Operating Systems

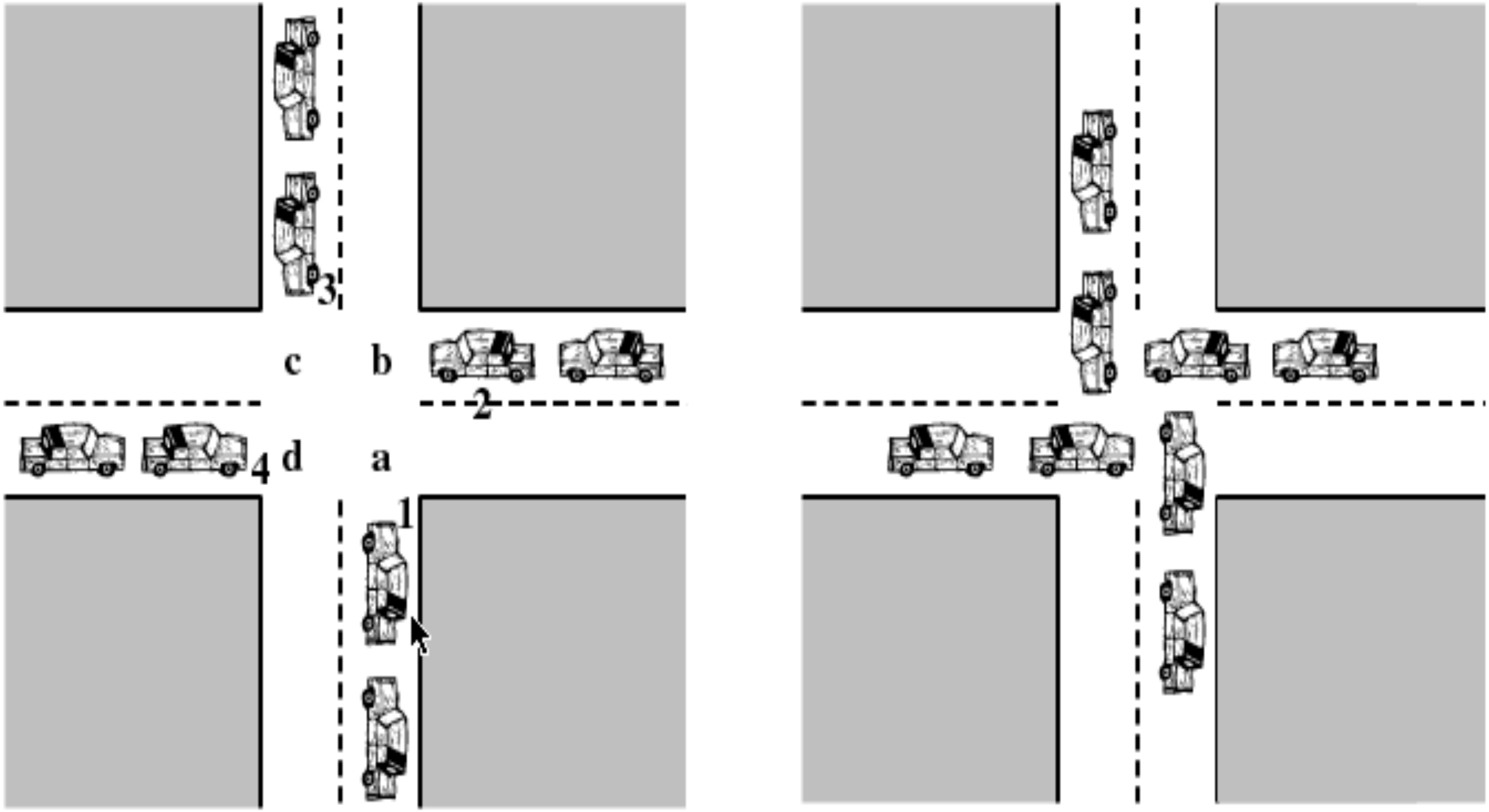
Chapter 6 Concurrency: Deadlock(死锁) and Starvation(饥饿)

6.1 Principles of Deadlock

- <u>6.1 Principles of Deadlock</u>
- 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

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution in the general case(通用)
- 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).
  - A common example is the traffic deadlock (交通死锁)



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

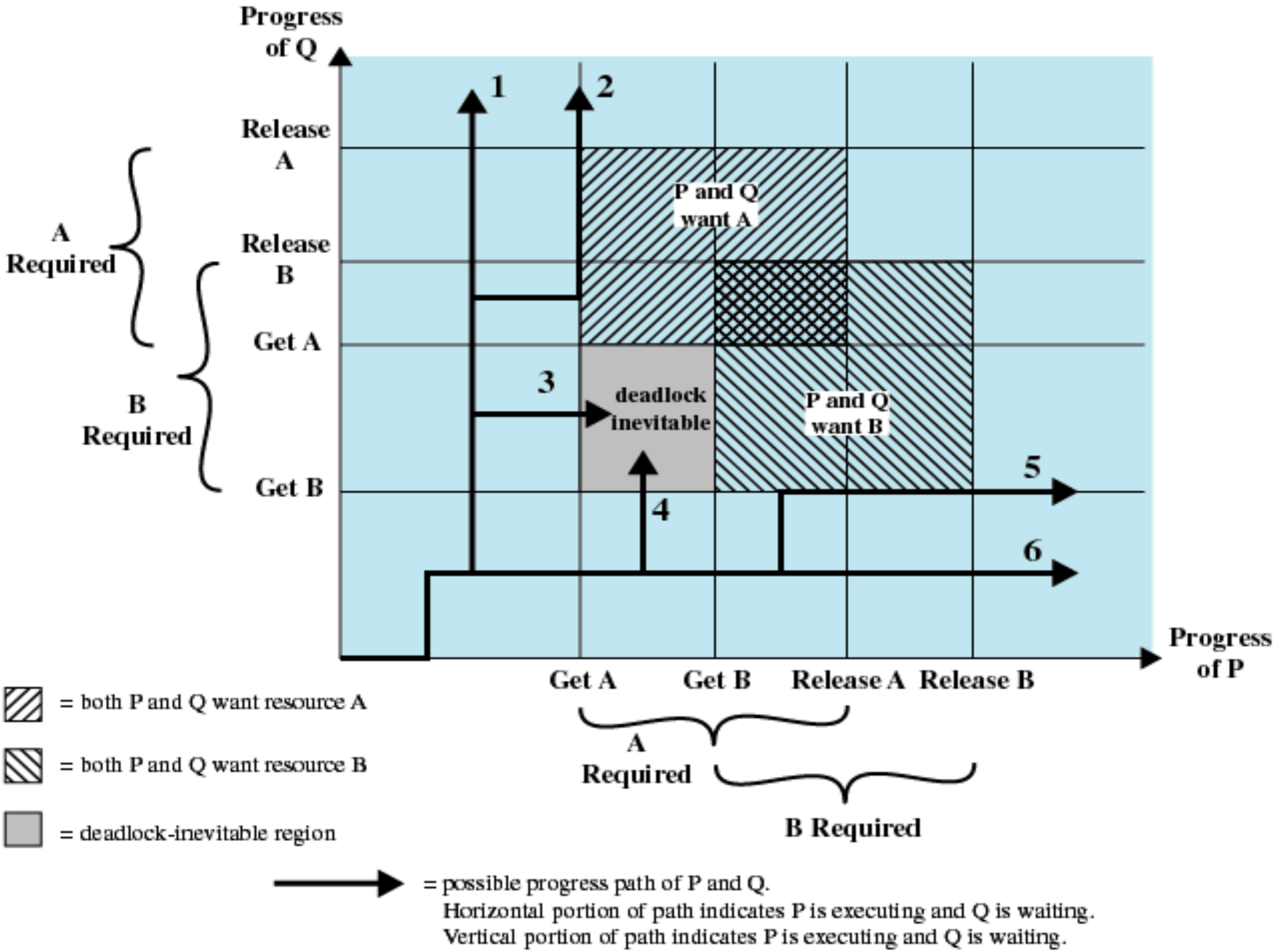


Figure 6.2 Example of Deadlock

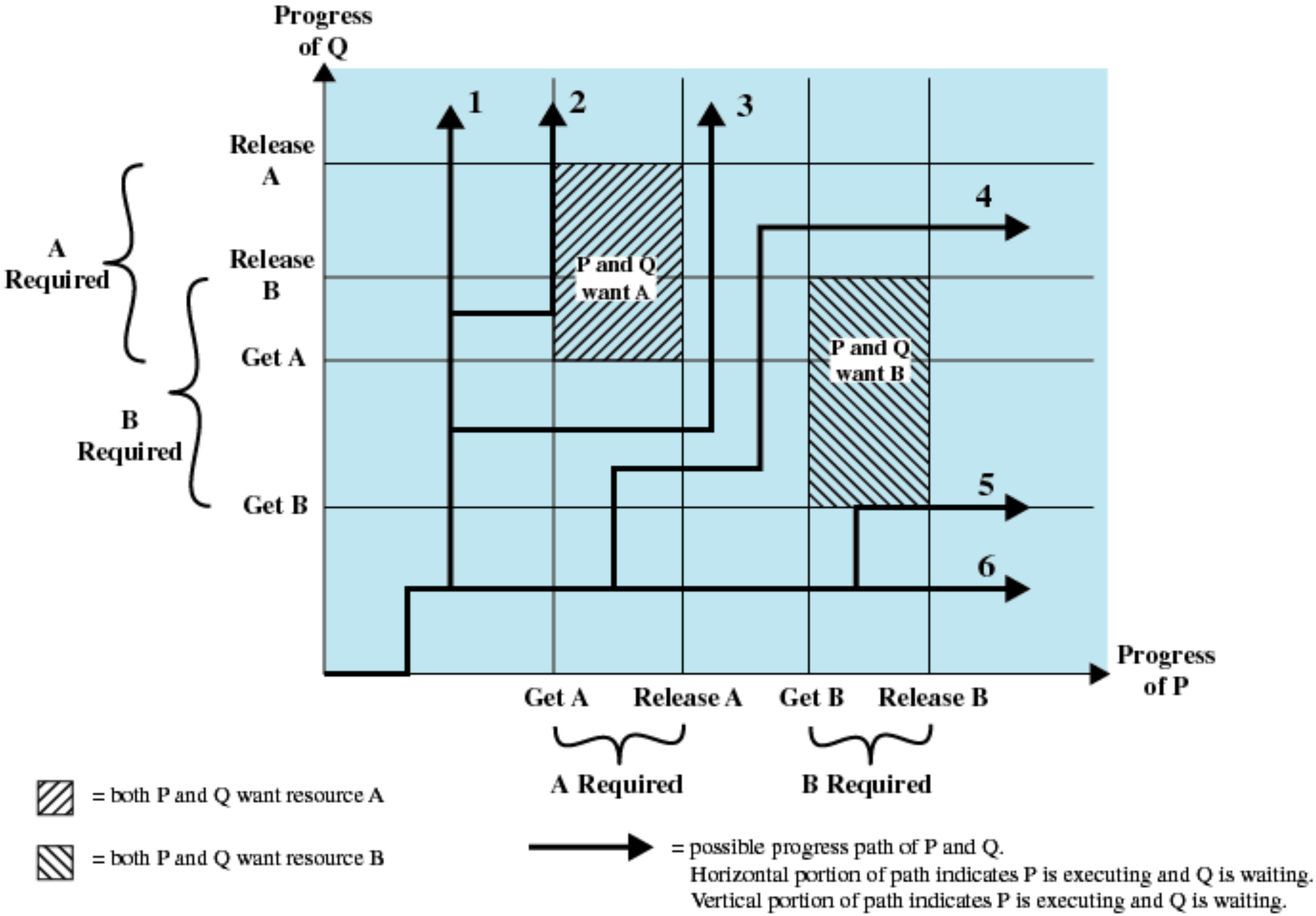


Figure 6.3 Example of No Deadlock [BACO03]

Resources Categories(资源的分类)

- Reusable Resources(可重用资源)
- Consumable Resources(可消费资源)

6.1 Principles of Deadlock

- 6.1.1 Reusable Resources
- 6.1.2 Consumable Resources
- 6.1.3 Resource Allocation Graphs
- 6.1.4 The Conditions for Deadlock

Reusable Resources(可重用资源)

- Used by only one process at a time and not depleted(耗尽) by that use
- Processes obtain resources that they later release for reuse by other processes
- Examples include processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases

Deadlock Example of Reusable Resources

Interleaves the execution: p0 p1 q0 q1 p2 q2

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Another Deadlock Example of Reusable Resources

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

P1	P2
. . .	. . .
Request 80 Kbytes;	**Request 70 Kbytes;**
. . .	. . .
Request 60 Kbytes;	**Request 80 Kbytes;**

- Deadlock occurs if both processes progress to their second request

6.1 Principles of Deadlock

- 6.1.1 Reusable Resources
- <u>6.1.2 Consumable Resources</u>
- 6.1.3 Resource Allocation Graphs
- 6.1.4 The Conditions for Deadlock

Consumable Resources(可消费资源)

- May be created (produced) and destroyed (consumed) by processes
- Examples include interrupts, signals, messages, and information in I/O buffers

Deadlock Example of Consumable Resources

- Deadlock occurs if receive is blocking

P1	P2
. . .	**. . .**
Receive(P2);	**Receive(P1);**
. . .	**. . .**
Send(P2, M1);	**Send(P1, M2);**

6.1 Principles of Deadlock

- 6.1.1 Reusable Resources
- 6.1.2 Consumable Resources
- <u>6.1.3 Resource Allocation Graphs</u>
- 6.1.4 The Conditions for Deadlock

Resource Allocation Graphs(资源分配图)

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

(a) Resouce is requested

(b) Resource is held

Resource Allocation Graphs

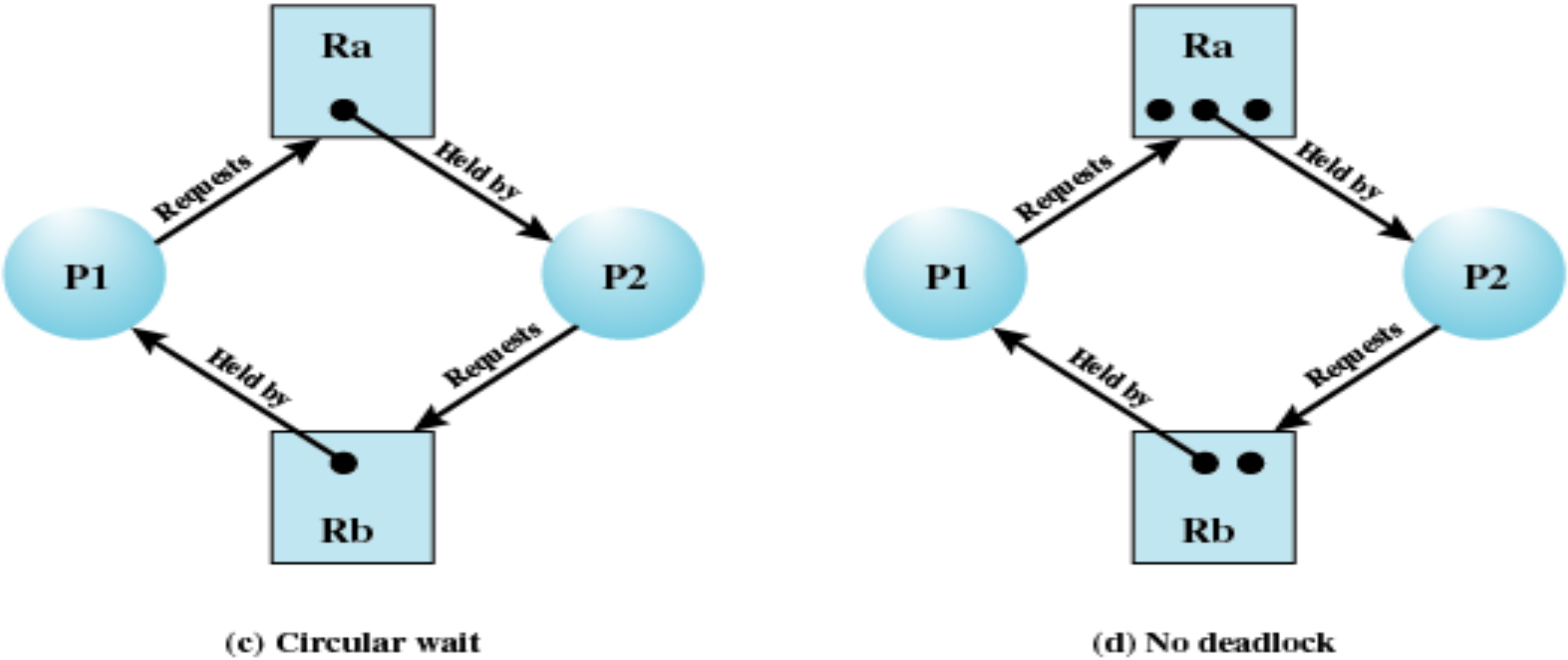


Figure 6.5 Examples of Resource Allocation Graphs

6.1 Principles of Deadlock

- 6.1.1 Reusable Resources
- 6.1.2 Consumable Resources
- 6.1.3 Resource Allocation Graphs
- <u>6.1.4 The Conditions for Deadlock</u>

Conditions for Deadlock(死锁的条件)

- 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

Conditions for Deadlock

- 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

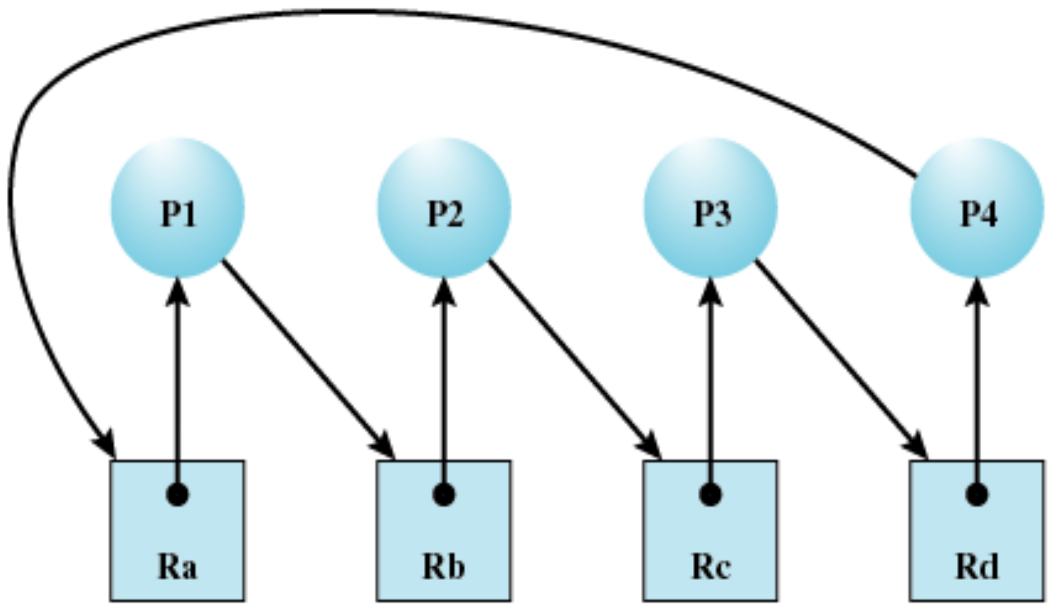


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Possibility of Deadlock(死锁的可能性)

- Mutual Exclusion
- Hold and wait
- No preemption（抢占权）

Existence of Deadlock(死锁的存在性)

- Mutual Exclusion
- Hold and wait
- No preemption
- Circular wait

Agenda

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

Deadlock Prevention(死锁预防)

- Mutual Exclusion
  - Must be supported by the operating system
- Hold and Wait
  - Require a process request all of its required resources at one time

Deadlock Prevention

- No Preemption
  - Process must release resource and request again
  - Operating system may preempt a process to require it releases its resources
- Circular Wait
  - Define a linear ordering of resource types

Agenda

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

Deadlock Avoidance(死锁避免)

- 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

Two Approaches to Deadlock Avoidance

- 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*)
- 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*)

Resource Allocation Denial

- Referred to as the banker's algorithm
- 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 that does not result in deadlock
- Unsafe state(不安全状态) is a state that is not safe

Determination of a Safe State
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
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

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
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

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

Determination of an Unsafe State

P1 request for one additional unit each of R1 and R3

	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 an Unsafe 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	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

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

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 Pk in rest such that
         claim [k,*] - alloc [k,*] <= currentavail;>
      if (found)                        /* simulate execution of Pk */
      {
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else
         possible = false;
   }
   return (rest == null);
}
```

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

Restrictions of Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- 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
- <u>6.4 Deadlock Detection</u>
- 6.5 An Integrated Deadlock Strategy
- 6.6 Dining Philosophers Problem
- 6.7 Summary

Deadlock Detection

Reference the textbook for deadlock detection algorithm

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

Available vector

Figure 6.10 Example for Deadlock Detection

Recovery Strategies Once Deadlock Detected (死锁检测到后的解锁策略)

- Abort all deadlocked processes
- Back up (回滚) each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- 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

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
- Least total resources allocated so far
- Lowest priority

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	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•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	•Never delays process initiation •Facilitates on-line handling	•Inherent preemption losses

Whatever, Deadlock(1/1)

- What's it meaning?
- Reasonable?

Agenda

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

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

- 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)
- 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)

Agenda

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

Dining Philosophers Problem(哲学家就餐问题)

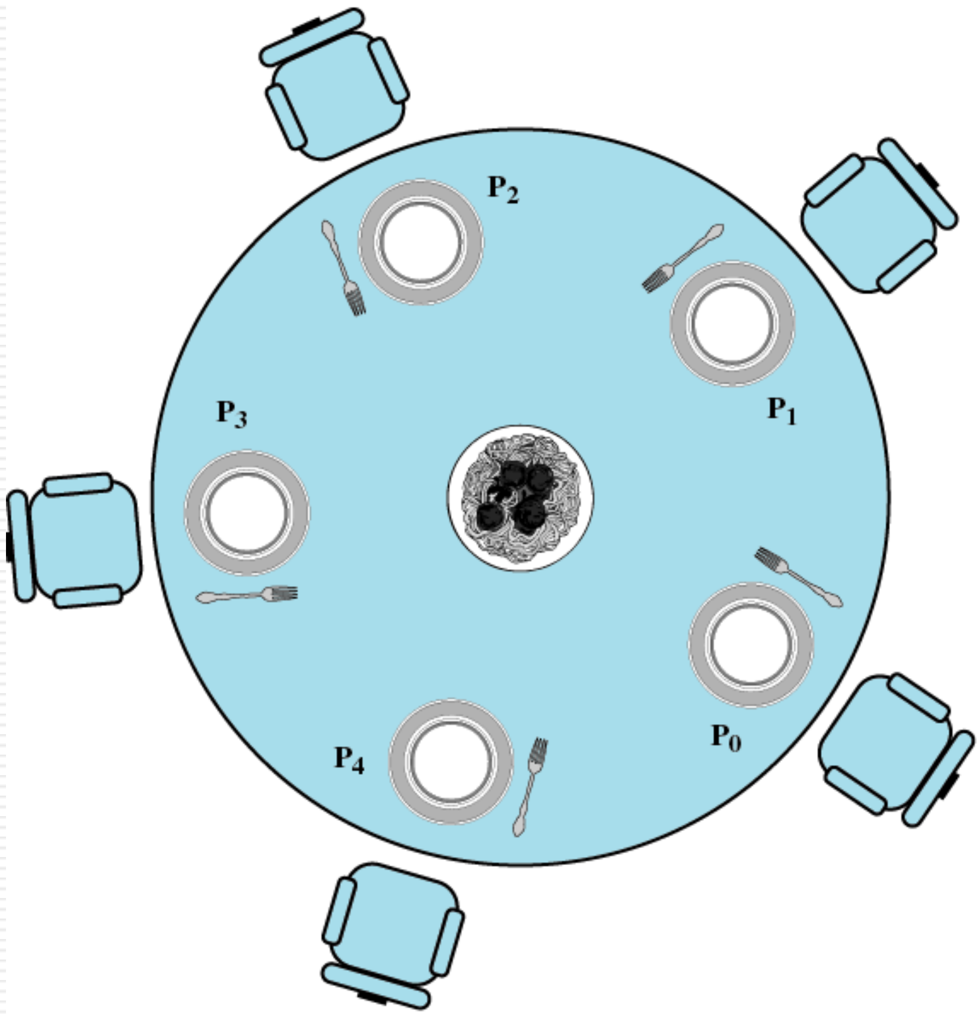


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem (incorrect)

```
/* 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));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Dining Philosophers Problem

```
/* 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));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

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