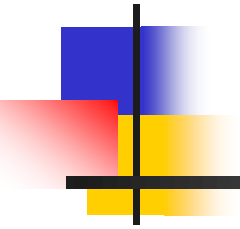
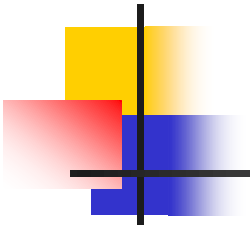
操作系统原理及应用

李　伟
xchlw@seu.edu.cn

计算机科学与工程学院、软件学院
江苏省网络与信息安全重点实验室



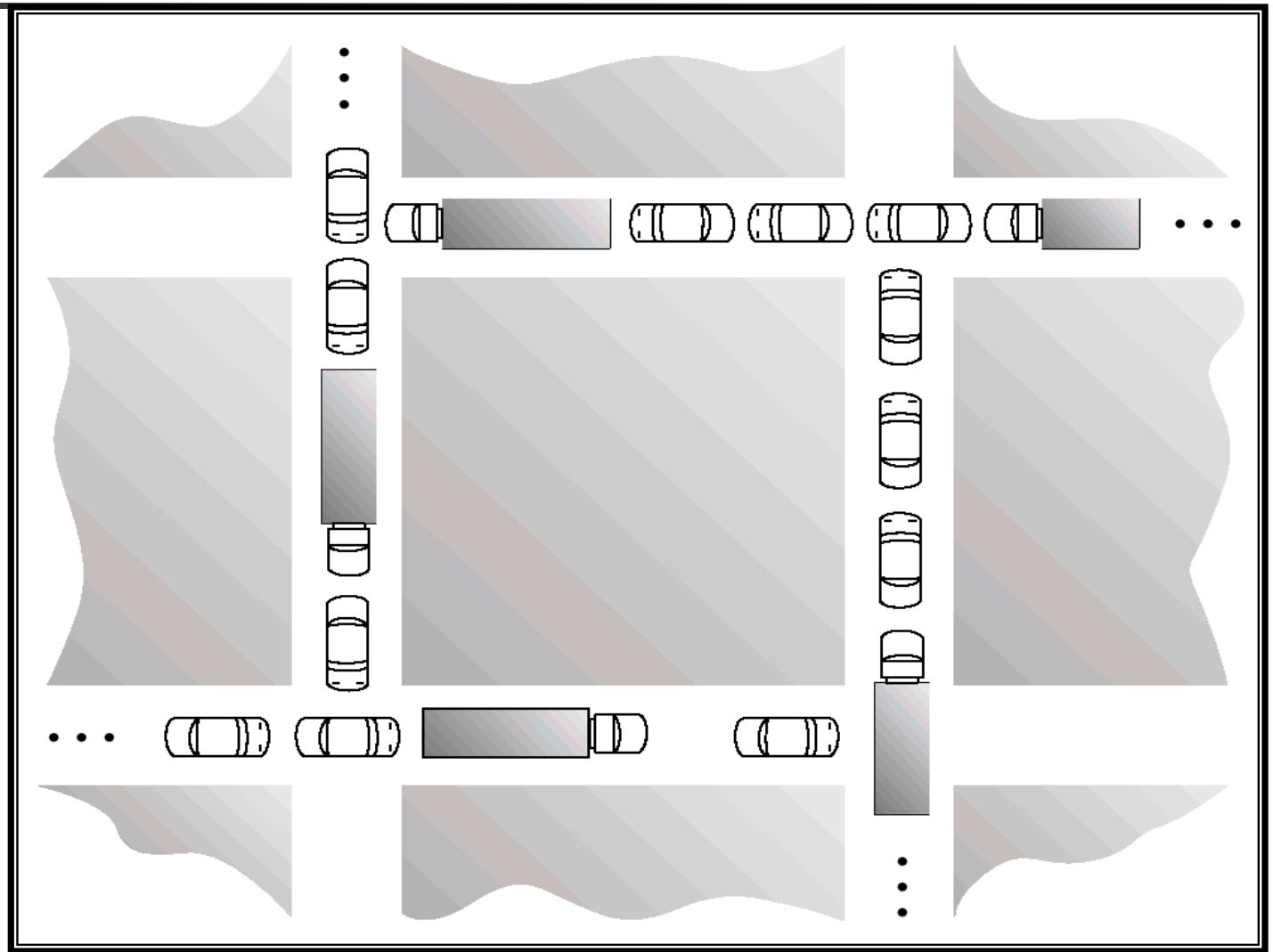
Chapter 7 Deadlocks



Outline

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

The Deadlock Problem



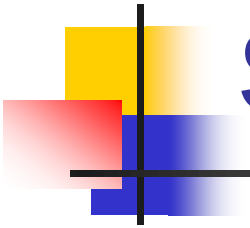
System Model

- **Resource types $R_1, R_2, \ldots, R_m$**

  CPU cycles, memory space, I/O devices, files and so on

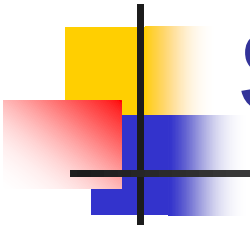
- **Each resource type R_i has W_i instances.**
- **Each process utilizes a resource as follows:**

  request → use → release



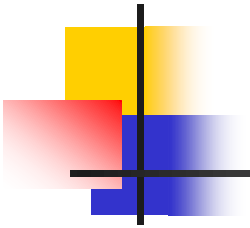
System Model

- **Request**
  - **If a process makes a request to use a system resource which cannot be granted immediately, then the requesting process is blocked until it can acquire the resource.**
- **Use**
  - **The process can operate on the resource.**
- **Release**
  - **The process releases the resource.**



System Model

- ***Deadlock***
  - **A set of process is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.**



Outline

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

Deadlock Characterization

- **For a deadlock to occur, each of the following four conditions must hold.**
  - **Mutual exclusion: only one process at a time can use a resource.**
  - **Hold and wait (request and hold): A process must be holding a resource and waiting for another.**
  - **No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.**
  - **Circular wait: A waits for B, B waits for C, C waits for A.**

Resource-Allocation Graph

A set of vertices V and a set of edges E.

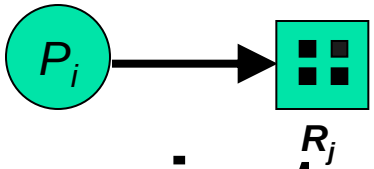
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

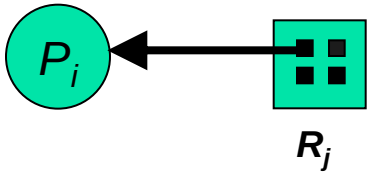
- **Process**

Resource Type with 4 instances

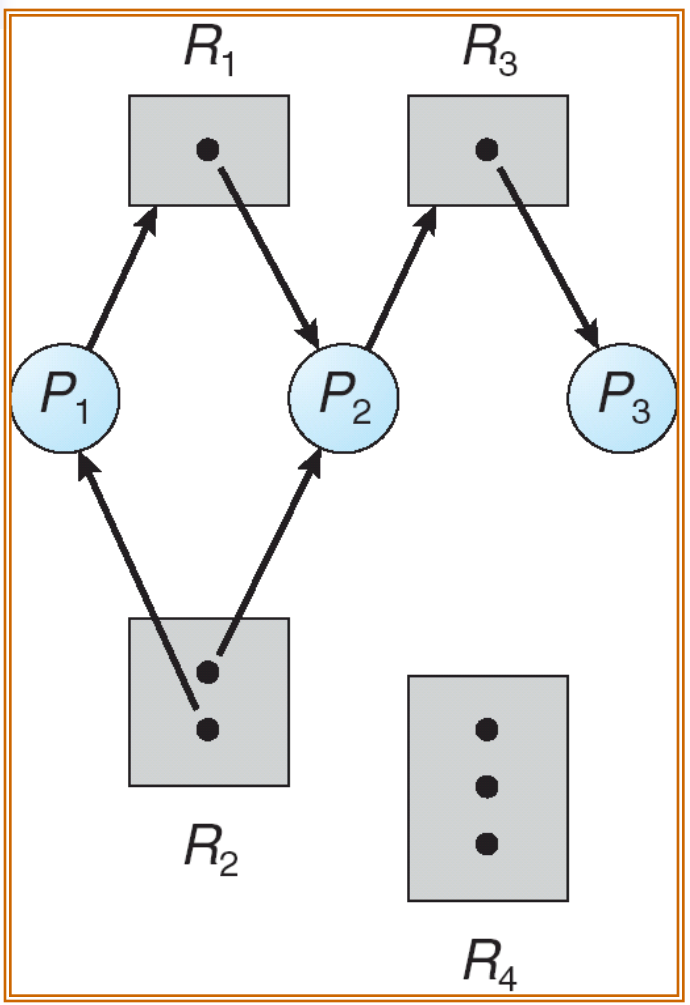
- **P_i requests instance of R_j**

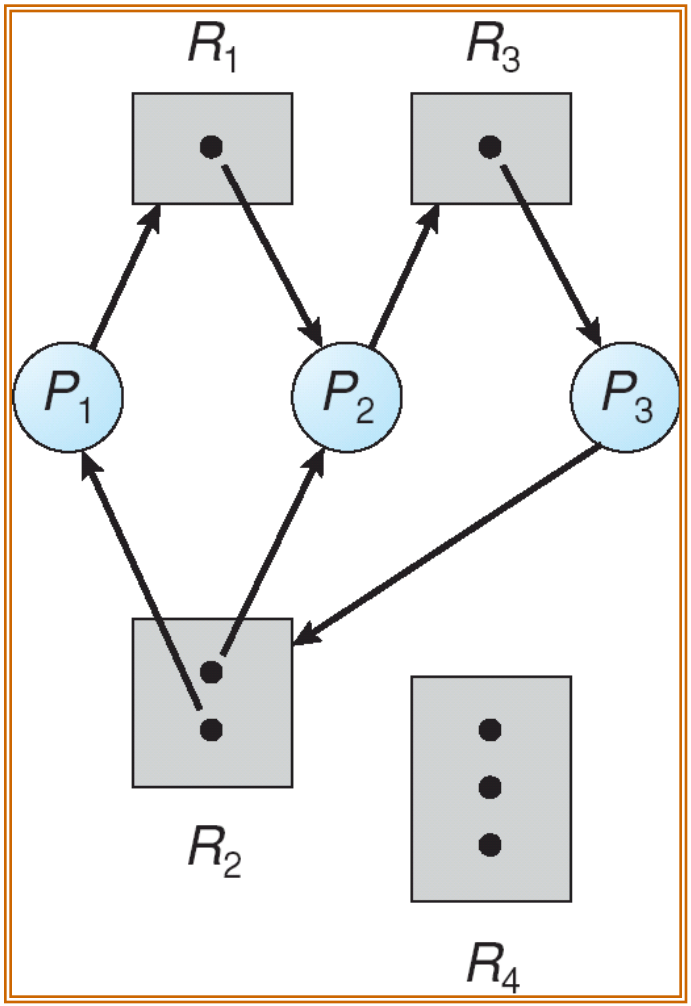


- **P_i is holding an instance of R_j**

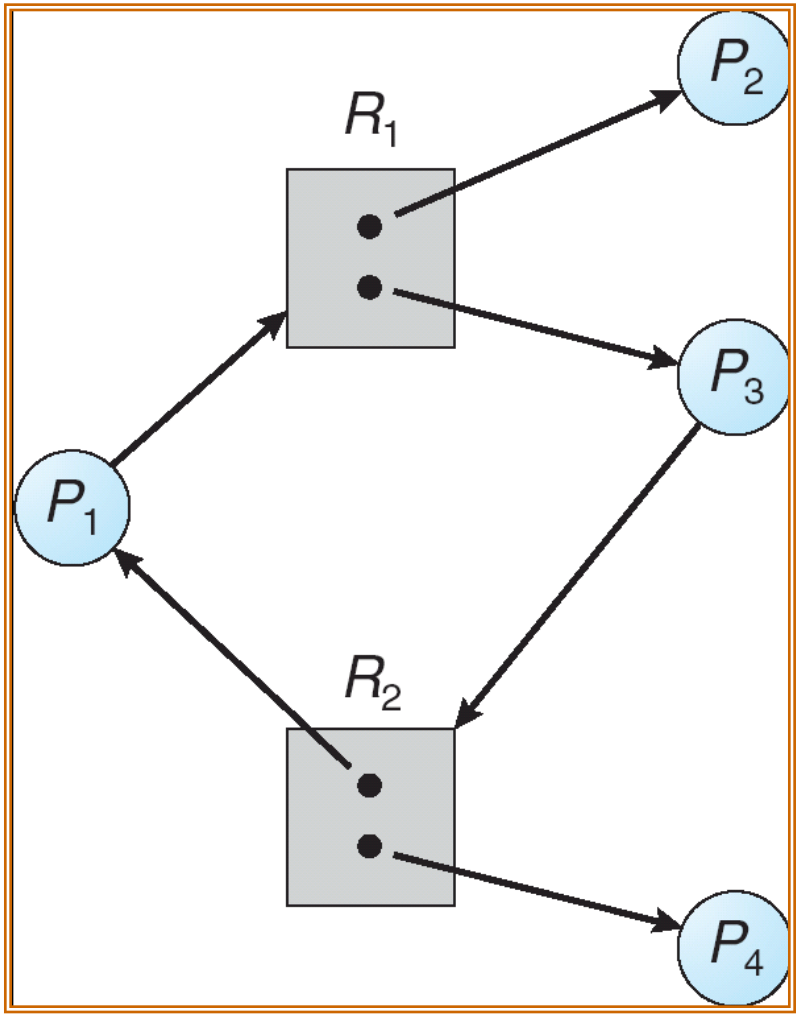


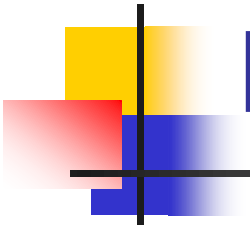
Example of a Resource Allocation Graph





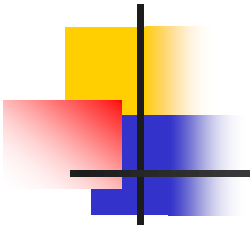
Resource Allocation Graph With a Cycle but No Deadlock





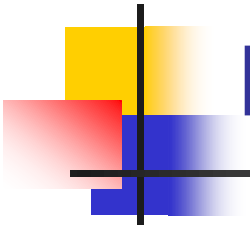
Basic Facts

- 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.



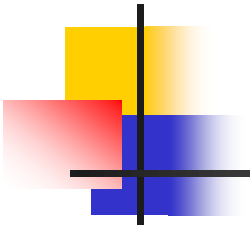
Outline

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



Methods for Handling Deadlocks

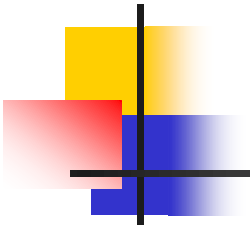
- **Ensure that the system will *never* enter a deadlock state.**
  - **Prevention: Ensure one of the four conditions fails.**
  - **Avoidance: The OS needs more information so that it can determine if the current request can be satisfied or delayed.**
- **Allow the system to enter a deadlock state, detect it, and recover.**
- **Ignore the problem and pretend that deadlocks never occur in the system**



Outline

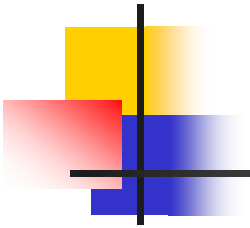
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock Prevention: Mutual Exclusion



- **By ensuring that at least one of the four conditions cannot hold, we can prevent the occurrence of a deadlock.**
- **Mutual Exclusion: Some nonsharable resources must be accessed exclusively (e.g., printer), which means we cannot deny the mutual exclusion condition.**

Deadlock Prevention: Hold and Wait



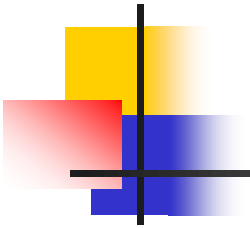
- **No process can hold some resources and then request for other resources.**
- **Two strategies are possible:**
  - **A process must acquire *all* resources before it runs.**
  - **When a process requests for resources, it must hold none (*i.e.*, returning resources before requesting for more).**
- **Resource utilization may be low, since many resources will be held and unused for a long time.**
- **Starvation is possible. A process that needs some popular resources may have to wait indefinitely.**

Deadlock Prevention: No Preemption

- 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 preempted.
- If the requested resources are not available:
  - If they are being held by processes that are waiting for additional resources, these resources are preempted and given to the requesting process.（抢别人）
  - Otherwise, the requesting process waits until the requested resources become available. While it is waiting, its resources may be preempted.（被抢）

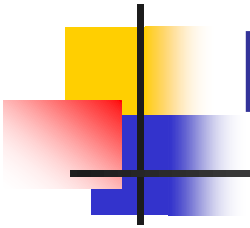
Deadlock Prevention: Circular Wait

- To break the circular waiting condition, we can order all resource types (*e.g.*, tapes, printers).
- A process can only request resources higher than the resource types it holds.
- A process must release some higher order resources to request a lower order resource.
- Suppose the ordering of tapes, disks, and printers are 1, 4, and 8. If a process holds a disk (4), it can only ask a printer (8) and cannot request a tape (1). To get tapes (1), a process must release its disk (4).
- In this way, no deadlock is possible. Why?



Outline

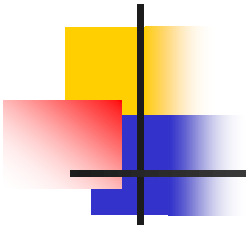
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock



Deadlock Avoidance

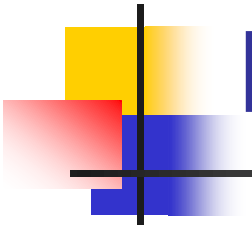
Requires that the system has some additional *a priori* information available.

- **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.**



Safe State

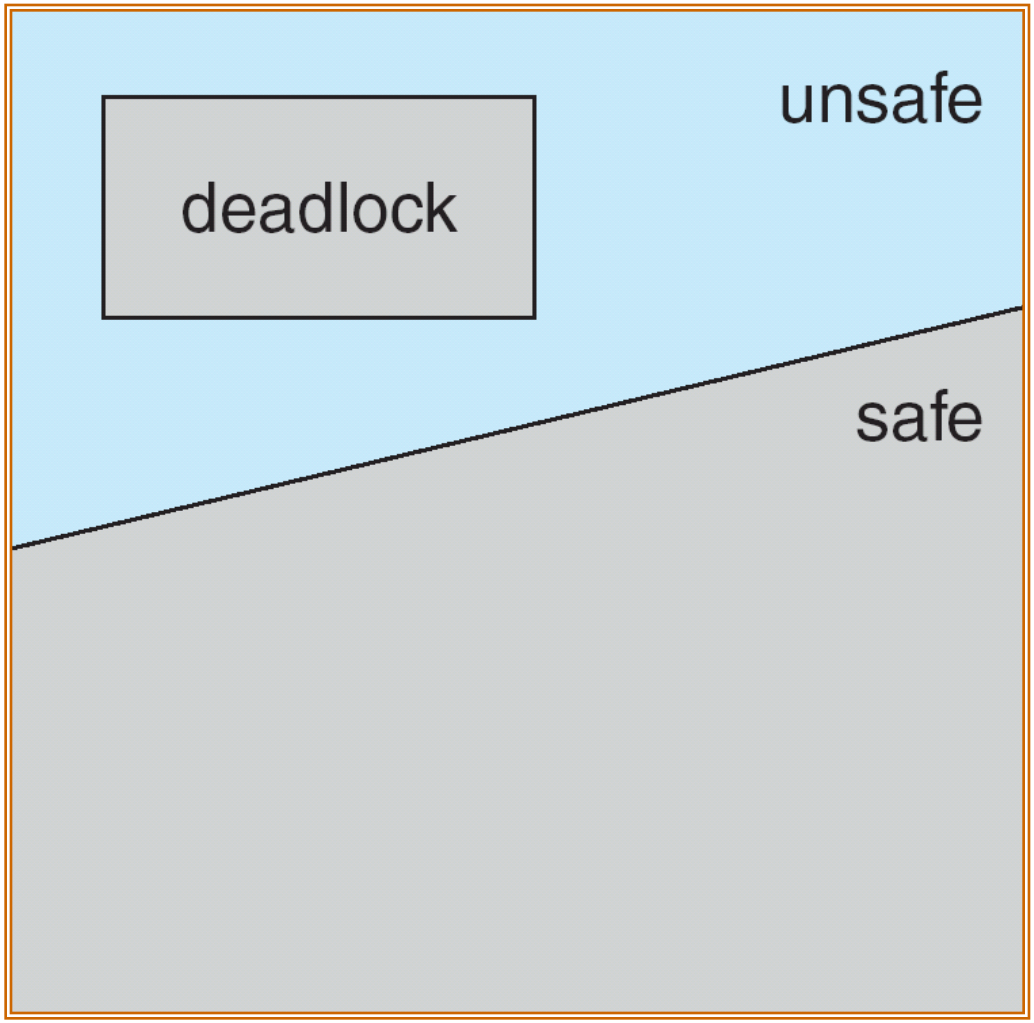
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a ***safe state***.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources *plus* resources held by all the P_j, with $j<i$.

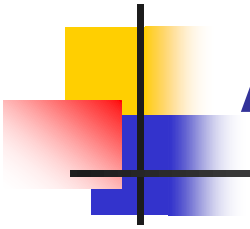


Basic Facts

- 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.

Safe, Unsafe , Deadlock State





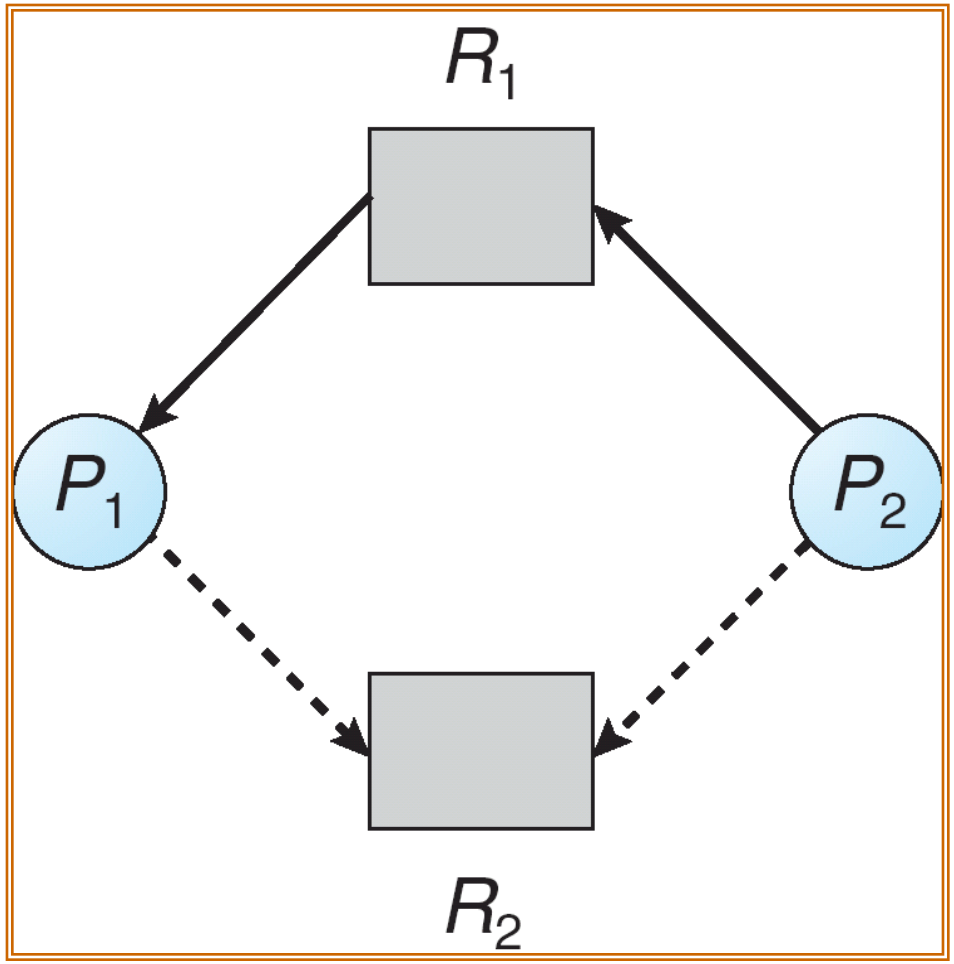
Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation-graph algorithm
- Multiple instances of a resource type
  - Use the banker's algorithm

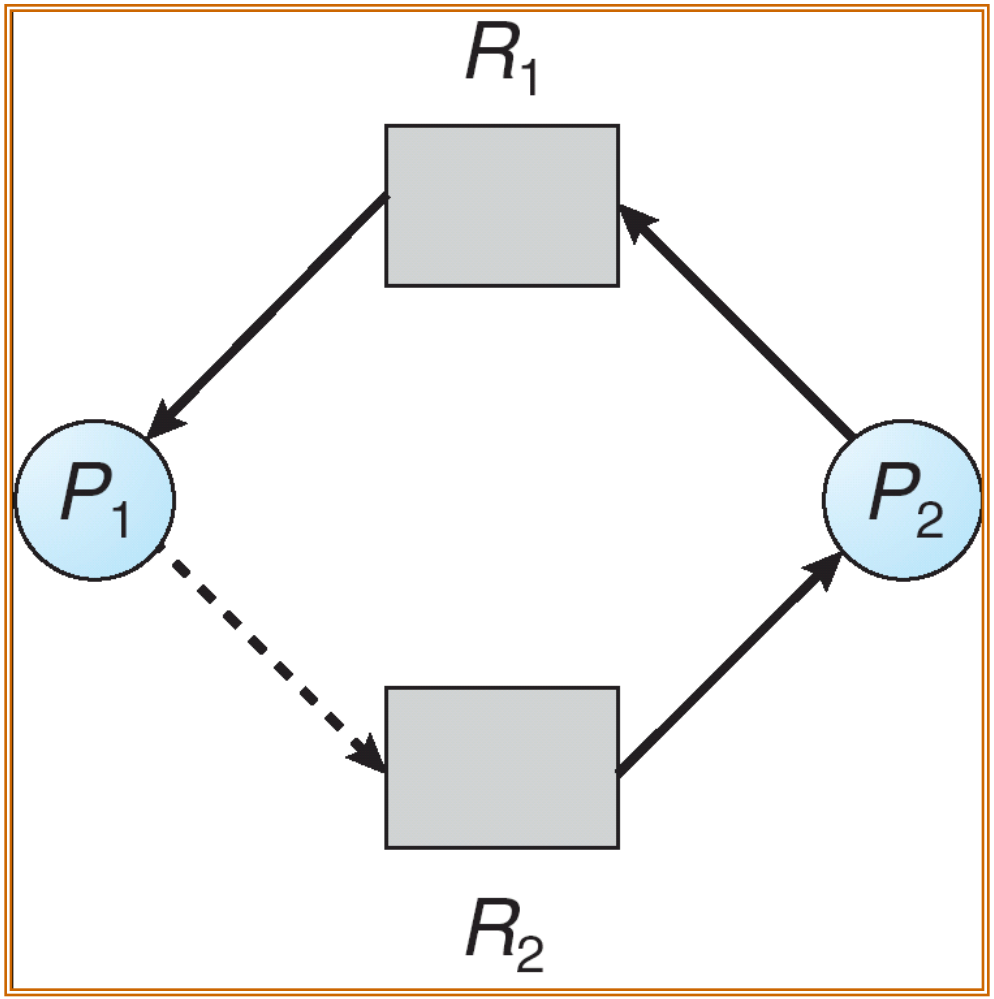
Resource-Allocation-Graph Algorithm

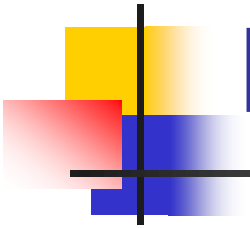
- ***Claim edge*** $P_i \rightarrow R_j$ **indicated that process** P_i **may request resource** R_j**; represented by a dashed line.**
- **Claim edge converts to request edge when a process requests a resource.**
- **When a resource is released by a process, assignment edge reconverts to a claim edge.**
- **Resources must be claimed a priori in the system.**

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph





Banker's 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 Structures for the Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

- ***Available:*** **Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available.**
- ***Max:*** ***n x m*** **matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.**
- ***Allocation:*** ***n*** **x** ***m*** **matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j.**
- ***Need:*** ***n*** **x** ***m*** **matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.**

Resource-Request Algorithm for Process Pi

- Let $Request_i$ be the 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.

Resource-Request Algorithm for Process Pi

3. **Pretend 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 safe*** $\Rightarrow$ ***the resources are allocated to*** P_i.
- ***If unsafe*** $\Rightarrow$ P_i ***must wait, and the old resource-allocation state is restored***

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively. Initialize:

   Work = Available; Finish [i] = false for i = 0, 1, …, n.

2. **Find an *i* such that both:**

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   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*] == true for all *i*, then the system is in a safe state.**

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:

	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 of Banker's Algorithm

- **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 < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.**

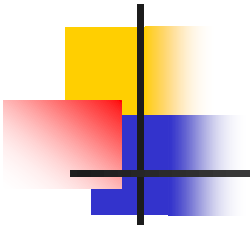
Example P1 Request (1,0,2)

- **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	

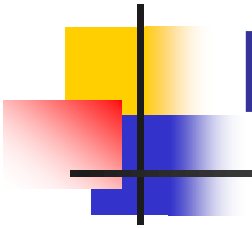
Example P1 Request (1,0,2)

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?



Outline

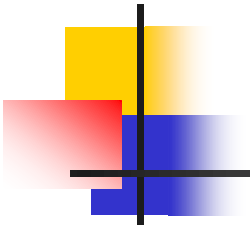
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock



Deadlock Detection

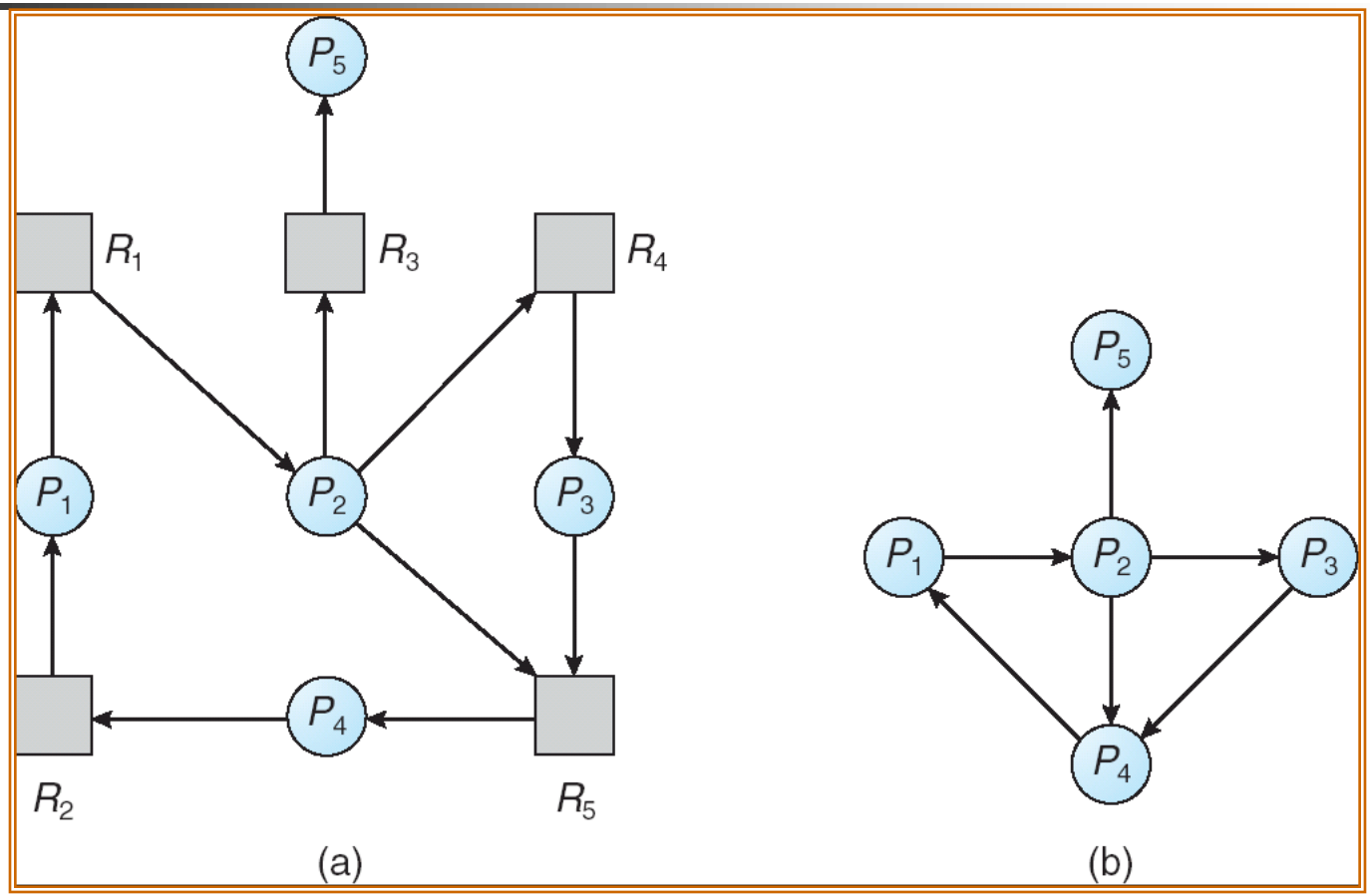
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type



- **Maintain *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 an order of n^2 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**

Several Instances of a Resource 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* [*i,j*] = *k*, then process P_i is requesting *k* more instances of resource type R_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 $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.

Detection Algorithm

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 deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.**

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*.

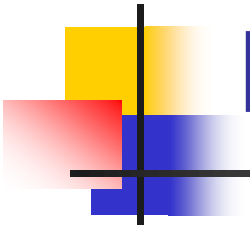
Example of Detection Algorithm

- P_2 requests an additional instance of type *C*.

Request

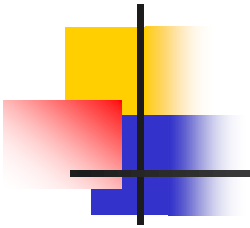
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?



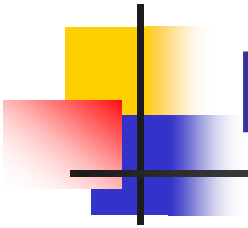
Detection-Algorithm Usage

- When and how often to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- 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.



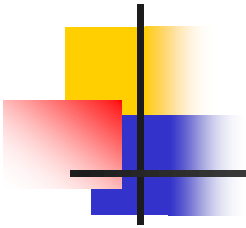
Outline

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



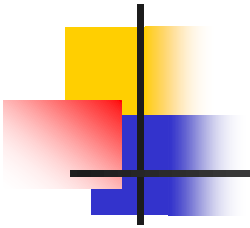
Recovery from Deadlock

- **Process Termination**
- **Resource Preemption**



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?



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.**
- **The most common solution is to include the number of rollback in the cost factor.**

作业

- 考虑下面的一个系统在某一时刻的状态：

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

使用银行家算法回答下面问题：

a. Need矩阵的内容是怎样的？

b. 系统是否处于安全状态？

c. 如果从进程P1发来一个请求（0,4,2,0），这个请求能否立刻被满足？

Part 2 小结（1/2）

- 进程的概念、组成、特征（4个）、状态（转换）、PCB
- 进程调度：调度队列（3种），调度程序（3种），上下文切换
- 进程操作：创建（fork()和exec()、CreateProcess()），终止（exit()和wait()、TerminateProcess()和WaitForSingleObject()）
- 进程间通信（IPC）：消息传递（直接通信、间接通信），共享内存（生产者——消费者），同步和异步，C/S系统通信（Socket、RPC、RMI）
- 线程的概念、组成、分类（用户、内核），程序、进程和线程的异同
- 多线程模型（多对一、一对一、多对多、二级模型）
- 线程库：pthread_create()、pthread_join()、pthread_exit()、pthread_cancel()、pthread_kill()、CreateThread()
- 多线程问题：线程取消（异步取消、延迟取消），信号机制（和中断机制的区别）

Part 2 小结（2/2）

- CPU调度：调度的可行性、发生调度的时机、抢占式调度与非抢占式调度、调度器与分派器、调度准则（5个）
- 调度算法：FCFS、SJF（指数平均）、优先级调度（Aging）、轮转法调度、多级队列调度、多级反馈队列调度
- 多处理器调度（负载均衡、处理器亲和性）、实时调度、线程调度
- 进程同步：竞争条件、临界区问题（4个部分、3个条件）、硬件同步机制（中断管理、特殊指令），软件同步机制（信号量、管程（条件变量））
- Peterson算法和Bakery算法、三大经典同步问题的解决方案
- 死锁的概念（资源分配图）、特征（4个）、处理方法（3种）
- 死锁预防（破坏四个条件之一）、死锁避免（安全状态、资源分配图算法、银行家算法）
- 死锁发现（单资源实例、多资源实例），死锁恢复（进程终止、资源抢占）