

COMMONWEALTH OF AUSTRALIA
Copyright Regulation 1969

WARNING

This material has been copied and communicated to you by or on behalf
of **Curtin University of Technology** pursuant to Part VB of the
Copyright Act 1968 (**the Act**)

The material in this communication may be subject to copyright under the
Act. Any further copying or communication of this material by you
may be the subject of copyright protection under the Act.

Do not remove this notice

Operating Systems

COMP2006

Deadlocks

Lecture 5

Deadlocks

References:

Silberschatz, Galvin, and Gagne, *Operating System Concepts*, Chapter 7.

Topics:

- Deadlock system model and characterization.
- Methods for handling deadlocks.
- Deadlock prevention, avoidance, and detection.
- Recovery from deadlock.
- Combined approach to deadlock handling.

Introduction

- ★ Several processes may compete for a finite number of resources, and some of them may wait for the resources forever because the resources are held by other waiting processes → deadlock.
- ★ A set of processes is in a deadlock state if every process in the set is waiting for an event that can be caused only by another process in the set.

Example

- ★ System has two tape drives.
- ★ P1 and P2 each hold one tape drive and each needs another one.

Example

- ★ Semaphores A and B, initialized to 1.

P0

wait (A)

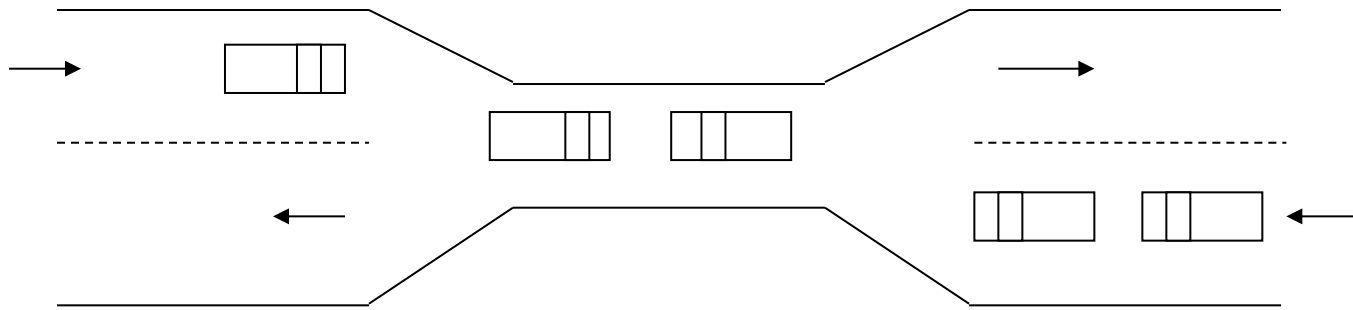
wait (B)

P1

wait (B)

wait (A)

Example (bridge crossing)



- ★ 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 back up if a deadlock occurs.
- ★ Starvation is possible.

System Model

- ★ Resources are partitioned into several types, each consists of some number of identical *instances*.
 - **Identical**: allocation of *any* instance of the type will satisfy process's request.
 - Resources may be physical resources (printers, tape drives, CPU cycles), or logical resources (files, semaphores, and monitors).
 - A **pre-emptible** resource is one that can be taken away from a process with no ill effect to the process; e.g., memory.
 - A **non-preemptible** resource is one that cannot be taken away from its user since it will make the user fails; e.g., printers
 - ★ In general, potential deadlocks involve this resource type.
- ★ Each process uses a resource as follows:
 - **Request** the resource; a process must wait if the resource is being used by another process.
 - **Use** the resource; e.g., the process can print on the printer.
 - **Release** the resource.

Necessary conditions for deadlock

Four conditions must hold for a deadlock to occur (Coffman et al.):

1. **Mutual exclusion condition.** Only one process at a time can use the resource.
 - or each resource is either currently assigned to exactly one process or is available.
 2. **Hold and wait condition.** A process holding at least one resource is waiting to acquire additional resources held by other processes.
 3. **No pre-emption condition.** A resource can be released only voluntarily by the process holding it after that process has completed its task.
 4. **Circular wait condition.** There exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .
- Note:** the four conditions are not completely independent, e.g., the circular-wait condition implies the hold-and-wait condition.

Deadlock Modelling

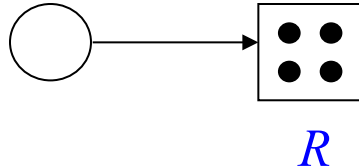
- ★ Deadlocks can be described more precisely in terms of a directed graph $G(V, E)$
 - called *System resource-allocation graph*
- ★ V is partitioned into two types:
 - Set of processes in the system: $P = \{P_1, P_2, \dots, P_n\}$.
 - Set of all resource types in the system: $R = \{R_1, R_2, \dots, R_n\}$
- ★ *Request edge* – directed edge $P_i \rightarrow R_j$
 - process P_i requests an instance of resource R_j
- ★ *Assignment edge* – directed edge $R_j \rightarrow P_i$
 - an instance of resource R_j has been allocated to process P_i

Model Symbols

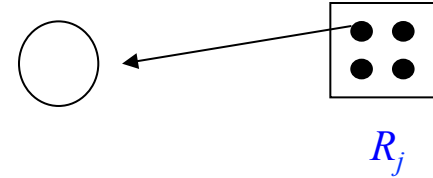
★ Process: ○

Resource type with 4 instances: □

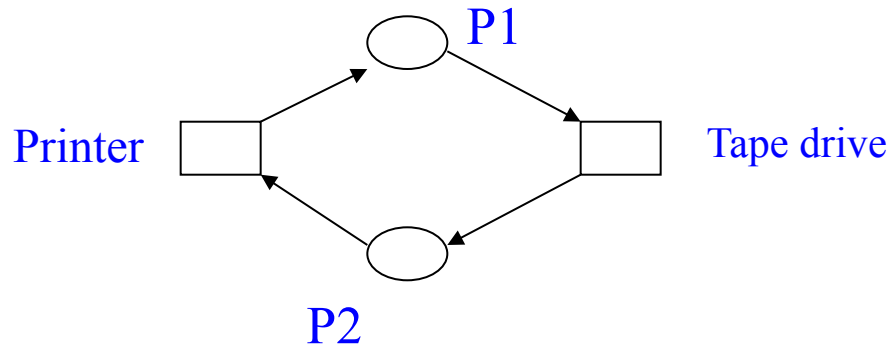
★ P_i requests R_j :



P_i uses R_j :

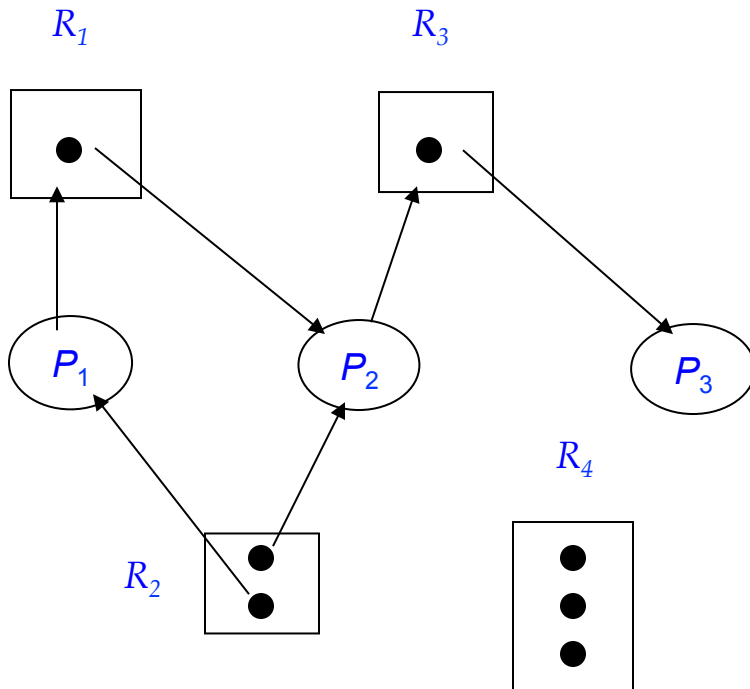


Example: Deadlock



- ★ If the graph contains **no cycles**, no process in the system is deadlocked.
- ★ If the graph contains **a cycle**, deadlock *may* exist.
 - If each resource type has **one instance**, **cycle means deadlock**.
 - If each resource type has **several instances**, cycle is necessary but **not sufficient condition for deadlock**.

Example: resource allocation graph (with no cycles)



The sets P, R, and E:

$$P = \{P_1, P_2, P_3\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

Resource instances:

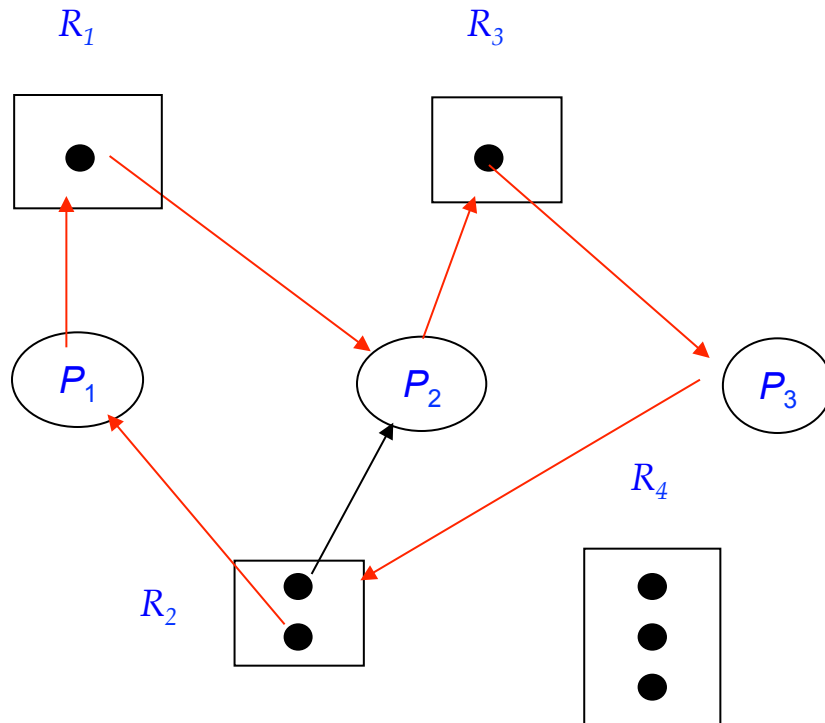
- ★ One instance of resource type R_1
- ★ Two instances of resource type R_2
- ★ One instance of resource type R_3
- ★ Three instances of resource type R_4

Process states:

- ★ P_1 is holding an instance of R_2 , and waiting for an instance of R_1
- ★ P_2 is holding an instance of R_1 and R_2 , and is waiting for an instance of R_3
- ★ P_3 is holding an instance of R_3

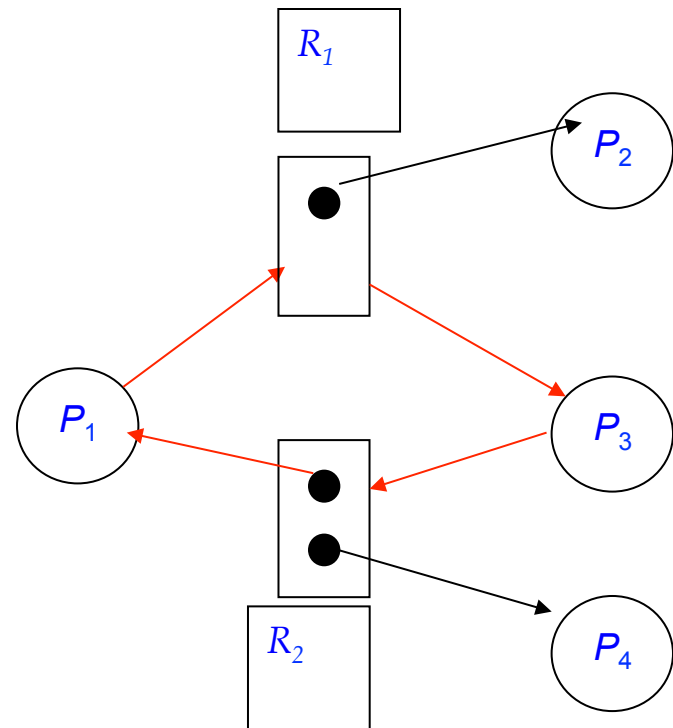
Example

A cycle and deadlock



A cycle but no deadlock

P_4 can release R_2 which gets allocated to P_3 ; breaking the cycle



Example: how deadlock occurs and how to avoid it

Consider:

- ★ Three resources: R, S, and T
- ★ Three processes: A, B, and C
- ★ The following is sequences of resource requests/releases:

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

Example (cont.)

(a) Request/release order:

(1) A requests R,

(2) B requests S,

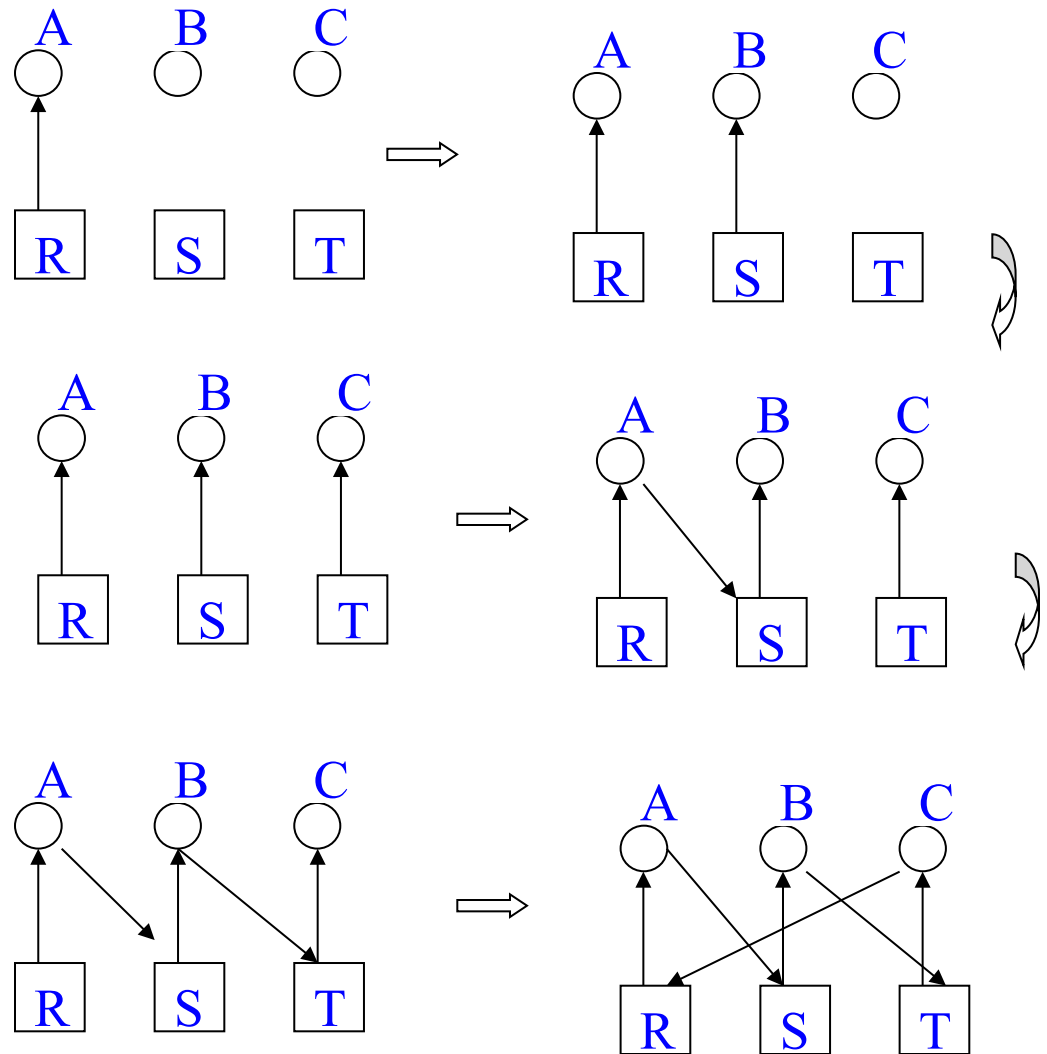
(3) C requests T,

(4) A requests S,

(5) B requests T,

(6) C requests R

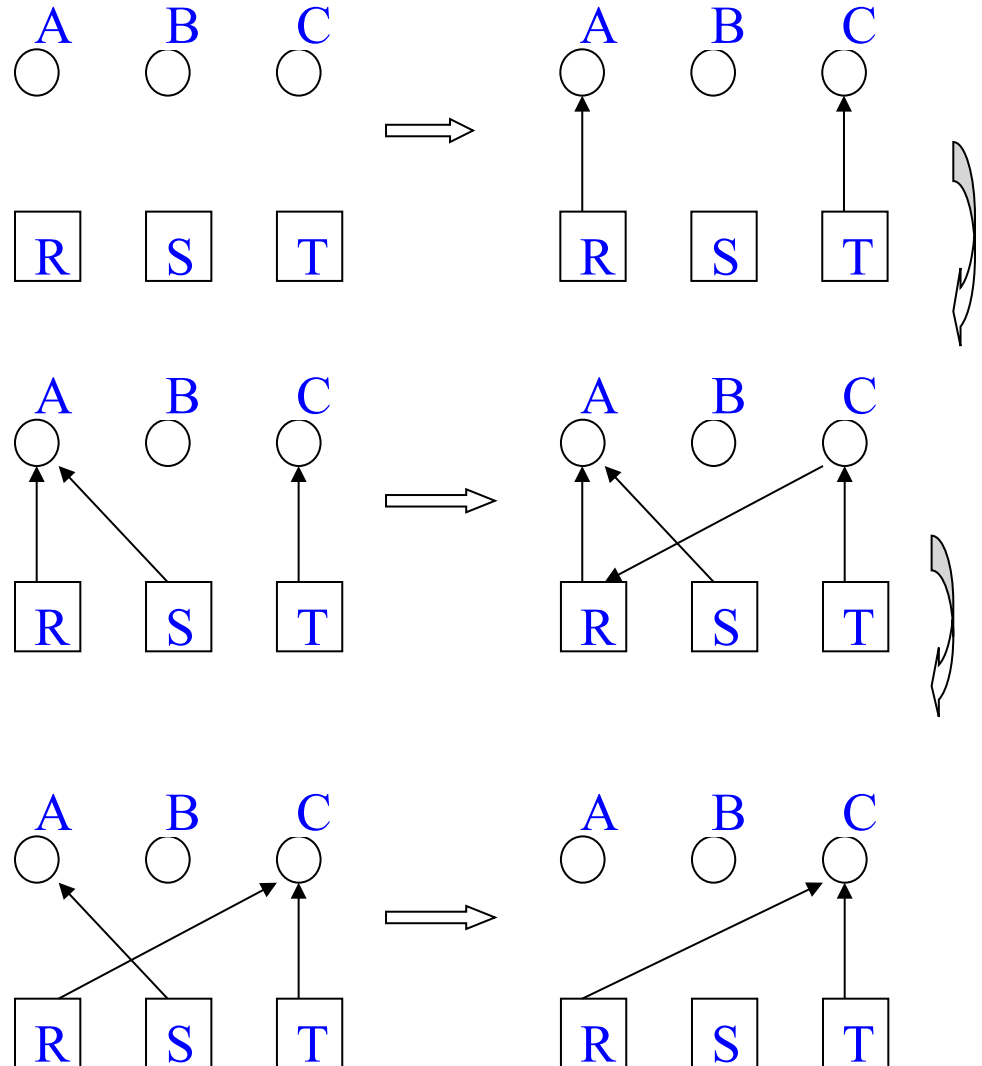
→ **deadlock.**



Example (cont.)

(b) Request/release order:

- (1) A requests R,
 - (2) C requests T,
 - (3) A requests S,
 - (4) C requests R,
 - (5) A release R,
 - (6) A release S
- no deadlock.



Three Methods for handling deadlock

- ★ Use a protocol to ensure that the system will *never* reaches deadlock
 - Using *deadlock prevention* and/or *deadlock avoidance* techniques
- ★ Allow the system to enter a deadlock state and then recover
 - needs *deadlock detection* and *deadlock recovery* algorithms
- ★ Ignore the problem and pretend that deadlocks never occur in the system
 - used by most OS, including UNIX
 - Also called the **ostrich** algorithm!

Deadlock prevention

- ★ Restrain the ways resource requests can be made
 - Use a set of methods to ensure that **any one** of the four deadlock conditions cannot hold

(1) Deny mutual exclusion

- Not required for sharable resources (e.g., read-only files, cannot be in deadlock)
- Must hold for non-sharable resources (a printer cannot be simultaneously shared by several processes)
- **In general**, it is not possible to prevent deadlock by denying mutual-exclusion condition since some resources are non-sharable

(2) Deny hold and wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources

Options:

- Each process is granted all resources before it starts
- Allows a process to request resources only when it has none
 - ★ If a process needs more resources, release all resources before requesting new ones

Problem:

- Resource utilisation is low
- Possible starvation.
 - ★ A process that needs popular resources may have to wait indefinitely

Deadlock prevention (cont.)

(3) Prevent no pre-emption (i.e., allow pre-emption)

- When a process holding some resources requests other resource that cannot be immediately allocated, it must release all resources currently being held
 - ★ The pre-empted resources are added to the process's list of requested resources
 - ★ The process is restarted when it regains its old resources and obtains the new one it is requesting

Problem:

- Can be applied easily to resources whose state can be saved easily (e.g., memory), but not so easily for others (e.g., printer)

Deadlock prevention (cont.)

(4) Deny circular wait

- All resource types are ordered, e.g.,
 - * $F(\text{card reader}) = 1$ $F(\text{disk drive}) = 5$
 - * $F(\text{tape drive}) = 7$ $F(\text{printer}) = 12$
- Each process must request increasing order of resources
- Protocol:
 - * Each process requests resources in increasing order
 - * Initially a process can request for any R_i
 - * After that, it can request R_j only if $F(R_j) > F(R_i)$
- **Problem:** It may be impossible to find a resource ordering that satisfies everyone

Deadlock avoidance

- ★ The system must have some additional *a priori* information about which resources a process will request and use during its lifetime
 - With the additional information, the system can decide for each request whether or not the process should wait
 - The 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**
- ★ A resource-allocation *state* is defined by:
 - The number of **available** and **allocated** resources, and
 - The **maximum demands** of the processes

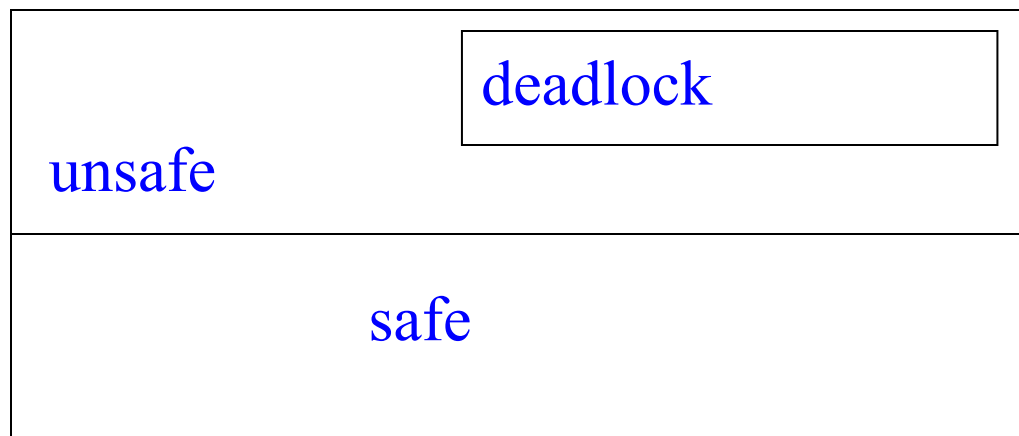
Safe State

- ★ When a process requests an available resource, the system checks if its allocation keeps the system in *safe state*
- ★ The system is in *safe state* if there exists a *safe sequence* of all processes
- ★ A sequence $\langle P_1, P_2, \dots, P_n \rangle$ is *safe* if, for each P_i , the resources requested by P_i can be allocated from the currently available resources + resources held by all P_j , with $j < i$
 - If P_i 's resource needs are not immediately available, P_i waits until all P_j have finished
 - When all P_j are finished, P_i obtains the needed resources, executes, returns the allocated resources, and terminates
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe State (cont.)

Basic facts

- ★ If a system is in safe state → no deadlocks
- ★ If a system is in unsafe state → possibility of deadlock
- ★ Avoidance ensures that the system never enters an unsafe state
- ★ A process requesting for a currently available resource may have to wait
 - Thus, resource allocation is lower than without deadlock avoidance algorithm



Example

Consider a system with 12 resources of the same type, and 3 processes with the following resource needs and allocation

	<u>Maximum needs</u>	<u>Allocation</u>	<u>Current need</u>
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

- ★ At time t_0 , **available resource = 3**, and the system is in safe state
 - There is a safe sequence $\langle P_1, P_0, P_2 \rangle$
- ★ What if at t_1 one more resource is allocated to process P_2 ?
 - The system is in unsafe state
 - ★ Deadlock can occur

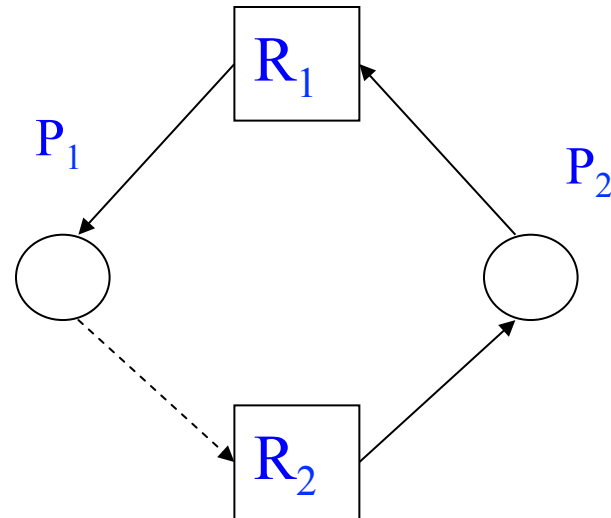
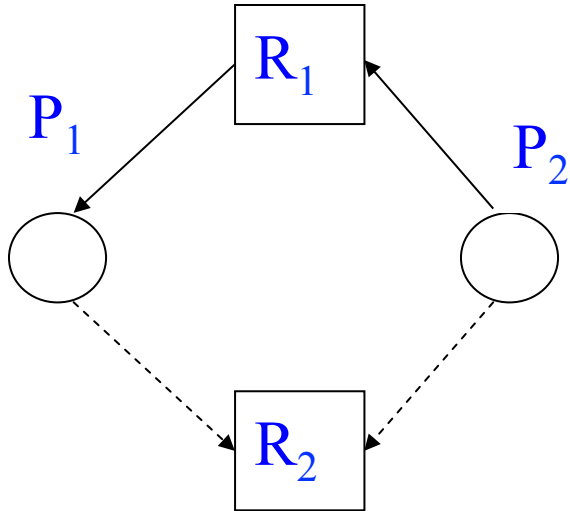
Resource-Allocation Graph Algorithm

- ★ *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j
 - 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* converts to a *claim edge*
- ★ Resources must be claimed *a priori* in the system
- ★ Need a cycle detection algorithm $\rightarrow O(n^2)$
- ★ This algorithm **can not be used** for system comprising resource types with **multiple instances**

Example

Suppose P_2 requests R_2

Although R_2 is currently free, allocating it to P_2 may lead to unsafe state (a cycle in right figure)



Banker's Algorithm

- ★ The algorithm for a system comprising resource types with **multiple instances**
- ★ Similar to a bank: never allocates its available cash if it can no longer satisfy the needs of all customers
- ★ Each process must *a priori* claim maximum number of instances of each resource type that it may need
- ★ When a process requests a resource:
 - It may have to wait (if resource allocation may lead to unsafe state) until some other process releases enough resources
- ★ When a process gets all its resources:
 - It must return them in a finite amount of time

Banker's Algorithm (cont.)

Algorithm

Let n = number of processes, and m = number of resource types

Data structures:

- ★ *Available*: Vector of length m
 - $available[j] = k$; means k instances of resource type R_j are available
- ★ *Max*: $n \times m$ matrix
 - $Max[i, j] = k$; means process P_i may request at most k instances of resource type R_j .
- ★ *Allocation*: $n \times m$ matrix
 - $Allocation[i, j] = k$; means process P_i is currently allocated k instances of resource type R_j
- ★ *Need*: $n \times m$ matrix
 - $Need[i, j] = k$; means process P_i may need k more instances of resource type R_j to complete its task.
 - $Need[i, j] = max[i, j] - allocation[i, j]$

Implementation of the safety algorithm

// Time complexity = $O(mn^2)$

1. Let *work* and *finish* be vectors of length *m* and *n*, respectively

 initialise:

work = *available*

finish [*i*] = *false* for *i* = 1, 2, ..., *n*

// Find an unfinished process *i*; it still needs resources

2. Find a value of *i* such that both:

- *finish*[*i*] = *false*, and
- *need*_{*i*} ≤ *work*
- If no such *i* exists, go to step 4

// process *i* pretends to finish, so it releases its resources i.e., *allocation*_{*i*}

3. *work* = *work* + *allocation*_{*i*}

finish[*i*] = *true*

 go to step 2

4. If *finish*[*i*] = *true* for all *i*, the system is in safe state.

Resource-request algorithm for process P_i

$Request_i$ = request vector for process P_i

If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j

1. If $request_i \leq need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $request_i \leq available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. The system pretends to allocate requested resources to P_i by modifying the state as follows:
 - $available = available - request_i$
 - $allocation_i = allocation_i + request_i$
 - $need_i = need_i - request_i$
 - If resulting state is safe, resources are allocated to P_i
 - else P_i must wait, and the old resource-allocation state is restored.

Example of Banker's algorithm

- ★ 5 processes P_0 through P_4 ; 3 resource types A (10 instances), B (5 instances), and C (7 instances)
- ★ Snapshot at time T_0 :

	<i>Allocation</i>			<i>Max</i>			<i>Available</i>			<i>Need</i>		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

- ★ The content of matrix *Need* is defined to be $Max - Allocation$
- ★ The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria

Example (P_1 requests (1,0,2)):

- ★ Check that $request \leq need$ (that is, $(1, 0, 2) \leq (1, 2, 2)) \rightarrow \text{true}$
- ★ Check that $request \leq available$ (that is, $(1, 0, 2) \leq (3, 3, 2)) \rightarrow \text{true}$

Before Adjustment

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

After Adjustment

	Alloc.			Need			Avail.		
	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	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- ★ $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ or $\langle P_1, P_4, P_3, P_0, P_2 \rangle$ satisfies safety requirement
- ★ Can request for (3, 3, 0) by P_4 be granted? (0, 2, 0) by P_0 ?

Deadlock detection

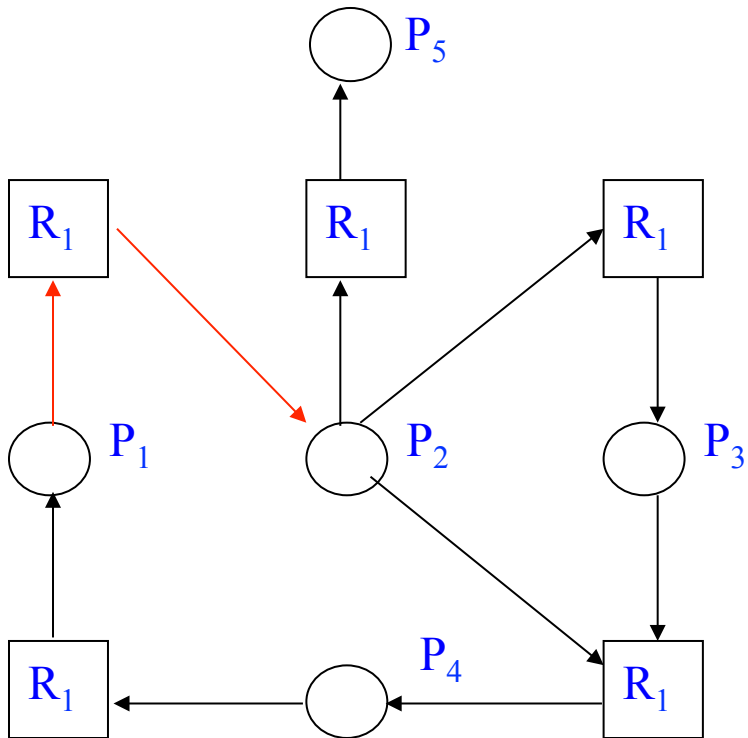
- ★ If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur
- ★ Need a *deadlock detection* algorithm that examines the state of the system to determine whether a deadlock has occurred
- ★ Need a *recovery* algorithm to recover from deadlock

Deadlock detection for single instance of each resource type

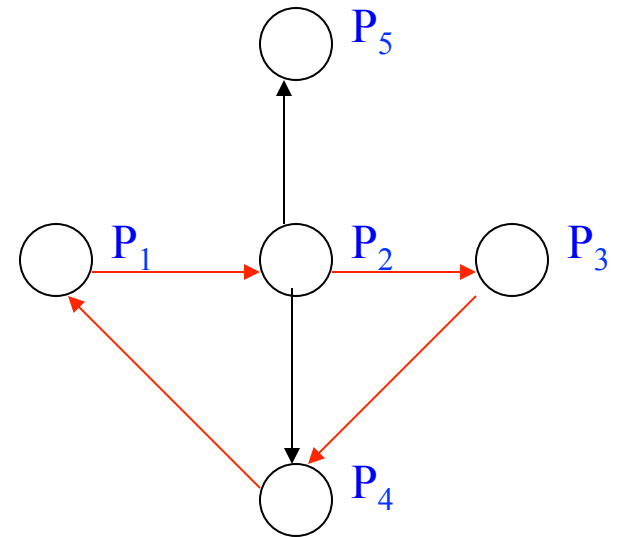
- ★ Maintain a *wait-for graph*
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- ★ Periodically invoke an algorithm that searches for a cycle in the graph
 - An algorithm to detect a cycle in a graph requires $O(n^2)$ operations,
 - ★ n is the number of vertices in the graph

Example

Resource Allocation Graph



Wait for Graph



Deadlock detection (cont.)

Deadlock detection for several instances of a resource type

Data structures:

Available: Vector of length m

Indicates the number of available resources of each type

Allocation: $n \times m$ matrix

Defines the number of resources of each type currently allocated to each process

Request: $n \times m$ matrix

Indicates the current request of each process

$request[i, j] = k$; // means P_i is requesting k more instances of resource R_j

Detection algorithm

// Requires an $O(mn^2)$ operations to detect if the system is in a deadlocked state

1. Let *work* and *finish* be vectors of length m and n , respectively.
 initialise: $work = available$
 for $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $finish[i] = false$; otherwise $finish[i] = true$;
2. Find an index i such that both $finish[i] = false$ and $request_i \leq work$
 if no such i exists, go to step 4
3. $work = work + allocation_i$
 $finish[i] = true$; go to step 2
4. if $finish[i] = false$, for some i , $1 \leq i \leq n$, then the system is in a deadlock state.
 Moreover, $finish[i] = false \rightarrow P_i$ is deadlocked.

Example of detection algorithm

- ★ Five processes: P_0 through P_4
- ★ Resources: A (7 instances), B (2 instances), and C (6 instances)
- ★ Snapshot at time t_0 :

	<i>Allocation</i>	<i>Request</i>	<i>Available</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$
for all $i \rightarrow$ no deadlock

Example (cont.)

What if P_2 requests an additional instance of type C ?

	<i>Request</i>
	<i>A B C</i>
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system:

- ★ Can reclaim resources held by process P_0 ,
but insufficient resources to fulfil requests
from other processes
 - ★ **Deadlock exists, consisting of processes P_1 ,
 P_2 , P_3 , and P_4**

Detection algorithm usage

- ★ When should the algorithm be invoked? It depends on:
 - How often we believe deadlocks occur
 - How many processes will be affected by deadlock when it occurs
- ★ Invoke every time a process makes a request → expensive
 - But it can identify the processes that caused the deadlock
 - ★ Good for debugging
- ★ Invoke at less frequent intervals
 - E.g., when CPU utilisation falls below 40% (perhaps because of deadlock), or every hour.

Deadlock recovery

1) Terminate processes

- ★ Kill (abort) all deadlocked processes
- ★ Kill one process at a time until deadlock cycle eliminated

- ★ In which order should we choose process to abort?
 - The process with lowest priority
 - How long the process has computed, and how much longer to completion
 - Resources the process has used
 - Resources the process needs to complete
 - How many processes will need to be terminated
 - Is process interactive or batch?

Problem: what if the process is in the middle of updating a file?

Aborting the process may lead to incorrect file

Deadlock recovery

2) Pre-empt a resource from a process.

- ★ How to select a victim (process) to minimize cost?
- ★ Roll back the process to some safe state and restart from there
 - How do we find a safe state?
 - ★ Easiest way: destroy the process and restart
 - ★ Use checkpoints during execution
- ★ Starvation – same process may always be picked as victim
 - How do we ensure no starvation?
 - ★ Include number of rollbacks in cost factor

Combined approach to deadlock handling

- ★ Combine the three basic approaches
 - Prevention
 - Avoidance
 - Detection
- ★ Allow the use of the optimal approach for each class of resources in the system
 - Partition resources into hierarchically ordered classes
 - Use most appropriate technique for handling deadlocks within each class.