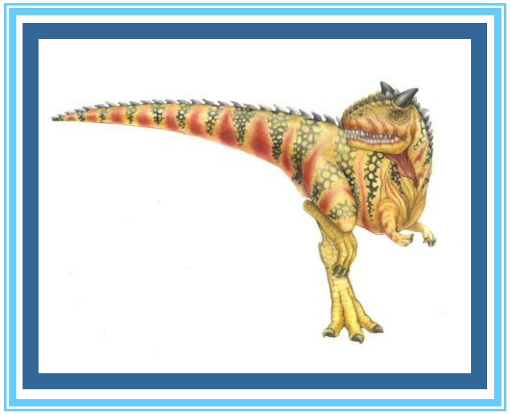
Chapter 7: Deadlocks



Chapter 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

Chapter 7: Deadlocks

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

SYSTEM MODEL

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles,

  memory space,

  I/O devices

System Model (cont.)

- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:
  - **request 申请**
  - **use 使用**
  - **Release 释放**

DEADLOCK CHARACTERIZATION

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion 互斥:** only one process at a time can use a resource
- **Hold and wait 占有并等待:** a process holding at least one resource is waiting to acquire additional resources held by other processes

Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

- **No preemption 非抢占:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

- **Circular wait 循环等待:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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

  $P_2, \ldots, P_{n-1}$ 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.

Deadlock with Mutex Locks

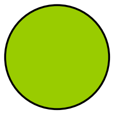
- Deadlocks can occur via system calls, locking, etc
- See example box in text page 318 for mutex deadlock

Resource-Allocation Graph 资源分配图

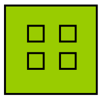
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** 申请边
  - directed edge $P_i \rightarrow R_j$
- **assignment edge**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

Resource-Allocation Graph (Cont.)

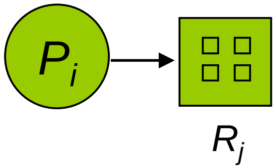
- Process



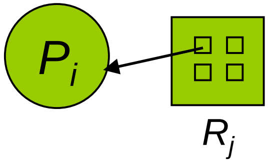
- Resource Type with 4 instances



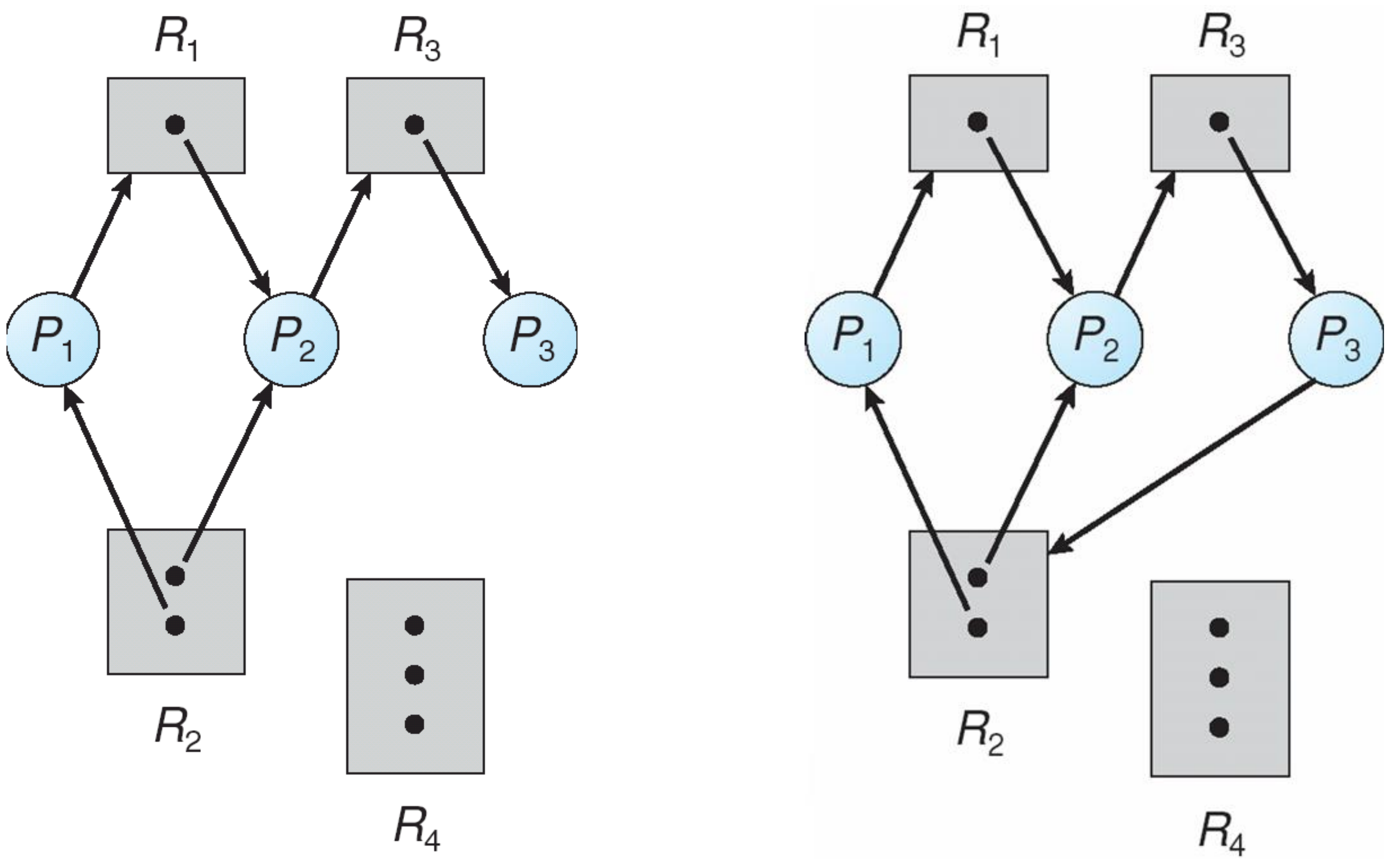
- P_i requests instance of R_j



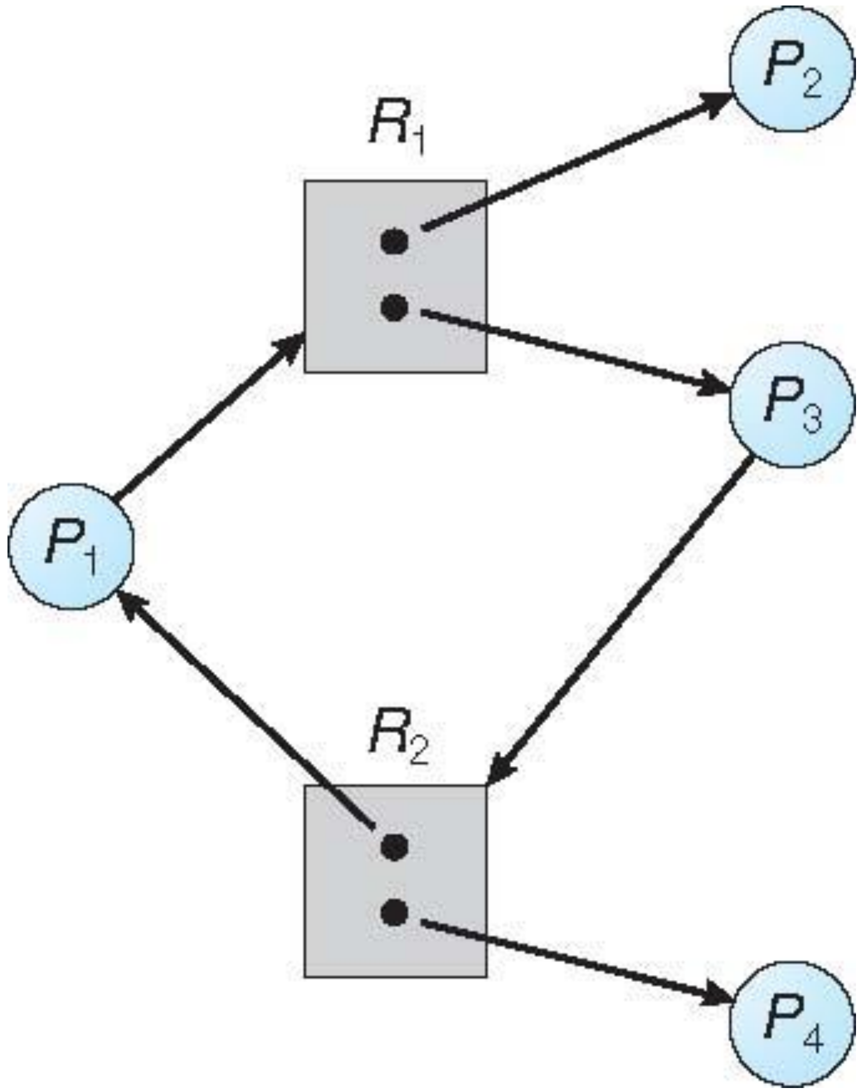
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



Graph With A Cycle But No Deadlock



Basic Facts

- 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

HANDLING DEADLOCKS

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

DEADLOCK PREVENTION

Deadlock Prevention

Restrain 限制 the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

Deadlock Prevention (cont.)

Restrain the ways request can be made

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization 低资源使用率; starvation possible

Deadlock Prevention (Cont.)

- **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 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

Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

DEADLOCK AVIODANCE

Deadlock Avoidance

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

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need

Silberschatz, Galvin and Gagne ©2013

Deadlock Avoidance (cont.)

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

- 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

Silberschatz, Galvin and Gagne ©2013

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 sequence $<P_1, P_2, \ldots, P_n>$ 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 $j < i$

Safe State (cont.)

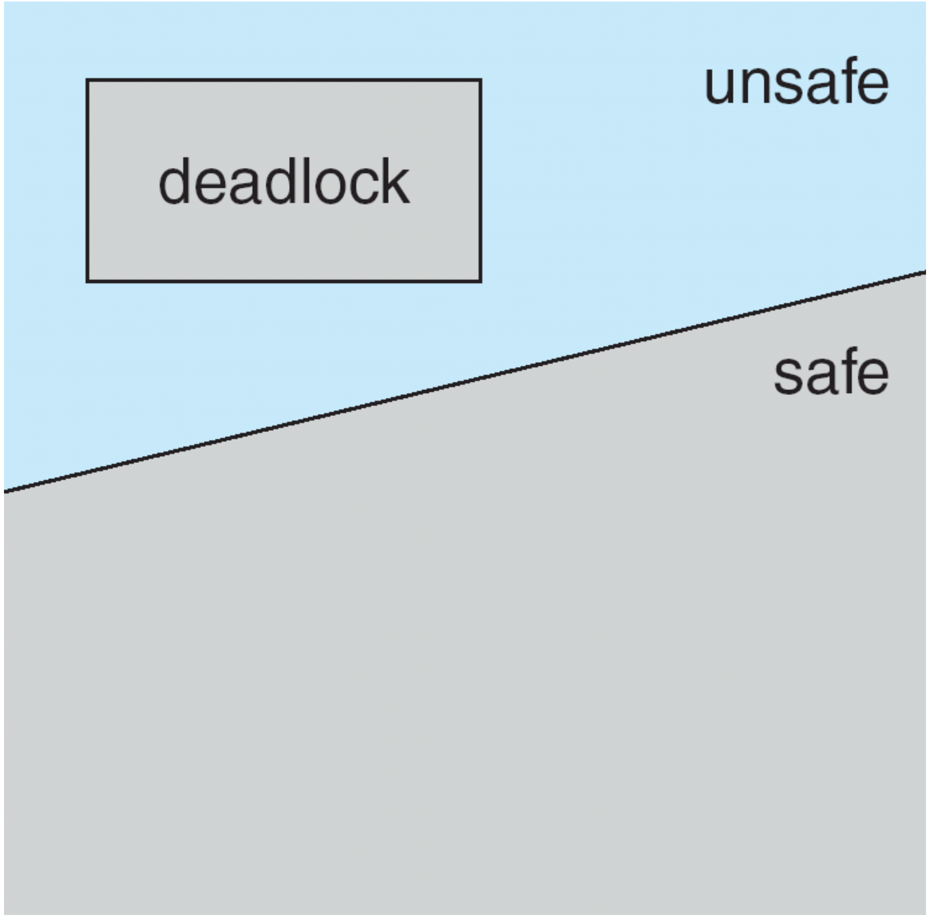
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Silberschatz, Galvin and Gagne ©2013

Basic Facts

- 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.

Safe, Unsafe, Deadlock State



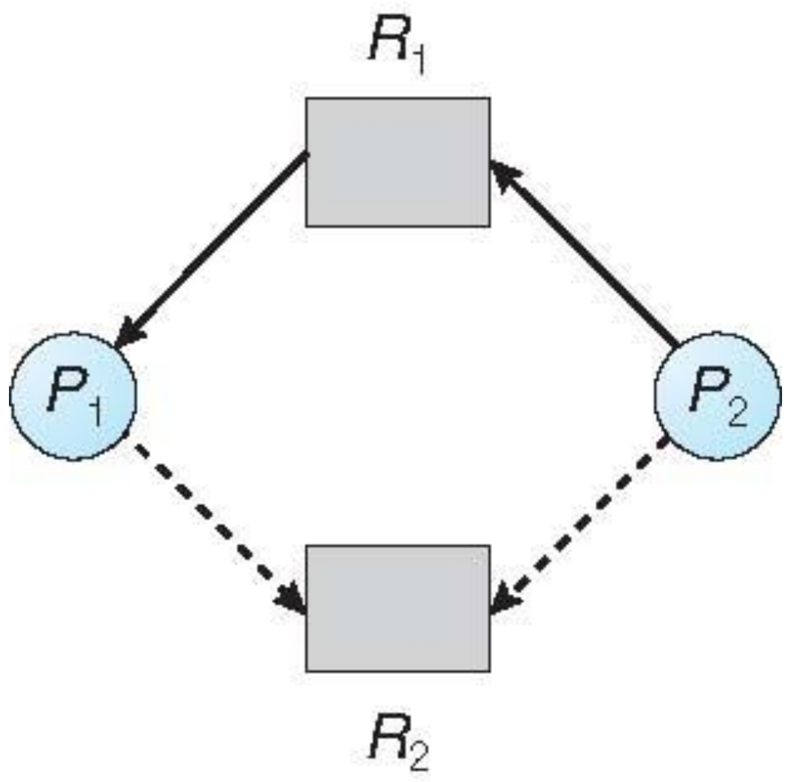
Avoidance algorithms

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

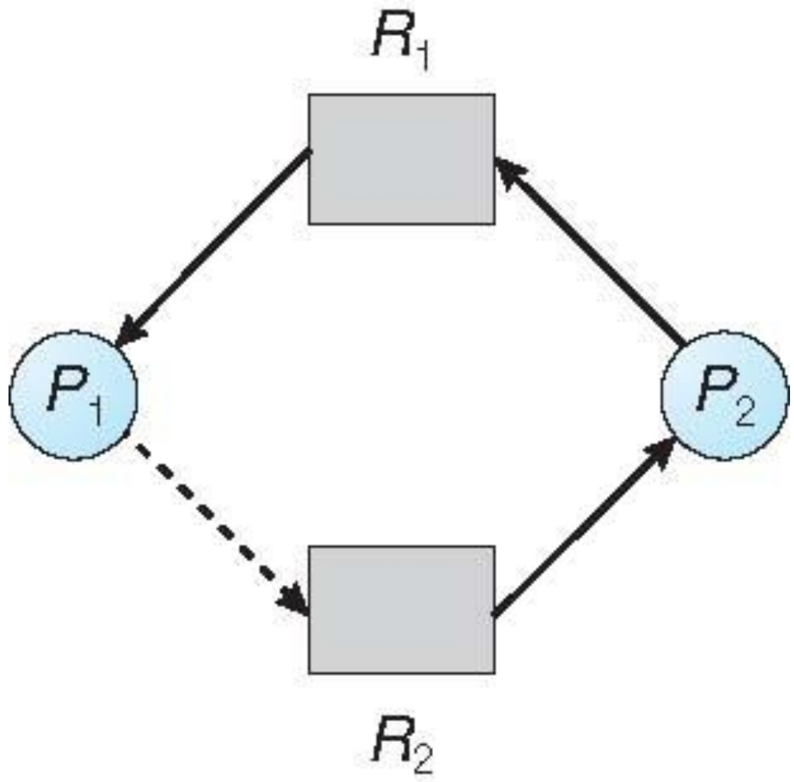
Resource-Allocation Graph Scheme

- **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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Scheme

- 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

- 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

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

Data Structures for the Banker's Algorithm

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

- **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

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

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*- 1**

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

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

Safety Algorithm (cont.)

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

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

Resource-Request Algorithm for Process P_i

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

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

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

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

- 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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances),

  B (5 instances),

  C (7 instances)

Example (Cont.)

- 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 (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

Example (Cont.)

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ 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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2) (cont.)

- 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?

DEADLOCK DETECTION

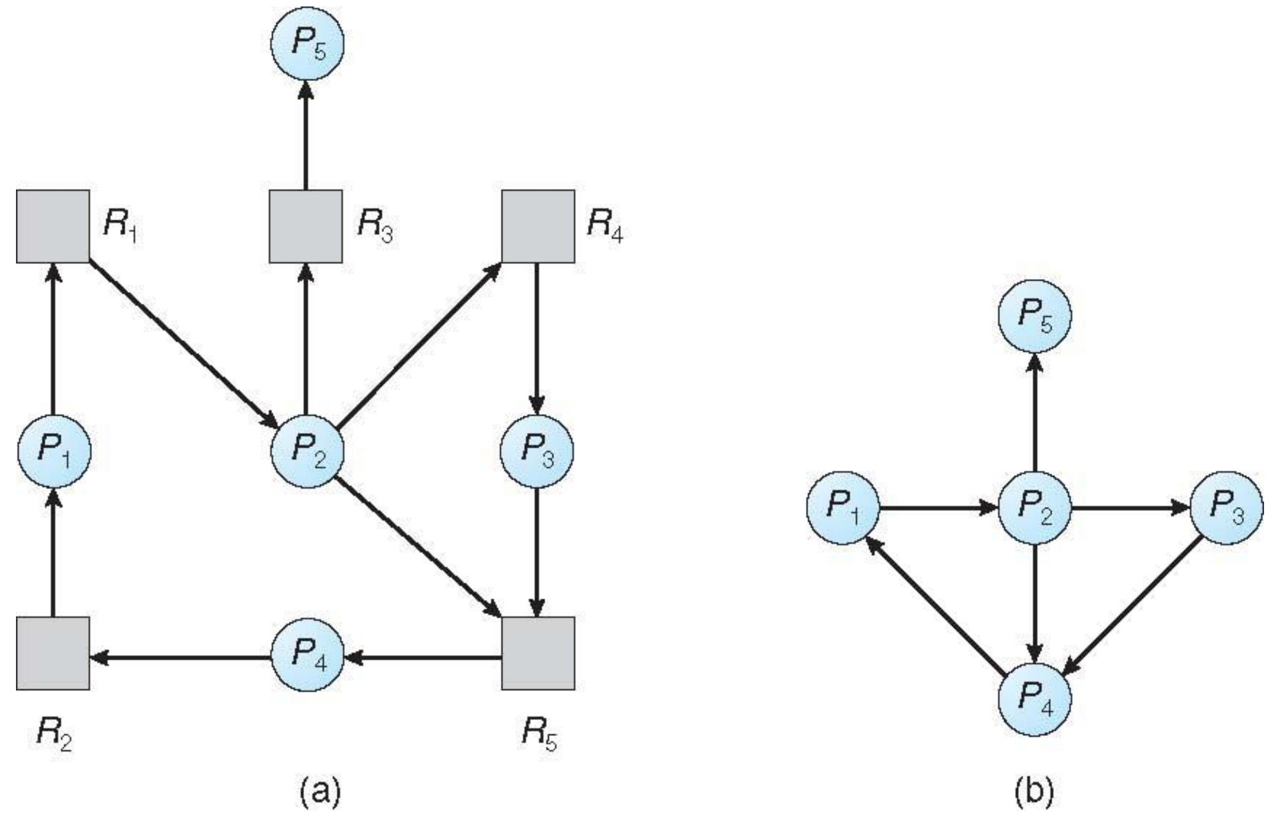
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme 恢复方法

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. If there is a cycle, there exists a deadlock
- 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 $\boldsymbol{Allocation_i \neq 0}$, then
   Finish*[i] = *false;
   otherwise, ***Finish*[i] = *true***

Detection Algorithm (cont.)

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

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

   (b) $\boldsymbol{Request_i \leq Work}$

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

Silberschatz, Galvin and Gagne ©2013

Detection Algorithm (cont.)

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then $\boldsymbol{P_i}$ is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes ***P*$_0$** through ***P*$_4$**;
- Three resource types

  A (7 instances)

  B (2 instances)

  C (6 instances)

Example (cont.)

- 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 (cont.)

- 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

Example (cont.)

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

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

Detection-Algorithm Usage (cont.)

- 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

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes

- Abort one process at a time until the deadlock cycle is eliminated

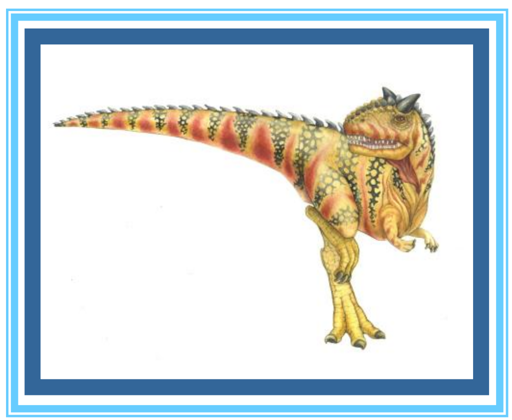
Recovery from Deadlock: Process Termination

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from 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

End of Chapter 7



Silberschatz, Galvin and Gagne ©2013