Deadlocks — Chapter 7

2022年10-11月

薛哲

School of Computer Science (National Pilot Software Engineering School)

Deadlock Concepts

- Definition
  - a situation or system states in which a set of ***blocked*** processes (e.g. in the waiting state) each holding some resources and waiting to acquire a resource held by another process in the set, and these processes will never proceed
- E.g.1 There are two tape drivers **A** and **B** in the system
  - P_1 and P_2 , each hold one tape drive and each needs another one
  - semaphores *A* and *B* corresponding to two tape drivers respectively, initialized to 1

P_1	P_2
wait (A)	
	wait(B)
wait (B)	
	wait(A)

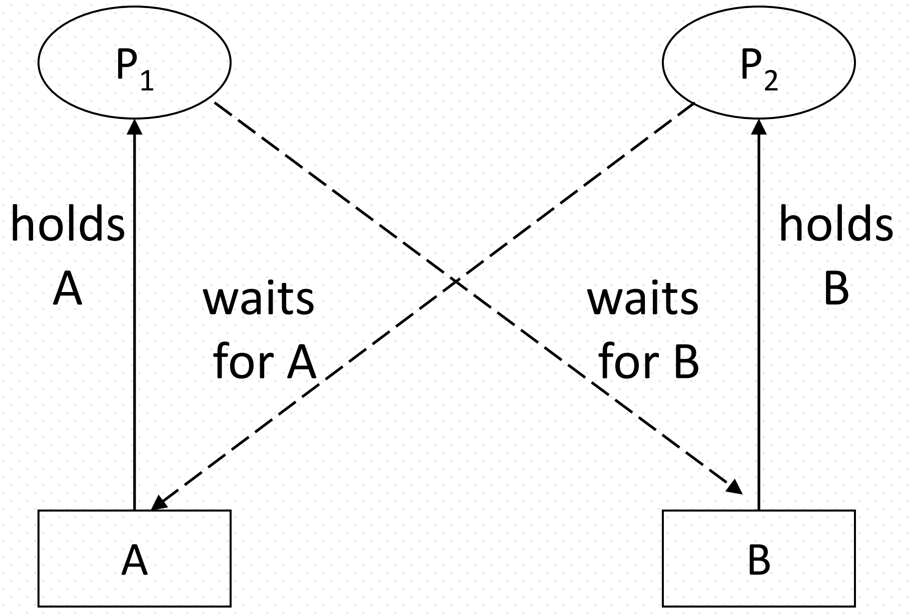


Fig.0.1 Deadlock Concept

Deadlock Examples

- E.g.2 Bridge Crossing

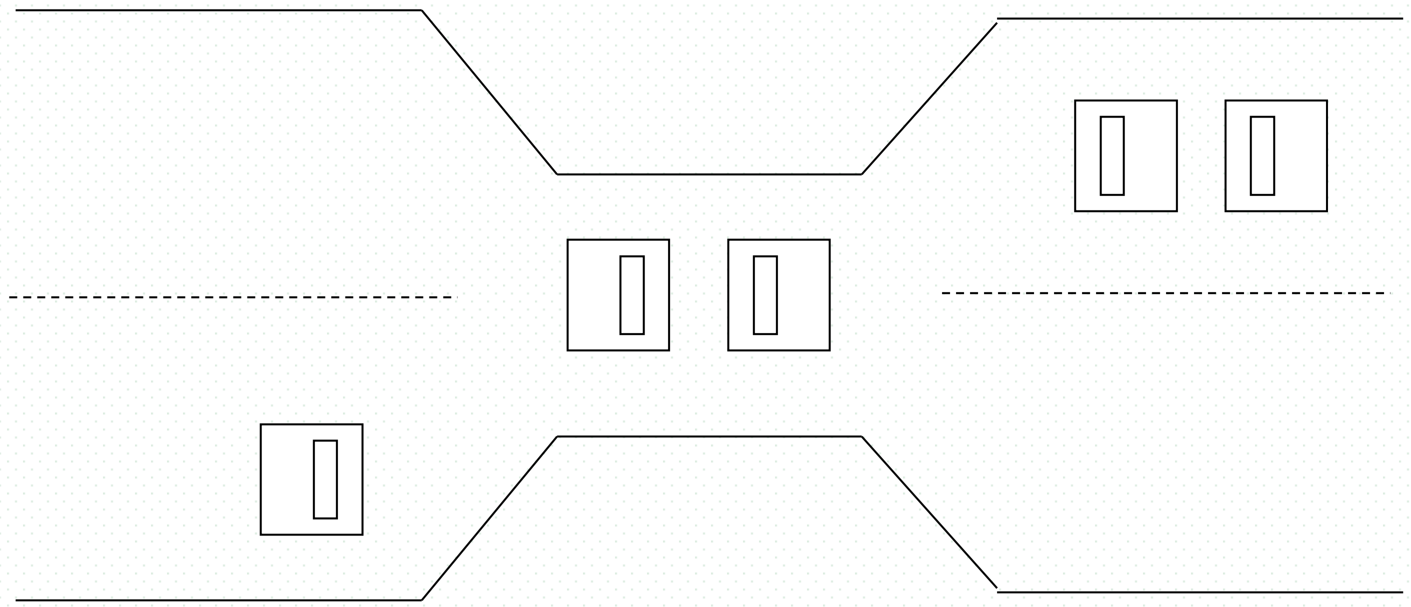


Fig. 7.0.2

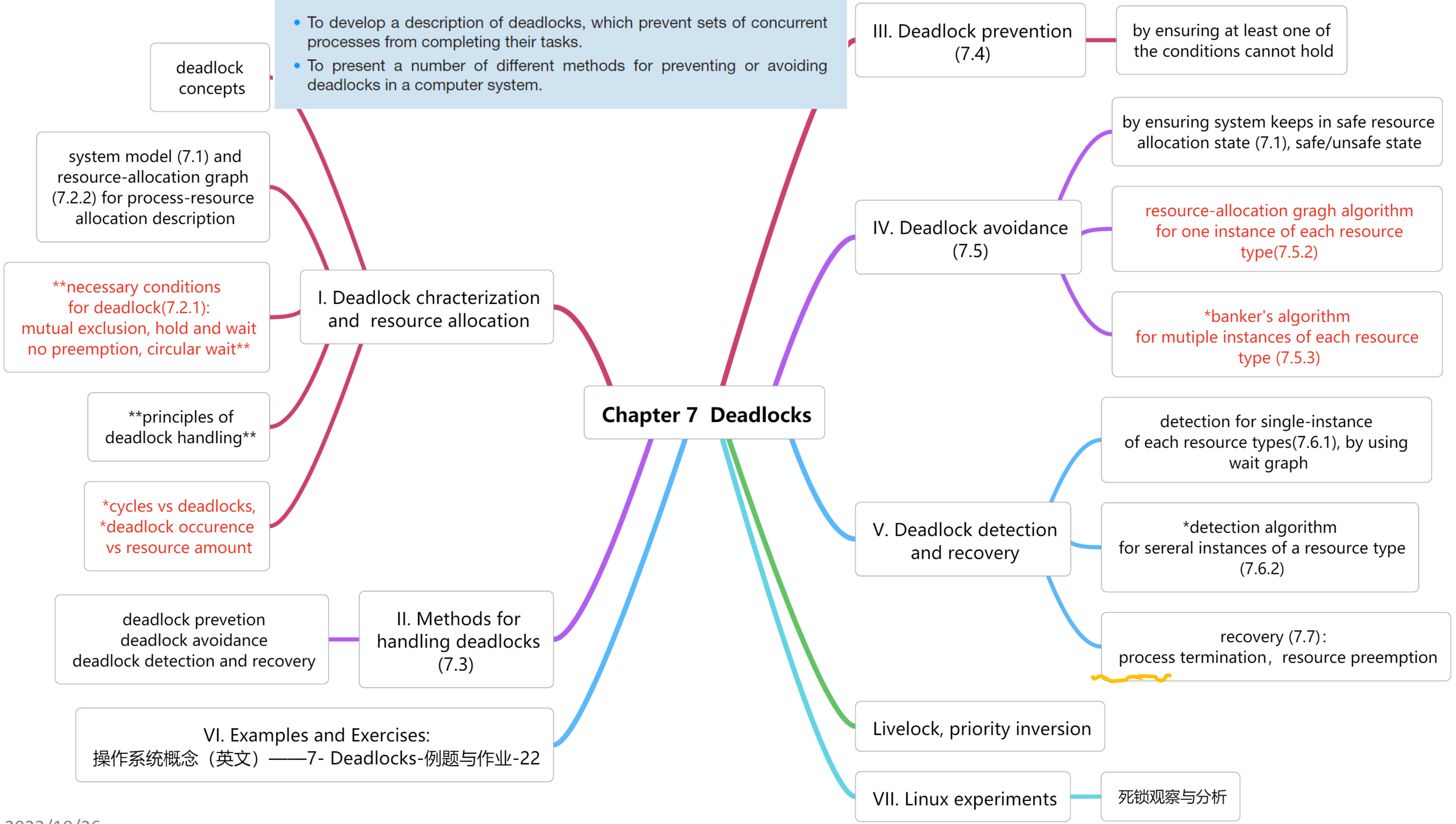
- traffic only in one direction.
- each section of a bridge can be viewed as a resource
- if a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- several cars may have to be backed up if a deadlock occurs
- starvation is possible.

Formal Descriptions of Deadlocks

- 死锁形式化描述
  - 并发进程$P_l, P_2, \ldots, P_n$， 竞争使用共享资源$R_l, R_2, \ldots, R_m$ $(n > 0, m > 0, n \geq m)$
  - 每个P_i $(1 \leq i \leq n)$ 拥有资源R_j $(1 \leq j \leq m)$，直到不再有剩余资源；同时，各P_i又在不释放R_j的前提下要求得到R_k $(k \neq j, l \leq k \leq m)$，造成资源的互相占有和互相等待
  - 在没有外力驱动的情况下，该组并发进程停止继续执行，陷入永久等待状态
- 死锁起因
  - 并发进程对共享资源的竞争
  - 提供的可用资源数目少于并发进程所要求的该类资源数目

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

- I. Deadlock chracterization and resource allocation
  - deadlock concepts
  - system model (7.1) and resource-allocation graph (7.2.2) for process-resource allocation description
  - **necessary conditions for deadlock(7.2.1): mutual exclusion, hold and wait no preemption, circular wait**
  - **principles of deadlock handling**
  - *cycles vs deadlocks, *deadlock occurence vs resource amount
- II. Methods for handling deadlocks (7.3)
  - deadlock prevetion
  - deadlock avoidance
  - deadlock detection and recovery
- III. Deadlock prevention (7.4)
  - by ensuring at least one of the conditions cannot hold
- IV. Deadlock avoidance (7.5)
  - by ensuring system keeps in safe resource allocation state (7.1), safe/unsafe state
  - resource-allocation gragh algorithm for one instance of each resource type(7.5.2)
  - *banker's algorithm for mutiple instances of each resource type (7.5.3)
- V. Deadlock detection and recovery
  - detection for single-instance of each resource types(7.6.1), by using wait graph
  - *detection algorithm for sereral instances of a resource type (7.6.2)
  - recovery (7.7)：process termination，resource preemption
- VI. Examples and Exercises: 操作系统概念（英文）——7- Deadlocks-例题与作业-22
- Livelock, priority inversion
- VII. Linux experiments
  - 死锁观察与分析

7.1 System Model

- System model
  - resource-allocation model, to describe
    - resources allocated to processes
    - processes requesting resources
- Finite number of resources in systems, resource types $R_1, R_2, . . ., R_m$
  - physical resources, *e.g., CPU cycles, memory space, I/O devices*
  - logical resources*, e.g., files, semaphores, monitors*
- Each resource type R_i has W_i instances, a number of competing processes request the instances of resource types, e.g. 16G main memory consists of a number of pages of 4K-in-size
- Each process utilizes a resource as follows:
  - **request** (as a system call, in queue of ***blocked*** processes waiting for the resource)
  - **use**
  - **release** (as a system call)

System Model

- If resources are controlled by **semaphores**, requesting and releasing resources are conducted by **wait** and **signal** operations**.**

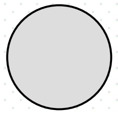
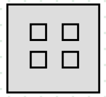
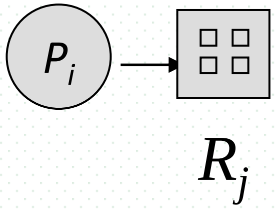
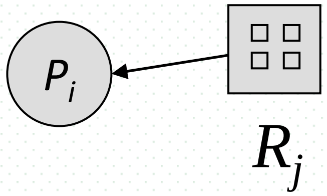
- Resources can be classified as
  - sharable resources, being used by processes concurrently

    — e.g. data items permitted to be read by several readers
  - non-sharable resources/critical resources, being used by processes exclusively

7.2.2 Resource-allocation Graph

- A system resource allocation graph *G*(*V*, *E*) is a directed graph and consists of a set of vertices *V* and a set of edges *E*
  - V is partitioned into two types
    - P = {P_1, P_2, …, P_n}, the set consisting of all the processes in the system
    - R = {R_1, R_2, …, R_m}, the set consisting of all resource types in the system.
  - E is partitioned into two types
    - request edge – directed edge $P_1 \rightarrow R_j$
    - assignment edge – directed edge $R_j \rightarrow P_i$

Resource-allocation Graph

- Representation of **resource-allocation graph** *G*
  - process 
  - resource type with 4 instances 
  - P_i requests instance of R_j 
  - P_i is holding an instance of R_j 

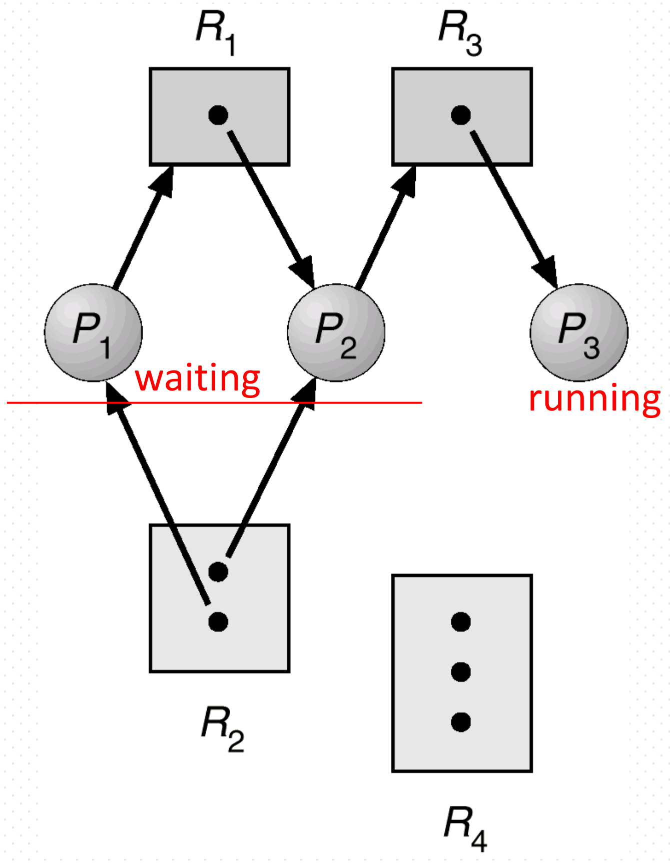


Fig.7.2

7.2 Deadlock Characterization

- **Necessary conditions for deadlock**
- If deadlock arises, then the following four conditions hold simultaneously
  - **mutual exclusion**
    - a resource (or ***a resource instance***) can be used at one time by only one process
  - **hold and wait (or**部分分配)
    - a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **no preemption**
    - a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Necessary conditions for deadlock

- **circular wait**
  - 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, …, 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.

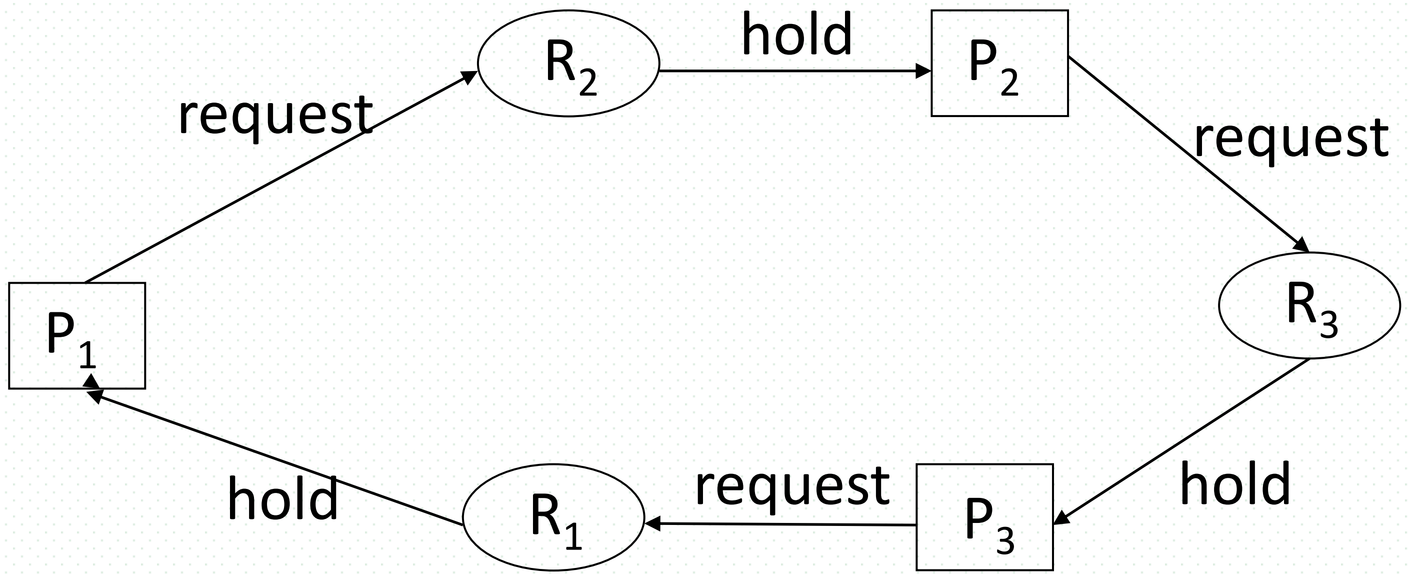


Fig. 7.0.3

Principles of deadlock handling

- *if* deadlock occurs

```
then {      (mutual exclusion)
       and  (hold and wait )
       and  (no preemption)
       and  (circular wait )
     }
```

- /(等价逆否命题) Equivalent converse-negative proposition

```
if {      not (mutual exclusion)
       or not (hold and wait )
       or not (no preemption)
       or not (circular wait )      }
then deadlock will not occur
```

◁

Principles of deadlock handling

- 通过破坏四个必要条件之一，阻止死锁的发生
  - 实际系统中主要采取破坏hold and wait 和circular wait的方法
- 三种死锁处理机制
  - 死锁的预防
  - 死锁的避免
  - 死锁的检测与恢复

Circular Wait/Cycles vs Deadlocks

- E.g.2
  - no cycle, no deadlock
  - process execution order
    - P_3, P_2, P_1

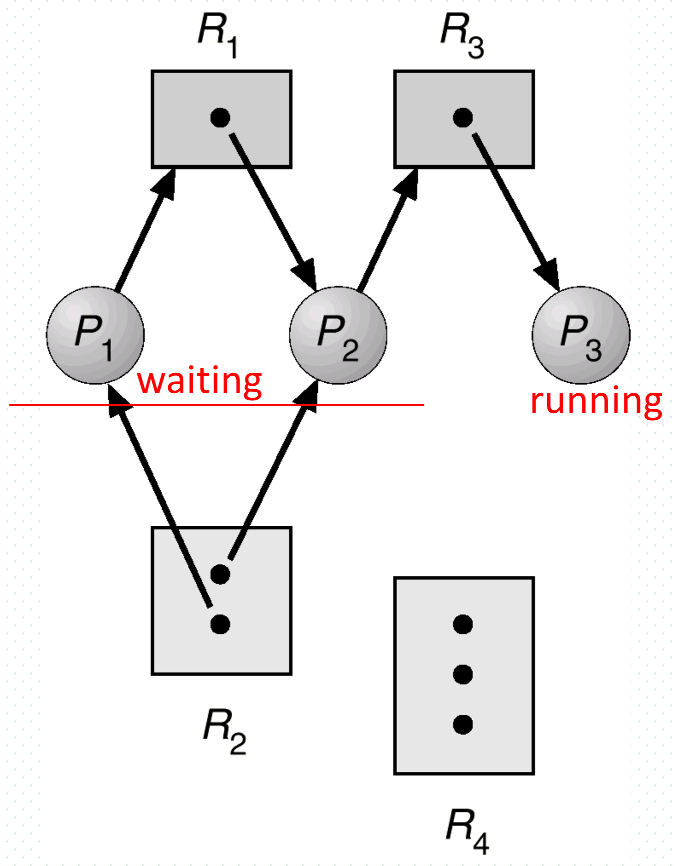
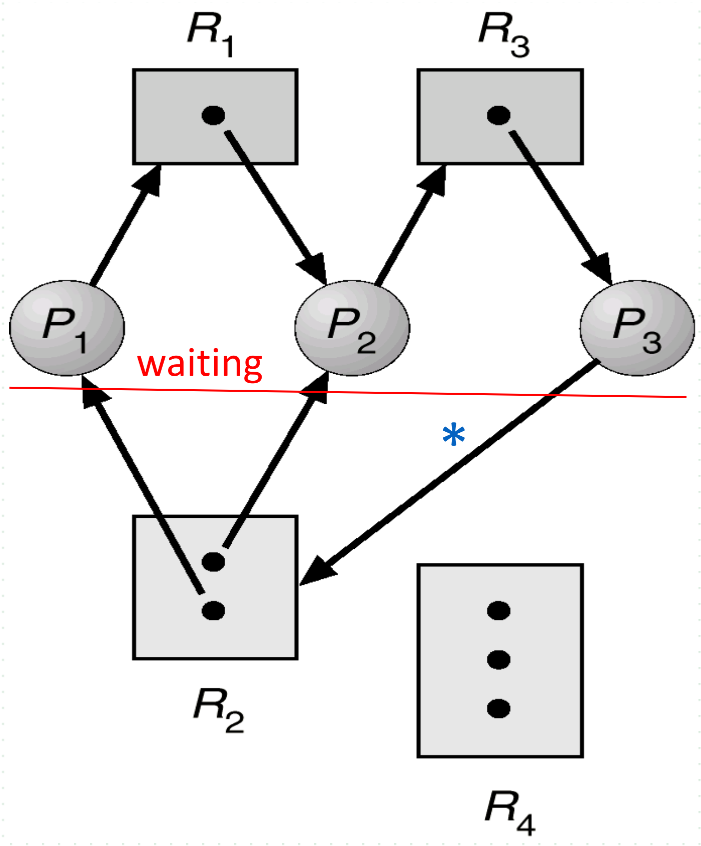


Fig.7.2

Circular wait/Cycles vs Deadlocks

- E.g.2
  - resource allocation graph with a deadlock
  - two cycles in the graph

1. P1 → R1 →P2 → R3 → P3 → R2→P1
2. P2 → R3 → P3 → R2→P2



F.g.7.3

Cycles vs Deadlocks

- E.g.2
  - resource allocation graph with a cycle but no deadlock
    - resource instance of R_2 occupied by P_4 is released to P_3 after being used

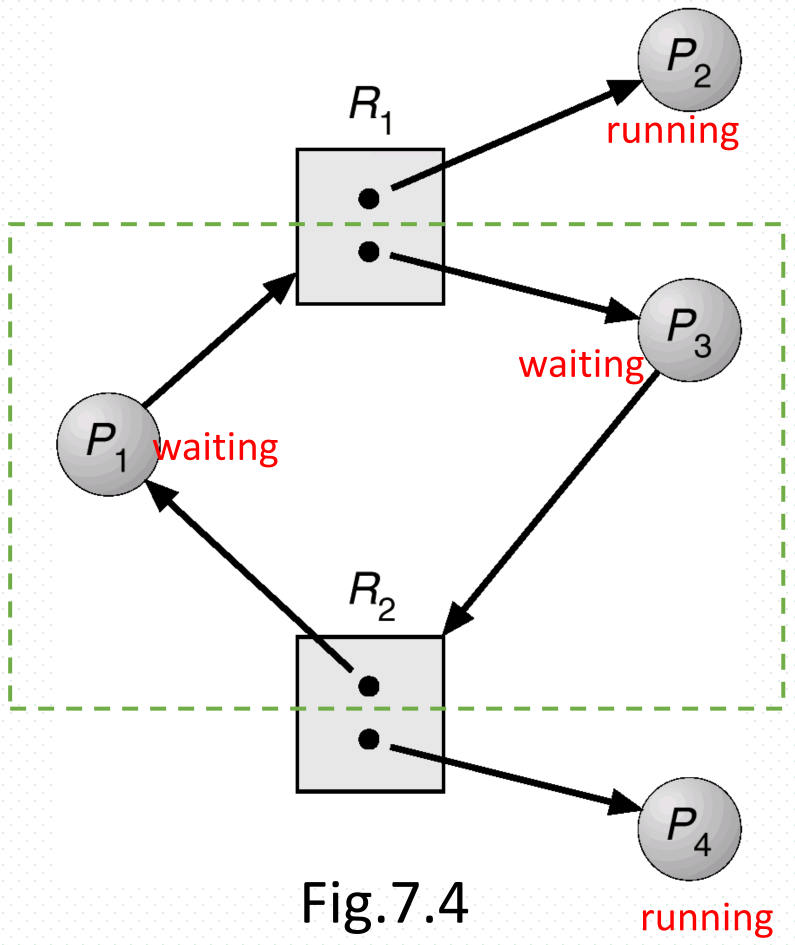


Fig.7.4

Cycles vs Deadlocks

- Conclusions !
  - if the graph contains no cycles $\Rightarrow$ no deadlock in the system
  - if graph contains a cycle $\Rightarrow$
    - if only one instance per resource type, then deadlock
    - if several instances per resource type, possibility of deadlock

Example 4

- 系统在某一时刻的资源分配表和线程等待表如下图所示，问
  - 该系统中是否存在死锁?
  - 如果存在，终止哪个线程解决死锁问题所付出的代价最小?

资源分配表

进程号	资源号
T_1	Q_1
T_2	Q_3
T_3	Q_2
T_3	Q_5
T_4	Q_4

占有边

线程等待表

线程号	资源号
T_1	Q_2
T_2	Q_1
T_3	Q_3
T_3	Q_4
T_4	Q_5

请求边

资源分配表

进程号	资源号
T_1	Q_1
T_2	Q_3
T_3	Q_2
T_3	Q_5
T_4	Q_4

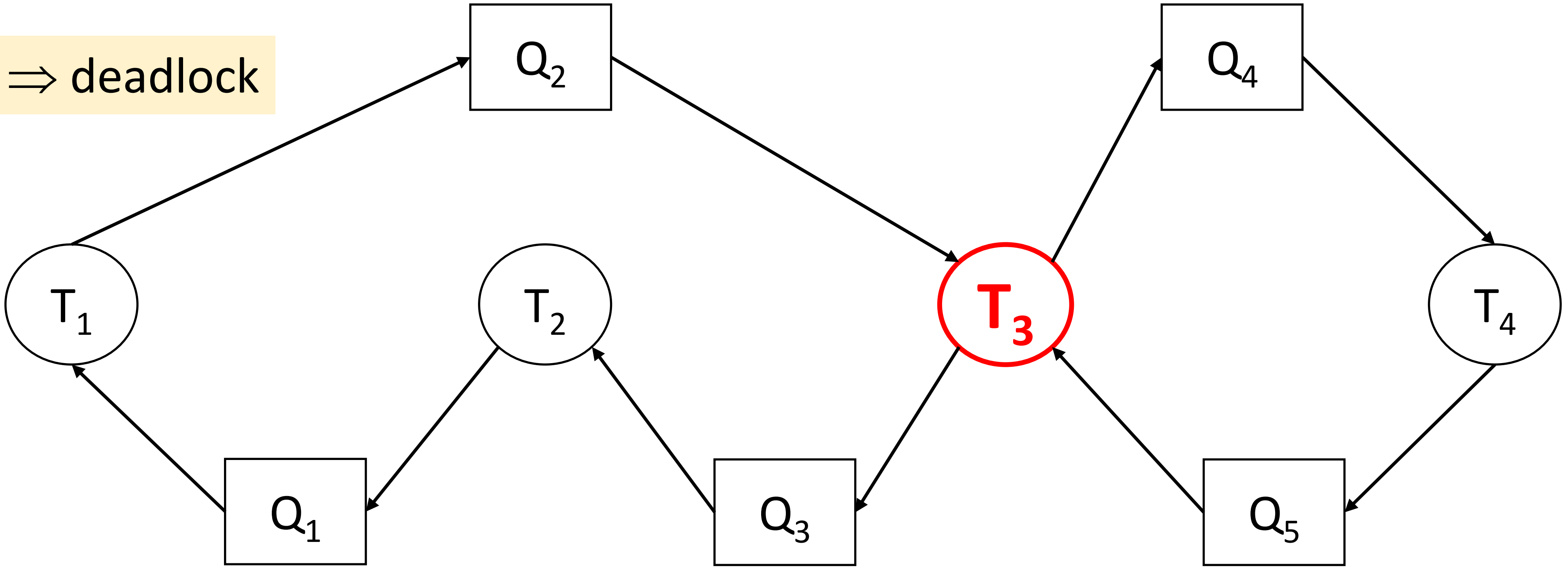
占有边

线程等待表

线程号	资源号
T_1	Q_2
T_2	Q_1
T_3	Q_3
T_3	Q_4
T_4	Q_5

请求边

two cycles ⇒ deadlock



Deadlock Occurrence vs Resource Amount

- Example
  - 某系统有K台互斥使用的同类设备， n=3个并发进程分别需要使用2、3、4台设备，可导致系统发生死锁的设备数目K<u>**最大**</u>为多少？

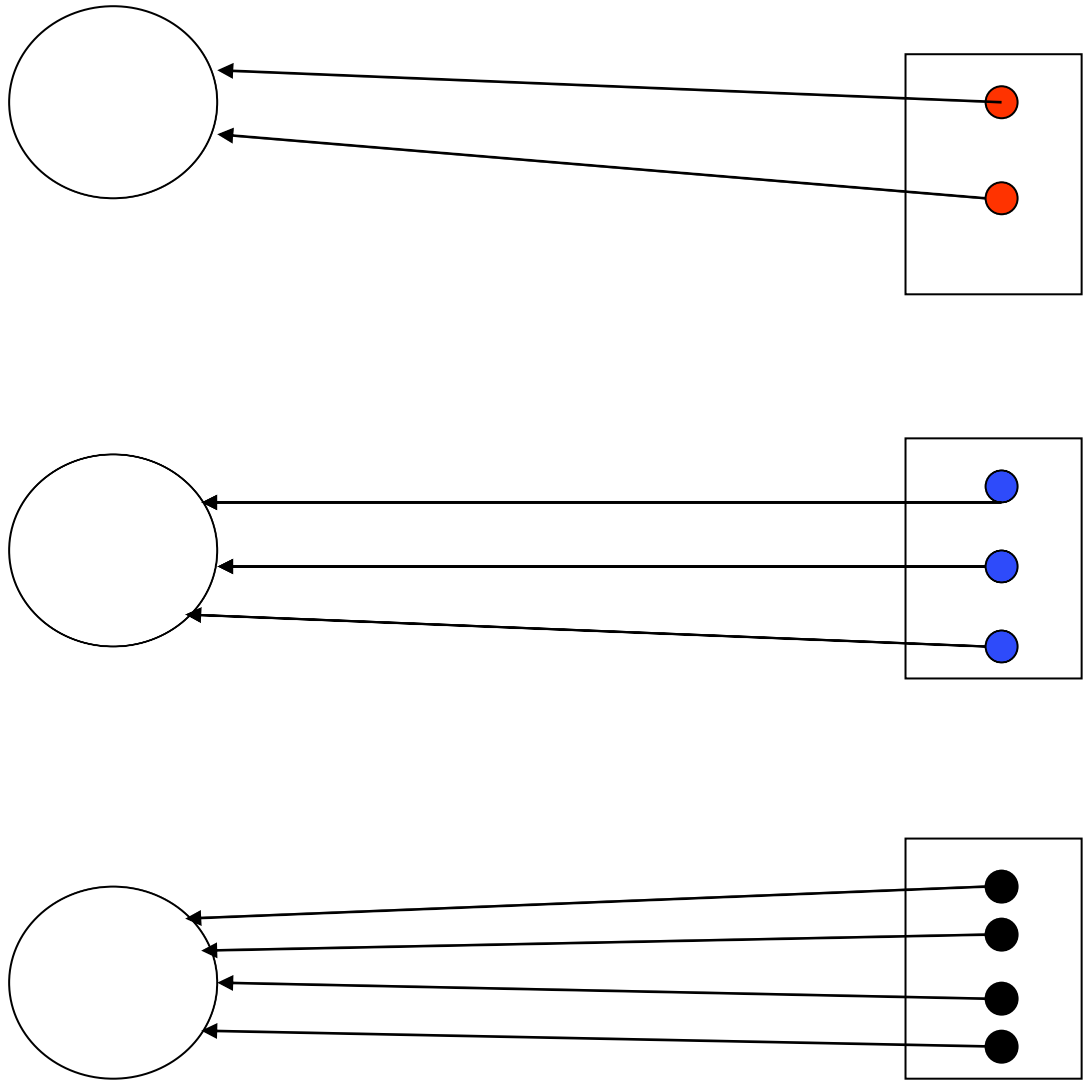
- 答案
  - 资源种类m=1， K个资源实例，
  - 导致发生死锁的

K = 所有进程的资源- 进程总数n

= (2+3+4) - 3= 6

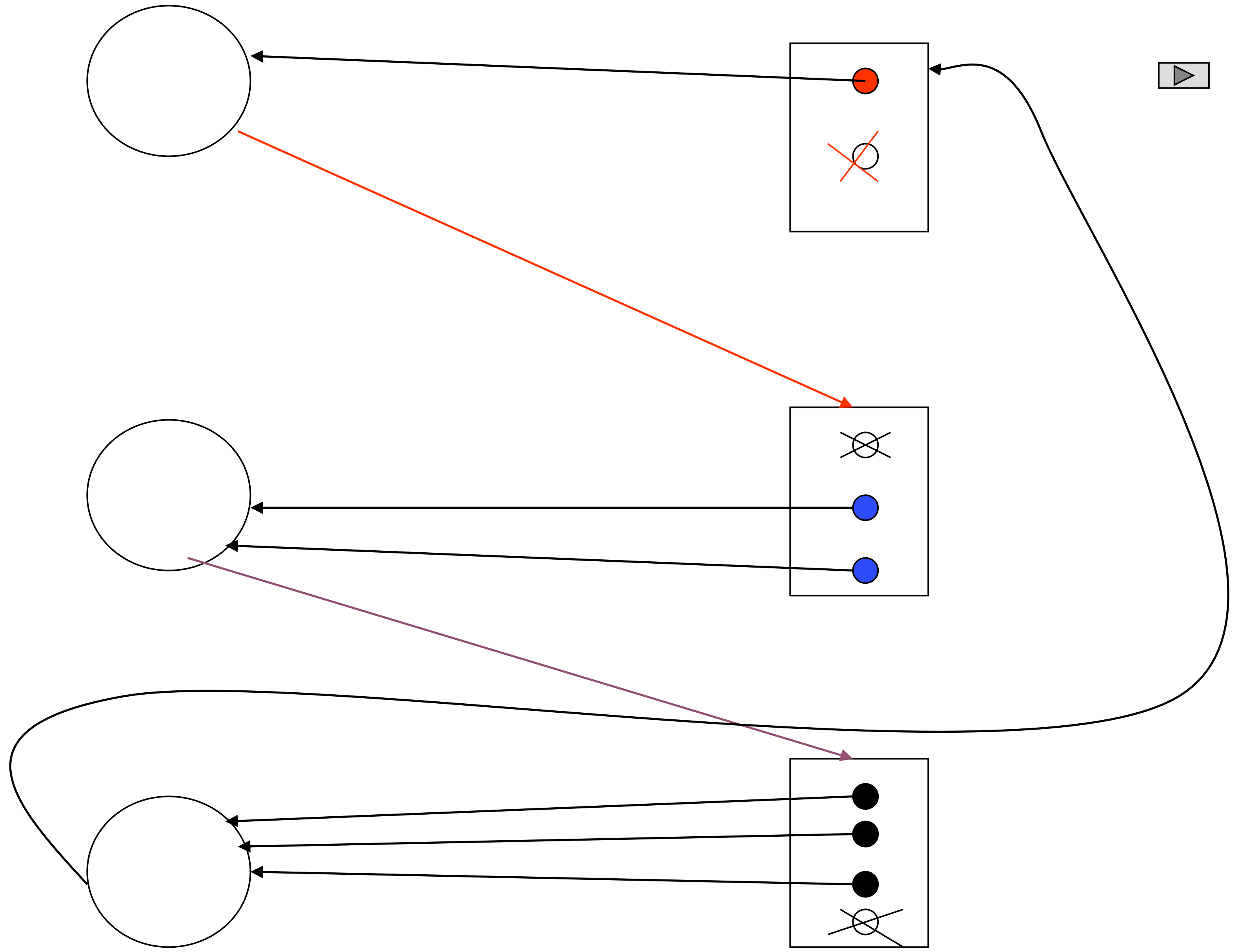
保证无死锁的最少资源实例总数=K+1

- 当K=9， n=3个进程的设备需求完全满足，进程无等待



why ?

- K=9-3=6，每个进程占有的设备数缺少1个，形成循环等待



- 在进程数目n、资源种类m固定情况下，已知进程对资源的最大总需求数L，导致发生死锁的资源实例总数K的最大值
  - K= L – n
- 保证无死锁的最少实例总数：K + 1

有环路、有死锁

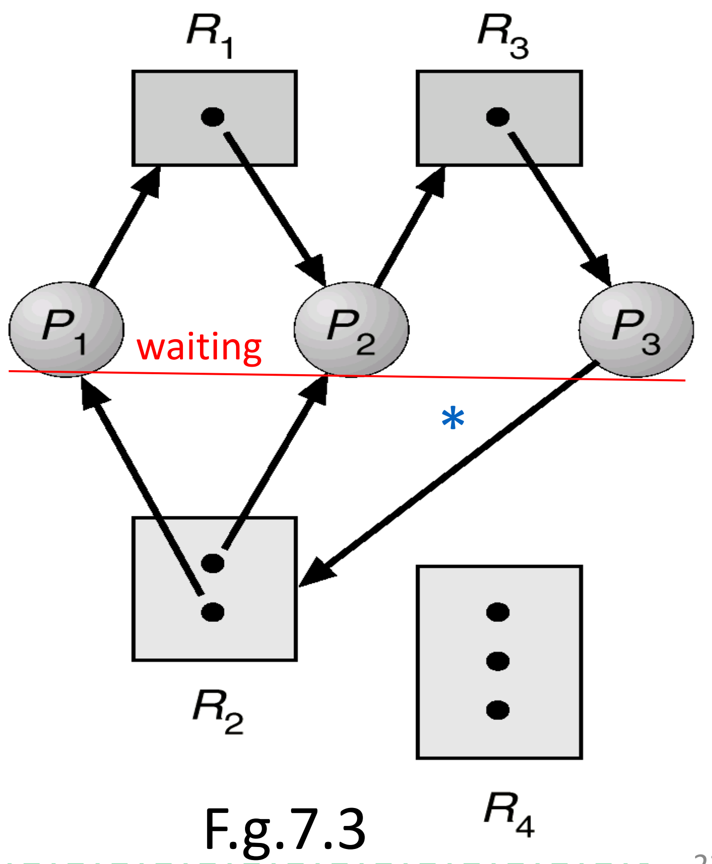
Fig. 7.3

进程数目n=3，有访问需求的资源总类m=3，

R_1, R_2, R_3的资源实例总数 =(1+ 2 +1)=4

3个进程对资源的总需求 L=(2 + 3 + 2) = 7

K= L- n= 7-3=4，等于目前资源实例总数，发生死锁



F.g.7.3

- 在进程数目n、资源种类m固定情况下，已知进程对资源的最大总需求数L，导致发生死锁的资源实例总数K的最大值
  - K= L – n
- 保证无死锁的**最少**实例总数：K + 1

Fig. 7.4

进程数目n=4，资源总类m=2，

2类资源的实例总数

=(2+ 2)=4

4个进程对资源的总需求L

=(2 + 1 + 2 + 1)=6

导致发生死锁的K= L- n=6-4=2

目前，2类资源的实例总数4 > K=2，因此，无死锁！

原因：P_2、P_4没有处于环路中

有环路、无死锁

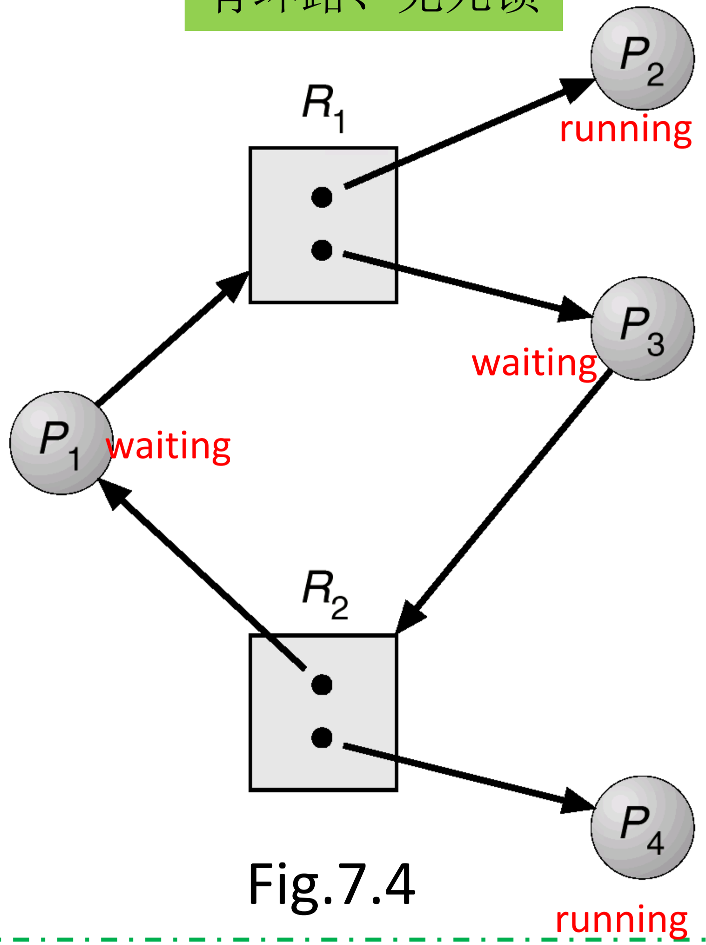


Fig.7.4

7.3 Methods for Handling Deadlocks

Deadlock can be handled in several ways as follows

- Ensure that the system will *never* enter a deadlock state
  - deadlock prevention
  - deadlock avoidance
- Allow the system to enter a deadlock state and then recover.
  - deadlock detection and recovery
- Ignore the problem and pretend that deadlocks never occur in the system, i.e. OS does not handle deadlock.
  - user or other systems are responsible for deadlock handling
  - <u>*used by most operating*</u> systems, including UNIX/Linux (?)

7.4 Deadlock Prevention

- The principle
  - Ensuring that at least one of the necessary conditions cannot hold by constraining the ways resources request can be made
    - /*通过破坏四个必要条件之一，阻止死锁的发生
    - /*实际系统中主要采取破坏**hold and wait** 和**circular wait**的方法 ▶

1. Mutual Exclusion

- Method
  - mutual exclusion constraints can be relaxed for not sharable resources (e.g., read-only files), i.e. permitting several processes to access sharable resources simultaneously.
- However, mutual exclusion constraint must hold for nonsharable resources (e.g., printers).
- Conclusions
  - cannot prevent deadlock by denying the mutual-exclusion condition

2. Hold and Wait

- Method

  must guarantee that whenever a process requests a resource, it does not hold any other resources.

  - Protocol 1
    - the process can only request and be allocated **all required resources** before it begins execution
    - /* 预先全部分配所需资源（第6章信号量作业，哲学家就餐）
  - Protocol 2
    - the process is allowed to request resources several times during its lifetime
    - it must release all the resources that it is currently allocated before it requests any additional resources, *e.g. 2PL in DBS*

T_1和T_2并发访问数据项A和B

	T_1	T_2
	wait(A)	
	wait(B)	
growing	read (A)	
phase	read (B)	
	signal (B)	
shrinking		wait (B)
phase	write (A)	
		read (B)
	signal (A)	
		write (B)
		signal (B)

Fig. A concurrent schedule S in DBS under 2PL protocol constraints

Hold and Wait

- E.g. P253. processes and tape driver, disk files and printers
- Conclusions
    - an applicable method
    - demerits: low resource utilization; starvation possible.

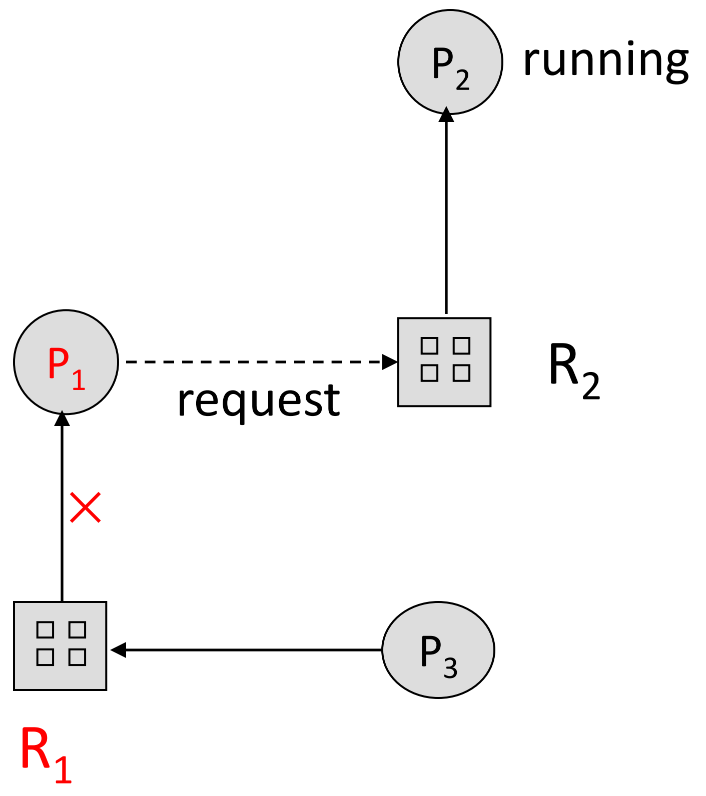
3. No Preemption

- Principle
  - /*允许进程抢占处于**waiting态**的进程占有的资源
    - e.g. swap in/swap out
- To ensure ***No Preemption*** does not hold, the protocol as follows is used
  - if P_1 is now holding some resources R_1 and requests another resource R_2, which is now occupied by P_2 **and cannot be immediately allocated to P**$_1$,
- for example, P_2 is in the running state and using R_2, then

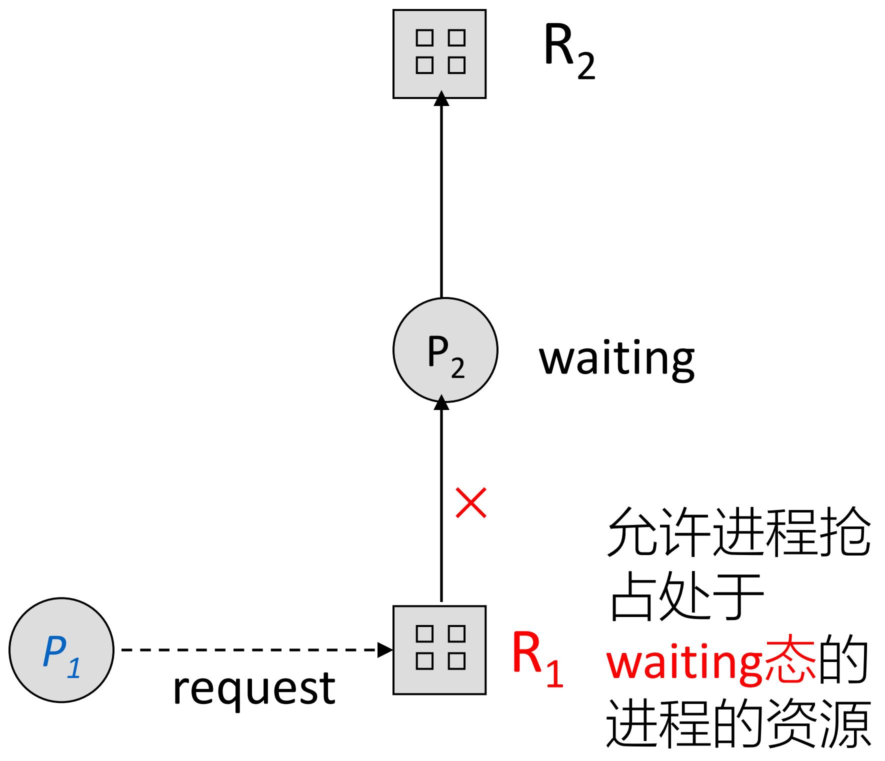
    refer to Fig.7.0.4(a)

  - P_1 turns into the waiting state and R_1 held by P_1 is released
  - preempted resources R_1 is added to the list of resources for which the process P_3 is waiting, so the waiting process P_3 may be enabled to run
  - the resources-preempted process P_1 will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Waiting 态下进程释放/不占有资源



(a) R_1 is released and P_1 turns into *waiting* state, thus the waiting process P_3 will be enabled to run



(b) P_1 preempts R_1 from the *waiting* process P_2

Fig. 7.0.4

No Preemption

- if P_1 requests R_1 that is not available, and if R_1 is allocated to a waiting process P_2 that is waiting for another resource R_2, then
  - P_1 preempts R_1 from the waiting process P_2
  - refer to Fig. 7.0.4(b)

- Conclusion
  - this method can be applied to resources *whose state can be easily saved and restored later*, such as CPU registers and memory

4. Circular Wait

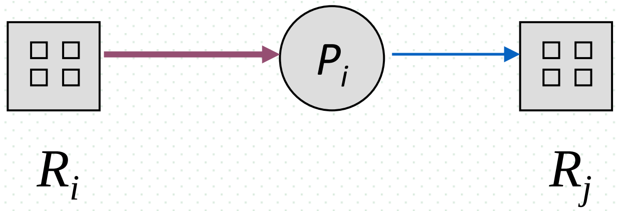
- To ensure this condition cannot hold
  - impose a total ordering of all resource types
  - require that each process requests resources in an increasing order of enumeration. /* 有序资源使用法
- Implementation
  - $R=\{R_1, R_2, R_3,\ldots, R_m\}$,
  - resource ordering function F: $R \rightarrow N$
  - e.g. F(tape driver) =1,

    F(disk driver) =5, F(printer) =12

Circular Wait

- Protocol 1
  - if P_i has requests or holds the instances of R_i, then afterwards it can only request resource R_j such that $F(R_j) > F(R_i)$

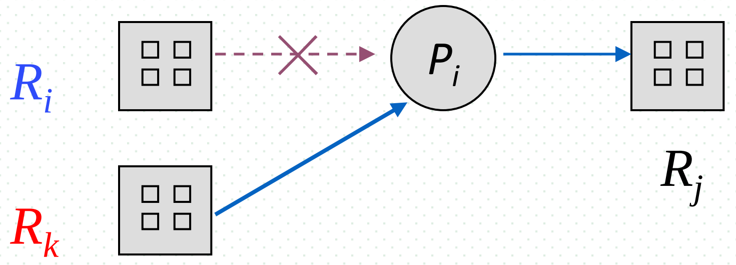
/* 按资源序号递增的顺序请求资源



$F(R_j) > F(R_i)$

Circular Wait

- Protocol 2
  - whenever P_i requests an instances of R_j , it must have released all resources R_i such that $F(R_i) \geq F(R_j)$



$F(R_i) \geqslant F(R_j)$

$F(R_k) < F(R_j)$

- Conclusion
  - in line with these two protocols, the **Circular Wait** condition cannot hold and the deadlock is prevented
  - more commonly used, as the basis of deadlock avoidance and detection

7.5 Deadlock Avoidance

- Assuming OS resources allocator knows some additional/*priori* information about how resources are to be requested by processes
  - each process declares the *maximum number* of resources of each type that it may need
  - e.g. 批处理系统中，使用作业说明书描述各个作业的资源需求
- When a process requests resources, on the basis of these *a priori* information, the allocator uses deadlock-avoidance algorithm to examines the resource-allocation state
  - to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.
  - 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.
  - < available, allocated, maximum>
  - e.g. ▷

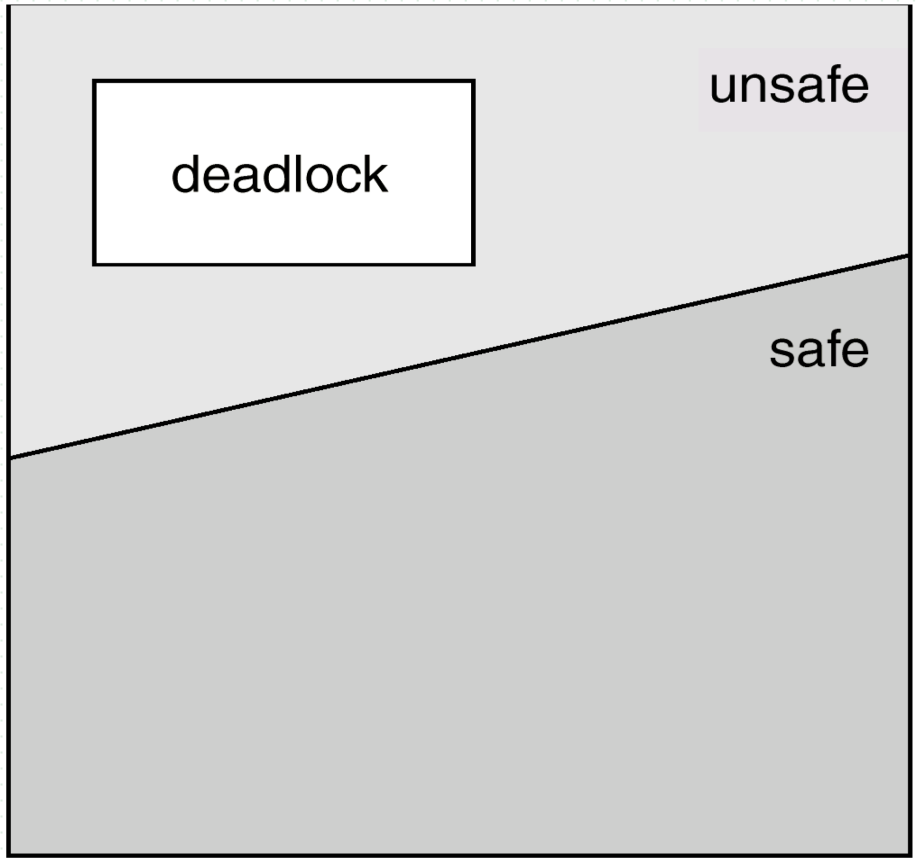
Safe State

- Intuitively, system is in safe state if it can allocate resources to processes and still avoid deadlock
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a *safe state*
- Safe state
  - formally, the system is in the safe state only if there exists a safe sequence of all processes $<P_1, P_2, \ldots, P_n>$ executing
  - sequence $<P_1, P_2, \ldots, P_n>$ is safe for the current allocation state, if for each P_i, *the resources that P_i can still request* can be satisfied by *currently available resources* plus *resources held by all the P_j, with $j<i$.*
    - if the resource needs of P_i are not immediately available, then P_i can wait until all P_j have finished
    - when P_j finished, P_i obtains needed resources, executes, returns allocated resources and terminates

Safe State

- Note
  - for a safe state, there may exist more than one safe process sequences

- Conclusion !!!
  - if a system is in a safe state $\Rightarrow$ no deadlocks
  - if a system is in an unsafe state $\Rightarrow$ possibility of deadlock
  - avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state, denying the circular-wait condition
  - the safe state is a *sufficient condition* for non-deadlock

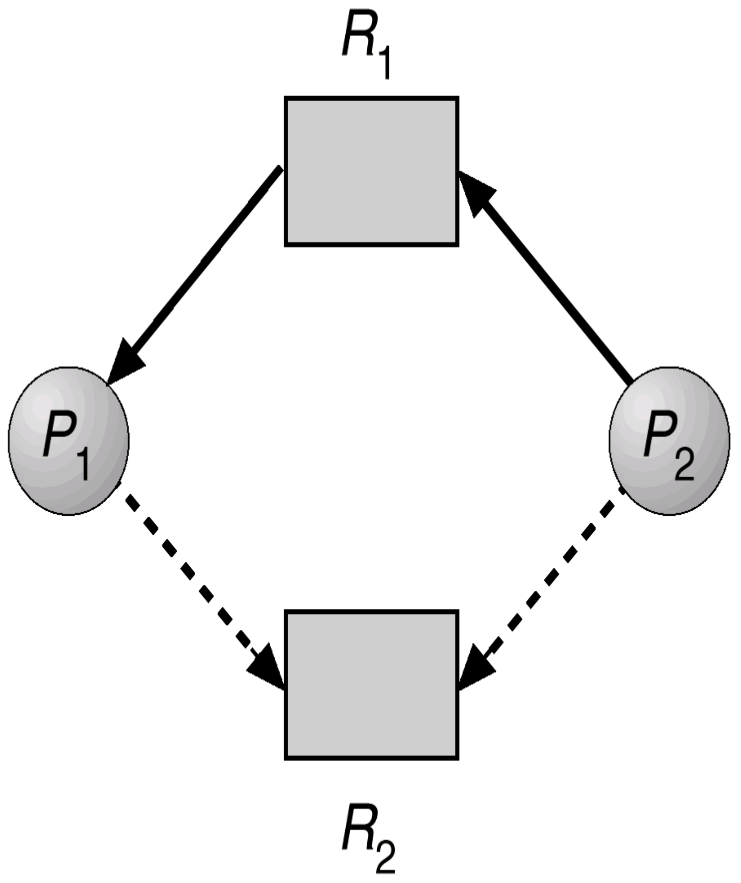


F.g.7.5 Safe, Unsafe, Deadlock State

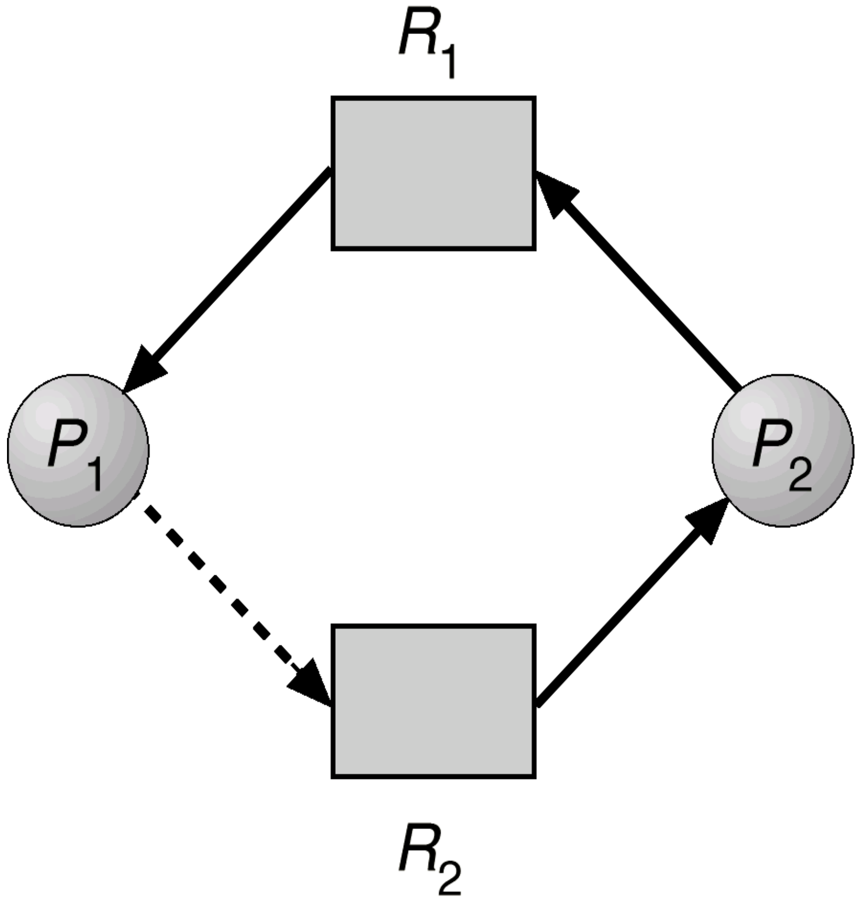
Resource-allocation graph algorithm

- This algorithm can only be applied to the systems with only one instance of each resource type.
- Resource-Allocation Graph G (V,E) (Fig.7.6/7.7)
  - vertex: processes, resources
  - edge: request edge, assignment edge, claim edge
  - claim edge $\mathbf{P_i} \rightarrow \mathbf{R_j}$ indicated that process $\mathbf{P_i}$ ***may*** request resource $\mathbf{R_j}$; represented by a dashed line.
  - claim edge converts to request edge, when a process requests a resource.
  - assignment edge reconverts to a claim edge, when a resource is released by a process

Fig. 7.4



E.g.1 Resource-allocation graph for deadlock avoidance (Fig.7.6)



E.g.2 Unsafe state in resource-allocation graph (Fig.7.7)

Resource-allocation graph algorithm

- Cycle-checking based deadlock avoid algorithm
  - resources must be claimed *a priori* in the system, i.e. *the graph can be built beforehand.*
  - **the system state is safe, that is, there is no deadlock, only if there is no cycle in the resource-allocation graph**
    - the resource requests from processes are granted only if these requests do not result in the formation of a cycle in the resource-allocation graph

Banker’s Algorithm

- Applicable to the system with multiple instances of each resource type***, but less efficient***
- Principles
  - each process must declare the maximum number of instances of each resource type that it may need
    - e.g. 批处理系统，作业说明书
  - when a process requests a set of resources, the system determines whether the allocation of these resources will leave the system in a safe state, and then the process may get the resources or have to wait.
  - when a process gets all its resources, it must return/release them in a finite amount of time.

Banker’s Algorithm

- Data structures for the Banker’s algorithm
  - *n* : number of processes, *m:* number of resources types.
  - Available[m]: if available [j] = k, there are k instances of resource type R_j available/idle
  - Max[n, m]: if Max [i, j] = k, then process P_i may request at most k instances of resource type R_j.
  - Allocation[n, m]: if Allocation[i, j] = k, then P_i is currently allocated k instances of R_j.
  - Need[n, m]: 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]

Banker’s Algorithm

- Safety Algorithm (§7.5.3.1)
  - goal: to determine whether or not the system is in a safe state, i.e. to evaluate a safe system state
  - algorithm steps

Step1. Let Work and Finish be vectors of length *m* and *n*, respectively. Initialize:

Work = Available   /* 系统空闲/可用/回收资源

Finish [i] = false for i = 1,2, …, n.   /*P_i是否已结束*/

Step2. Find an *i* such that both:

/*找出1个可安全分配资源的进程 P_i*/

(a) Finish [i] = false

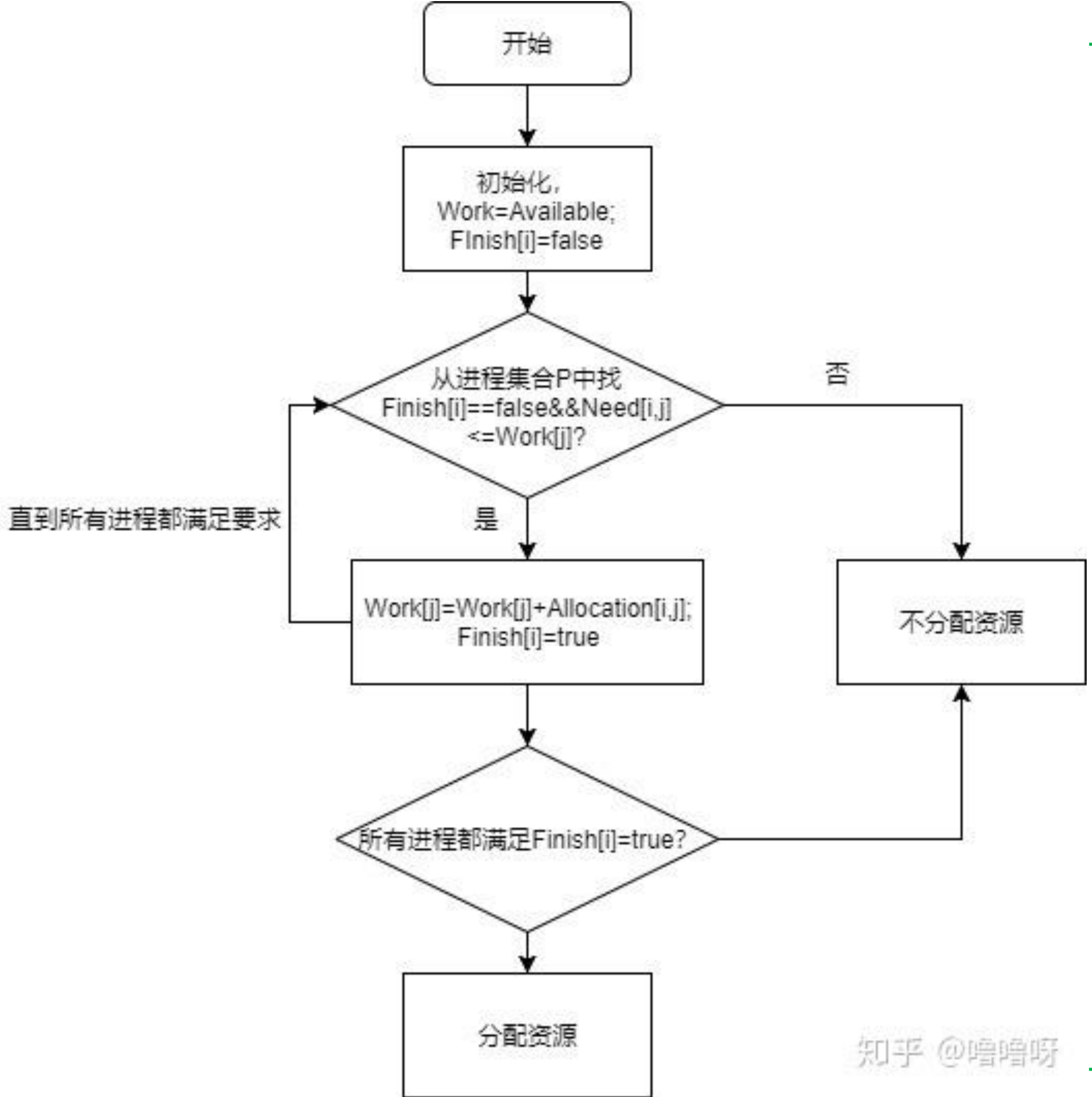
(b) $Need_i \leq Work$

if no such *i* exists, go to step 4.

Banker's Algorithm

Step3. Work = Work + $Allocation_i$ /* P_i执行；
Finish[i] = true P_i结束，释放其占有的资源*/
go to step 2   /*继续考察其它进程

Step4. If Finish [i] == true for all i,
then the system is in a safe state.
otherwise, the system is in a unsafe state


开始
初始化,
Work=Available;
FInish[i]=false
从进程集合P中找
Finish[i]==false&&Need[i,j]
<=Work[j]?
否
是
直到所有进程都满足要求
Work[j]=Work[j]+Allocation[i,j];
Finish[i]=true
不分配资源
所有进程都满足Finish[i]=true?
分配资源

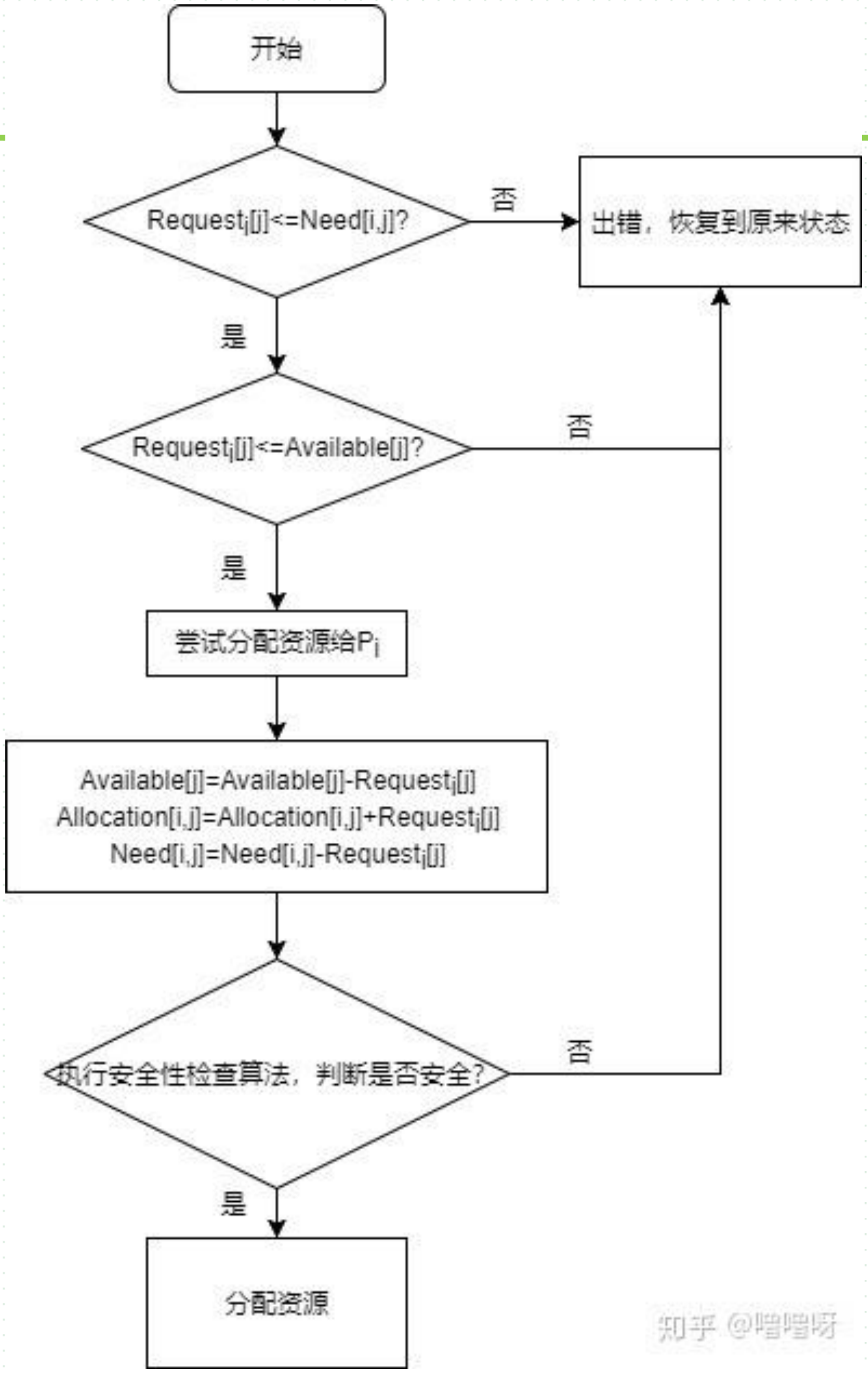
Banker's Algorithm

- Resource-Request Algorithm for Process P_i (§7.5.3.2) ▷
  - Goal: to evaluate a safe resource request from a process P_i
  - $Request_i$ [m]: request vector for process P_i.

    $Request_i$ [j] = k : process P_i wants k instances of resource type $R_{j.}$

  - note: for process P_i , its resource requirement $Need_i$ can be satisfied through several $Request_{i1}$, $Request_{i2}$, …, $Request_{ik}$, where

    $Request_{i1}$ + $Request_{i2}$ + …, $Request_{ik}$ = $Need_i$


开始
$Request_i[j]<=Need[i,j]$?
否
出错，恢复到原来状态
是
$Request_i[j]<=Available[j]$?
否
是
尝试分配资源给P_i
$Available[j]=Available[j]-Request_i[j]$
$Allocation[i,j]=Allocation[i,j]+Request_i[j]$
$Need[i,j]=Need[i,j]-Request_i[j]$
执行安全性检查算法，判断是否安全?
否
是
分配资源
知乎 @喵喵呀

Banker's Algorithm

- algorithm steps

when P_i makes a resource request $Request_i\,[m]$,

Step1. If $Request_i \leq Need_i$ go to step 2;

otherwise, raise error condition, since process has exceeded its maximum claims

Step2. If $Request_i \leq Available$, go to step 3;

/*系统有可用资源供P_i使用*/

otherwise P_i must wait, since resources are not available

Banker's Algorithm

Step3. Pretend to allocate requested resources $Request_i$ [m] to P_i by modifying the state as follows:

- Available := Available – $Request_i$;

  $Allocation_i$ = $Allocation_i$ + $Request_i$;

  $Need_i$ = $Need_i$ – $Request_i$;

- use safety algorithm in§7.5.3.1 to decide whether or not the system is safe after allocating resources to P_i

(1) if safe $\Rightarrow$ the resources are allocated to P_i.

(2) if unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Banker's Algorithm —Example One

- There are 5 processes P_0 , P_1, P_2, P_3, P_4; 3 resource types *A* with 10 instances (e.g. memory), *B* with 5 instances (disk), and *C* with 7 instances (tapes)
  - snapshot at time T_0
  - safe sequence < P_1, P_3, P_4, P_2, P_0>, < P_3, P_1, P_4, P_2, P_0>

	Allocation	Max	Available	Need=MAX —Allocation
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		> 1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		> 0 1 1
P_4	0 0 2	4 3 3		4 3 1

total resources

:	A	B	C
	10	5	7

Example

- total resources=(10 5 7)
- Need is defined to be Max – Allocation

Need=MAX-- Allocation

- apply safety algorithm, for sequence $< P_1, P_3, P_4, P_2, P_0>$

	Need: A B C	Work
step5. P_0	7 4 3	< (7 4 5) + Allocat2(3 0 2) =(10 4 7)
step1. P_1	1 2 2	< (3 3 2)
Step4. P_2	6 0 0	< (7 4 3) + Allocat4(0 0 2) =(7 4 5)
step2. P_3	0 1 1	< (3 3 2) + Allocat 1(2 0 0)=(5 3 2)
Step3. P_4	4 3 1	< (5 3 2) + Allocat3 (2 1 1)=(7 4 3)

- In the current situation, the system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria. ▶

- decide whether or not the P_1's request(1, 0, 2) can be granted
  - request (1, 0, 2) < $Need_1$ = (1, 2, 2)
  - check that Request ≤ Available (that is, (1, 0, 2) ≤ (3, 3, 2) ⇒ true), (1 0 2) are allocated to P_1, and system enters a new safety state

	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

3 3 2 –1 0 2

1 2 2 –1 0 2

2 0 0 + 1 0 2

- executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement

补充作业

- Can request for (3, 3, 0) by P_4 be granted?
- Can request for (0, 2, 0) by P_0 be granted?

Example Two

- For the system described in the following table
  - (a) is the system in a safe or unsafe state ?   Why ?
  - (b) if P_3 request resource of (0, 1, 0, 0), can resources be allocated to it ? Why ?

process	Current allocation				Maximum allocation				Resource available			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	2	1	0	0
P_2	2	0	0	0	2	7	5	0				
P_3	0	0	3	4	6	6	5	6				
P_4	2	3	5	4	4	3	5	6				
P_5	0	3	3	2	0	6	5	2				

Example Two

- [a]
  - the system is in a safe state, because there exists a safe process sequence <P_1, P_4, P_5, P_2, P_3>
  - the *Need* Matrix (*=MAX– Allocation)* is:

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	6	6	2	2
P_4	2	0	0	2
P_5	0	3	2	0

- according to safety algorithm,
  - firstly, Work:=(2, 1, 0, 0), Finish[i]:=false i:=1, 2, 3, 4, 5
  - for i:=1, Need1=(0, 0, 0, 0) < Work, Finish[1]:=true, and Work:=(2, 1, 1, 2)
  - for i:=4, Need4=(2, 0, 0, 2) < Work, Finish[4]:=true, and Work:=(4, 4, 6, 6)
  - for i:=5, Need5=(0, 3, 2, 0) < Work, Finish[5]:=true, and Work:=(4, 7, 9, 8)
  - for i:=2, Need2=(0, 7, 5, 0) < Work, Finish[2]:=true, and Work:=(6, 7, 9, 8)
  - for i:=3, Need3=(6, 6, 2, 2) < Work, Finish[3]:=true, and Work:=(6, 7, 12, 12)
  - Finish[i]=true for i:=1, 2, 3, 4, 5, so the system is in a safe state.

Example Two

- [b]
  - No, because if the resources are allocated to P_3, then the system will be in a unsafe state.
  - matrix *Allocation* , *Need* , and *Available* are as follows:

process	Current allocation				Need				Available			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	0	0	2	0	0	0
P_2	2	0	0	0	0	7	5	0				
P_3	0	1	3	4	6	5	2	2				
P_4	2	3	5	4	2	0	0	2				
P_5	0	3	3	2	0	3	2	0				

- according to safety algorithm,
  - firstly, *Work*:= (2, 0, 0, 0), *Finish*[i]:=false i:=1, 2, 3, 4, 5
  - for i:=1, *Need1*=(0, 0, 0, 0) < *Work*, *Finish*[1]:=true, and *Work*:=(2, 0, 1, 2)
  - for i:=4, Need4=(2, 0, 0, 2) < Work, Finish[4]:=true, and Work:=(4, 3, 6, 6)
  - for i:=5, Need5=(0, 3, 2, 0) < Work, Finish[5]:=true, and Work:=(4, 6, 9, 8)
  - for i:=2, Need2=(0, 7, 5, 0) < Work is not true;
  - and for i:=3, Need3=(6, 5, 2, 2) < Work is not true

- It is not true that Finish[i]=true for i:=1, 2, 3, 4, 5
- so there not exists a safe sequence, and the system is in a unsafe state.
- resource request from P3 can not be satisfied immediately

注意

- 关于安全状态、安全序列是针对死锁避免、<u>**不是针对死锁检测机制**</u>
- 死锁检测中无安全状态、安全序列的概念，只有死锁、非死锁状态

7.6/7.7 Deadlock Detection and Recovery

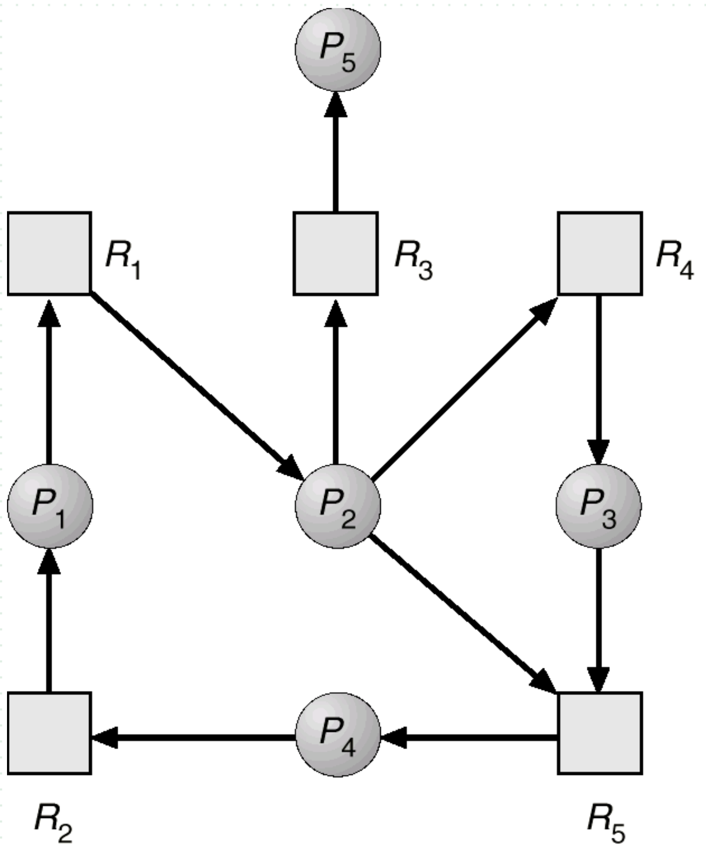
- Demerits of deadlock prevention and avoidance
  - algorithms for handling deadlock are somewhat complex

- Principle of deadlock detection and recovery
  - allow system to enter deadlock state, find deadlock by detection algorithm, and overcome deadlock by recovery scheme

- Detection for single-instance-resource systems
- Detection for multiple-instance-resource systems

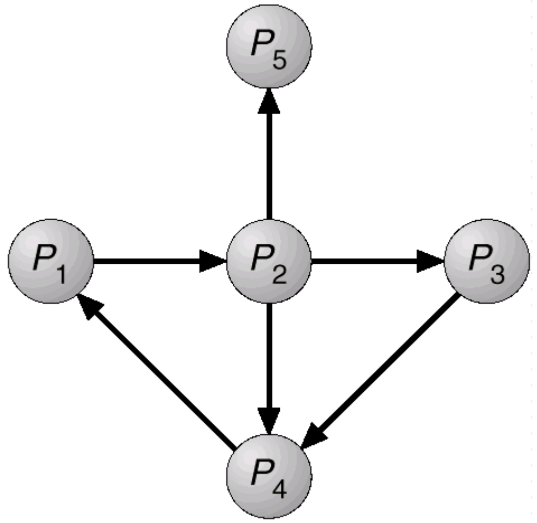
Detection for Single-instance-resource Systems

- For systems in which each resource type has only single instance, maintain wait-for graph (deriving from resource allocation graph)
  - nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an detection algorithm that searches for a cycle in the graph
  - *cycle* $\Rightarrow$ *deadlock*
- The 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



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

Detection for Multiple-instance-resource Systems

- Data structures for detection algorithm
  - *Available*[m]: a vector of length m indicates the number of available/idle resources instances of each type.
  - *Allocation*[n, m] : defines the number of resources of each type currently allocated to each process.
  - *Request*[n, m]:   indicates the current request of each process. If *Request* [i, j] = k, then process P_i is requesting k instances of resource type. R_j.
  - *Work* and *Finish* be vectors of length m and n, respectively
  - **没有Max, Need矩阵**

- The system state is defined by Available, Allocation, and Request

- Detection algorithm Principles

  /* 每一时刻，系统内进程P_i 分为3类

  - Allocation$_i$ =0， 并且 Request$_i$ = 0或Request$_i \neq 0$： 标记为Finish[i] = true
    - 不占有资源，不会引起死锁（不出现在环路中）
  - Allocation$_i \neq 0$， Request$_i \leq$ Work： 标记为Finish[i] = true
    - 已经占有资源，又申请新资源，并且资源请求能够满足, 可立即执行并结束
  - *Allocation*$_i \neq 0$， *Request*$_i$ > *Work*： *Finish*[i] = false
    - 已经占有资源，又申请新资源，但资源请求目前无法满足，标记暂时仍为 需要等到以后某一时刻其他进程结束并归还资源后，再判断本进程的资源请求能否得到满足

- 检测算法考虑 (2), (3) 类进程, very similar to the safety algorithm in Banker algorithm

- Algorithm Steps
  - step1. Initialize:

```
(a) Work = Available
(b) For i = 1, 2, …, n,
if  allocation_i ≠ 0
  then  Finish[i] = false
   /*占有资源, 涉及死锁*/
  else  Finish[i] = true
 /排除不占有资源，不导致死锁 的进程
```

  - step2. Find an index i such that both:

(a)   Finish[i] == false
(b) $Request_i \leq Work$
/* P_i 可以立即获得资源并执行

, if no such i exists, go to step 4

  - step3. $Work = Work + Allocation_i$
    Finish[i] = true
    go to step 2.
    / *为Pi 分配资源使之执行

  - step4. If Finish[i] == false, for some i, 1 $\leq i \leq$ n, then:

(1) the system is in deadlock state.

(2) moreover, if Finish[i] == false, then P_i is deadlocked

/*当前状态下，不存在一个可分配的 process sequence（按此序列为进程分配资源，不会导致死锁

Example

- Five processes P_0 through P_6, three resource types
  - A with 7 instances, B with 2 instances, and C with 6 instances.
- Snapshot at time T_0:

	Allocation	Request	Available	Finish
	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		
P_5	0 0 0	0 0 0		true
P_6	0 0 0	3 2 6		true

Example

- The system is not in a deadlock state, because
  - the sequence <P_0, P_2, P_3, P_1, P_4> and will result in Finish[i] = true for all i. or: < P_2, P_0, P_3, P_1, P_4>
- 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

- then the system is in a deadlock state, processes P_1, P_2, P_3, and P_4. is deadlocked

	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

补充作业2

- Five processes P_0 through P_4; three resource types
  A with 6 instances, B with 3 instances, and C with 6 instances
- Given the 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	1 0 1
P_1	2 0 1	2 0 2	
P_2	1 1 1	0 0 2	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

补充作业2

- Answer the following questions on the basis of deadlock-detecting algorithm
  - is the system in a deadlock-free state?   and why?
  - if P_2 requests two additional instance of type A, is there a deadlock in the system, and why?

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	1 0 1
P_1	2 0 1	2 0 2	
P_2	1 1 1	2 0 2	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

When and how often to invoke the detection algorithm

- Depends on /*何时启动算法*/ (§7.6.3)
  - how often a deadlock is likely to occur?
  - how many processes will need to be rolled back?
    - one for each disjoint cycle ▷
  - 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
- When deadlock recovery occurs,
  - handled by operators, **or**
  - recovery automatically by
    - process termination
    - resource preemption

oepnGauss数据库事务死锁检测与处理：
1.针对表级锁的等待队列，进行环路检测
2.回滚某些处于环路中的事务

Process Termination

break the circular wait

- How to terminate the process in deadlock ?
  - abort (夭折、撤销) all deadlocked processes
  - abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - priority of the process.
  - how long process has computed, and how much longer to completion.
  - resources the process has used.
  - resources the process needs to complete.
  - how many processes will need to be terminated.
  - is process interactive or batch?

Resource Preemption

- Selecting a process as a victim
  - minimize the costs
- Rollback (回滚，卷回) this victim
  - return to some safe state, restart the process for that state.
- The number of rollbacks should be included in cost factor
- Maybe resulting in starvation
  - a process may always be picked as victim

Livelock（活锁）

- Livelock
  - 竞争锁的进程/线程长时间处于竞争、但无法获得锁的状态，自身也没有进入等待/阻塞态
  - 带有busy-waiting无阻塞的锁/信号量造成的
- 示例
  - 假设：为预防死锁，采用“不允许hold-and-wait”的策略，即：一次申请获得全部所需资源，并且当任意需要获取的资源不可用时，放弃已经占有的资源，并尝试重新申请全部所需资源
  - e.g. 线程需要申请资源A、资源B
- Linux非阻塞加锁操作trylock()，线程执行trylock()申请加锁
  - 如果加锁成功，返回SUCC
  - 如果加锁失败，返回FAIL，线程立即返回，不会进入阻塞态等待获取锁

thread T_1

```
while (true)  {
  if  (trylock(A)== SUCC) {
       if  (trylock(B)==SUCC) {  //加锁A、B成功，进入临界区
          //临界区
          ....
          unlock(B);
          unlock(A);
          break;
       }  else  {                //加锁B失败，释放已经获得的锁A
          unlock(A);
       }
  }
}
```

thread T_2

```
while (true)  {
  if  (trylock(B)== SUCC) {
       if  (trylock(A)==SUCC) {  //加锁B、A成功，进入临界区
          //临界区
          ....
          unlock(A);
          unlock(B);
          break;
       }  else  {                //加锁A失败，释放已经获得的锁B
          unlock(B);
       }
  }
}
```

Livelock

- T_1、T_2采用时间片调度，交替执行各自步骤
- 如果T_1在第2步获得A上的锁，但第3步加锁B失败，因为之前B锁已经被T_2获得
  - T_1释放A上的锁，回到第2步，重新申请加锁A
- 类似地，T_2在第2步获得B上的锁，但第3步加锁A失败，因为之前B锁已经被T_1获得
  - T_2释放B上的锁，回到第2步，重新申请加锁B
- 如果T_1、T_2执行速度相似，则下一次还会出现类似情况，线程不断重复“尝试-失败-尝试-失败”的过程，导致两个线程在非阻塞态下，都无法同时获取需要的两把锁、进入临界区——活锁

- 解决方案：依赖于合理的调度顺序、线程自身业务逻辑处理步骤
  - 线程申请失败、释放锁后，随机等待一段时间，再重新申请，降低活锁发生概率

数据库中的多粒度锁及死锁检测

- 多个并发用户对DB中不同粒度的数据对象互斥访问时，施加不同粒度的锁，
- E.g.1 The levels, starting from the coarsest (top) level are
  - database
  - area, e.g. extent, page in SQL Server
  - file
  - record, tuple
- E.g. 2 四级加锁DB– table- row- attribute
- 锁粒度越大，系统中可以被加锁的数据项就越少，事务并发执行度也越低，但同时系统开销也小；
- 当锁粒度比较小时，事务并发度高，但系统开销也较大

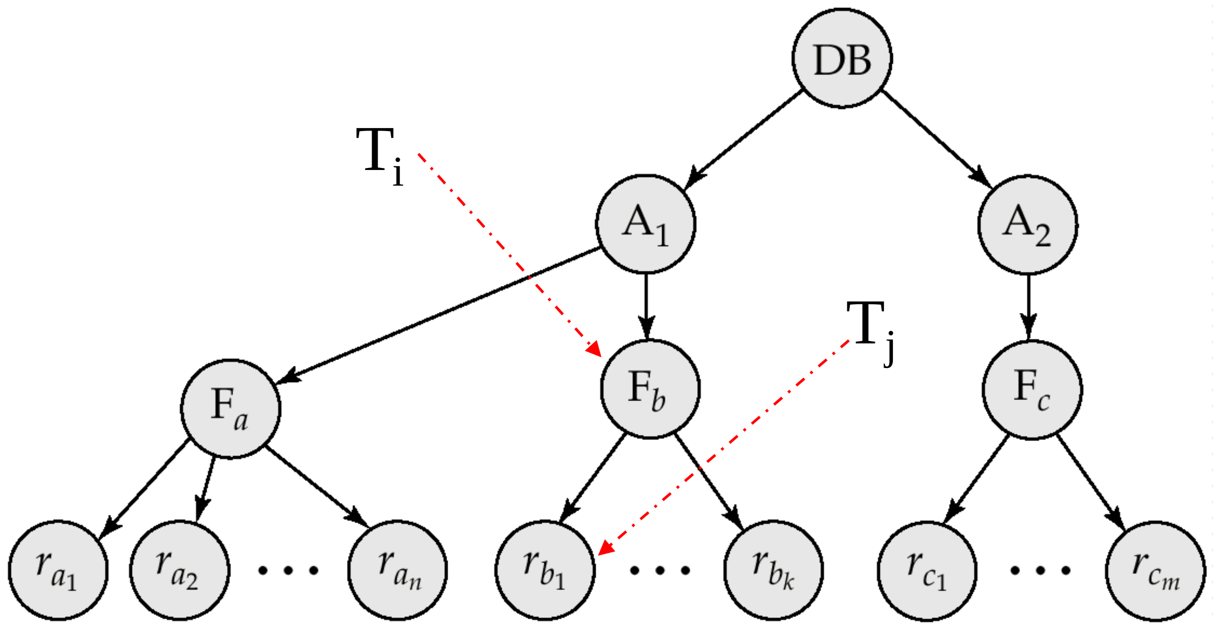


Fig. 15.15 Example of Granularity Hierarchy

自上而下四级加锁：DB– table- row- attribute

T_1

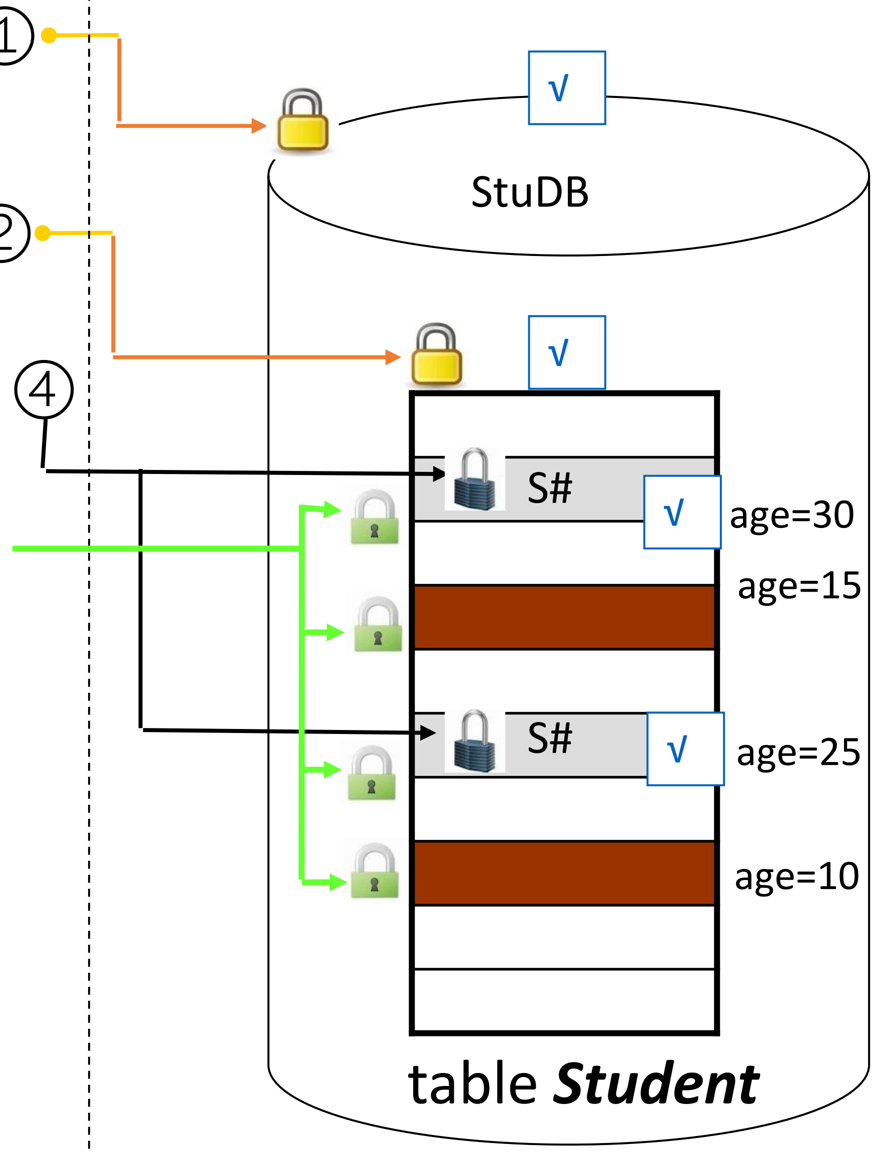
USE **StuDB** ①

Update ***Student*** ②

Set S#=S#+1 ④

Where **age>22** ③

unlock
unlock
unlock
unlock

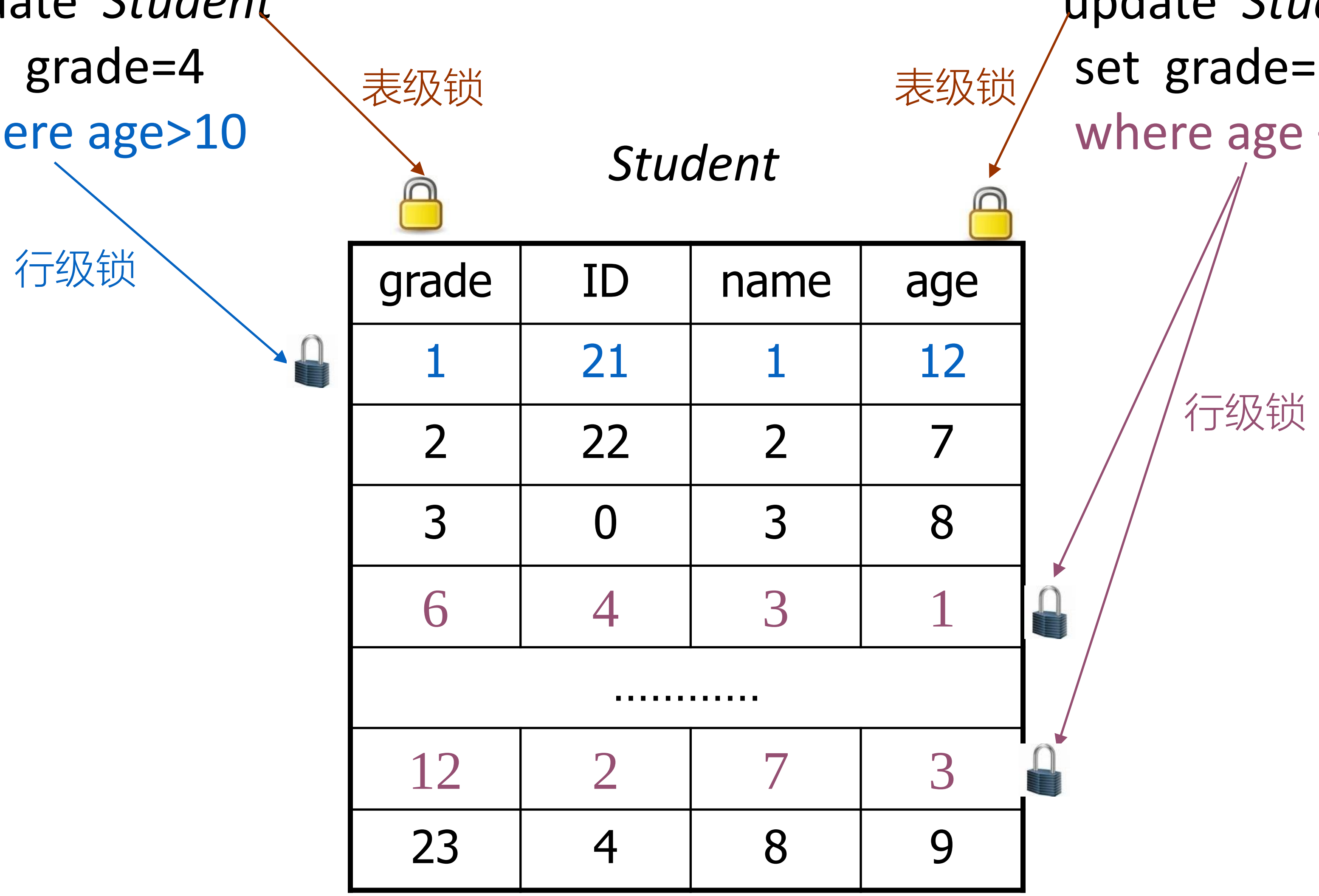


transaction T_i :

```
update  Student
 set  grade=4
 where age>10
```

transaction T_j :

```
update  Student
 set  grade=6
 where age < 5
```



Student

grade	ID	name	age
1	21	1	12
2	22	2	7
3	0	3	8
6	4	3	1
…………			
12	2	7	3
23	4	8	9

openGauss加锁及死锁检测

- 采用表级锁，提供各类SQL语句对于表的访问控制
- 根据访问控制的排他性，表级锁分为1到8级锁
  - 查询语句（select）需要获取1级锁
  - DML（insert，update，delete）语句需要获取3级锁
  - DDL（create，alter）语句需要获取8级锁
- 对定义在同一张表上的两个表级锁的两个持有者，如果他们所持有的表级锁的级别之和大于等于8，则这两个持有者会相互阻塞
  - E.g. 对同一张表，insert，alter，不允许修改表结构时，向表中插入数据
- 为表级锁的等待者维护等待队列，基于该等待队列，提供死锁检测
  - 在所有表级锁的等待队列中，寻找环路
  - 当发现环路，回滚环路中某个等待事务，使其退出队列，打破环路，解除死锁

30．进程P1和P2均包含并发执行的线程，部分伪代码描述如下所示。

```
//进程 P1
    int x=0;
    Thread1( )
    {    int a;
    a=1;    x+=1;
    }
    Thread2( )
    {    int a;
         a=2;  x+=2;
    }
```

```
//进程 P2
    int x=0;
    Thread3( )
    {    int a;
    a=x;    x+=3;
    }
    Thread4( )
    {    int b;
         b=x;  x+=4;
    }
```

1. 不同进程内的同名变量不需互斥写，如P1中的x和P2中的x

2. 同一进程内的不同线程，各自访问线程内局部变量，无需互斥，如A

3. 不同进程内的不同线程，各自访问各自进程内的全局变量，无需互斥，如D

4. 同一进程内不同线程，对定义在进程内的全局变量进行读-写、写-写，需要互斥，如C；但同时进行读，无需互斥

写thread1和thread2中的局部变量，不冲突

下列选项中，需要互斥执行的操作是

读P2中全局变量x，分别写thread3和thread4各自局部变量，不冲突

A．a=1与a=2　　B．a=x与b=x

c．x+=1与x+=2　　D．x+=1与x+=3

答案C：对P1中全局变量x，分别由thread1和thread1写，冲突

thread1对P1中全局变量x写，thread3对P2中的全局变量x写，不冲突

Example

- 如下代码片段展示了两个线程thread1、thread2所执行的函数，以及执行这2个线程前的初始化代码init。
- 如果需要保证这两个线程都一定可以终止运行，填写下面代码中的空出部分。
- 注意
  - thread1、thread2函数中，只能填写signal()、wait()，每个空位可填写的操作数量不限
- 原理
  - thread1、thread2对共享变量x分别进行乘二、乘三操作，需要保证x的值经过这两个线程的访问之后，x=12，从而thread1、thread2跳出while循环，可exit()结束
  - 设置同步、互斥信号量，使得两个线程从x=1开始，按照：

    x=x*2;　x=x*3; x=x*2，

  或者：　x=x*2;　x=x*2; x=x*3，　...

```
int x=1;
struct  sem a, b, c

void  init()
 {
   a->value=______;
   b->value=______;
   c->value=______;
  }

```

```
void thread1();
 {
    while (x !=12)   {
      ______________ ;
       x = x*2;
      _______________;
   }
   exit(0);
 }

```

```
void thread2();
 {
    while (x !=12)   {
      ______________ ;
       x = x*3;
      _______________;
   }
   exit(0);
 }

```

原理：

1. 12=2*2*3
2. 通过thread1和thread2间的同步、互斥信号量，控制thread1的循环体执行2次，thread2的循环体执行1次

Appendix 7-1 计算题类型

- 根据资源、进程间的请求关系，画出Resource-Allocation Graph
- 根据进程数目、进程资源需求、可用资源数目，判断
  - 是否发生死锁，导致死锁发生的最大资源数目，保证不发生死锁的最小资源数目
  - 原理：资源分配图，环路
- Deadlock Avoidance
  - 每类资源只有一个资源实例，利用Resource-Allocation Graph Algorithm判断有无死锁/系统是否安全
  - 资源有多个资源实例，利用Banker Algorithm判断系统是否安全?(无死锁?), 进程的资源请求是否be granted
- Deadlock Detecting
  - 每类资源只有一个资源实例，利用waiting graph判断有无死锁?
  - 资源可有多个资源实例时，利用deadlock detecting algorithm 判断死锁?

Thanks for your attention