

Operating Systems

Chapter 6 Concurrency: Deadlock(死锁) and Starvation(饥饿)

6.1 Principles of Deadlock

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

6.1 Principles of Deadlock(1/9)

- Why: compete for finite resources

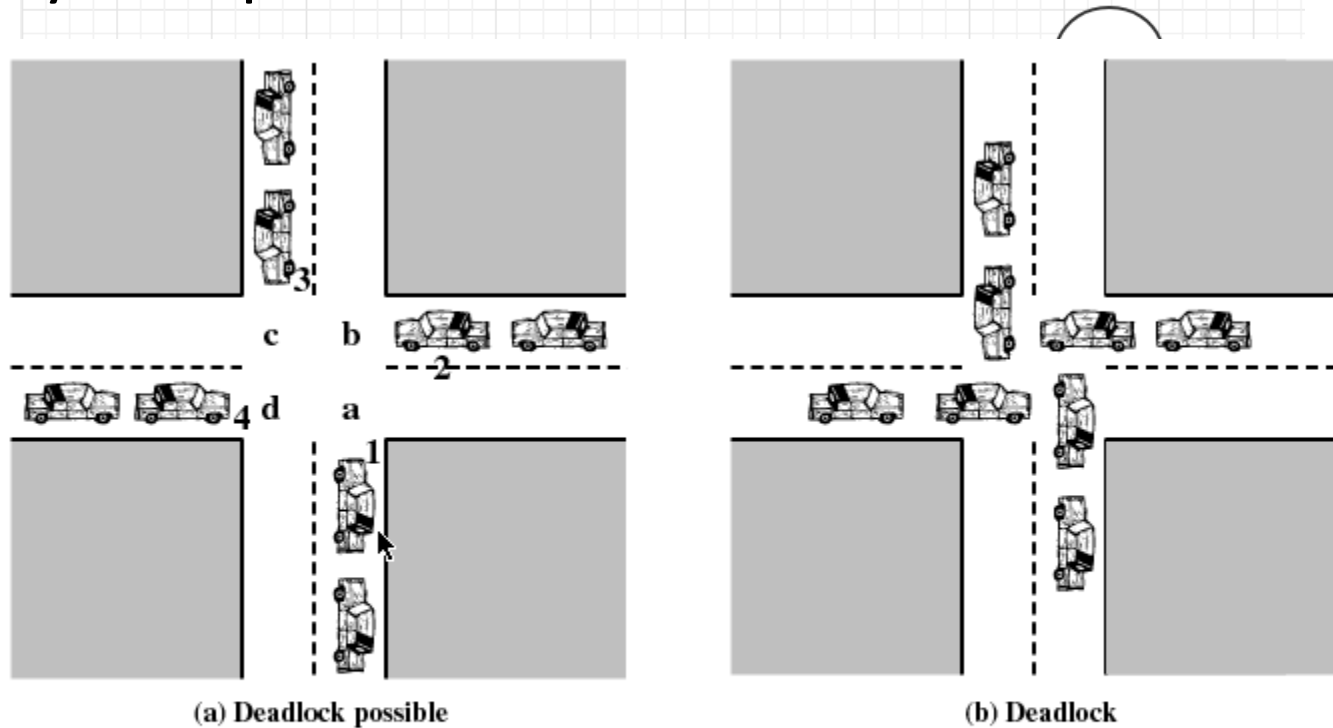


Figure 6.1 Illustration of Deadlock

6.1 Principles of Deadlock(2/9)

- Reusable Resources 可重用
 - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases
 - Used by only one process at a time
 - Obtainable after release
- Consumable Resources 消费型
 - interrupts, signals, messages, and information in I/O buffers
 - created (produced) and destroyed (consumed) by processes

6.1 Principles of Deadlock(3/9)

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- All deadlock involve conflicting needs for resources by two or more processes (死锁源于两个或者多个进程的资源需求冲突).
 - Reusable Resources(可重用资源)
 - Consumable Resources(可消费资源)
- No efficient solution in the general case

6.1 Principles of Deadlock(4/9)

Uncertainty

Process P

••••

Get A

••••

Get B

••••

Release A

••••

Release B

••••

Process Q

••••

Get B

••••

Get A

••••

Release B

••••

Release A

••••

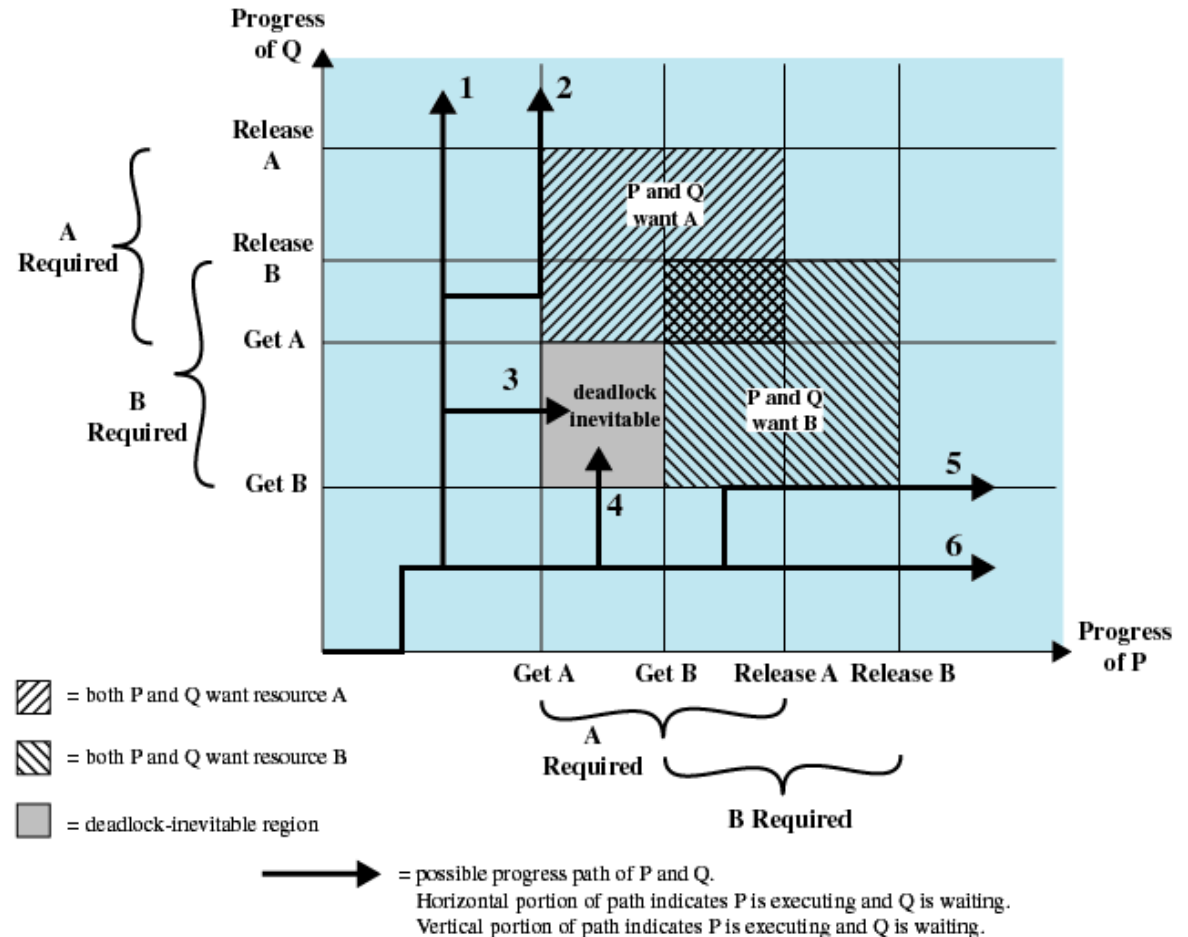


Figure 6.2 Example of Deadlock

6.1 Principles of Deadlock(5/9)

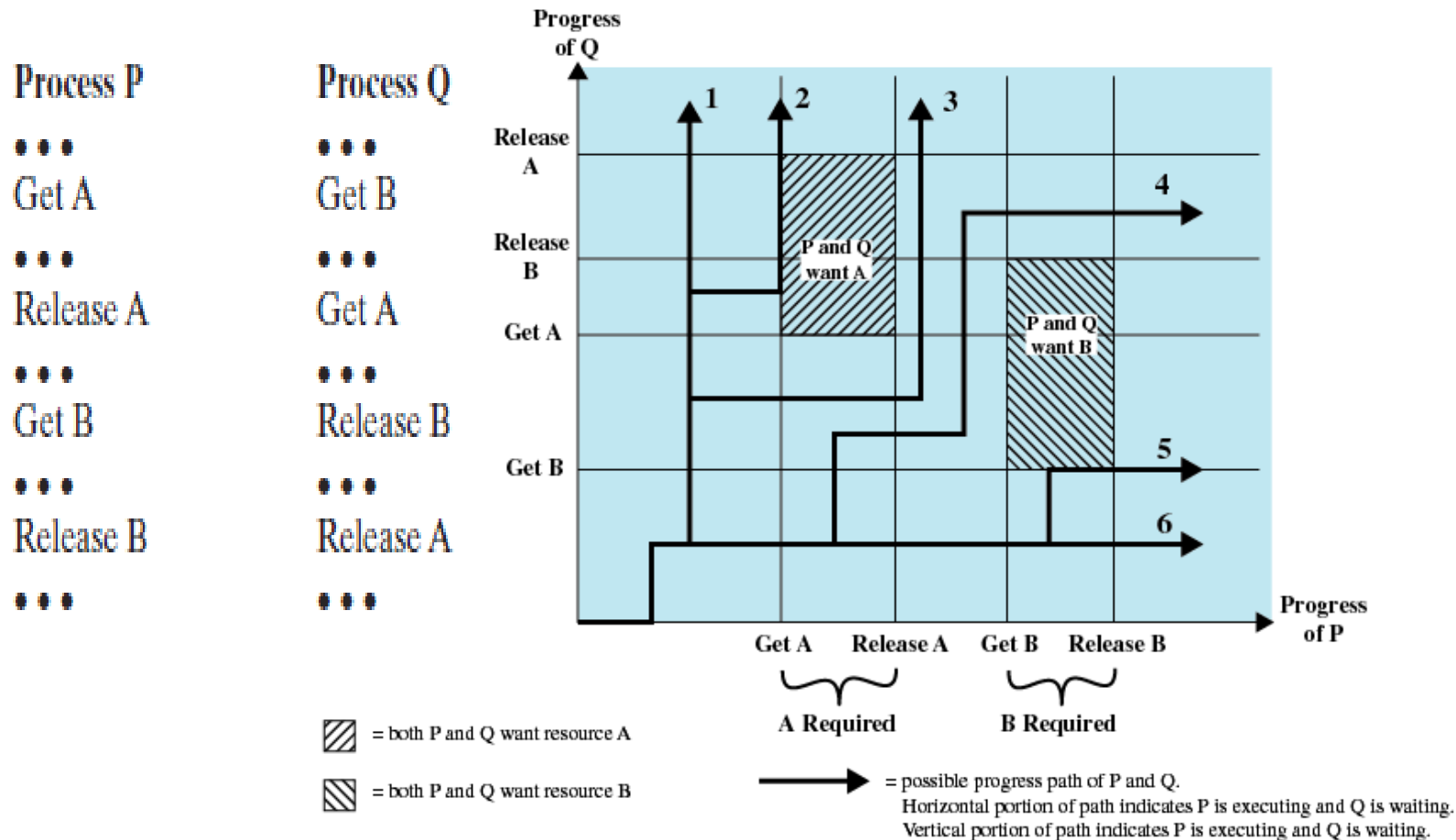


Figure 6.3 Example of No Deadlock [BACO03]

6.1 Principles of Deadlock(6/9)

Interleaves the execution: p0 p1 q0 q1 p2 q2

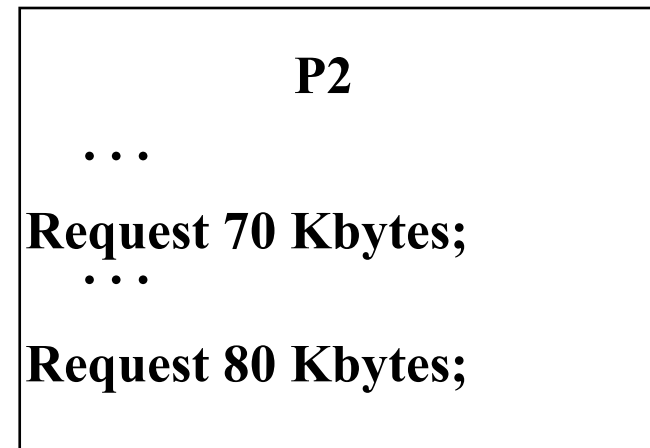
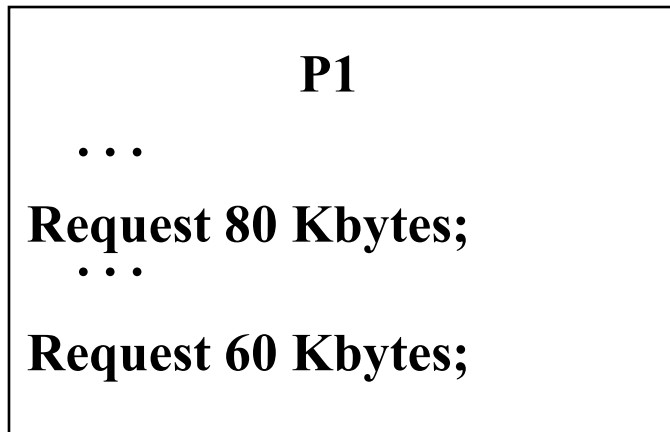
Process P		Process Q	
Step	Action	Step	Action
p ₀	Request (D)	q ₀	Request (T)
p ₁	Lock (D)	q ₁	Lock (T)
p ₂	Request (T)	q ₂	Request (D)
p ₃	Lock (T)	q ₃	Lock (D)
p ₄	Perform function	q ₄	Perform function
p ₅	Unlock (D)	q ₅	Unlock (T)
p ₆	Unlock (T)	q ₆	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

One strategy : require resources in order
(wait 信号的位置不要随意变换)

6.1 Principles of Deadlock(7/9)

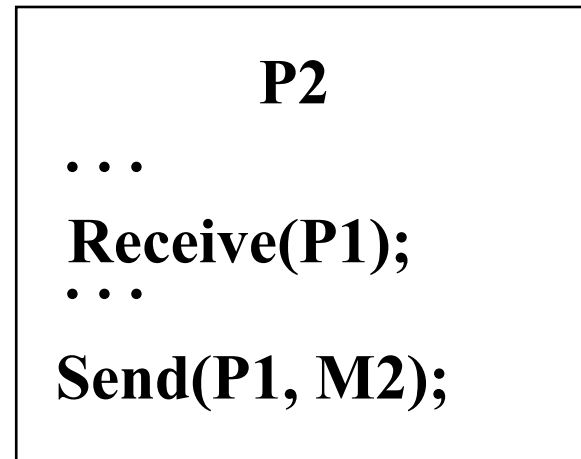
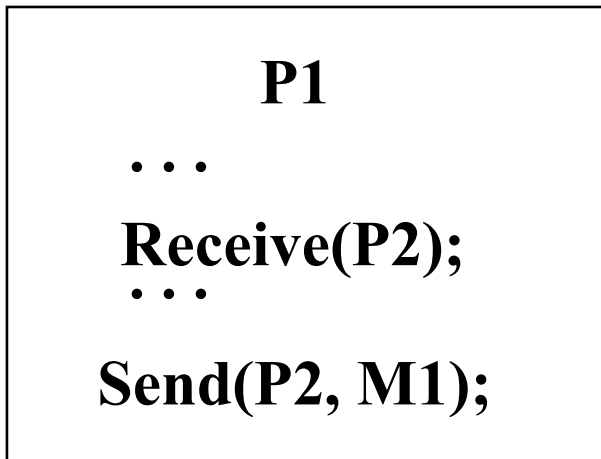
- Space is available for allocation of 200Kbytes, and the following sequence of events occur



One strategy : virtual memory 缓解发生概率

6.1 Principles of Deadlock(8/9)

- communicate designed incorrect (using blocking receive)



Design the program with caution

6.1 Principles of Deadlock(9/9)

- By now, we got that...

Processes may or may not cause deadlock



deadlock may be solved by different ways
according to resources and situation



How to generate a model to describe all
conditions



How to solve the problem

6.1 Principles of Deadlock

- 6.1.3 Resource Allocation Graphs
- 6.1.4 The Conditions for Deadlock

6.1.3 Resource Allocation Graphs(资源分配图)(1/5)

- Directed graph(有向图) that depicts(表述) a state of the system of resources and processes

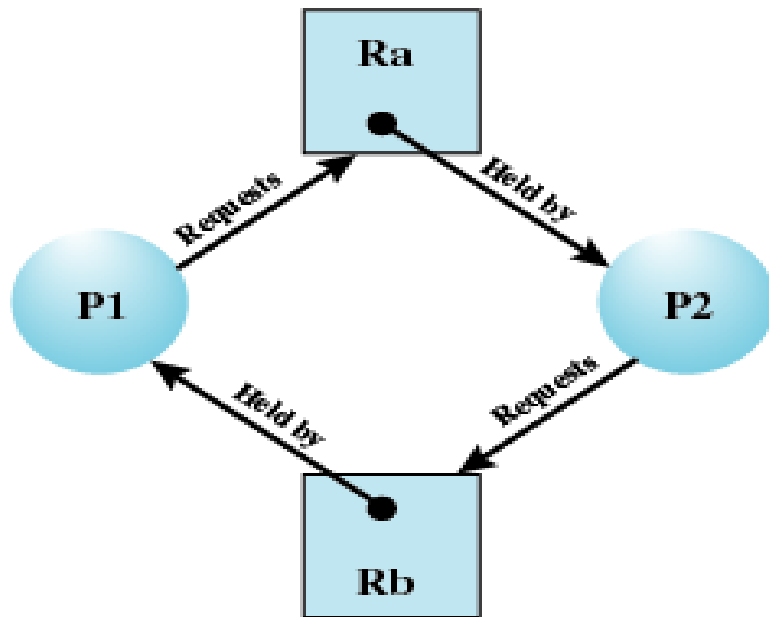


(a) Resource is requested

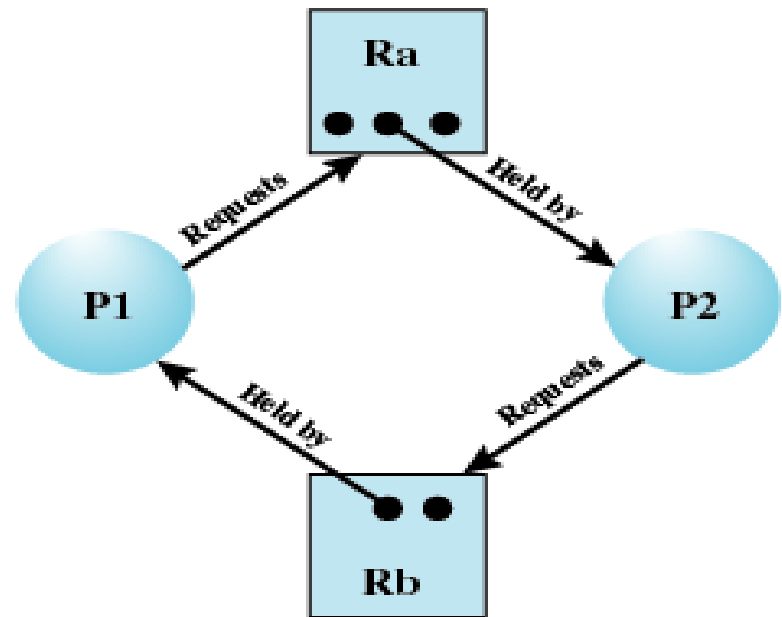


(b) Resource is held

6.1.3 Resource Allocation Graphs(2/5)



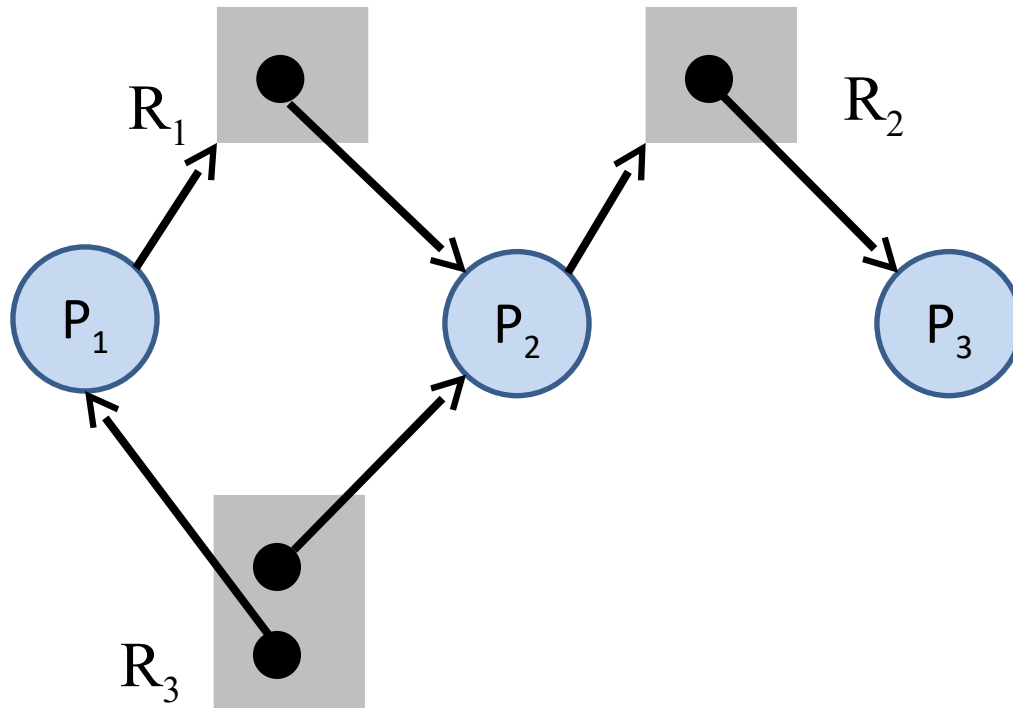
(c) Circular wait



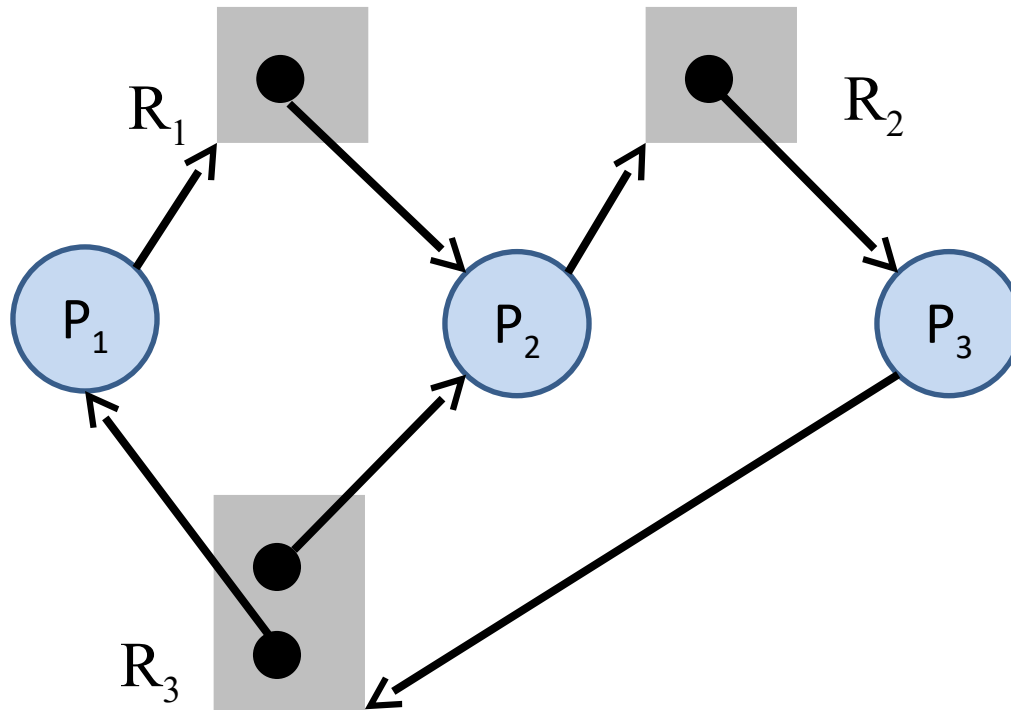
(d) No deadlock

Figure 6.5 Examples of Resource Allocation Graphs

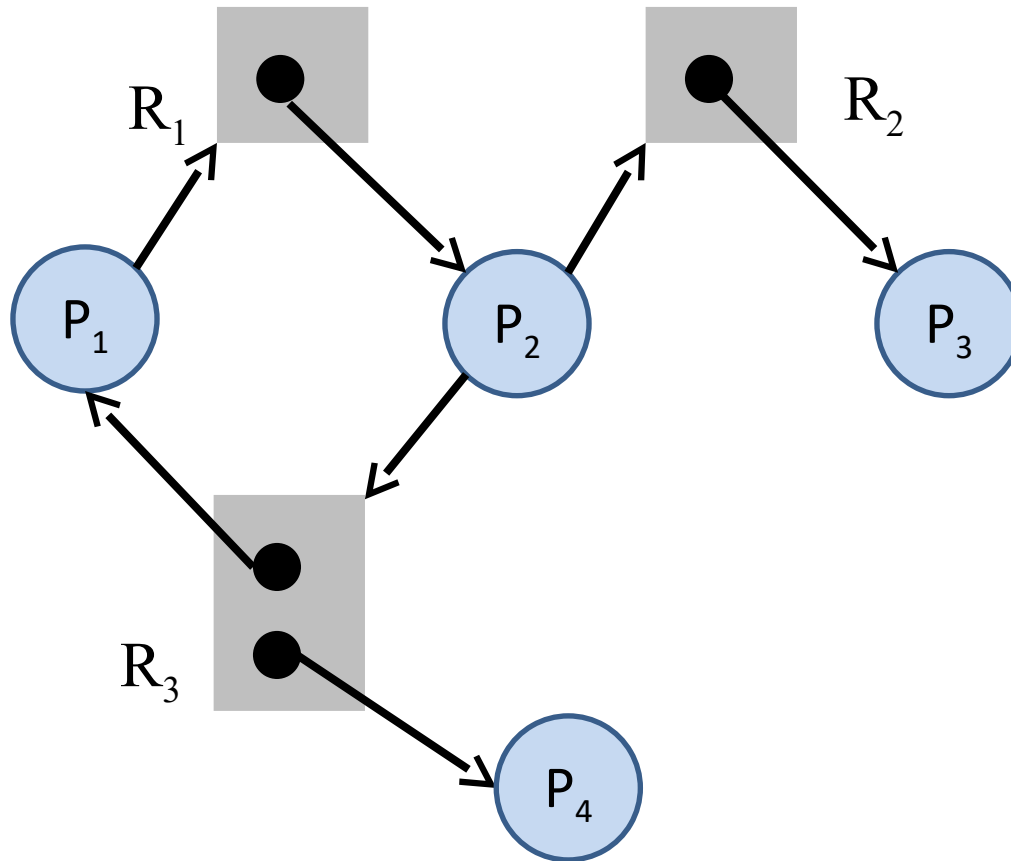
6.1.3 Resource Allocation Graphs(3/5)



6.1.3 Resource Allocation Graphs(4/5)



6.1.3 Resource Allocation Graphs(5/5)



6.1 Principles of Deadlock

- 6.1.3 Resource Allocation Graphs
- 6.1.4 The Conditions for Deadlock

6.1.4 The Conditions for Deadlock (死锁的条件)(1/3)

- Mutual exclusion(互斥)
 - A resource may used only by one process at a time
- Hold-and-wait(占有且等待)
 - A process may hold allocated resources while awaiting assignment of others
- No preemption(非抢占)
 - No resource can be forcibly removed from a process holding it

6.1.4 The Conditions for Deadlock(2/3)

- Circular wait(循环等待)
 - A **closed chain** of processes exists, such that **each process holds at least one resource needed by the next process** in the chain

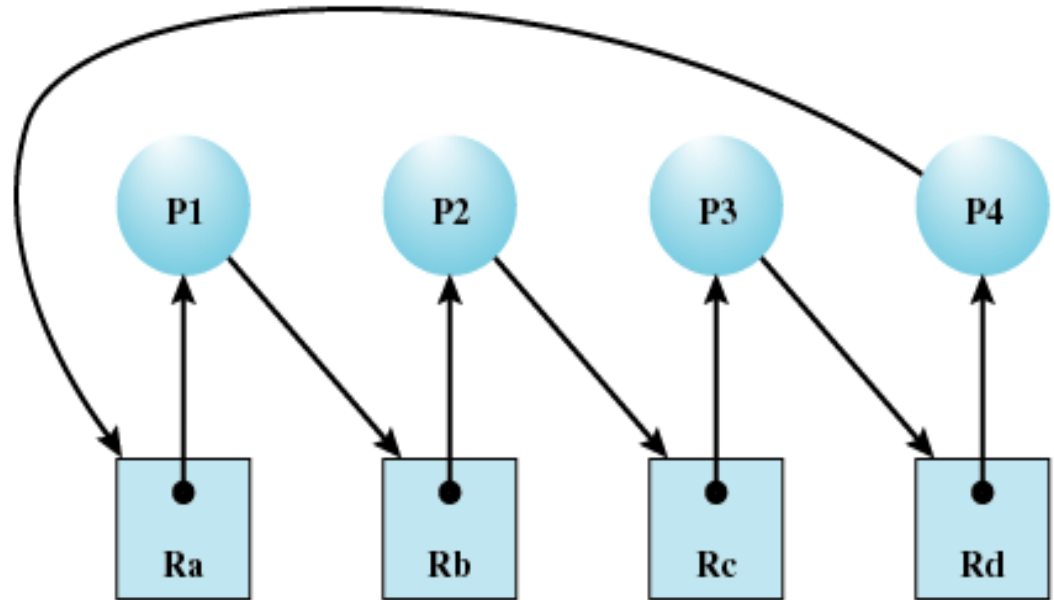
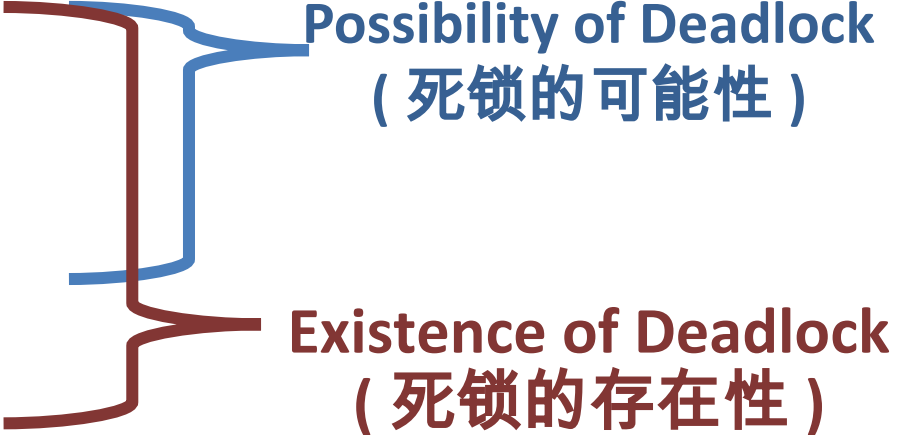


Figure 6.6 Resource Allocation Graph for Figure 6.1b

6.1.4 The Conditions for Deadlock(3/3)

- Mutual Exclusion
 - Hold and wait
 - No preemption
 - Circular wait
- 
- Possibility of Deadlock**
(死锁的可能性)
- Existence of Deadlock**
(死锁的存在性)

Agenda

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

↓
约束
强度
依次
减弱

6.2 Deadlock Prevention(死锁预防)(1/2)

- Mutual Exclusion
 - Must be supported by the operating system
 - 不太可能去掉互斥
- Hold and Wait
 - Require a process request all of its required resources at one time 拿到全部资源
 - 其它进程时效性、新需求的出现 (交互)

6.2 Deadlock Prevention(死锁预防)(2/2)

- No Preemption 无抢占
 - if a further request is denied , Process must release resource and request again
 - Operating system may preempt a process to require it releases its resources(kill the process, higher priority)
- Circular Wait
 - Define a linear ordering of resource types 线性排序
 - 时效性

Agenda

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

6.3 Deadlock Detection(1/13)

- 在资源申请前进行判断
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future resource requests

6.3 Deadlock Detection(2/13)

- Two Approaches to Deadlock Avoidance
 - 1. Process Initiation Denial
 - Do not start a process if its demands might lead to deadlock(如果一个进程的请求会导致死锁，则不启动此进程，**进程启动拒绝**)
 - 2.Resource Allocation Denial
 - Do not grant an incremental resource request to a process if this allocation might lead to deadlock(如果一个进程增加资源的请求会导致死锁，则不容许此分配，**资源分配拒绝**)

6.3 Deadlock Detection(3/13)

- 1. Process Initiation Denial 进程启动拒绝

Resource = $\mathbf{R} = (R_1, R_2, \dots, R_m)$	Total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \dots, V_m)$	Total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

1. $R_j = V_j + \sum_{i=1}^n A_{ij}$, for all j i process ID; j resource ID
Start a new process P_{n+1} only if

2. $C_{ij} \leq R_j$, for all i, j

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij} \quad \text{for all } j$$

3. $A_{ij} \leq C_{ij}$, for all i, j

6.3 Deadlock Detection(4/13)

- 2.Resource Allocation Denial 分配资源拒绝 --
Referred to as the **banker's algorithm** 银行家算法
 - State of the system(系统状态) is the current allocation of resources to process
 - Safe state(安全状态) is where there is at least one **sequence 序列** that does not result in deadlock
 - Unsafe state(不安全状态) is a state that is not safe

6.3 Deadlock Detection(5/13)

- 安全状态示范

a. Initial State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

Determination of a Safe State

6.3 Deadlock Detection(6/13)

b.P2 Runs to Completion

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

need

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

Determination of a Safe State

6.3 Deadlock Detection(7/13)

c. P1 Runs to Completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

Determination of a Safe State

6.3 Deadlock Detection(8/13)

d. P3 Runs to Completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

safe sequence : P2,P1,P3,P4

Determination of a Safe State

6.3 Deadlock Detection(9/13)

- 不安全状态示范

Initial state is safe, if p2 completes first.

But before p2 requests the remaining resources and completes, P1 requests for one additional unit of R1 and R3 each

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

Determination of a Unsafe State³⁴

6.3 Deadlock Detection(10/13)

If P1's request is granted, the state is **unsafe**. 满足 p1 请求，会进入不安全状态

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

Determination of a Unsafe State ₃₅

6.3 Deadlock Detection(11/13)

Deadlock Avoidance Logic(死锁避免逻辑)

```
struct state
{
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
    < error >;                                /* total request > claim*/
else if (request [*] > available [*])
    < suspend process >;
else                                           /* simulate alloc */
{
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*] >;
}
if (safe (newstate))
    < carry out allocation >;
else
{
    < restore original state >;
    < suspend process >;
}
```

(b) resource alloc algorithm

6.3 Deadlock Detection(12/13)

Deadlock Avoidance Logic

```
boolean safe (state S)
{
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible)
    {
        <find a process  $P_k$  in rest such that
            claim  $[k, *] - \text{alloc } [k, *] \leq \text{currentavail};$ >
        if (found)                                /* simulate execution of  $P_k$  */
        {
            currentavail = currentavail + alloc  $[k, *]$ ;
            rest = rest -  $\{P_k\}$ ;
        }
        else
            possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

6.3 Deadlock Detection(13/13)

- Restrictions 限制 of Deadlock Avoidance
 - Maximum resource requirement must be stated in advance 最大资源需求量需要提前申明
 - Processes under consideration must be **independent**; **no synchronization** requirements (进程间相互独立, 否则影响序列的搜索)
 - 比如等待某个进程的消息
 - There must be a fixed number of resources to allocate
 - No process may exit while holding resources

Agenda

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

6.4 Deadlock Detection(1/6)

- **1. Mark** each process that has a row in the Allocation matrix of all zeros. A process that has no allocated resources cannot participate in a deadlock.
- **2.** Initialize a temporary vector **W** to equal the **Available** vector.
- **3.** Find an index i such that process i is currently **unmarked** and the i th row of **Q(request)** is less than or equal to **W**. That is, $Q_{ik} \leq W_k$, for $k: 1 \dots m$. If no such row is found, terminate the algorithm.
- **4.** If such a row is found, mark process i and add the corresponding row of the allocation matrix to **W**. That is, set $W_k = W_k + A_{ik}$, for $1 \dots k \dots m$. Return to step 3.

6.4 Deadlock Detection(2/6)

Reference P.265 of the textbook for deadlock detection algorithm

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
T	0	0	0	0	1
T	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	1	1

Available vector

P1 & P2 cause deadlock

Figure 6.10 Example for Deadlock Detection

6.4 Deadlock Detection(3/6)

- Deadlock Avoidance \neq Deadlock Detection
 - Deadlock Avoidance : 在分配新资源时，进行是否安全，是否导致死锁的检查
 - Deadlock Detection : 所有新资源分配的请求都允许，周期性的检测当前的资源状况是否已经死锁

6.4 Deadlock Detection(4/6)

- Recovery Strategies Once Deadlock Detected (死锁检测后的解锁策略)
- 复杂度递增
 - Abort all deadlocked processes (most common)
 - Back up (回滚) each deadlocked process to some previously defined checkpoint, and restart all process
 - Original deadlock may occur, however nondeterminancy of concurrent processing may ensure this doesn't happen.
 - Successively abort (连续取消) deadlocked processes until deadlock no longer exists
 - Successively preempt (连续剥夺) resources until deadlock no longer exists

6.4 Deadlock Detection(5/6)

- Selection Criteria Deadlocked Processes (被剥夺或者取消进程的选择标准)
 - Least amount of processor time consumed so far
 - Least number of lines of output produced so far
 - Most estimated time remaining (not easy to measure)
 - Least total resources allocated so far
 - Lowest priority

6.4 Deadlock Detection(6/6)

Strengths and Weaknesses of the Strategies

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none">•Works well for processes that perform a single burst of activity•No preemption necessary	<ul style="list-style-type: none">•Inefficient•Delays process initiation•Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none">•Convenient when applied to resources whose state can be saved and restored easily	<ul style="list-style-type: none">•Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none">•Feasible to enforce via compile-time checks•Needs no run-time computation since problem is solved in system design	<ul style="list-style-type: none">•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none">•No preemption necessary	<ul style="list-style-type: none">•Future resource requirements must be known by OS•Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none">•Never delays process initiation•Facilitates on-line handling	<ul style="list-style-type: none">•Inherent preemption losses

Agenda

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

6.5 An Integrated Deadlock Strategy(一种综合死锁策略)(1/2)

- Group resources into a number of different resource classes (资源分类)
- Use the linear ordering strategy(线性排序策略) defined previously for the prevention of circular wait to prevent deadlocks between resource classes(类与类之间用线性排序策略避免死锁)
 - e.g. steps process may follow during lifetime
 - 可交换空间
 - 进程资源
 - 内存
 - 外部资源 (I/O)

6.5 An Integrated Deadlock Strategy(一种综合死锁策略)(2/2)

- Within a resource class, use the algorithm that is most appropriate for that class(每个类中使用适合于该类的策略)
 - Swappable space 可交换空间：外存一次性分配
 - Process resources：死锁避免或排序
 - Mem: page/segment, 置换到辅存
 - 内部资源：排序

Agenda

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

6.6 Dining Philosophers Problem(1/7)

哲学家就餐问题

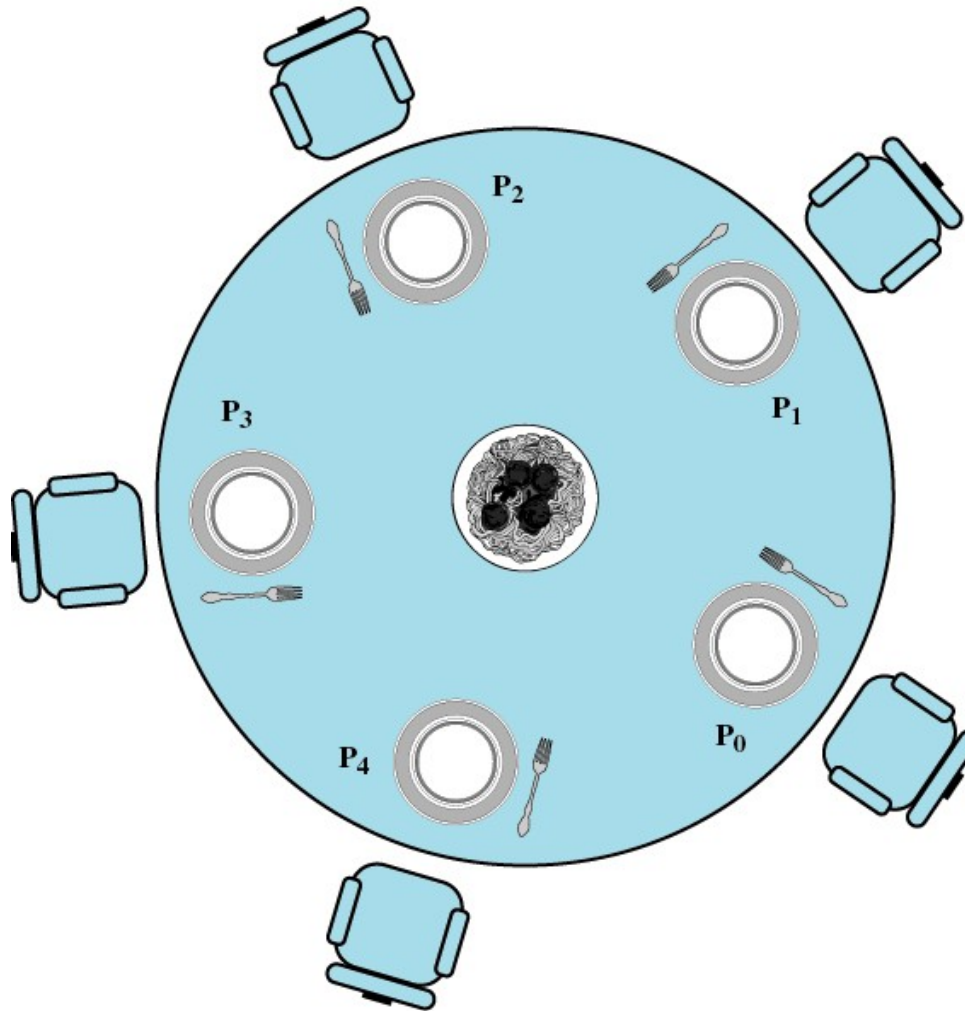


Figure 6.11 Dining Arrangement for Philosophers

6.6 Dining Philosophers Problem (2/7)

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

incorrect

Figure 6.12 **A First Solution to the Dining Philosophers Problem**

6.6 Dining Philosophers Problem(3/7)

```
/*program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

method 1

Figure 6.13 A Second Solution to the Dining Philosophers Problem

Dining Philosophers Problem(4/7)-method 2

- 先测试 , 只有左右都有叉子才吃饭
- 1 : thinking
- 2. feel hungry
- 3. test: if right **OR** left neighbor is eating, wait; if not, continue
- 4. take two forks
- 5. eating
- 6. put left fork, test left neighbor, if its left neighbor is not eating , invoke it
- 7. put right fork, test right neighbor, if its right neighbor is not eating , invoke it
- 8. jump to 1
- Example Code

Dining Philosophers Problem(5/7)-method 3 Monitor

```
void philosopher[k=0 to 4]    /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k);    /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k); /* client releases forks via the monitor */
    }
}
```

ε **Figure 6.14** A Solution to the Dining Philosophers Problem Using a Monitor

Dining Philosophers Problem(6/7)-method 3 Monitor

```
monitor dining_controller;
cond ForkReady[5];    /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid) /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork[left])
        cwait(ForkReady[left]); /* queue on condition variable */
    fork[left] = false;
    /*grant the right fork*/
    if (!fork[right])
        cwait(ForkReady[right]); /* queue on condition variable */
    fork[right] = false;
}
```

Dining Philosophers Problem(7/7)-method 3 Monitor

```
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left])/*no one is waiting for this fork */
        fork[left] = true;
    else          /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right])/*no one is waiting for this fork */
        fork[right] = true;
    else          /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```


Agenda

- 6.1 Principles of Deadlock
- 6.2 Deadlock Prevention
- 6.3 Deadlock Avoidance
- 6.4 Deadlock Detection
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary