

# 0117401: Operating System

## 计算机原理与设计

### Chapter 7: Deadlock

陈香兰

`xlanchen@ustc.edu.cn`

`http://staff.ustc.edu.cn/~xlanchen`

Computer Application Laboratory, CS, USTC @ Hefei

Embedded System Laboratory, CS, USTC @ Suzhou

April 29, 2015

## 温馨提示：



为了您和他人的工作学习，  
请在课堂上**关机或静音**。

**不要**在课堂上接打电话。

# Objectives

- To develop a description of **deadlocks**, which prevent sets of concurrent processes from completing their tasks
- To present a number of different **methods for preventing or avoiding deadlocks** in a computer system.

## 1 Background and System Model

## 2 Deadlock Characterization

- Necessary Conditions
- Resource-Allocation Graph
- Methods for Handling Deadlocks

## 3 Deadlock Prevention (死锁预防)

## 4 Deadlock Avoidance (死锁避免)

- Safe State (安全状态)
- Resource-Allocation Graph Scheme
- Banker's Algorithm (银行家算法)

## 5 Deadlock Detection (死锁检测) and Recovery

## 6 小结和作业

## 1 Background and System Model

# The Deadlock Problem

## deadlock situation

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

### Example 1

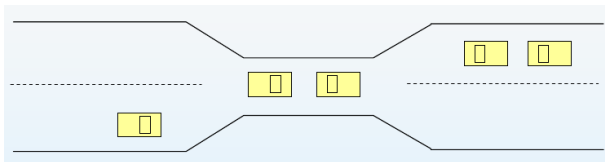
- System has 2 disk drives.
- $P_1$  and  $P_2$  each hold one disk drive and each needs another one.

### Example 2

- 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.

# System Model

- A system consists of a finite number of resources
- The resources are partitioned into several types, each consisting of some number of **identical** instance.
  - physical resources: CPU cycles, memory space, I/O devices
  - logical resources: files, semaphores, and monitors
- System model
  - Resource types  $R_1, R_2, \dots, R_m$
  - Each resource type  $R_i$  has  $W_i$  instances.
  - Each process utilizes a resource as follows:
    - **request**: may wait until it can acquire the resource
    - **use**
    - **release**



- 2 Deadlock Characterization
  - Necessary Conditions
  - Resource-Allocation Graph
  - Methods for Handling Deadlocks

- 2 Deadlock Characterization
  - Necessary Conditions
  - Resource-Allocation Graph
  - Methods for Handling Deadlocks

# Deadlock Characterization: Necessary Conditions

- Deadlock can arise if four conditions hold simultaneously.

- ① **Mutual exclusion**(互斥):

only one process at a time can use a resource.

- ② **Hold and wait**(持有并等待):

a process holding at least one resource is waiting to acquire additional resources held by other processes.

- ③ **No preemption**(不剥夺):

a resource can be released only voluntarily by the process holding it, after that process has completed its task.

- ④ **Circular wait**(循环等待):

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  $P_0$ .

## 2 Deadlock Characterization

- Necessary Conditions
- Resource-Allocation Graph
- Methods for Handling Deadlocks

# Deadlock Characterization: Resource-Allocation Graph

- System resource-allocation graph: A directed graph
  - A set of vertices  $V$  and a set of edges  $E$ .
  - $V$  is partitioned into two types:
    - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system.
    - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system.
    - request edge(请求边) — directed edge  $P_i \rightarrow R_j$
    - assignment edge(分配边) — directed edge  $R_j \rightarrow P_i$

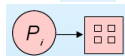


- Process

- Resource Type with 4 instances



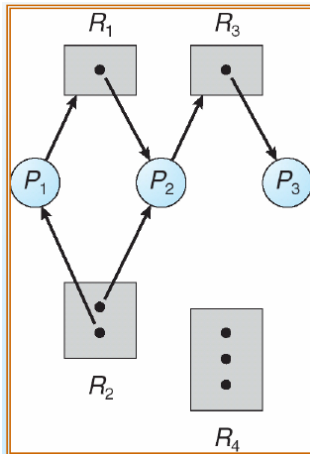
- $P_i$  requests instance of  $R_j$



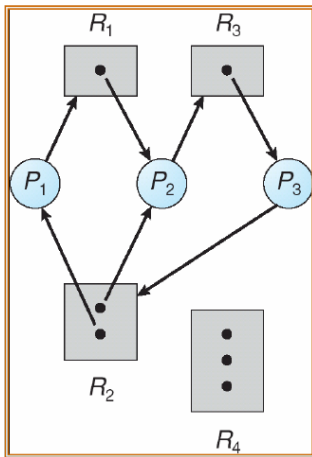
- $P_i$  is holding an instance of  $R_j$



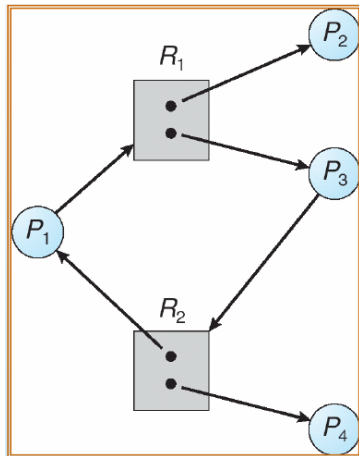
# Example of a Resource Allocation Graph



# Example of a resource Allocation Graph With A Deadlock



# Graph With A Cycle But No Deadlock



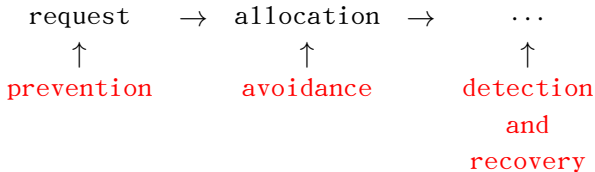


- If graph contains no cycles  $\Rightarrow$  no deadlock.
- If graph contains a cycle  $\Rightarrow$ 
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

- 2 Deadlock Characterization
  - Necessary Conditions
  - Resource-Allocation Graph
  - Methods for Handling Deadlocks

# Methods for Handling Deadlocks

- ① Ensure that the system will **never** enter a deadlock state.
  - ① Deadlock **prevention**
  - ② Deadlock **avoidance**
- ② **Allow** the system to enter a deadlock state and then recover.
  - ① Deadlock **detection** and **recovery** from deadlock
- ③ **Ignore** the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.



## 3 Deadlock Prevention (死锁预防)

# Deadlock Prevention (死锁预防)

- Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.
- Restrain the ways **request** can be made.

## 1 Mutual Exclusion

- not required for **sharable** resources (read-only files); must hold for **nonsharable** resources. (printer)
- In general, therefore, we **cannot deny the mutual-exclusion condition**

# Deadlock Prevention (死锁预防)

- Restrain the ways **request** can be made.

## ② **Hold and Wait**

- must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - ① Require process to **request and be allocated all** its resources before it begins execution, or
  - ② allow process to request resources only when the process **has none**.
- **Disadvantage:**
  - ① Low resource utilization;
  - ② starvation possible.

# Deadlock Prevention (死锁预防)

- Restrain the ways **request** can be made.

## ⑧ No Preemption

- ① 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**.
  - 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.
- ② **preempt the desired resources** from the waiting process and allocate them to the requesting process
  - if the resource are neither available nor held by a waiting process, the requesting process must wait. While waiting, some of its resources may be preempted by other requesting process
  - a process can be **restarted** only when it is **allocated** the new resources it is requesting and **recovers** any resources that were preempted.

# Deadlock Prevention (死锁预防)

- Restrain the ways **request** can be made.

## ④ **Circular Wait**

- impose a total **ordering** of all resource types, and require that each process requests resources in an increasing order of enumeration.
  - ① always in an increasing order
  - ② may release some higher ordered resource before requesting lower ordered resource



## 4 Deadlock Avoidance (死锁避免)

- Safe State (安全状态)
- Resource-Allocation Graph Scheme
- Banker' s Algorithm (银行家算法)

# Deadlock Avoidance (死锁避免)

- 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.

## 4 Deadlock Avoidance (死锁避免)

- Safe State (安全状态)
- Resource-Allocation Graph Scheme
- Banker' s Algorithm (银行家算法)

# Safe State (安全状态)

- 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 (安全序列)**

$$\langle P_1, P_2, \dots, P_n \rangle$$

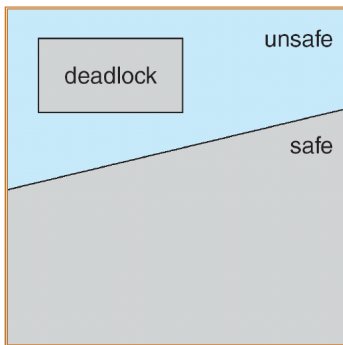
of ALL the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$ .

- That is:

- If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished.
- When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate.
- When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on.

# Basic Facts: Safe, Unsafe , Deadlock State I

- If a system is in **safe state**  $\Rightarrow$  **no deadlocks**.
- If a system is in **unsafe state**  $\Rightarrow$  **possibility** of deadlock.
- **Avoidance**  $\Rightarrow$  ensure that a system will never enter an unsafe state.



- Example, 12 tape drives and 3 processes, at  $T_0$

	MaxNeeds	current
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2 $\rightarrow$ 3

- $\langle P_1, P_0, P_2 \rangle$
- if at  $T_2$ ,  $P_2$  request and is allocated one more tape drive, ?

# Avoidance algorithms

- ❶ Single instance of a resource type.
  - Use a **resource-allocation graph**
- ❷ Multiple instances of a resource type.
  - Use the **banker's algorithm** (银行家算法)

- 4 Deadlock Avoidance (死锁避免)
  - Safe State (安全状态)
  - Resource-Allocation Graph Scheme
  - Banker's Algorithm (银行家算法)



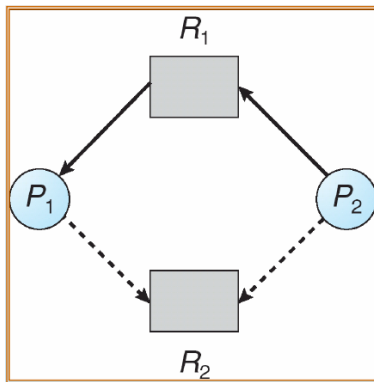
# 1. Resource-Allocation Graph Scheme

- Resource-Allocation Graph

- Claim edge (需求边)  $P_i \rightarrow R_j$ 
  - indicated that process  $P_j$  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 process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

# 1. Resource-Allocation Graph Scheme

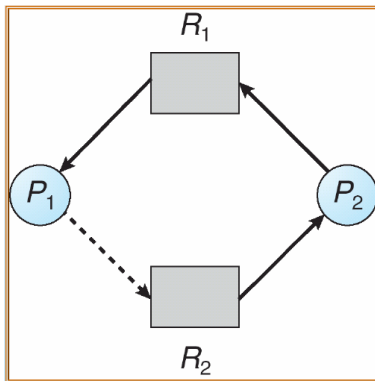
- Example: Safe State



safe sequence:  $\langle P_1, P_2 \rangle$

# 1. Resource-Allocation Graph Scheme

- Example: Unsafe State In Resource-Allocation Graph



# 1. Resource-Allocation Graph Scheme

- Resource-Allocation Graph **Algorithm**

- Suppose that process  $P_i$  requests a resource  $R_j$
- 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

- 4 Deadlock Avoidance (死锁避免)
  - Safe State (安全状态)
  - Resource-Allocation Graph Scheme
  - Banker's Algorithm (银行家算法)

## 2. Banker's Algorithm (银行家算法)

- Banker's Algorithm (银行家算法)

- 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.

- 1 Data structures
- 2 safety algorithm
- 3 resource-request algorithm

## 2. Banker's Algorithm (银行家算法): Data Structures

Let

$n$  = number of processes

$m$  = number of resource types.

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

$$Need[i, j] = Max[i, j] - Allocation[i, j].$$

## 2. Banker's Algorithm (银行家算法): Safety Algorithm

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

$Work = Available$

$Finish[i] = false$  for  $i = 0, 1, \dots, n-1$ .

- ② Find an  $i$  such that both:

①  $Finish[i] = false$

②  $Need_i \leq Work$

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**.



## 2. Banker's Algorithm: Resource-Request Algorithm for Process $P_i$

Request = request vector for process  $P_i$ .

If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$ .

- ① If  $\text{Request}_i \leq \text{Need}_i$  go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
- ② If  $\text{Request}_i \leq \text{Available}$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available.
- ③ Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- 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

## 2. Banker's Algorithm: Example

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

						Need = Max - Allocation			
Allocation			Max			Available			
A B C			A B C			A B C			
P <sub>0</sub>	0	1	0	7	5	3	3	3	2
P <sub>1</sub>	2	0	0	3	2	2			
P <sub>2</sub>	3	0	2	9	0	2			
P <sub>3</sub>	2	1	1	2	2	2			
P <sub>4</sub>	0	0	2	4	3	3			

Need			
A B C			
P <sub>0</sub>	7	4	3
P <sub>1</sub>	1	2	2
P <sub>2</sub>	6	0	0
P <sub>3</sub>	0	1	1
P <sub>4</sub>	4	3	1

- The system is in a safe state since the sequence

$\langle P_1, P_3, P_4, P_2, P_0 \rangle$

satisfies safety criteria.

## 2. Banker's Algorithm: Example: $P_1$ Request (1,0,2)

- ① Check that  $\text{Request}(1,0,2) \leq \text{Available}(3,3,2) \Rightarrow \text{true}$ .

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	4	3	3	3	2

- ②  $P_1$  2 0 0  $\rightarrow$  3 0 2 1 2 2  $\rightarrow$  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

- ③ Executing safety algorithm shows that sequence

$$\langle P_1, P_3, P_4, P_0, P_2 \rangle$$

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?

## 5 Deadlock Detection (死锁检测) and Recovery

# Deadlock Detection (死锁检测) and Recovery

- Allow system to enter deadlock state

- Detection algorithm

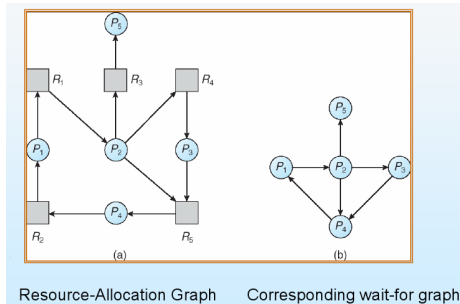
- ① single instance
- ② several instances

- Recovery scheme

- ① Process termination
- ② Resource preemption

# 1. Single Instance of Each Resource Type

- Maintain **wait-for graph**:
  - Nodes are processes.
  - $P_i \rightarrow P_j$ , if  $P_i$  is waiting for  $P_j$ .



Resource-Allocation Graph

Corresponding wait-for graph

- **Periodically** invoke an algorithm that **searches for a cycle** in the graph. **If there is a cycle, there exists a deadlock.**
- **COST**: 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.

## 2. Several Instances of a Resource Type

### 1 Data structures:

- **Available:**

A vector of length  $m$  indicates the number of available resources of each type.

- **Allocation:**

An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process.

- **Request:**

An  $n \times m$  matrix indicates the current request of each process. If  $\text{Request}[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .

## 2. Several Instances of a Resource Type

### ② Detection Algorithm

- ① Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:
    - **Work = Available**
    - For  $i = 1, 2, \dots, n$ , **if  $\text{Allocation}_i \neq 0$ , then  $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$ .**
  - ② Find an  $i$  such that both:
    - **$\text{Finish}[i] == \text{false}$**
    - **$\text{Request}_i \leq \text{Work}$**

**If no such  $i$  exists, go to step 4.**
  - ③ **Work = Work + Allocation <sub>$i$</sub> ,  $\text{Finish}[i] = \text{true}$ , go to step 2.**
  - ④ **If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked.**
- Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state.**



## 2. Several Instances of a Resource Type

### ⑧ Example of Detection Algorithm

- Five processes:  $P_0 \sim P_4$ ;
- 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 2	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  $\text{Finish}[i] = \text{true}$  for all  $i$ .

## 2. Several Instances of a Resource Type

### ⑧ Example of Detection Algorithm

- If  $P_2$  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	0
$P_3$	1	0	0
$P_4$	0	0	2

→ 0 0 1

- 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  $P_4$ .

## 2. Several Instances of a Resource Type

### ● 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 from Deadlock: 1. Process Termination

- ① Abort all deadlocked processes.
  - ② Abort one process at a time until the deadlock cycle is eliminated.
- To minimize cost: 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 from Deadlock: 2. Resource Preemption

- Three issues need to be addressed:
  - ① **Selecting a victim** — minimize cost.
  - ② **Rollback** — return to some safe state, restart process for that state.
  - ③ **Starvation** — same process may always be picked as victim, include number of rollback in cost factor.

## 6 小结和作业

# 小结

## 1 Background and System Model

## 2 Deadlock Characterization

- Necessary Conditions
- Resource-Allocation Graph
- Methods for Handling Deadlocks

## 3 Deadlock Prevention (死锁预防)

## 4 Deadlock Avoidance (死锁避免)

- Safe State (安全状态)
- Resource-Allocation Graph Scheme
- Banker's Algorithm (银行家算法)

## 5 Deadlock Detection (死锁检测) and Recovery

## 6 小结和作业

- 参见课程主页



谢谢！