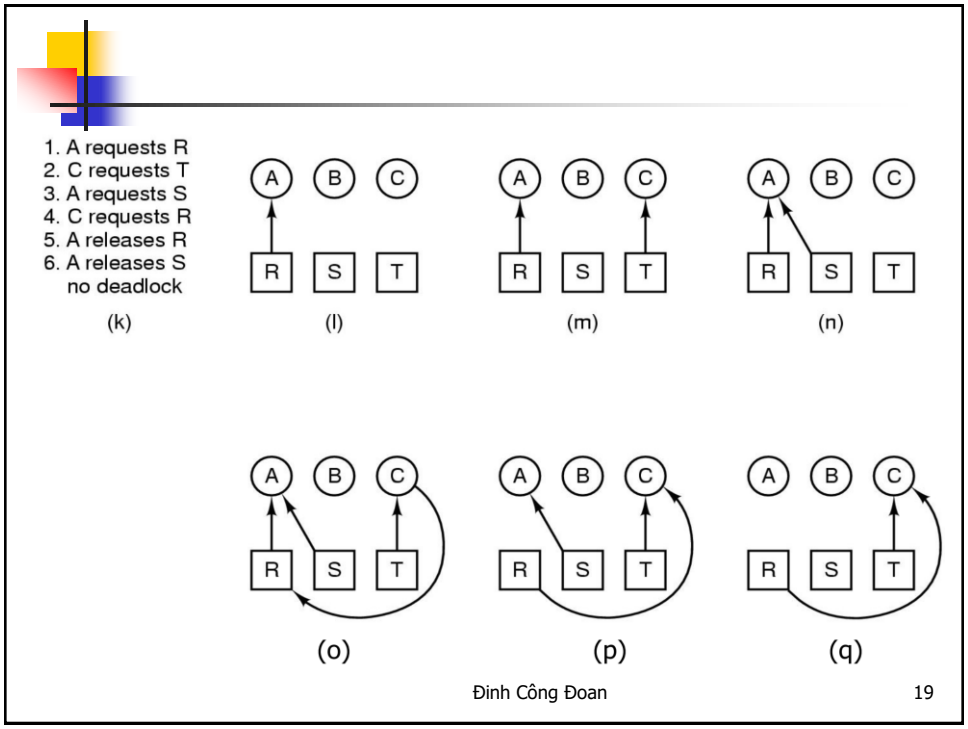


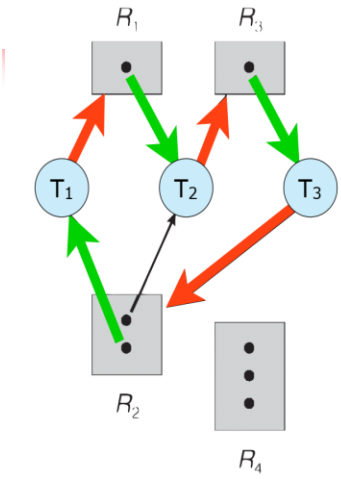
Resource allocation graph with a deadlock



Deadlock Modeling - **Example**







T_1

Holds one instance of the R_2 resource.

Waits for the R_1 resource held by T_2.

T_2

Holds the R_1 resource and one instance of the R_2 resource.

Waits for the R_3 resource held by T_3.

T_3

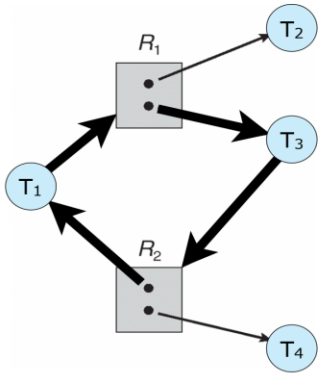
Holds the R_3 resource.

Waits for one instance of the two R_2 resources (one of which is held by T_1 and one which is held by T_2).

Deadlock? Deadlock!

All processes are waiting for a resource held by another process also waiting for a resource ...

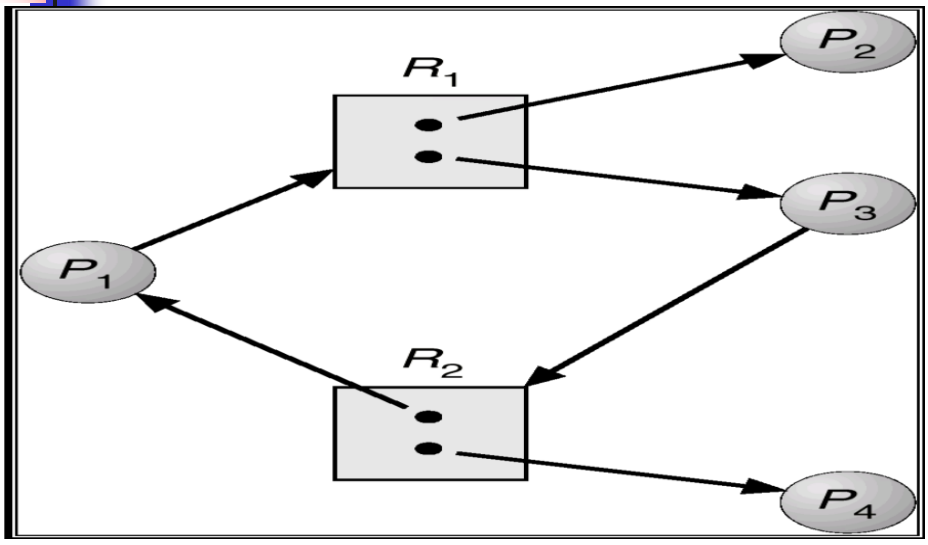
RAG with a cycle ... but no deadlock



Deadlock? **No deadlock!**

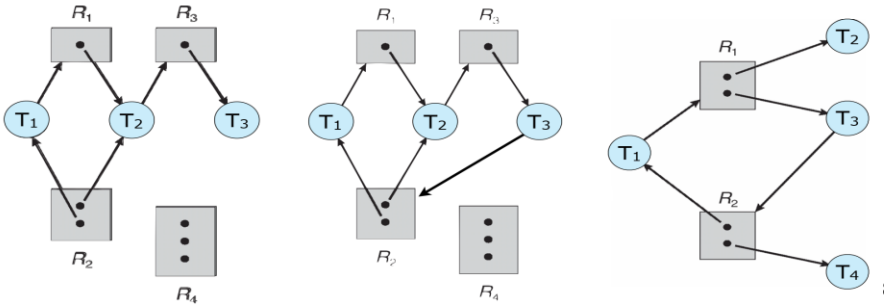
There are several instances of the resources R_1 and R_2. Once T_2 releases R_1, process T_1 will be unblocked. Once T_4 releases R_2, process T_3 will be unblocked.

Resource allocation graph with a circle but no deadlock



Basic facts

- If graph contains no cycles → no deadlock.
- If graph contains a cycle →
  - ✓ if only one instance per resource type,then deadlock.
  - ✓ if several instances per resource type, possibility of deadlock.



Methods for handing deadlocks

- Strategies for dealing with Deadlocks
- 1. just ignore the problem altogether: Ensure that the system will never enter a deadlock state.
- 2. prevention
  - ✓ negating one of the four necessary conditions
- 3. detection and recovery: Allow the system to enter a deadlock state and then recover.
- 4. dynamic avoidance
  - ✓ careful resource allocation

Approach 1: The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - ✓ deadlocks occur very rarely
  - ✓ cost of prevention is high
    - Example of "cost", only one process runs at a time
- UNIX and Windows takes this approach for some of the more complex resource relationships to manage
- It's a trade off between
  - ✓ Convenience (engineering approach)
  - ✓ Correctness (mathematical approach)

Deadlock prevention

- **Preventing** deadlocks by constraining how requests for resources can be made in the system and how they are handled (**system design**).
- The **goal** is to ensure that at least **one** of the four necessary **conditions** for deadlock can **never hold**

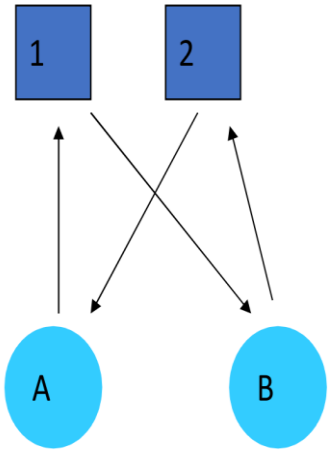
Approach 2: Deadlock Prevention

- Resource allocation rules prevent deadlock by prevent one of the four conditions required for deadlock from occurring
  - ✓ Mutual exclusion
  - ✓ Hold and wait
  - ✓ No preemption
  - ✓ Circular Wait
- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

- Hold and Wait Condition
  - ✓ Require processes to request resources before starting
    - a process never has to wait for what it needs
  - ✓ Issues
    - may not know required resources at start of run
      - → not always possible
    - also ties up resources other processes could be using
  - ✓ Variations:
    - process must give up all resources if it would block holding a resource
    - then request all immediately needed
    - prone to livelock

- No Preemption Condition
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all
  - resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- Circular Wait Condition
  - The displayed deadlock cannot happen
  - If A requires ***1***, it must acquire it before acquiring ***2***
  - Note: If B has ***1***, all higher numbered resources must be free or held by processes who doesn't need ***1***
  - Resources ordering is a common technique in practice!!!!!



Conditions for deadlock

Mutual exclusion: only one task at a time can use a resource instance.

Hold and wait: a task holding at least one resource is waiting to acquire additional resources held by other tasks.

No preemption: a resource can be released only voluntarily by the task holding it, after that task has completed its task.

Circular wait: there exists a set $\{T_0, T_1, \ldots, T_0\}$ of waiting tasks such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by $T_2, \ldots, T_{n-1}$ is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Deadlock prevention

Restrain the ways requests can be made:

- ★ **Mutual exclusion** – not required for sharable resource instances;
  - Must hold for non-sharable resource instances.
- ★ **Hold and wait** – must guarantee that whenever a task requests a resource, it does not hold any other resources:
  - Require processes to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none.
  - Low resource utilization; **starvation** possible.

Conditions for deadlock

Mutual exclusion: only one task at a time can use a resource instance.

Hold and wait: a task holding at least one resource is waiting to acquire additional resources held by other tasks.

No preemption: a resource can be released only voluntarily by the task holding it, after that task has completed its task.

Circular wait: there exists a set $\{T_0, T_1, \ldots, T_0\}$ of waiting tasks such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by $T_2, \ldots, T_{n-1}$ is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

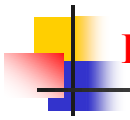
Deadlock prevention (continued)

Restrain the ways requests can be made:

- ★ **No preemption**

  If a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released:
  - Preempted resources are added to the list of resources for which the task is waiting.
  - The task will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- ★ **Circular wait**

  Impose a **total ordering** of all resource types, and require that each task requests resources in an increasing order of enumeration.



Deadlock avoidance

- The system **dynamically** considers every request and **decides** whether it is **safe** to grant it at this point,
- Requires additional **apriori information** regarding the overall potential use of each resource for each task.
- Allows for more concurrency.

Deadlock Avoidance

- Instead of detecting deadlock, can we simply avoid it?
  - ✓ YES, but only if enough information is available in advance.
  - ✓ Maximum number of each resource required
  - ✓ Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
  - ✓ The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - ✓ Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock avoidance

- Requires that the system has some additional **a priori information** available.
- **A priori** knowledge is independent of experience - information given to us before we start
  - ✓ Simplest and most useful model requires that each task declare the **maximum** number of **resources** of each type that it may **need**.
  - ✓ The deadlock-avoidance algorithm **dynamically examines** the resource-allocation **state** to **ensure** that there can **never** be a **circular-wait** condition.
  - ✓ Resource-allocation **state** is defined by the number of **available** and **allocated resources**, and the potential **maximum demands** of the tasks

Deadlock prevention vs deadlock avoidance

- The difference between deadlock prevention and deadlock avoidance is similar to the difference between a traffic light and a police officer directing traffic

Prevention

Follow static rules strictly to stay out of trouble.

Avoidance

Dynamic and more adaptive to the actual state of the whole system.

Safe state

- A state is *safe* if
  - The system is not deadlocked
  - There exists a scheduling order that results in every process running to completion, *even if they all request their maximum resources immediately*

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence <P1, P2, …, Pn> is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j < i.
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
  - When Pi terminates, Pi+1 can obtain its needed resources, and so on.

Safe and Unsafe States

- Demonstration that the state in (a) is safe

Note: We have 10 units of the resource

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Safe and Unsafe States

- *A* requests one extra unit resulting in (b)
- Demonstration that the state in b is not safe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

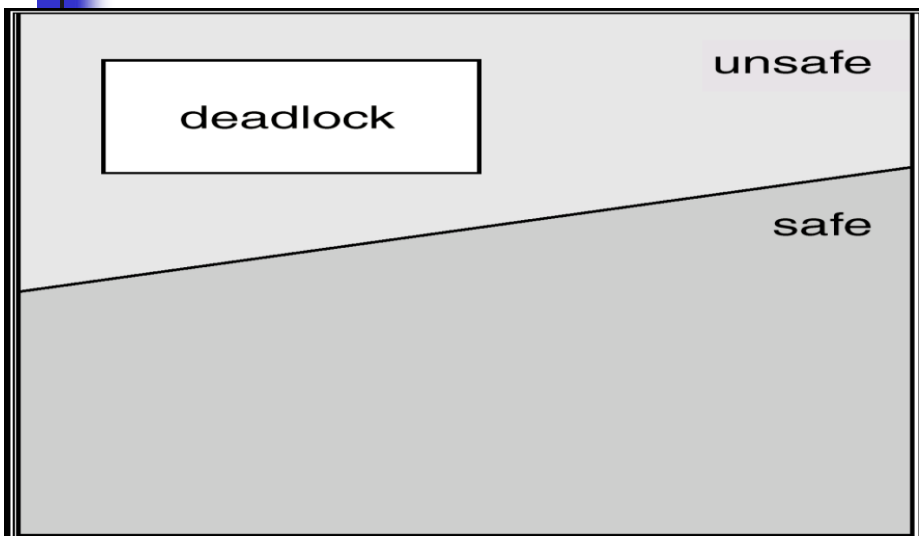
Free: 4

(d)

Basic facts

- If a system is in safe state → no deadlocks.
- If a system is in unsafe state → possibility of deadlock.
- Avoidance → ensure that a system will never enter an unsafe state.
- Unsafe states are not necessarily deadlocked
  - ✓ With a lucky sequence, all processes may complete
  - ✓ However, we *cannot guarantee* that they will complete (not deadlock)
- Safe states guarantee we will eventually complete all processes
- Deadlock avoidance algorithm:
  - ✓ Only grant requests that result in safe states

Safe, Unsafe Deadlock state



Resource allocation graph Algorithm

- Claim edge Pi → Rj indicated that process Pj may request resource Rj; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.

Resource allocation graph algorithm

★ Resources must be claimed **a priori** in the system.

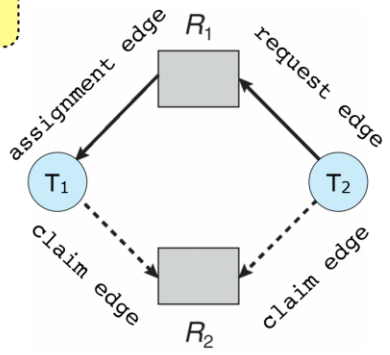
★ **Claim edge** $T_i \rightarrow R_j$ indicated that task T_j may request resource R_j; represented by a dashed line.

★ Claim edge converts to request edge when a process requests a resource.

★ **Request edge** converted to an assignment edge when the resource is allocated to the process.

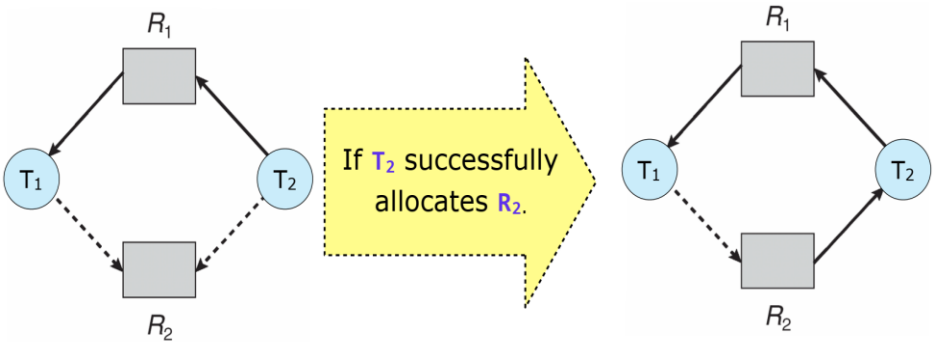
★ When a resource is released by a task, assignment edge reconverts to a claim edge.

a priori (latin "from the earlier")



Resource allocation graph algorithm

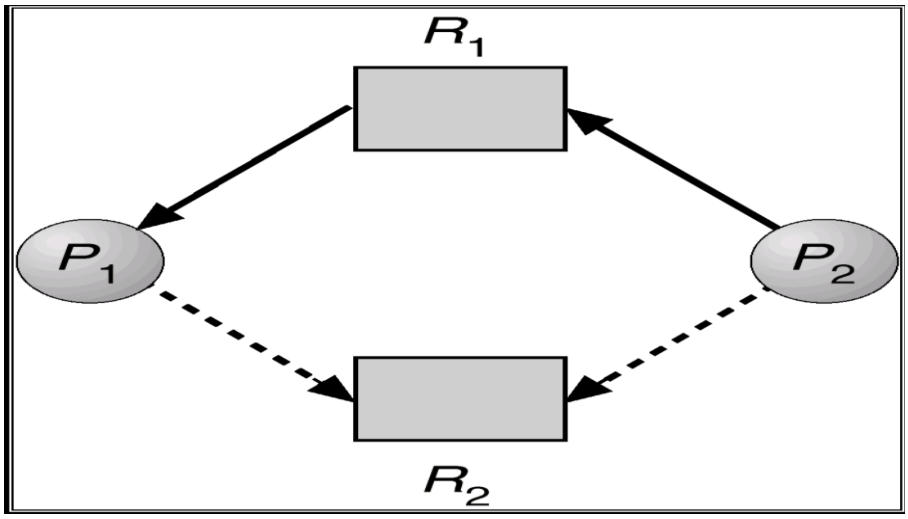
- Suppose that process **Pi** requests a resource **Rj**.
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.



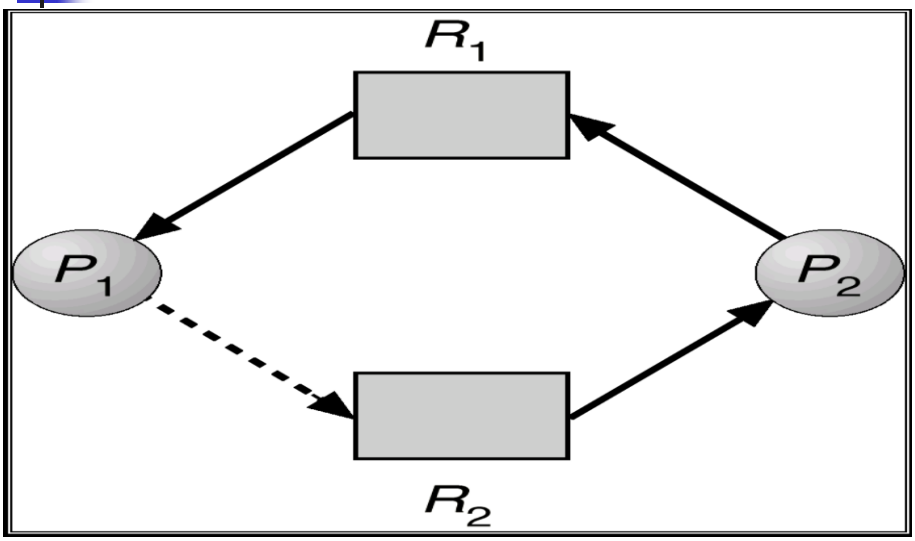
This state is **safe**

This state is **not safe**

Resource allocation graph for deadlock avoidance



Unsafe state in resource allocation graph



Banker's algorithm

- The Banker's algorithm is a resource allocation and **deadlock avoidance algorithm** developed by Edsger Dijkstra.
  - ✓ Tests for safety by **simulating** the **allocation** of **predetermined maximum possible amounts of all resources** ...
  - ✓ ... and then makes a "safe-state" check to **test for possible deadlock conditions for all other pending activities**, before deciding whether allocation should be allowed to continue.

- **Multiple instances** of each resource.
- Each task must **a priori claim maximum use**.
  - **A priori** knowledge is independent of experience - information given to us before we start.
  - When a new task enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system
- When a task **requests** a resource it **may have to wait**.
- When a task gets all its resources it must return them in a finite amount of time.

Bankers Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data structure for banker's algorithm

- Let n = number of processes, and m = number of esources types.
- Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available.
- Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
- Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
- Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.
  Need [i,j] = Max[i,j] – Allocation [i,j]

Safety algorithm

- 1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
  Work = Available
  Finish [i] = false for i - 1,3, …, n.
- 2. Find an i such that both:
  (a) Finish [i] = false
  (b) Needi ≤ Work
  If no such i exists, go to step 4.
- 3. Work = Work + Allocationi
  Finish[i] = true
  go to step 2.
- 4. If Finish [i] = = true for all i, then the system is in a safe state

Resource request algorithm for process Pi

- Request = request vector for process Pi. If Requesti[j]= k then process P i wants k instances of resource type Rj.
- 1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
- 2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.
- 3. Pretend to allocate requested resources to Pi by modifying the state as follows:

  Available = Available = Requesti;
  Allocationi= Allocationi + Requesti;
  Needi = Needi – Requesti;;
  - ✓ If safe → the resources are allocated to Pi.
  - ✓ If unsafe → Pi must wait, and the old resource-allocation state is restored

Example of banker's algorithm

- 5 processes P0 through P4; 3 resource types A (10 instances), B (5instances, and C (7instances).
- Snapshot at time T0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (cont)

- The content of the matrix. Need is defined to be Max– Allocation.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

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

Example P1 request (1, 0, 2) (Cont)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) → true

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0,P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

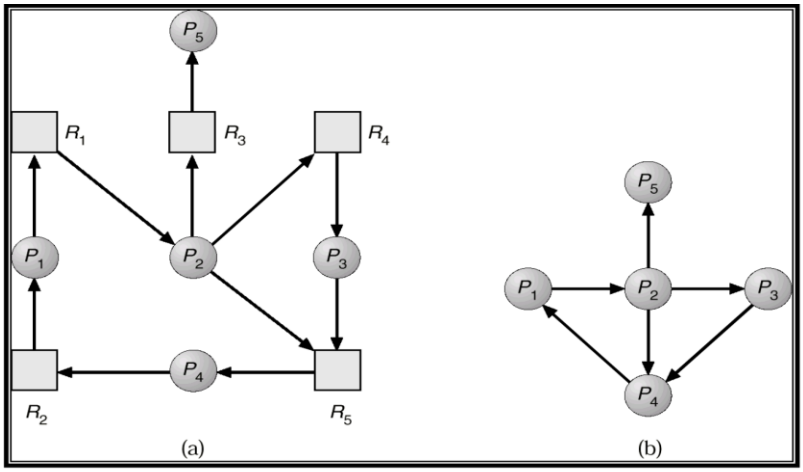
Deadlock detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Detection with One Resource of Each Type

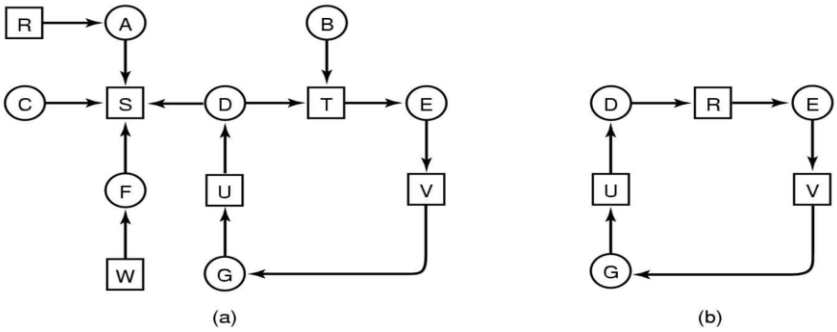
- Maintain wait-for graph
  - ✓ Nodes are processes.
  - ✓ Pi → Pj if Pi is waiting for Pj.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph.

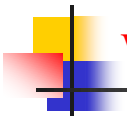
Resource allocation graph and wait for graph



Resource-Allocation Graph Corresponding wait-for graph

- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock





What about resources with multiple units?

- Examples
  - ✓ KBs of memory to execute
  - ✓ DVD drives to duplicate a disk
  - ✓ Blocks on disk to store files
- We need an approach for dealing with resources that consist of more than a single unit.

Detection with Multiple Resources of Each Type

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.
- Request: An n x m matrix indicates the current request of each process. If Request[ij] = k, then process Pi is requesting k more instances of resource type. Rj.

- Data structures needed by deadlock detection

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

- Note the following invariant
- Sum of current resource allocation + resources available = resources that exist

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Detection algorithm

- 1. Let Work and Finish be vectors of length m and n, respectively Initialize:
  (a) Work = Available
  (b) For i = 1,2, …, n, if Allocationi ≠ 0, then Finish[i] = false;otherwise, Finish[i] = true.
- 2. Find an index i such that both:
  (a) Finish[i] == false
  (b) Requesti ≤ Work
  If no such i exists, go to step 4.
- 3. Work = Work + Allocationi
  Finish[i] = true
  go to step 2.
- 4. If Finish[i] == false, for some i, 1 ≤ i ≤n, then the system is in deadlock state. Moreover, if Finish[i] ==false, then Pi is deadlocked.

Example of detection algorithm

- Five processes P0 through P4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T0:

	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 <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i.

- P2 requests an additional instance of type C
- State of system?
  - ✓ Can reclaim resources held by process P0, but insufficient resources to fulfill other processes; requests.
  - ✓ Deadlock exists, consisting of processes P1, P2, P3, and P4.

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

Detection algorithm Usage

- 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?
  - ✓ one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery form deadlock: Process termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - ✓ Priority of the process.
  - ✓ How long process has computed, and how much longer to completion.
  - ✓ Resources the process has used.
  - ✓ Resources process needs to complete.
  - ✓ How many processes will need to be terminated.
  - ✓ Is process interactive or batch?

Recovery form deadlock: resource preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Combined approach to Deadlock handling

- Combine the three basic approaches
  - ✓ Prevention
  - ✓ Avoidance
  - ✓ detection

  allowing the use of the optimal approach for each of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.