操作系统

第6章 并发性：死锁和饥饿
Concurrency: Deadlock and Starvation

孙承杰
哈工大计算学部

E-mail： sunchengjie@hit.edu.cn
2025年秋季学期

Learning Objectives

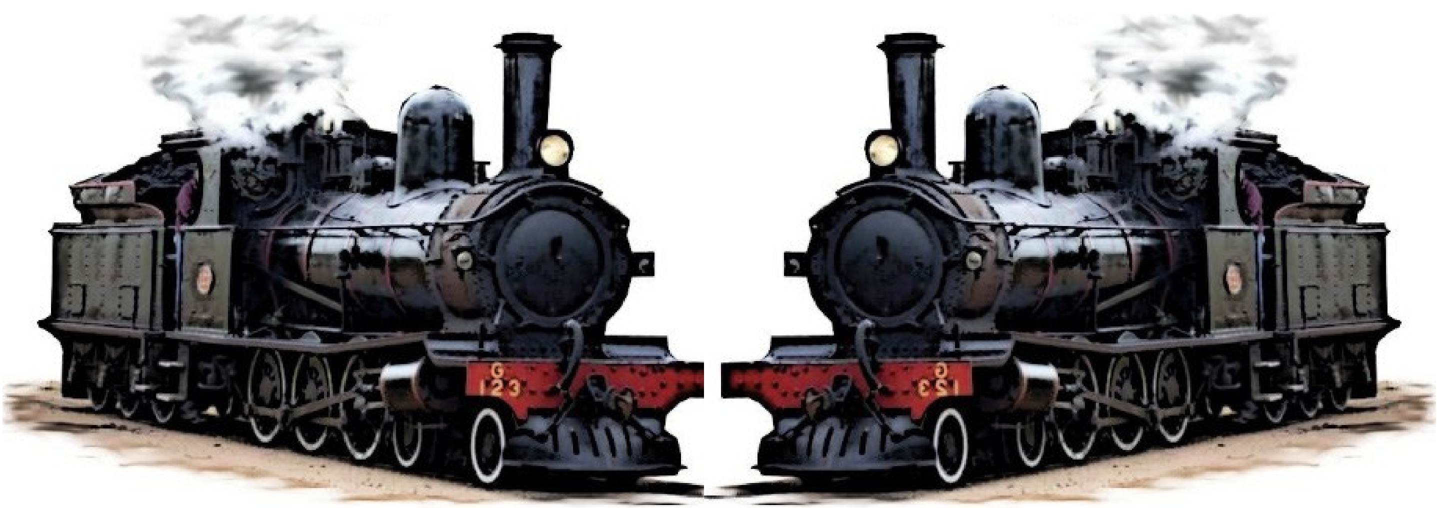
- List and explain the conditions for deadlock
- Define deadlock prevention and describe deadlock prevention strategies related to each of the conditions for deadlock
- Explain the difference between deadlock prevention and deadlock avoidance
- Understand two approaches to deadlock avoidance
- Explain the fundamental difference in approach between deadlock detection and deadlock prevention or avoidance
- Analyze the dining philosophers problem

Outline

- **Principles of Deadlock**
  - Reusable resources
  - Consumable resources
  - Resource allocation graphs
  - The conditions for deadlock
- **Deadlock Prevention**
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- **Deadlock Avoidance**
  - Process initiation denial
  - Resource allocation denial
- **Deadlock Detection**
  - Deadlock detection algorithm
  - Recovery
- **Dining Philosophers Problem**
  - Solution using semaphores
  - Solution using a monitor

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. Statute passed by the Kansas State Legislature, early in the 20th century.

-- *A TREASURY OF RAILROAD FOLKLORE,*
B. A. Botkin and Alvin F. Harlow



Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other

- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set

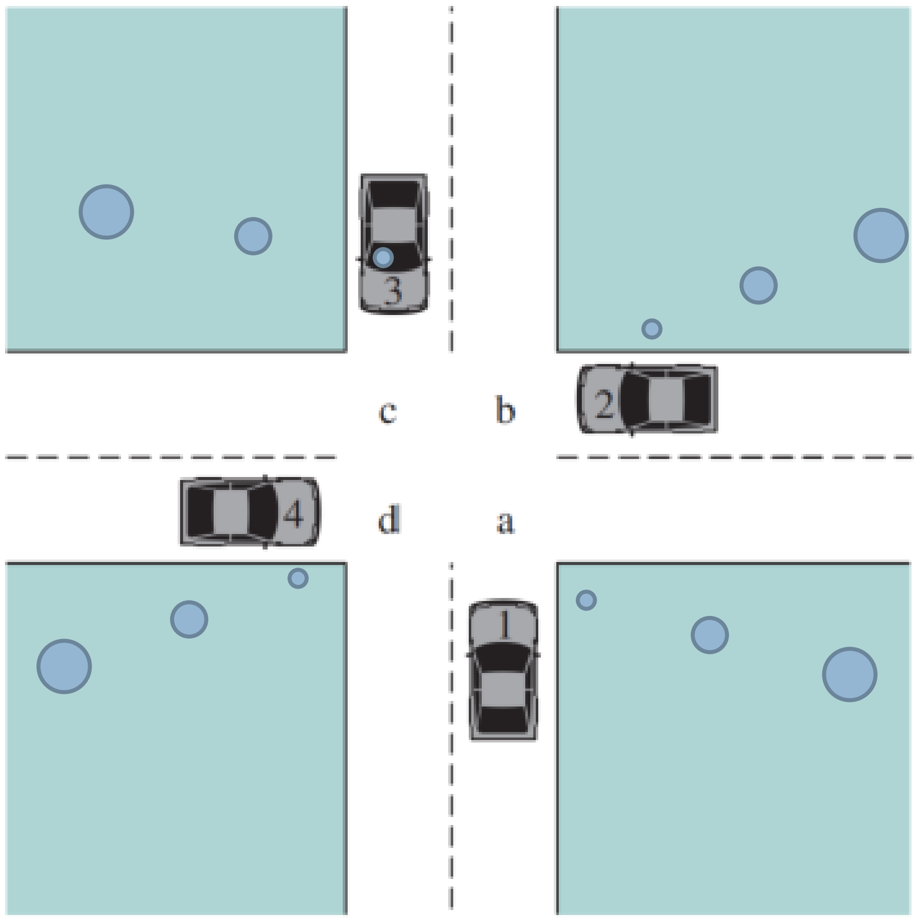
- Permanent

- No efficient solution

Potential Deadlock

I need quad
C and D

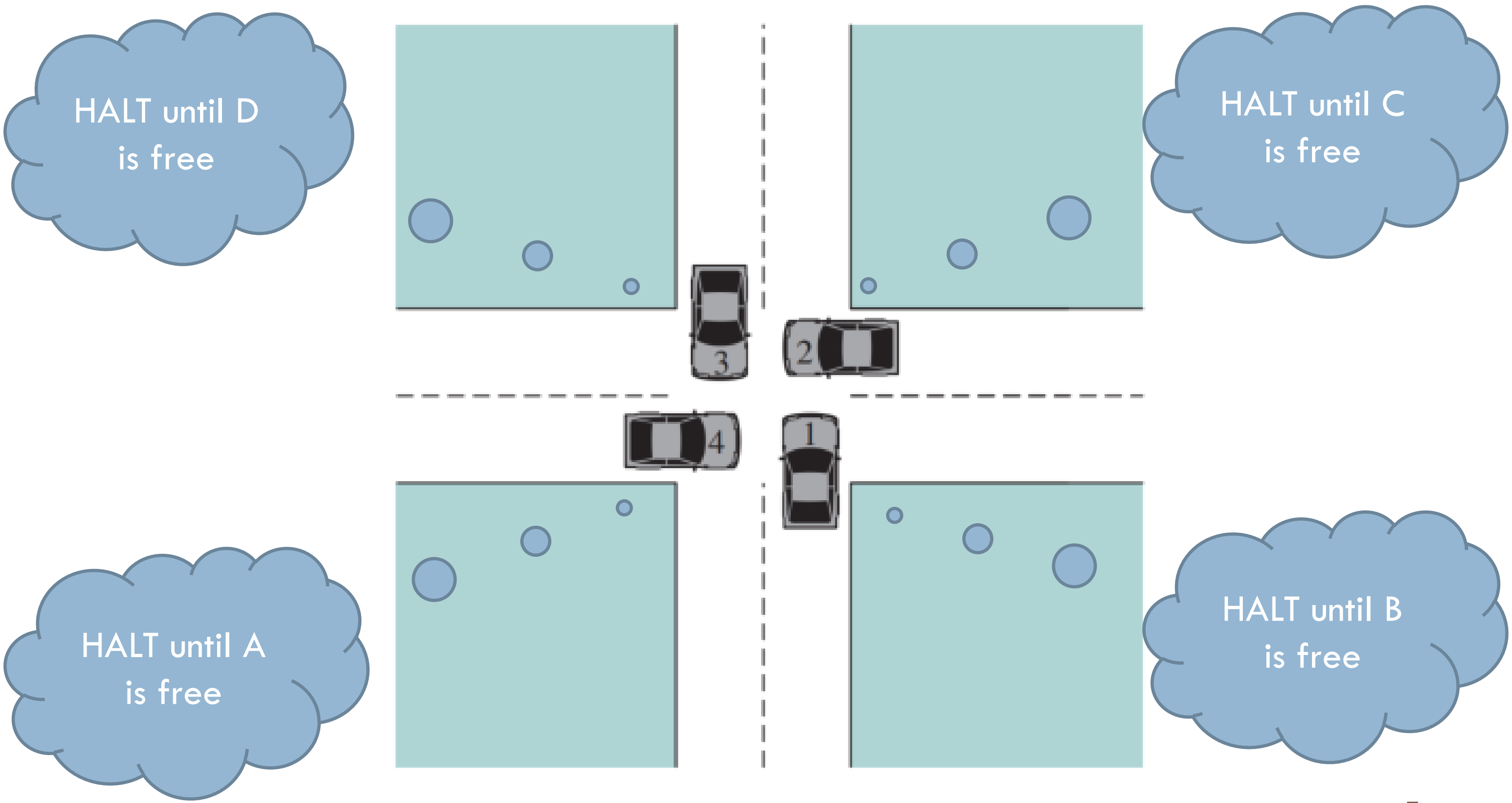
I need quad
B and C



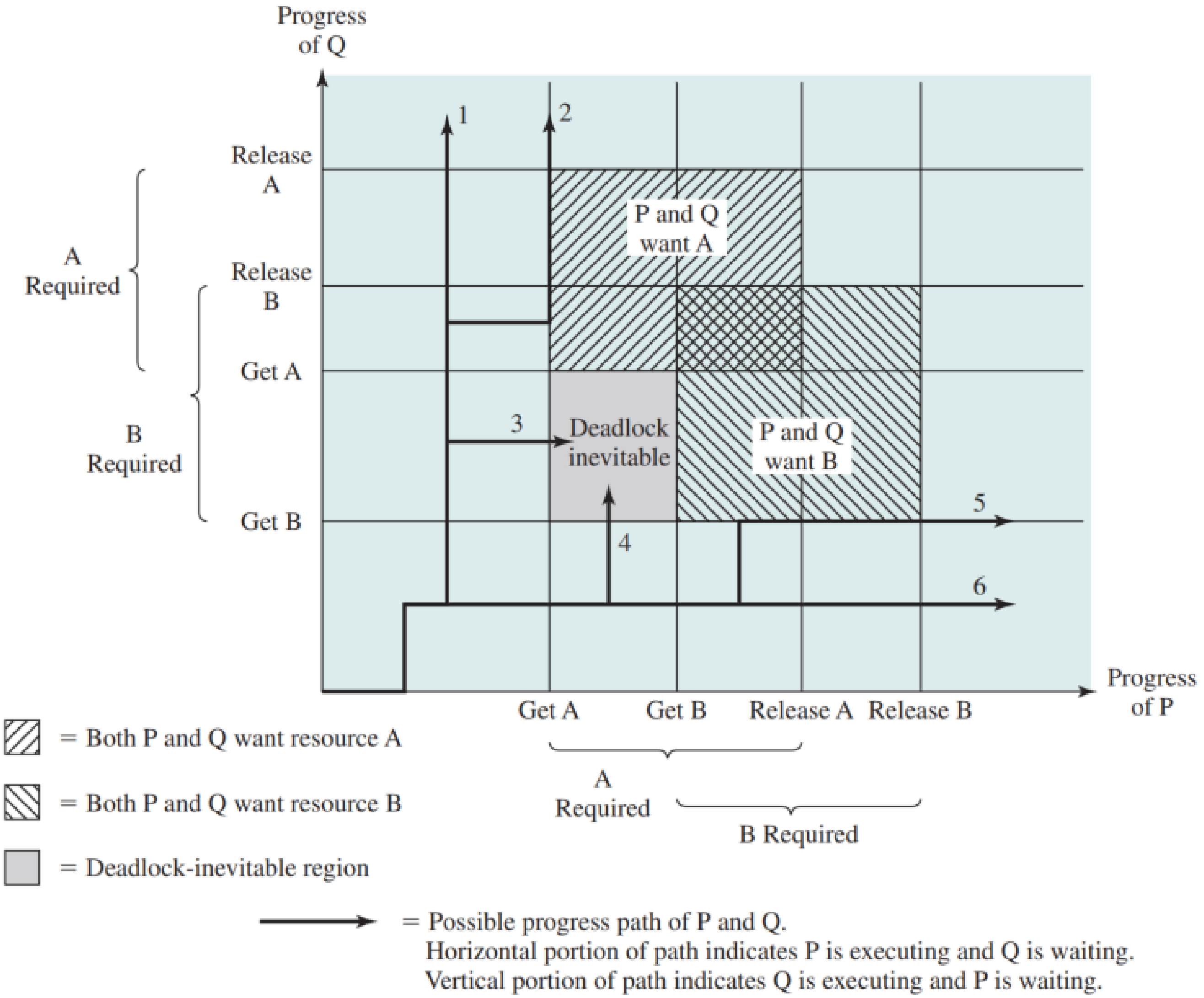
I need quad
D and A

I need quad
A and B

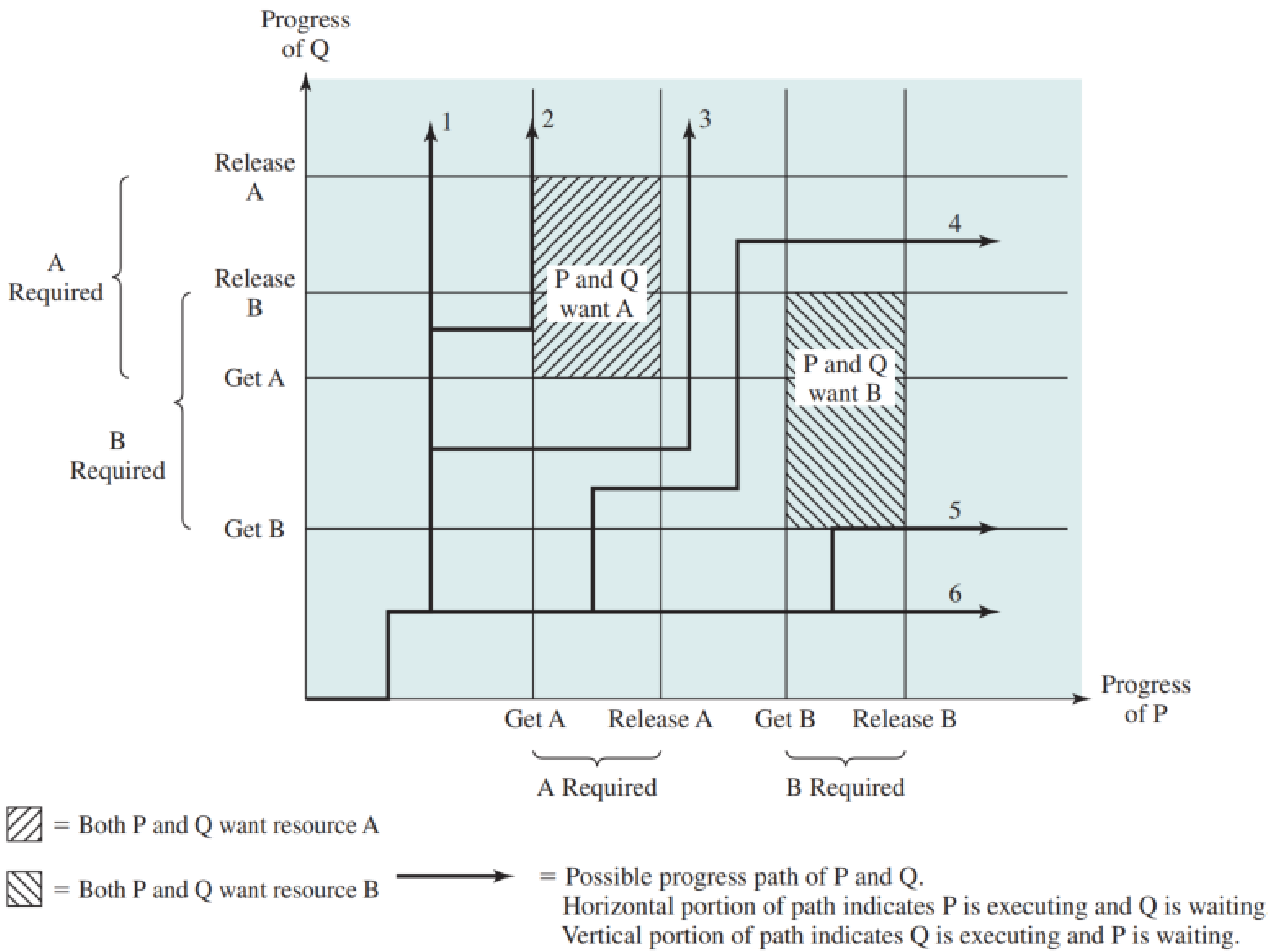
Actual Deadlock



Deadlock Example



No Deadlock Example



Resource Categories

Reusable(可重用)

- can be safely used by only one process at a time and is not depleted by that use
  - processors
  - I/O channels
  - main and secondary memory
  - devices
  - data structures such as files, databases, and semaphores

Consumable(可消耗)

- one that can be created (produced) and destroyed (consumed)
  - interrupts
  - signals
  - messages
  - information in I/O buffers

Reusable Resources: Example 1

Step		Process P Action
p_0	1	Request (D)
p_1	2	Lock (D)
p_2	5	Request (T)
p_3		Lock (T)
p_4		Perform function
p_5		Unlock (D)
p_6		Unlock (T)

Step		Process Q Action
q_0	3	Request (T)
q_1	4	Lock (T)
q_2	6	Request (D)
q_3		Lock (D)
q_4		Perform function
q_5		Unlock (T)
q_6		Unlock (D)

Example of Two Processes Competing for Reusable Resources

Example 2: Memory Request

- Space is available for allocation of 200 Kbytes, 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

Consumable Resources Deadlock

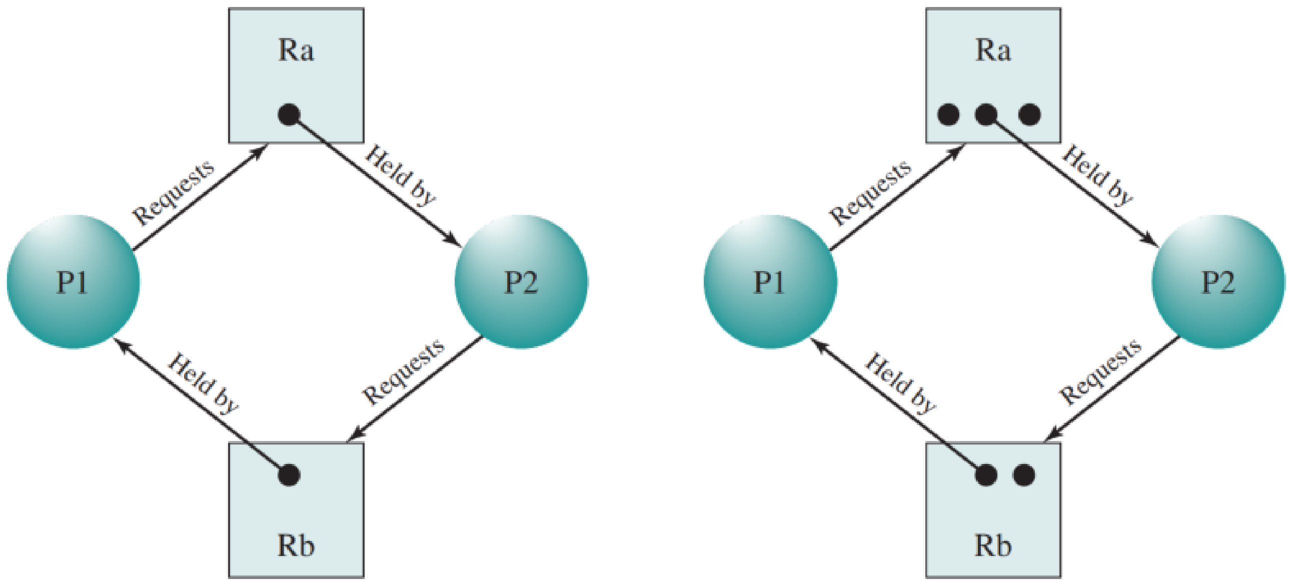
- **Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:**

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

- **Deadlock occurs if the Receive is blocking**

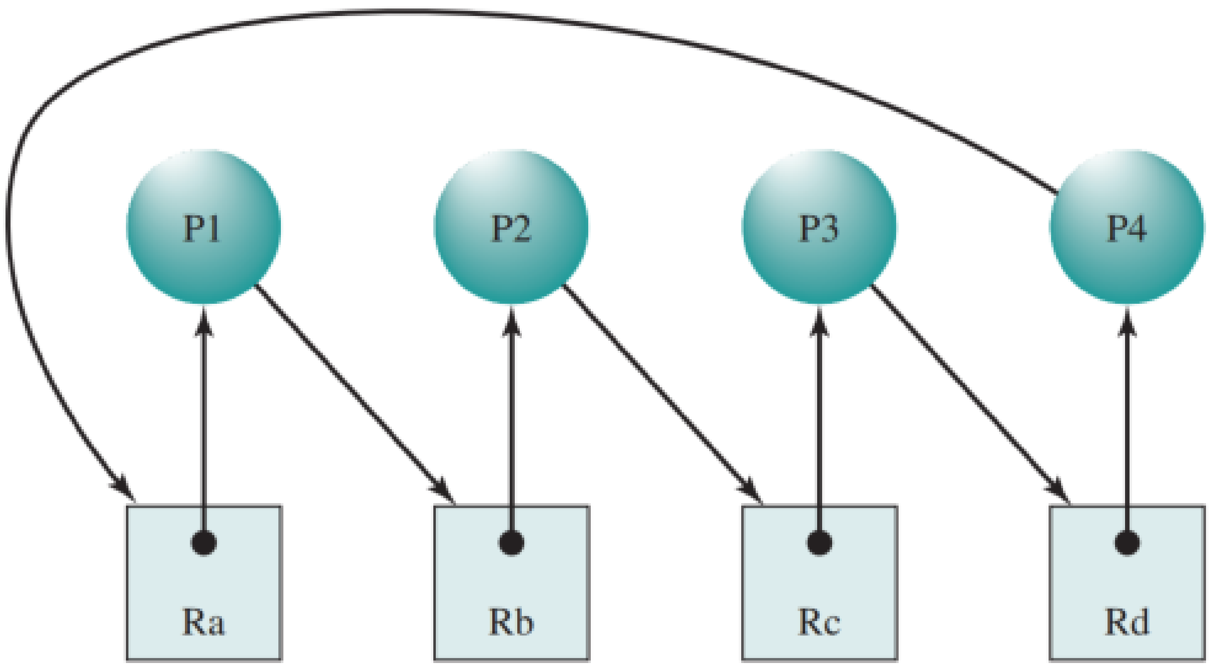
Resource Allocation Graphs

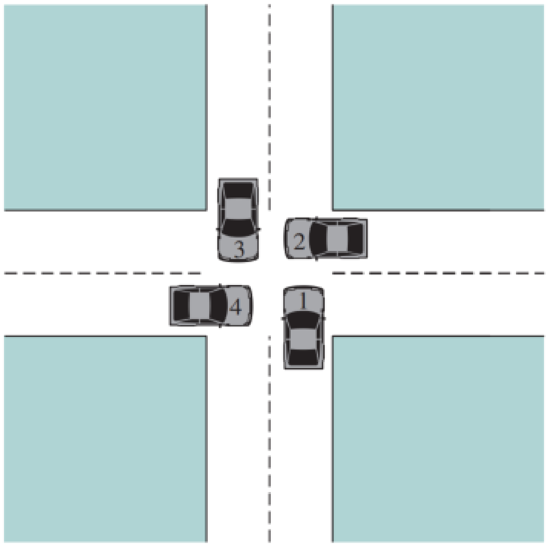
(a) Resource is requested (b) Resource is held



(c) Circular wait (d) No deadlock

Resource Allocation Graphs





Conditions for Deadlock

Policy decisions

Mutual exclusion

- Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

Hold and wait

- A process may hold allocated resources while awaiting assignment of other resources.

No preemption

- No resource can be forcibly removed from a process holding it.

Circular wait

- A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

Might occur

To summarize

- **Possibility of Deadlock**
  - Mutual exclusion
  - No preemption
  - Hold and wait
- **Existence of Deadlock**
  - Mutual exclusion
  - No preemption
  - Hold and wait
  - **Circular wait**

Dealing with Deadlock

Prevent Deadlock(死锁预防)

- by adopting a policy that eliminates one of the conditions

Avoid Deadlock(死锁避免)

- make the appropriate dynamic choices based on the current state of resource allocation

Detect Deadlock(死锁检测)

- attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention Strategy

- **Design a system in such a way that the possibility of deadlock is excluded**
- **Two main methods**
  - Indirect
    - prevent the occurrence of **one of the three** necessary conditions
  - Direct
    - prevent the occurrence of a **circular wait**

Deadlock Condition Prevention

- **Mutual Exclusion**
  - if access to a resource requires mutual exclusion then it must be supported by the OS
- **Hold and Wait**
  - require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
- **No Preemption**
  - if a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release its resources
    - different priority
- **Circular Wait**
  - define a linear ordering of resource types

Deadlock Avoidance

- **A decision is made dynamically**
  - whether the current resource allocation request will, if granted, potentially lead to a deadlock
- **Requires knowledge of future process requests**

Different from deadlock prevention
allows the three necessary conditions
but makes judicious(明智的) choices to assure that the **deadlock point is never reached**

Two Approaches to Deadlock Avoidance

Deadlock Avoidance

Process Initiation Denial:
Do not start a process if its demands might lead to deadlock

Resource Allocation Denial:
Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Resource Allocation Denial

- **Referred to as the banker's algorithm**
- **State of the system**
  - reflects the current allocation of resources to processes
- **Safe state**
  - is one in which there is **at least one sequence** of resource allocations to processes that **does not result in a deadlock**
- **Unsafe state**
  - is a state that is not safe

死锁避免之银行家算法

一个银行家：目前手里只有1亿，但是已经贷出很多钱
开发商A：已贷款15亿，资金紧张还需3亿。
开发商B：已贷款5亿，还需贷款1亿，运转良好能收回。
开发商C：已贷款2亿，欲贷款18亿
会不会出现楼盘烂尾?

开发商B还钱，再借给A，则可以继续借给C

银行家当前手里现金（Available）；
银行家可以利用的资金，即手里现金加上能收回的共有多少（work）；
各个开发商已贷款——已分配的资金（Allocation）；
各个开发商还需要贷款（Claim- Allocation）
钱就是资源，开发商就是进程，银行家的决策就是调度

A system of n processes and m resources

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

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j — All resources are either available or allocated.

2. $C_{ij} \leq R_j$, for all i, j — No process can claim more than the total amount of resources in the system.

3. $A_{ij} \leq C_{ij}$, for all i, j — No process is allocated more resources of any type than the process originally claimed to need.

Determination of a Safe State

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

(a) Initial state

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
6	2	3

(b) P2 runs to completion

Determination of a Safe State

	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

	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 a 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	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

	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

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

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>;
}
```

Resource allocation 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);
}
```

Test for safety algorithm (banker's algorithm)

Advantages and Restrictions

Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

Restrictions

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Deadlock Strategies

- **Deadlock prevention**
  - very **conservative**
  - **limit access** to resources by imposing restrictions on processes
- **Deadlock detection**
  - do the opposite
  - resource requests are granted **whenever possible**

Deadline Detection Algorithms

- A check for deadlock can be made as **frequently** as each **resource request** or, less frequently, depending on **how likely it is for a deadlock to occur**
- **Advantages:**
  - it leads to **early detection**
  - the algorithm is **relatively simple**
- **Disadvantage**
  - **frequent** checks
  - **consume** considerable processor time

Deadline Detection Algorithms

Example for Deadlock Detection

	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

The algorithm concludes with P1 and P2 unmarked, indicating these processes are deadlocked.

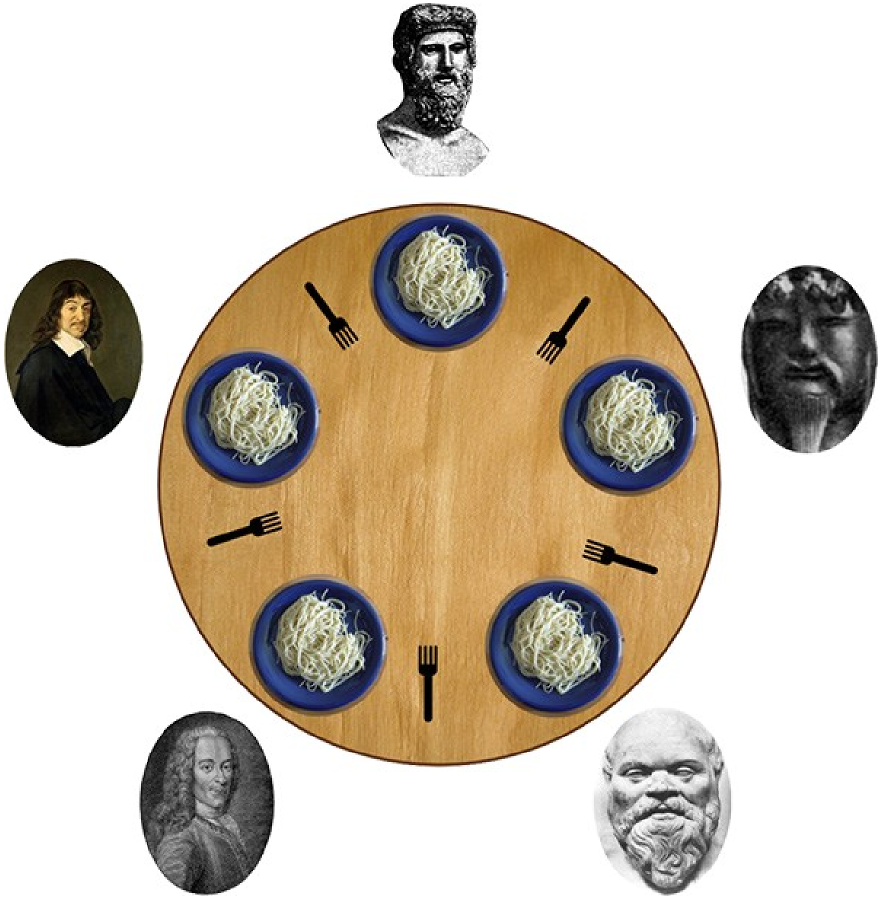
Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; under commits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • 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 online handling	• Inherent preemption losses

Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems

IMPORTANT

Dining Philosophers Problem

- **Mutual exclusion**
  - **No two philosophers** can use **the same fork** at the **same time**
- **Avoid deadlock and starvation**
  - No philosopher must starve to death



Using Semaphores

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


A First Solution to the Dining Philosophers Problem

Using Semaphores
A Second Solution

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

Using A 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 */
   }
}
```

```
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:
   }
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]);
}
```

Summary

- **Deadlock**
  - the blocking of a set of processes that either compete for system resources or communicate with each other
  - blockage is permanent unless OS takes action
  - may involve reusable or consumable resources
    - Consumable = destroyed
    - Reusable = not depleted or destroyed by use
- **Dealing with deadlock:**
  - **prevention**
    - guarantees that deadlock will not occur
  - **avoidance**
    - analyzes each new resource request
  - **detection**
    - OS checks for deadlock and takes action