# Concurrency: Deadlock

Chapter 6

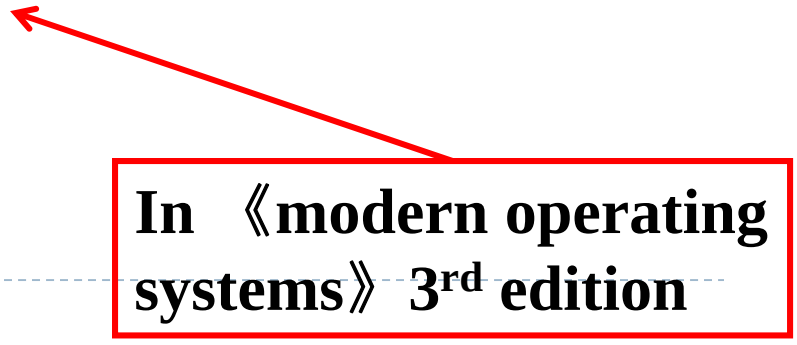
# Organization

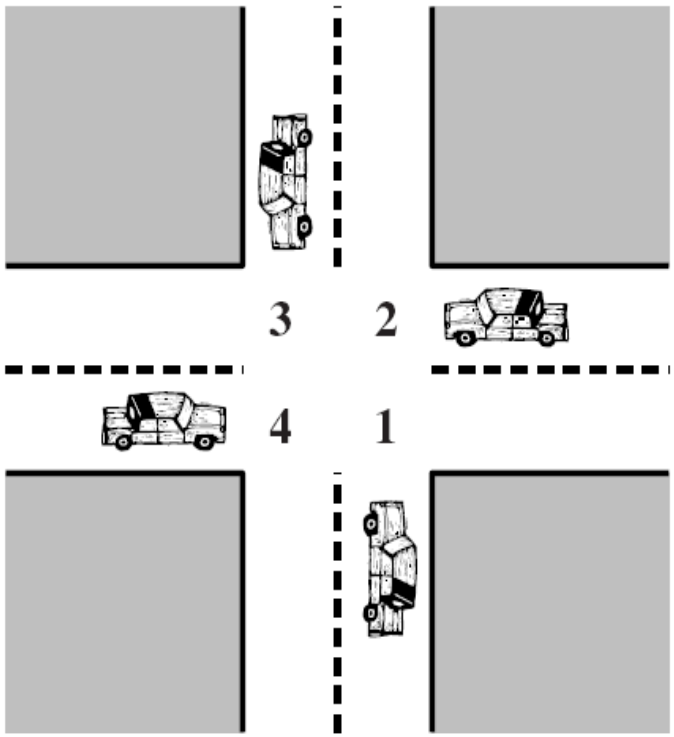
- **Principles of Deadlock**
- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection**

# Principles of Deadlock

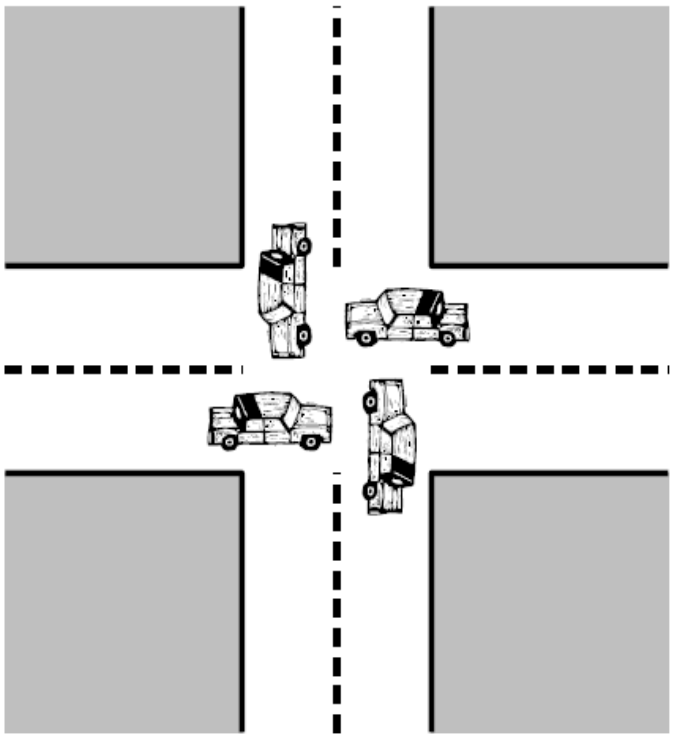
# Deadlock: The Definition

- **Definition**
  - ***Permanent blocking of a set of processes*** **that either compete for system resources or communicate with each other**
  - ***A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.***
- **No efficient solution!!**

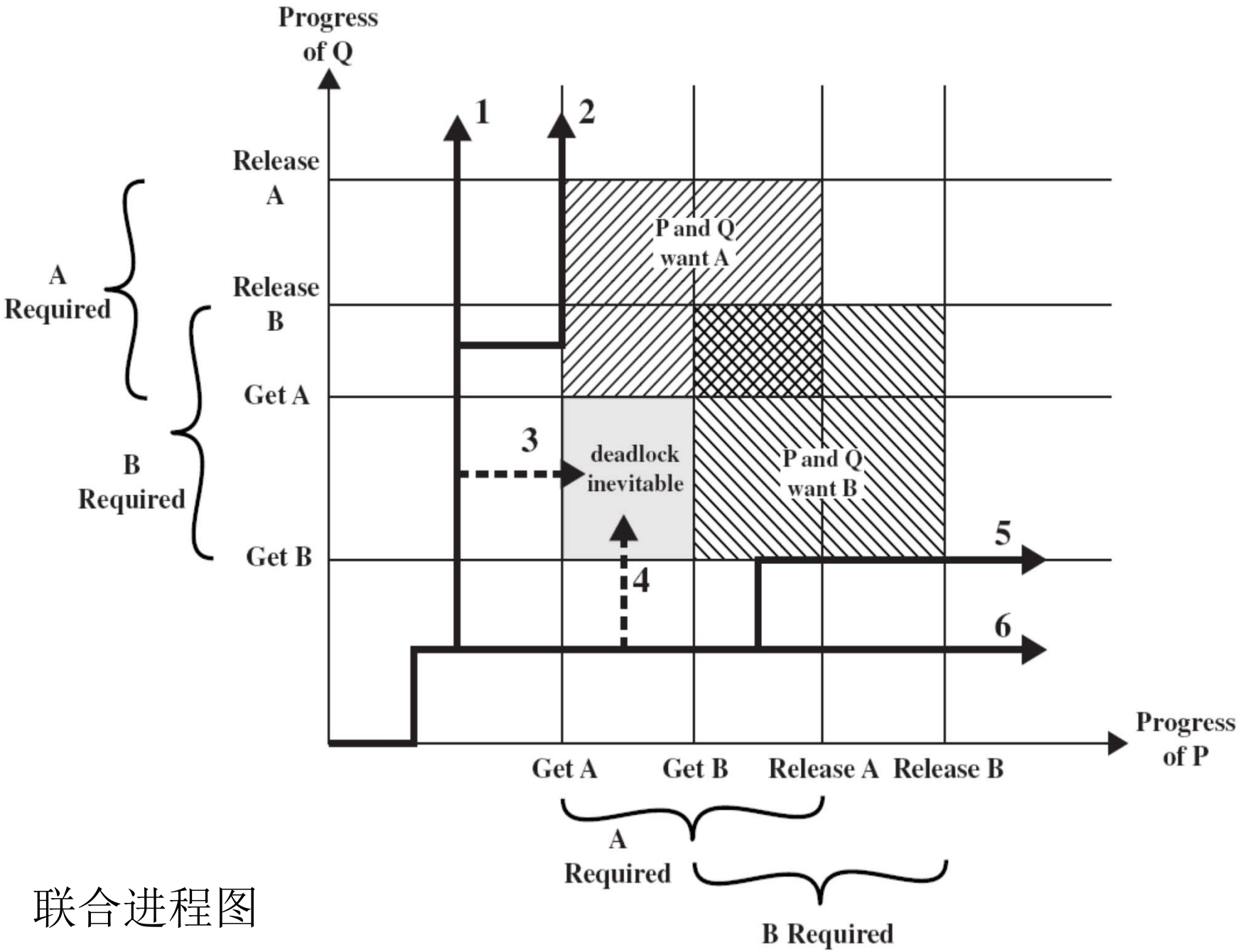




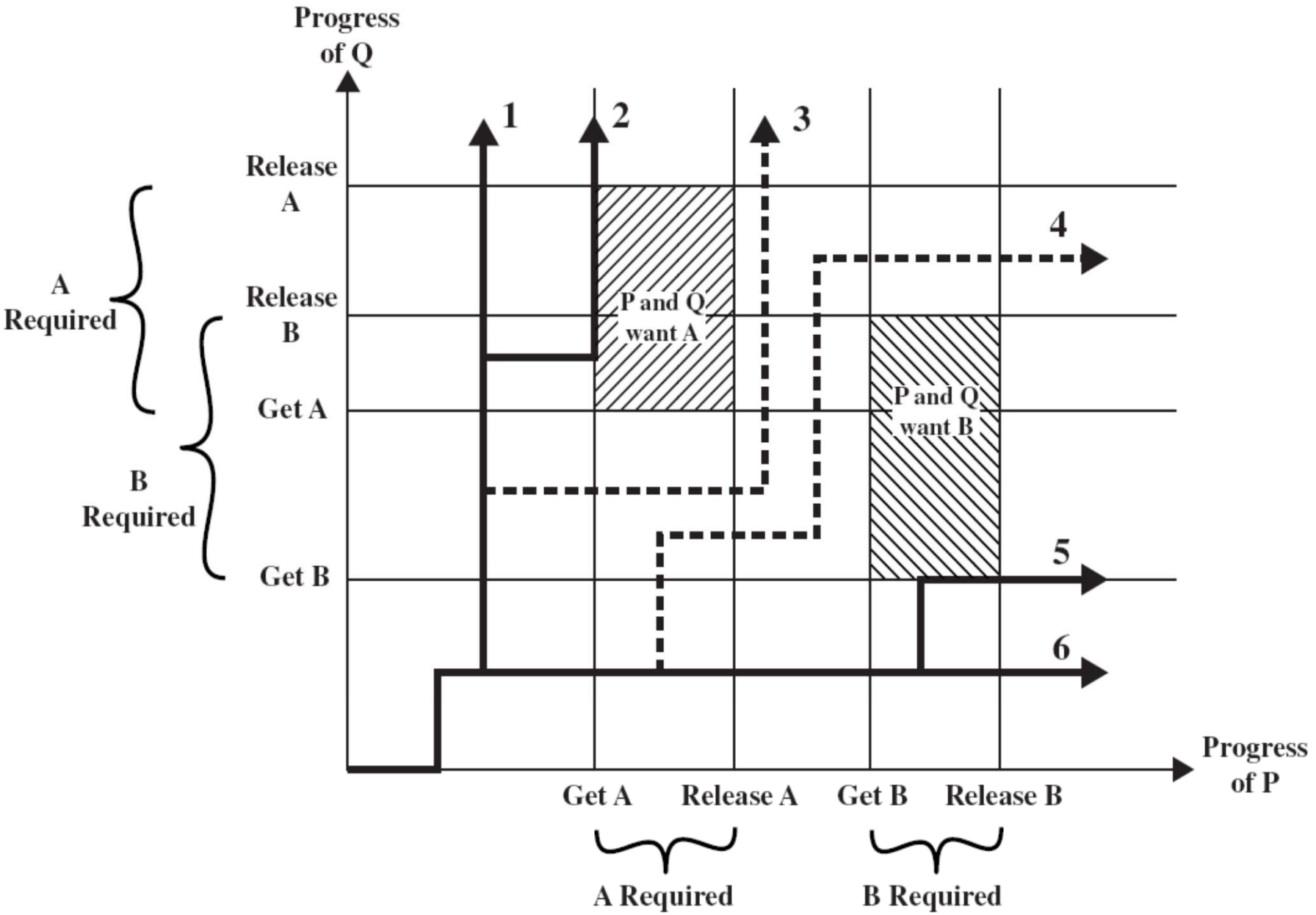

(a) Deadlock possible



(b) Deadlock


Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
deadlock inevitable
P and Q want B
Progress of P
Get A
Get B
Release A
Release B
A Required
B Required

联合进程图


Progress
of Q
Release
A
Release
B
Get A
Get B
A
Required
B
Required
1
2
3
4
5
6
P and Q
want A
P and Q
want B
Progress
of P
Get A
Release A
Get B
Release B
A Required
B Required

# Two Categories of Resources Resuable VS Consumable

- **Reusable Resources (可重用资源)**
  - *A reusable resource is one that can be safely used by only one process at a time and is not depleted (销毁) by that use*
- **Consumable Resources (可消费资源)**
  - *A consumable resource is one that can be created (produced) and destroyed (consumed)*

# Reusable Resources (1)

- ***Processes obtain resources that they later release for reuse by other processes***
- ***Examples of reusable resources***
  - ***Processors, I/O channels, main and secondary memory, devices, data structures(files, databases, and semaphores)***

# Reusable Resources (2)
# Resource Acquisition 资源获得

- ***Normally, a process may utilize a resource in only the following sequence:***
  - ***Request (or Get): If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.***
  - ***Use: The process can operate on the resource.***
  - ***Release: The process releases the resource***

system calls: can be accomplished by semaphores

# Reusable Resources (3)
# Example of Deadlock

| Process P | Process Q |
| --- | --- |
| Request(D) | Request(T) |
| Request(T) | Request(D) |
| Perform function | Perform function |
| Release(D) | Release(T) |
| Release(T) | Release(D) |

# Reusable Resources (4)
# Another Example of Deadlock

- **Space is available for allocation of 200K bytes, and the following sequence of events occur**

| P1 | P2 |
|---|---|
| . . . | . . . |
| Request 80K bytes; | Request 70K bytes; |
| . . . | . . . |
| Request 60K bytes; | Request 80K bytes; |

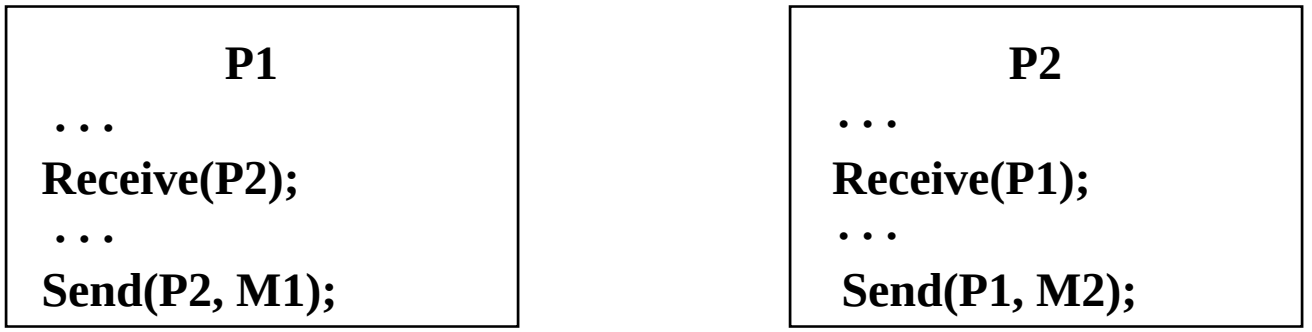
- **Deadlock occurs if both processes progress to their second request**

# Consumable Resources

- *Created (produced) and destroyed (consumed) by a process*
- *Examples of consumable resources*
  - *Interrupts, signals, messages, and information in I/O buffers*

# Consumable Resources Example of Deadlock

- Deadlock occurs if receive is blocking



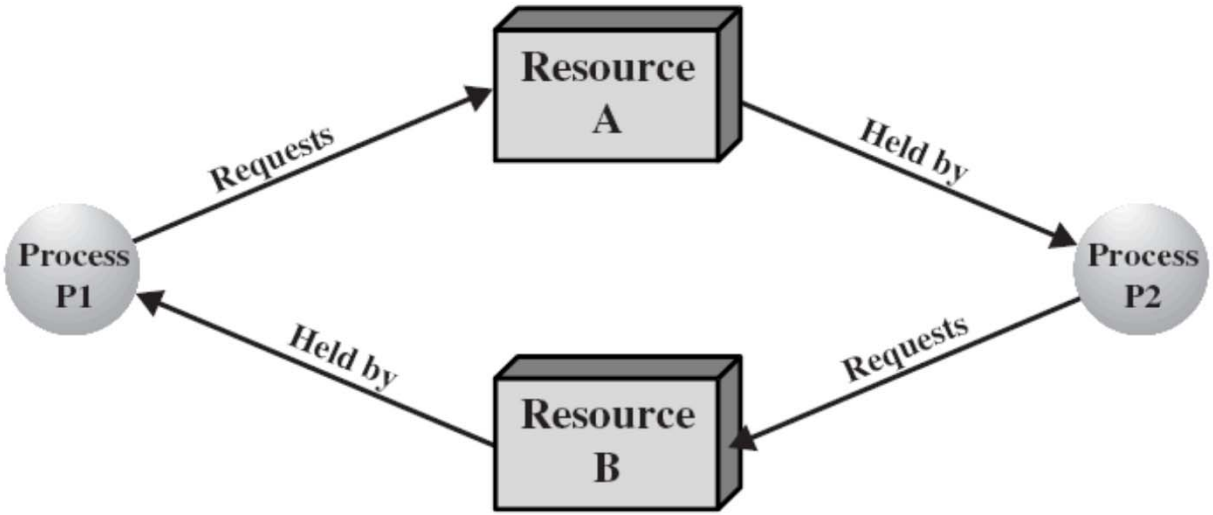

# Necessary Conditions for Deadlock (1)

- **Mutual Exclusion 互斥 (Resource)**
  - That is, only one process may use the resource at a time.
- **Hold-and-wait 占有且等待 (Process)**
  - A process may hold allocated resources while awaiting assignment of others

# Necessary Conditions for Deadlock (2)

- **No preemption 非抢占 (Holding relationship)**
  - No resource can be forcibly removed from a process holding it. In other words, a resource can be released only voluntarily by the process holding it.
- **Circular wait 循环等待**
  - There is a closed chain of processes such that each process holds at least one resource needed by the next process in the chain.

# Necessary Conditions for Deadlock (3)



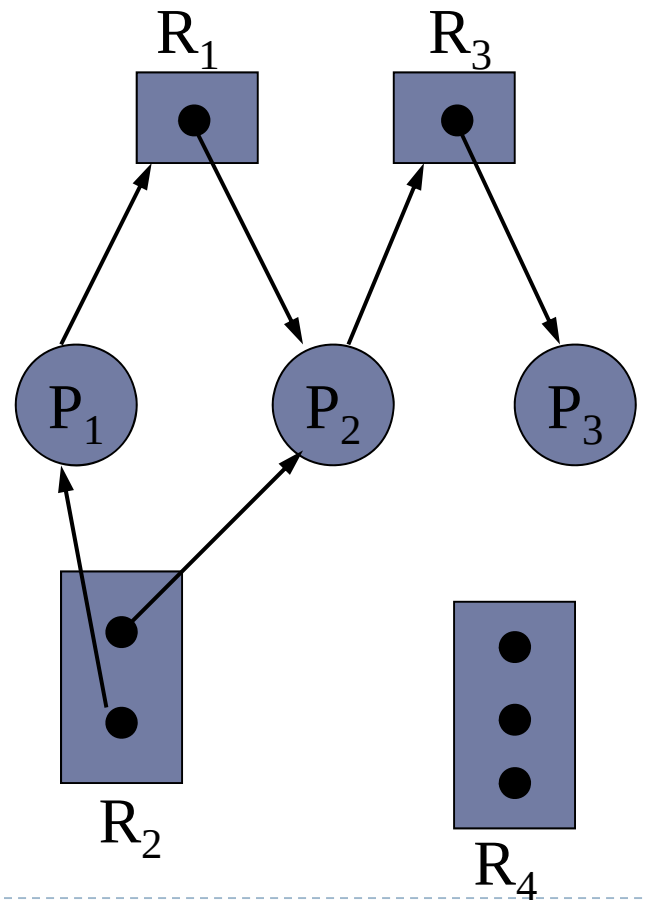

***These four conditions are not completely independent. For example, the circular-wait condition implies the hold-and-wait condition***

# Resource Allocation Graph (1)
# 资源分配图

- **A resource allocation graph consists of**
  - **A set of vertices *V*:**
    - **The set of vertices *V* is partitioned into two subsets $P=\{P_1, P_2, \ldots, P_n\}$ and $R=\{R_1, R_2, \ldots, R_m\}$**
  - **A set of edges *E*.**
    - **A directed edge from process $P_i$ to resource type $R_j$ is called a request edge ($P_i \rightarrow R_j$)**
    - **A directed edge from resource type $R_j$ to process $P_i$ is called an assignment edge ($R_j \rightarrow P_i$)**

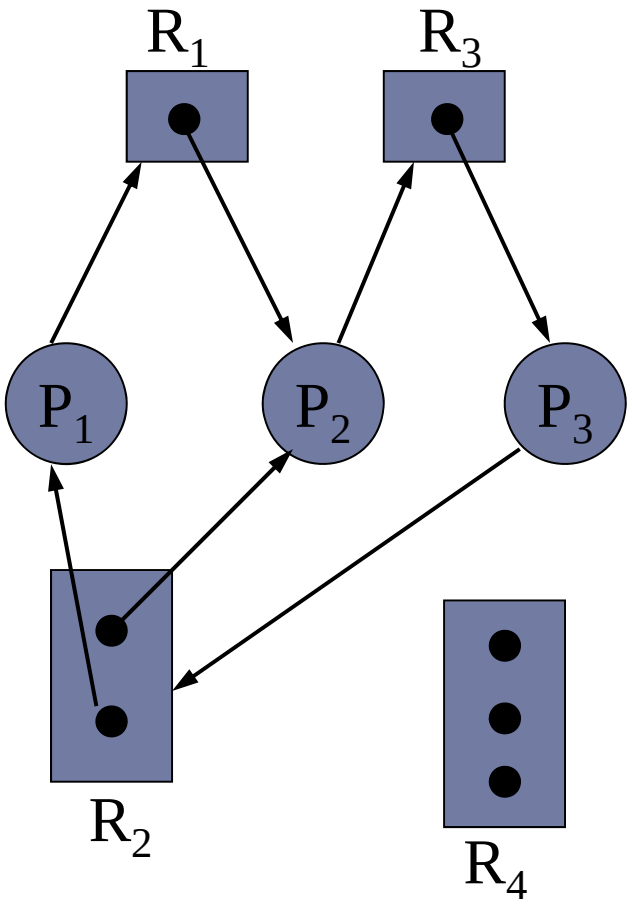
# Resource Allocation Graph (2)

- Each process $P_i$ is represented as a circle
- Each resource type $R_j$ as a square
  - Resource type $R_j$ may have multiple instances. Each instance is represented as a dot within the square

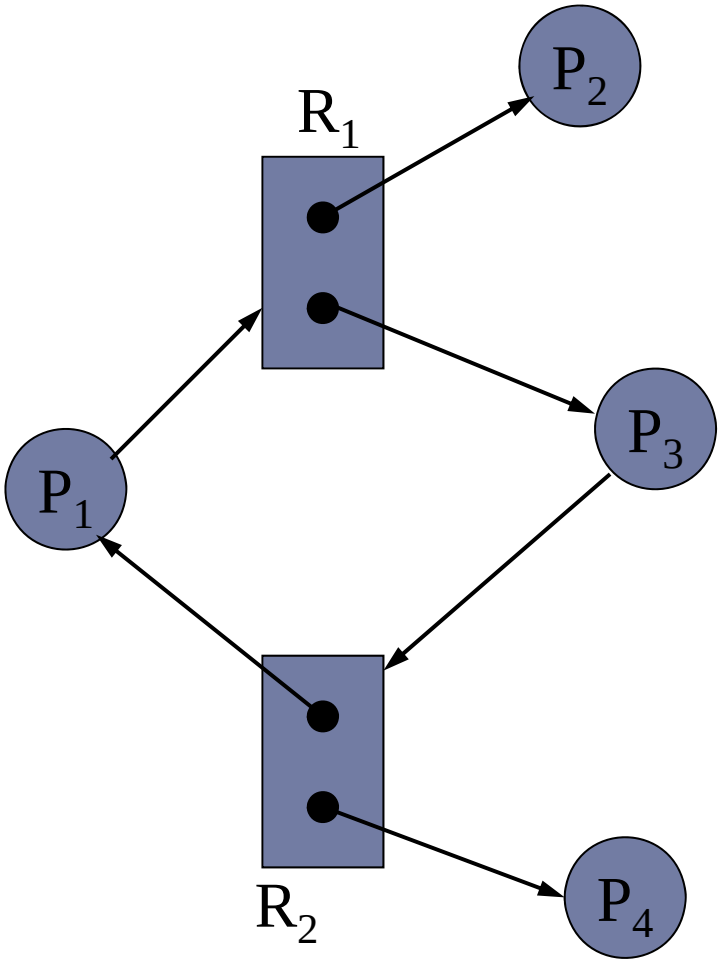


# Resource Allocation Graph (3)

- *If the graph contains no cycles, then no process in the system is deadlocked*
- *If the graph contains a cycle, then a deadlock may exist*
  - *If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred*
  - *If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred*




***Resource allocation graph with a deadlock***




***Resource allocation graph with a cycle but no deadlock***

# Four Strategies for Dealing with Deadlocks

- *The Ostrich algorithm: Just ignore the problem*
  *→Adopted by most operating systems*
- *Detection and recovery: Let deadlock occur, detect them, and take action*
- *Dynamic avoidance: by carefully resource allocation*
- *Prevention: by structurally negating one of the four required conditions*

# Deadlock Prevention

# Deadlock Prevention

- **This strategy is to design a system such that the possibility of deadlock is excluded**
- **Prevention algorithms prevent deadlocks by *restraining how requests can be made***
  - **The restraints ensure that at least one of the four necessary conditions can not hold**

# Deadlock Prevention Techniques (1)

- **Attacking "Mutual Exclusion"**
  - In general, mutual exclusion condition must hold for non-sharable resources
  - Mutual exclusion must be supported by OS if access to a resource requires mutual exclusion
  - In general, we can not prevent deadlocks by denying the mutual exclusion condition: Some resources are intrinsically nonsharable
- **Read-only files are a good example of a sharable resources**
  - A process never needs to wait for a sharable resource

# Deadlock Prevention Techniques (2)

- **Attacking "Hold and Wait"**
  - By requiring that a process *request all of its requested resources at one time* and blocking the process until all requests can be granted simultaneously
  - Inefficient in two ways:
    - A process may wait a long time for all of its resources
    - Resources allocated may remain unused for a long period (but other processes can not use it)
  - A process may not know in advance all of the resources required

# Deadlock Prevention Techniques (3)

- **Attacking "No Preemption"**
  - If a process holding certain resources is denied a further request, that process must release its original resources, and if necessary, request them again together with the additional resource (主动放弃)
  - If a process requests a resource held by another process, the OS may preempt the second process and request it to release its resources (被抢占)
- **Limitation: This approach is practical only for resources *whose state can be easily saved and restored later* (e.g. processors)**

# Deadlock Prevention Techniques (4)

- **Attacking "Circular Wait"**
  - Defining a linear ordering of all resource types
  - If a process has been allocated resource of type *R*, then it may request only those resources of types following *R* in the ordering
- **May be inefficient**

# Deadlock Avoidance

# Deadlock Avoidance

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

# Two Approaches to Deadlock Avoidance

- **Process Initiation Denial** (进程启动拒绝)
  - *Do not start a process if its demands might lead to deadlock*
- **Resource Allocation Denial** (资源分配拒绝)
  - *Do not grant an incremental resource request to a process if this allocation might lead to deadlock*

# Process Initiation Denial Problem Description

Resource = $(R_1, R_2, \ldots, R_m)$ → **total amount of each resource**

Available = $(V_1, V_2, \ldots, V_m)$ → **total amount of each resource not allocated to a process**

$$Claim = \begin{pmatrix} C_{11} & C_{12} & \Lambda & C_{1m} \\ C_{21} & C_{22} & \Lambda & C_{2m} \\ \mathrm{M} & \mathrm{M} & \mathrm{M} & \mathrm{M} \\ C_{n1} & C_{n2} & \Lambda & C_{nm} \end{pmatrix}$$

**requirement of each process for each resource**

$$Allocation = \begin{pmatrix} A_{11} & A_{12} & \Lambda & A_{1m} \\ A_{21} & A_{22} & \Lambda & A_{2m} \\ \mathrm{M} & \mathrm{M} & \mathrm{M} & \mathrm{M} \\ A_{n1} & A_{n2} & \Lambda & A_{nm} \end{pmatrix}$$

**current allocation**

# Process Initiation Denial

## Relationship Among These Matrices and Vectors

$$R_i = V_i + \sum_{k=1}^{n} A_{ki}$$

**for all *i*: All resources are either available or allocated**

$$C_{ki} \leq R_i$$

**for all *k*, *i*: No process can claim more than the total amount of resources in the system**

$$A_{ki} \leq C_{ki}$$

**for all *k*, *i*: No process is allocated more resources of any type than the process originally claimed to need.**

# Process Initiation Denial Policy

- **Refuse to start a new process if its resource requirements might lead to deadlock.**
- **A new process $P_{n+1}$ can be started only if:**

$$R_i \geq C_{(n+1)i} + \sum_{k=1}^{n} C_{ki} \qquad for\ all\ i$$

***It assumes the worst: all processes will make their maximum claims together***

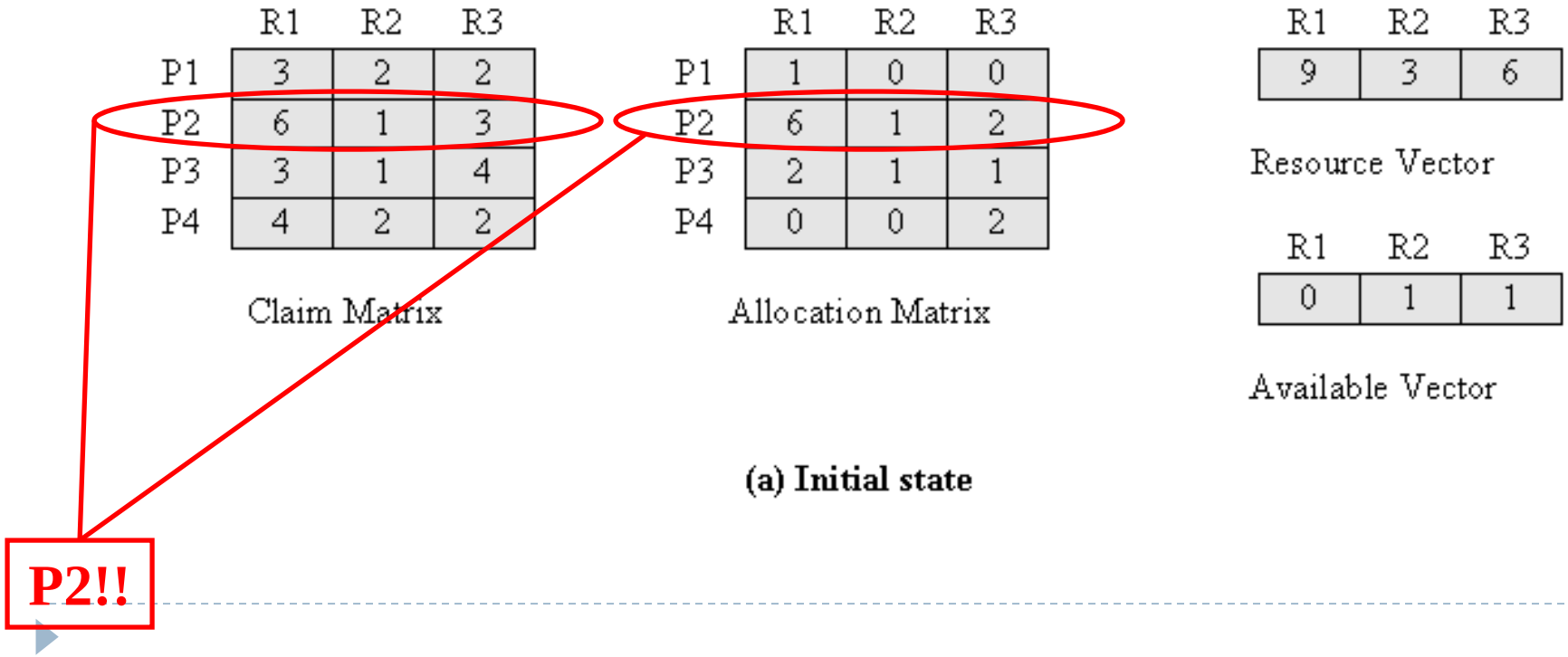
***This strategy is hardly optimal!!***

# Resource Allocation Denial Description

- **This strategy is also referred to as the banker's algorithm (银行家算法)**
- **State of the system is simply the current allocation of resources to process**
  - **The state consists of two vectors (Resource and Available), and two matrices (Claim and Allocation)**
- **Safe or Unsafe?**
  - ***Safe state*** is where there is at least one sequence that does not result in deadlock
  - ***Unsafe state*** is a state that is not safe

# (1)
# Initial State

***Which process can be run to completion with the resources available?***



# (2)
# 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

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

Allocation Matrix

| R1 | R2 | R3 |
|---|---|---|
| 6 | 2 | 3 |

Available Vector

**(b) P2 runs to completion**

# (3)
# P1 Runs to Completion

***After choosing P1 to be run to completion, the state is as follows:***

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

Claim Matrix

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

Allocation Matrix

| R1 | R2 | R3 |
|---|---|---|
| 7 | 2 | 3 |

Available Vector

**(c) P1 runs to completion**

# (4)
# P3 Runs to Completion

***Next, we can complete P3***

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

Claim Matrix

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

Allocation Matrix

| R1 | R2 | R3 |
|---|---|---|
| 9 | 3 | 4 |

Available Vector

**(d) P3 runs to completion**

***Finally, we can complete P4. All the 4 processes can be run to completion. Thus, the initial state is safe.***

# Resource Allocation Denial Solution

- One possible deadlock avoidance strategy is to ensure that the system of processes and resources is always in a safe state
- When a process makes a request for a set of resources,
  - Assume that the request is granted
  - Update the system state accordingly
  - And then determine whether the result is a safe state
    - If safe, grant the request; otherwise, block the process until it is safe to grant the request.

# Determination of an Unsafe State Initial State

***Consider the following safe initial state***

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

Claim Matrix

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

Allocation Matrix

| R1 | R2 | R3 |
|---|---|---|
| 9 | 3 | 6 |

Resource Vector

| R1 | R2 | R3 |
|---|---|---|
| 1 | 1 | 2 |

Available Vector

***Suppose that P2 makes a request for one additional unit of R1 and one additional unit of R3. Should this request be granted?***

- ***How about P1 making the same request instead?***

# 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

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

Allocation Matrix

| R1 | R2 | R3 |
|---|---|---|
| 0 | 1 | 1 |

Available Vector

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

# Deadlock Avoidance
# Advantages and Restrictions

- **Advantages**
  - It is not necessary to preempt or rollback processes (as in deadlock detection)
  - It is less restrictive than deadlock prevention
- **Restrictions on its use:**
  - 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

# Deadlock Detection and Recovery

# Deadlock Detection and Recovery

- **If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock may occur.**
- **System must provide:**
  - **An algorithm that *determines whether a deadlock has occurred***
  - **An algorithm to *recover from the deadlock***

# Problem Description

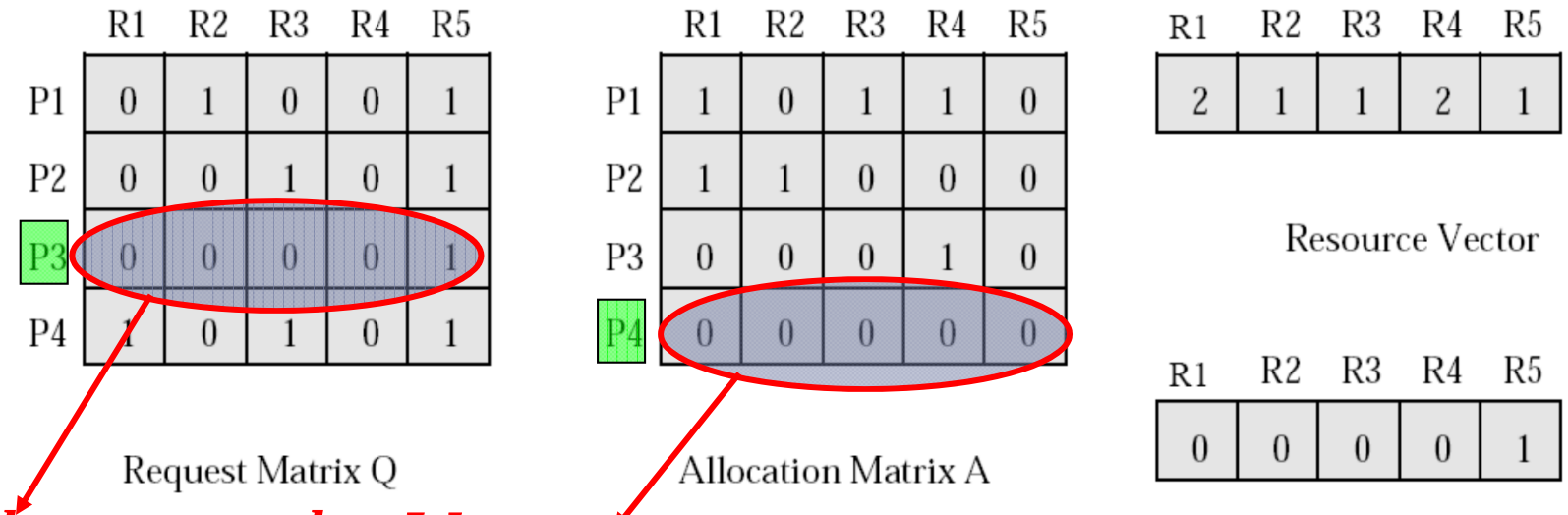
- **Allocation matrix and Available vector**
  - As described in deadlock avoidance
- **Request matrix Q**
  - $q_{ij}$ represents the amount of resources of type $j$ requested by process $i$.

**Note: The Claim matrix is not needed in deadlock detection, while it is a must in deadlock avoidance algorithms.**

# Deadlock Detection Algorithm

- **Step 1:** Mark each process that has a row in the Allocation Matrix of all zeros
- **Step 2:** Initialize a temporary vector W to equal the Available vector
- **Step 3:** Find an index *i* such that process *i* is currently unmarked and the *i*th row of **Q** is less than or equal to **W**. If no such row is found, terminate the algorithm
- **Step 4:** If such a row is found, mark process *i* and add the corresponding row of the allocation matrix to **W**. Return to Step 3.

# Example for Deadlock Detection




***P3 is less or equal to W***

***P4 has the row of all zero***

***W*** 0 0 0 0 1 ← ***Initialize W to equal the Available Vector***

***W*** 0 0 0 1 1 ← ***Add the row of P3 in Allocation matrix to W***

***No other unmarked process has a row in Q that is less than or equal to W!!!!!***

***Deadlock has occurred!!***

# When to Detect Deadlocks?

- ***There are several candidate strategies:***
  - ***To check every time a resource request is made (certain to detect deadlocks as early as possible, but potentially expensive in terms of CPU time)***
  - ***To check every k minutes***
  - ***To check only when the CPU utilization has dropped below some threshold (if enough processes are deadlocked, there will be few runnable processes)***

# Recovery from Deadlock

- **How to deal with it once a deadlock has been detected?**
  - To inform the operator that a deadlock has occurred, and let the operator to deal with it **manually**; or
  - To let the system recover from the deadlock **automatically**.
    - **Process termination: to abort one or more processes to break the circular wait**
    - **Resource preemption: to preempt some resources from one or more of the deadlocked process**

# Recovery Strategies once Deadlock Detected

- **Abort all deadlocked processes**
  - The simplest and the most common
- **Back up each deadlocked process to some previously defined checkpoint (检查点), and restart all process**
  - Rollback (回滚) and Restart (重启) mechanisms should be built into system
  - original deadlock may occur
- **Successively abort deadlocked processes until deadlock no longer exists**
- **Successively preempt resources until deadlock no longer exists**
  - A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

# Selection Criteria

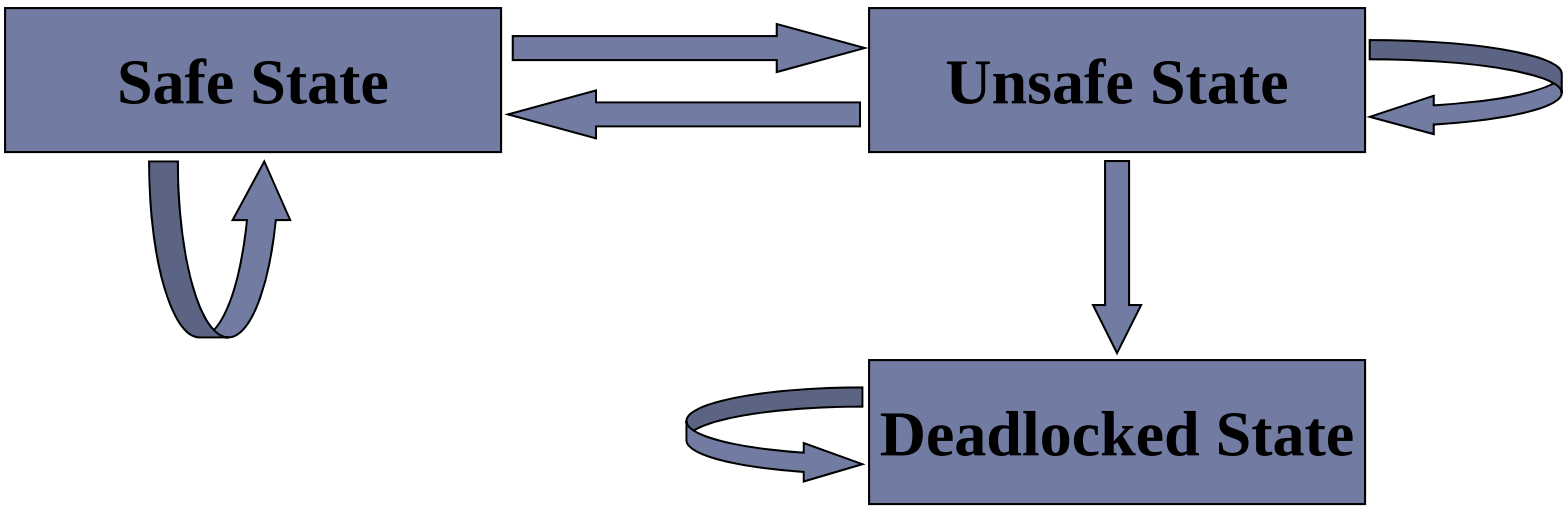
- **Which process to abort or resources from which process to be preempted?**
  - Least amount of processor time consumed so far
  - Least amount of output produced so far
  - Most estimated time remaining
  - Least total resources allocated so far
  - Lowest priority
  - Interactive or batch

# The Ostrich Algorithm

# The Ostrich Algorithm

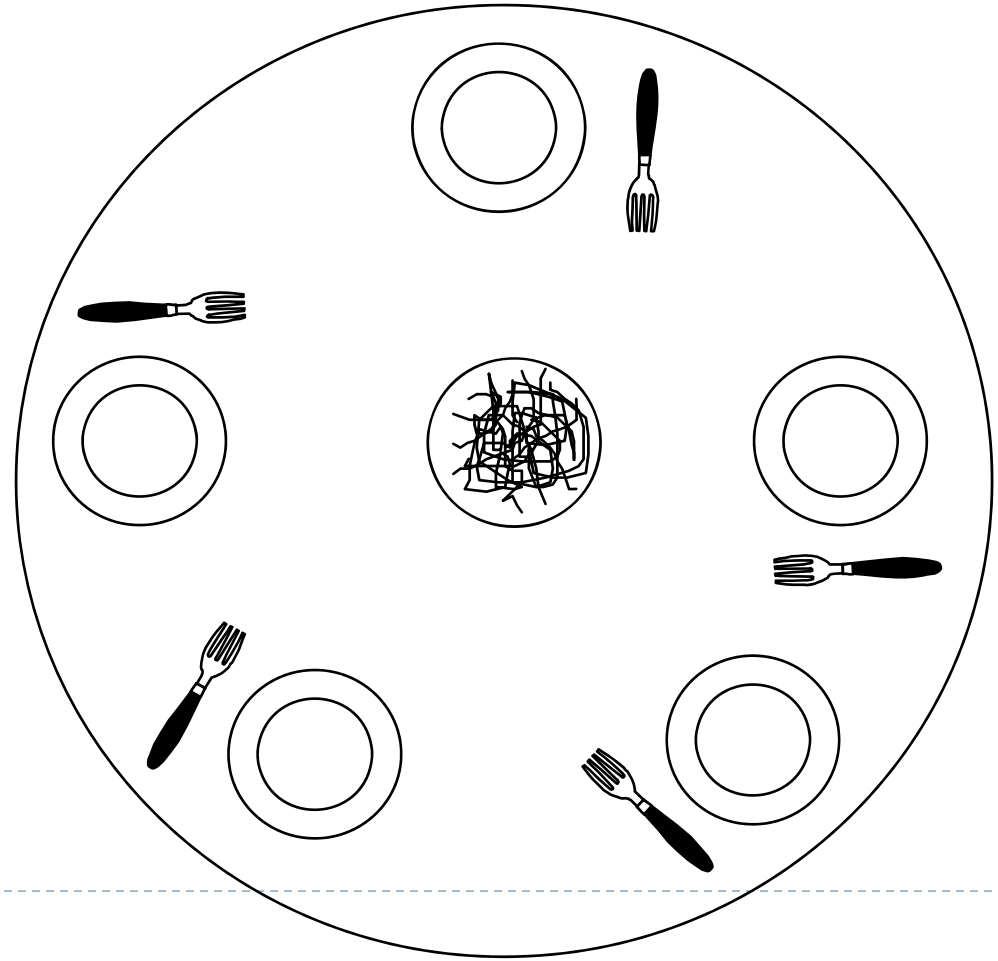
- *The basic idea: stick your head in the sand and pretend there is no problem at all.*
  - *It seems not to be a viable approach, but it is used in some operating systems (including UNIX)*
- *How often does a deadlock occur? How serious is a deadlock?*

# Safe State, Unsafe State, and Deadlocked State




# Dining Philosophers Problem

# Dining Philosophers Problem



# A First Solution to the Dining Philosophers Problem

```
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));
}
```

**This is a failed solution!!**

**How to use deadlock prevention to solve it?**

# A Second Solution to the Dining Philosophers Problem

```
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);
}
```

*At most 4 philosophers are allowed to get into the dining room at a time*

*At least one philosopher will have access to two forks.*