

# Process Synchronization

## — Chapter 6

2022年10月



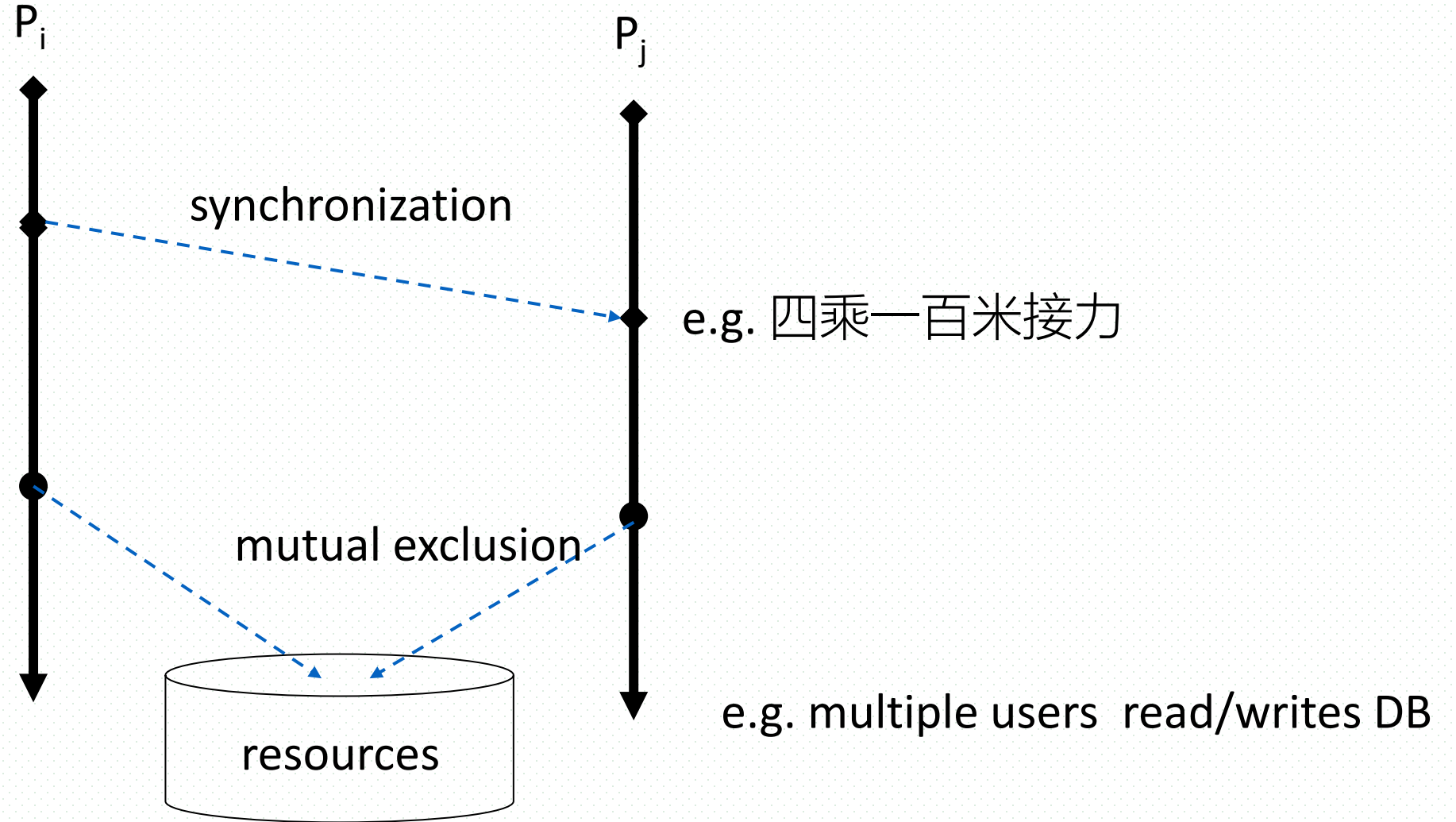
薛哲

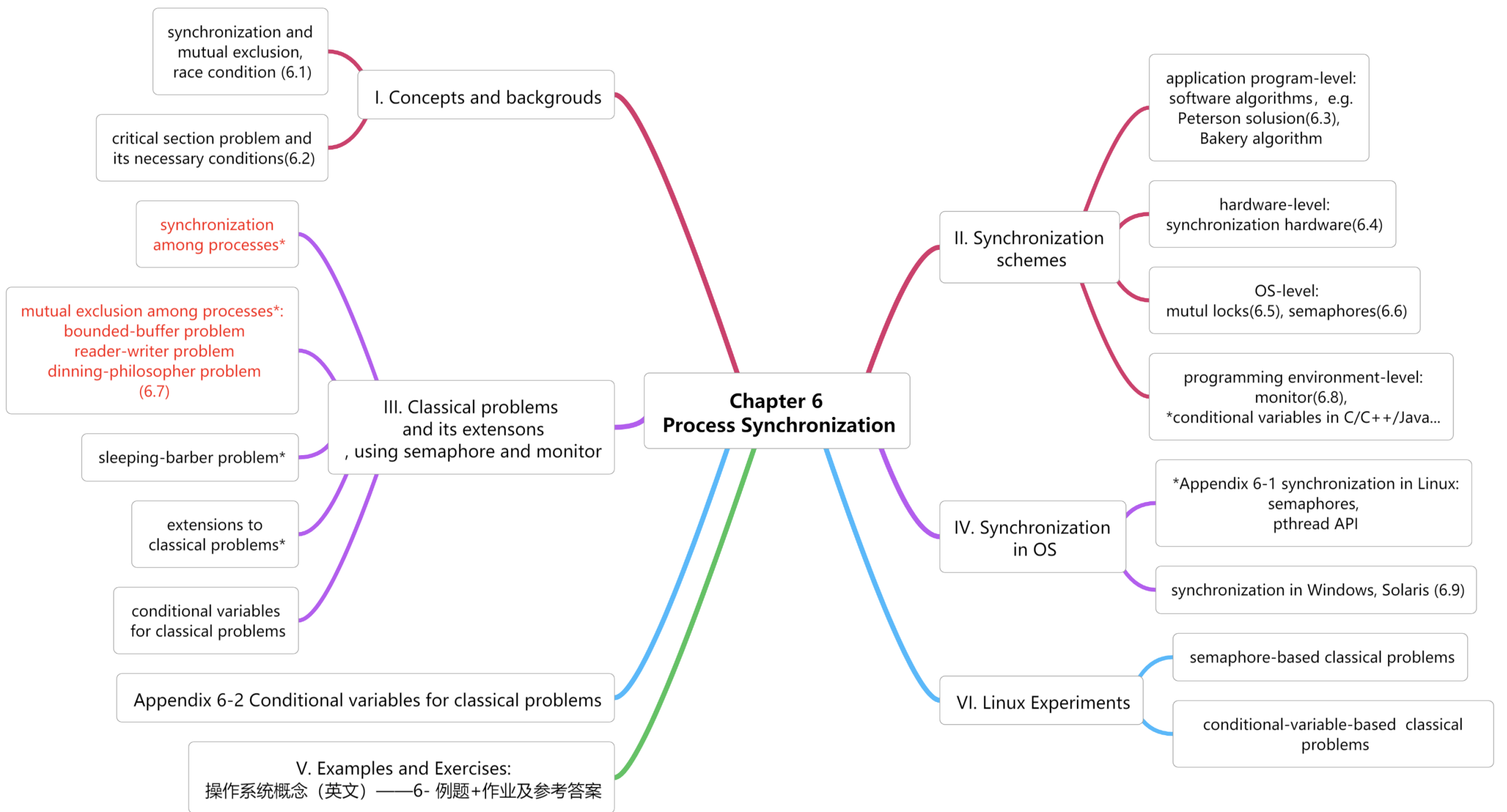
School of Computer Science (National Pilot Software Engineering School)



北京邮电大学

# Synchronization vs Mutual Exclusion





# Outline

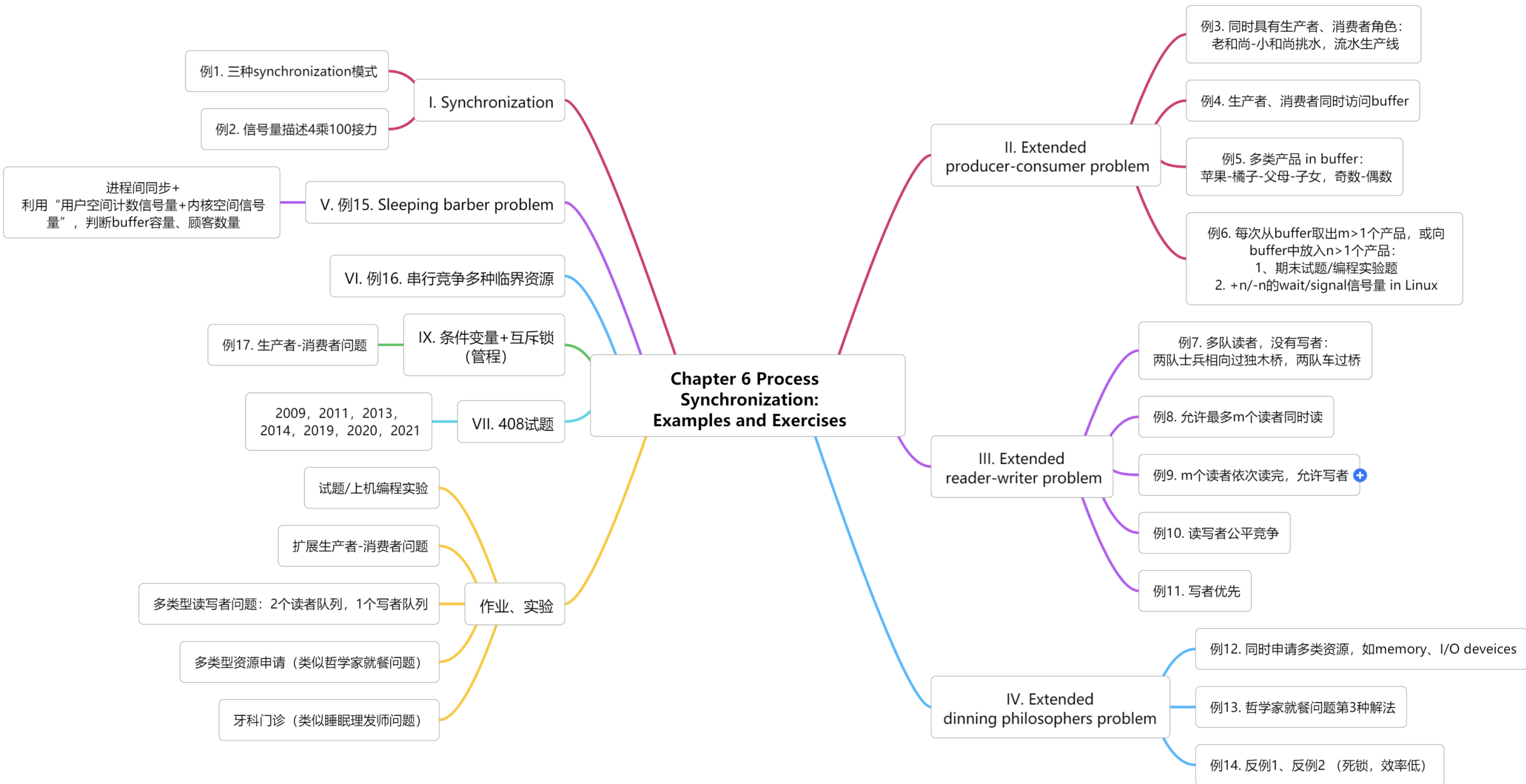
---

- Synchronization Concepts
  - why synchronization needed, race condition (§6.1)
  - critical section problem and its necessary conditions (§6.2)
    - process synchronization requirements can be described as the critical section problem
- Synchronizing mechanisms
  - application program-level
    - software algorithms (§6.3) , e.g. Peterson, Bakery
  - hardware-level
    - synchronization hardware (§6.4) (\*了解\*)
  - OS-level
    - semaphores (信号量, §6.5/6.6)
  - programming environments or system software-levels high-level language synchronization constructs, including
    - monitors (管程§6.7)

# Outline

---

- Classical problems of synchronization by semaphores (§6.6) (\*掌握\*)
  - ▣ bounded- buffer problem
  - ▣ readers-writers problem
  - ▣ dining-philosophers problem
- Case studies (§6.8) (\*知道\*)
  - ▣ synchronization in Linux
  - ▣ synchronization in Solaris2 & Windows XP
- *做题!!*



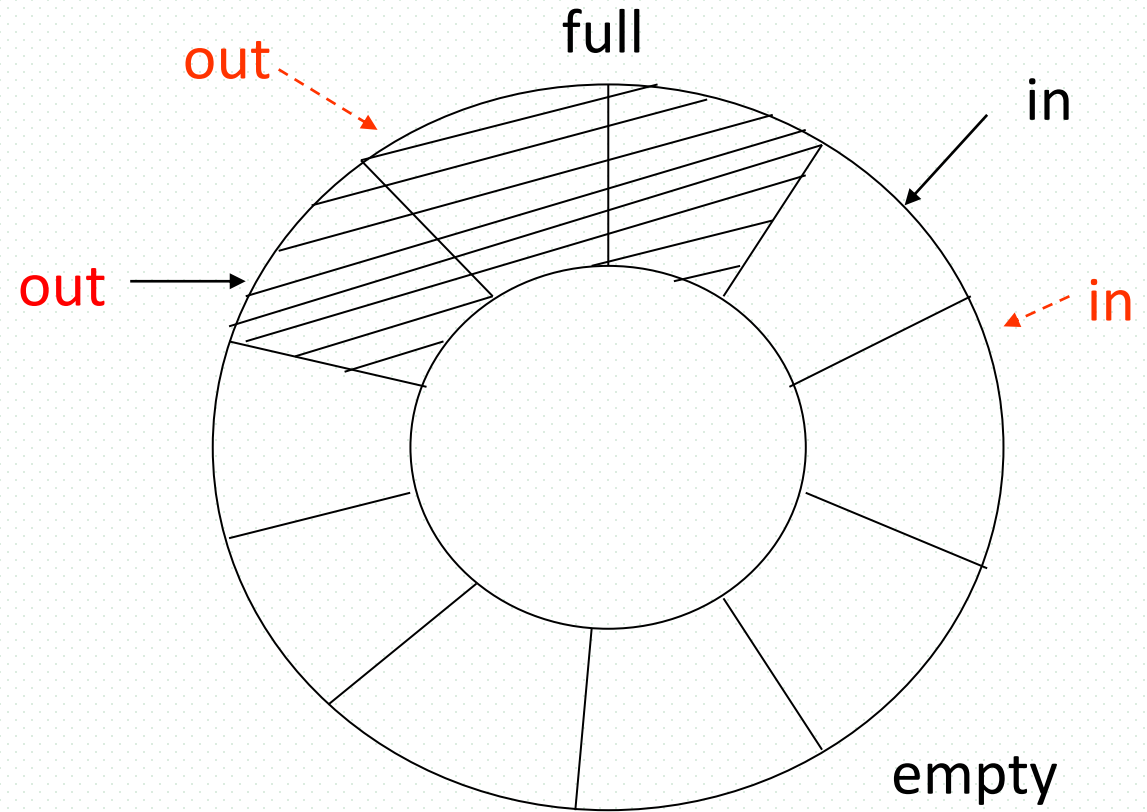
## 6.1 Background

---

- Cooperating processes vs independent processes in computers (§3.4, P96)
  - ▣ process cooperating in four conditions
- **Shared resources**
  - ▣ the resources (e.g., data, CPU, I/O ports, memory) that can be accessed by several cooperating processes **concurrently**
  - ▣ the shared resources cannot be used by several processes **simultaneously (or in parallel)**, only be used mutual **exclusively**(互斥).
- **Shared data** as shared resources
  - ▣ concurrent access to shared data in DBS may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
  - ▣ process synchronization is needed
  - ▣ e.g. transaction management in DBS (§6.9)

# Bounded-Buffer Problem

- Also known as Producer-consumer problem
- Synchronization among
  - producers
  - consumers
  - producers and consumers





# Bounded-Buffer Problem

---

- Shared-memory solution to bounded-buffer problem (Chapter 3)
  - ▣ allows at most  $n$  items in buffer at the same time
  - ▣ adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

- Codes for bounded-buffer problem

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0; int out = 0; int counter = 0;
```

# Bounded-Buffer Problem

## ■ Producer process

```
while (1) {  
    // producing an item in nextProduced */  
    while (counter == BUFFER_SIZE) ;  
    /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

## ■ Consumer process

```
while (1) {  
    while (counter == 0) ;  
    /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in nextConsumed  
    */  
}
```

# Bounded-Buffer Problem

- **counter** is shared data, the statements  
    `counter++;`  
    `counter--;`  
must be performed *atomically*
- **Atomic operation** means an operation that completes in its entirety without interruption. Otherwise, data inconsistency will occur.
- An Example: executing of `counter++` and `counter --` atomically
  - ▣ if the producer and the consumer access **counter** sequentially, data consistency can be maintained
  - ▣ e.g. `counter=5; counter ++; counter --;` the result remains `counter=5`
  - ▣ the statement “`count++`” and “`count--`” may be implemented in machine language as

```
register1 = counter  
register1 = register1 + 1  
counter  = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter  = register2
```

# Bounded-Buffer Problem

- Counter-example: executing of counter++ and counter-- in **interleaving** manner
  - both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved(交织)
  - interleaving depends upon how the producer and consumer processes are scheduled, and may result in **inconsistent data**
  - assume **counter** is initially 5. One interleaving of statements is

```
producer: register1 = counter      (register1 = 5)
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = counter      (register2 = 5)
consumer: register2 = register2 - 1 (register2 = 4)
producer: counter = register1      (counter = 6)
consumer: counter = register2      (counter = 4) .
```

- Problems
  - the value of count may be either 4 or 6, where the correct result should be 5

# Why synchronization needed: Race Condition

---

- Race condition(竞争条件)
  - ▣ the situation where several processes access and manipulate shared data concurrently
  - ▣ the final value of the shared data depends upon which process finishes **last**
- To prevent race conditions, concurrent processes must be synchronized, i.e. access shared resources in **mutual exclutory methods** (互斥方法)

## 6.2 The Critical-Section Problem

---

Some concepts related to Critical-Section Problem

- $n$  processes, all competing to use some shared data/resources
- **Critical section (临界区/段)**
  - ▣ each process has a code segment, called critical section, in which the shared data is accessed.
- **Critical resources**
  - ▣ resources accessed by critical section
- **Requirement** – ensure that
  - ▣ when one process is executing in its critical section to access critical resources, no other process is allowed to enter its critical section

# General Structure of Process

---

## ■ General Structure of Process

**do {**

.....

**entry** section(R) // apply

**critical** section(R) // use

**exit** section(R) // release

reminder section

**} while (1);**

# Solution to Critical-Section Problem

- **Necessary conditions** for correct solutions to critical-section problems
  - mutual exclusion
  - progress
  - bounded waiting
- **Mutual exclusion**
  - if process  $P_i$  is executing in its critical section specific to the resource  $R$ , then no other processes can be executing in their critical sections specific to  $R$ 
    - /\*互斥地执行临界区代码
  - what is mutual exclusion (互斥)
    - a programming technique ensuring that only one program or routine at a time can access some resources, such as a memory, an I/O port, or a file



# Mutual Exclusion

- Suppose Process  $P_1$  and  $P_2$  concurrently access resource  $R_1$  (e.g. main memory) and  $R_2$  (e.g. I/O devices), so each process has two critical sections responding to  $R_1$  and  $R_2$  respectively
- $P_1$  and  $P_2$  may be simultaneously stay in its critical sections, however access different resources, i.e.  $R_1$  and  $R_2$

$P_1$   
entry section( $R_1$ )  
critical section( $R_1$ )  
exit( $R_1$ )

entry section( $R_2$ )  
critical section( $R_2$ )  
exit( $R_2$ )

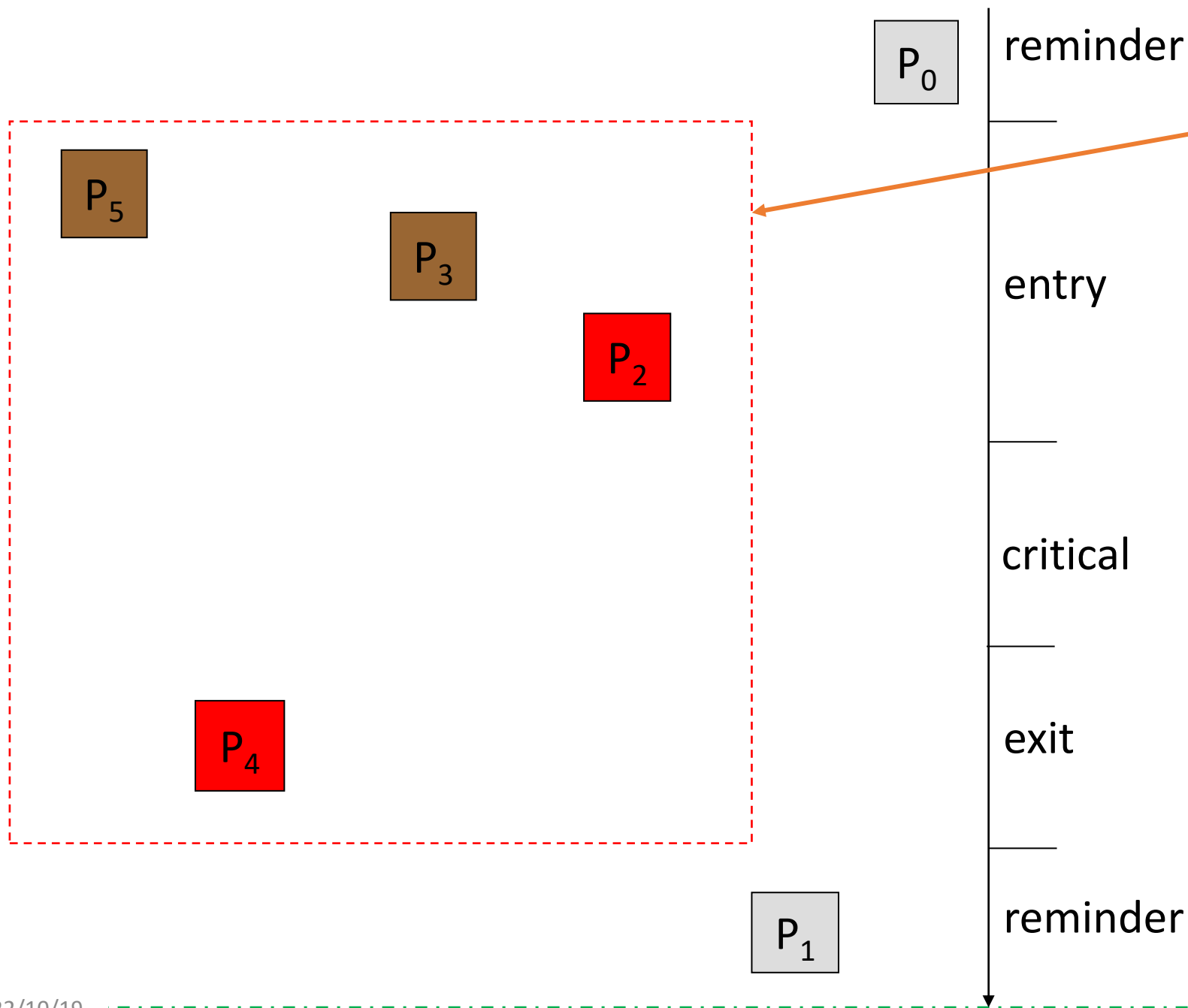
$P_2$   
entry section( $R_2$ )  
critical section( $R_2$ )  
exit( $R_2$ )

entry section( $R_1$ )  
critical section( $R_1$ )  
exit( $R_1$ )

# Progress

## ■ Progress

- if no process is executing in its critical section (共享资源空闲) and there exist some processes that wish to enter their critical section, then
  - (1) *only those processes that are not executing in their reminder sections can participate in the decision* on which will enter its critical section next, and  
// 挑选进入临界区的进程时，不考虑那些正在reminder 区中 执行的进程(已经使用完临界资源)；  
或：只考虑那些正处于entry(申请进入)、critical sections和exit sections 的进程  
临界区外的进程不应阻塞其它进程
  - (2) this selection *cannot be postponed indefinitely*  
// 挑选过程不应被无限期阻止



*participate in the decision* on which will enter its critical section next

左边方框中的四个进程  $P_2, P_3, P_4, P_5$  中,  $P_2, P_3, P_5$  处于进入段, 提出资源访问请求。 $P_4$  处于退出段, 正在释放资源。其它两个进程  $P_0, P_1$  处于剩余段, 与资源访问无关; 没有进程处于临界段, 资源空闲可用。

同步互斥机制判断挑选进程进入临界段访问互资源时, 只考虑进程  $P_2, P_3, P_4, P_5$ 。其中, 进程  $P_2, P_3, P_5$  有访问资源需求。进程  $P_4$  正在释放资源, 只有释放之后, 其它三个进程才有可能访问资源

# Bounded Waiting

## ■ Bound waiting

- ▣ a bound or limit must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - 当一个进程申请进入临界区并还未得到许可时，应当限制其它进程进入临界区的次数。否则，会导致该进程在临界区外无休止等待——进程在临界段内只应停留有限时间
- ▣ to avoid deadlock or starvation

## ■ Counter-example

- ▣ priority-based preemptive CPU scheduling, the process with lower priority may be starved

# 进程互斥使用临界资源的原则

---

- 在共享同一个临界资源的所有进程中，每次只允许一个进程处于它的临界段中
- 若有多个进程同时要求进入它的临界段，应在有限时间内让其中之一进入，而不应相互阻塞使得各进程都无法进入临界段
- 进程在临界段内只应停留有限时间
- 不要使需要进入临界段的进程无限制地等待在临界段之外
- 处于剩余段的进程不应阻止其他进程进入临界段

## 6.3 Peterson Solution

---

- In user mode, a software-based solution to process synchronization
- G.L.Peterson (1981) algorithm
  - ▣ for synchronizing of two processes
- Bakery Algorithm
  - ▣ critical section solution for  $n \geq 2$  processes

# Peterson Solution

---

- the structure of process  $P_i$ 
  - ▣ `enter_section ( i );` //判断是否可安全进入, 如不能, 等待  
`critical_section(i);`  
`exit_section ( i );` //退出时, 应允许其它进程进入临界段  
`remainder section (i);`

- For processes  $P_i$  and  $P_j$ , shared variables

- **boolean flag [2];**

- $/* \text{flag}[i] = \text{true} \Rightarrow P_i \text{ ready/want/apply to enter its critical section} */$

- initially **flag [i] = flag [j] = false.**

- **int turn;**  $/* \text{轮到谁?} */$

- turn == i**  $\Rightarrow P_i$  are allowed to enter its critical section

- initially **turn = 0**



■ Process  $P_i$

do {

enter\_section {

- flag [i] = true;**      /\*表明自身请求进入临界区\*/
- turn = j;**      /\* 假设先轮到对方进入临界区 \*/
- while (flag [j] and turn == j)**
- { do-nothing };**
- /\*对方正在临界区中,  $P_i$ 忙等待\*/

critical section

exit\_section {

- flag [i] = false**      /\*表明自身不再需要进入临界区\*/

remainder section

**} while (1);**

- 当 $P_i$ 和 $P_j$ 同时申请访问临界资源时，微观上先申请者(先置位turn)先进入临界区，假设：单CPU系统

$P_i$

flag [i]= true

turn = j;

false: flag [j] and turn == j

*critical section i*

$P_j$

flag [j]= true

turn = i;

true: flag [i] and turn == i

*busy waiting*

timeline



# Peterson Solution (cont.)

---

- Features

- ▣ **this algorithm is correct**

- meets requirements of **mutual exclusion, progress, bounded-waiting**;

- a solution to critical-section problem for two processes ()

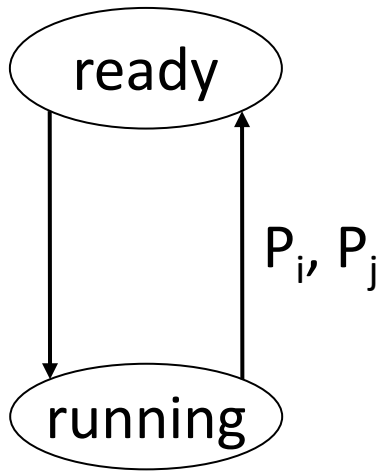
- ▣ combining the shared variables in algorithms 1 and 2.

- ▣ demerit

- **busy-waiting**

- $P_i$  alternatively stays in **running** and **ready** states

processes pass two states:



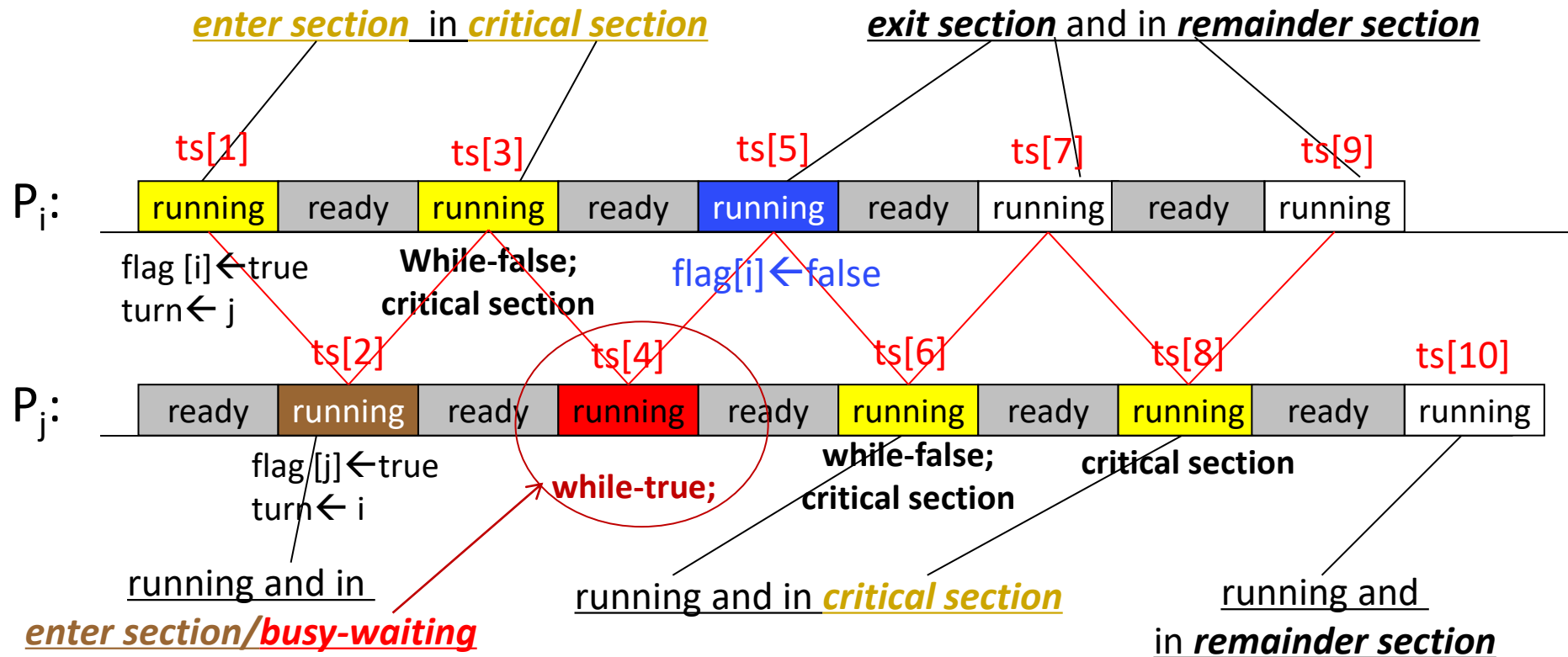
while (flag [j] and turn == j)

while (flag [i] and turn == i)

busy-waiting

It is assumed that:

1.  $P_i$  is in its critical section, and  $P_j$  is in the busy- waiting state
2. time-sharing scheduling:  
*time slots*  $ts[i]$  (e.g. 50ms) are assigned to  $P_i$  and  $P_j$  alternatively



# Bakery Algorithm

---

- An software solution for synchronize among **n processes**
- Principles
  - ▣ before entering its critical section, process receives a **number** (denoted for **ticket #**)
  - ▣ the holder of the smallest number enters the critical section.
  - ▣ if processes  $P_i$  and  $P_j$  receive the **same number**, if  $i < j$ , then  $P_i$  is served first; else  $P_j$  is served first.
  - ▣ the **numbering scheme** always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

# Bakery Algorithm (cont.)

- Notes : for lexicographical order (ticket #, process id# )
  - $(a,b) < (c,d)$  if  $a < c$  or if  $a = c$  and  $b < d$
  - **max**  $(a_0, \dots, a_{n-1})$  is a number  $k$ , such that  $k \geq a_i$  for  $i = 0, \dots, n-1$
- Shared data
  - **boolean choosing[n];**  
/\* choosing[i]=true:  $P_i$  begins/wants to apply for ticket\*/
  - **int number[n];** /\* number[i]: ticket# for  $P_i$ ,  
number[i]>0 表示 $P_i$ 需要进入临界区: \*/
  - data structures are initialized to **false** and **0** respectively

Process  $P_i$

*time slots* (e.g. 50ms) are assigned to processes alternatively

do {

    choosing[i] = true;      /\*begin to apply for ticket\*/

    number[i]

        = max(number[0], number[1], ..., number[n - 1]) + 1;

        /\* a ticket whose number is maximum is given ,  
        first come first given, in non-decreasing order \*/

    choosing[i] = false;      /\* end application \*/

for (j = 0; j < n; j++)      /\*考虑所有进程\*/

{

    while (choosing[j]) ;      /\*如有其它进程正在申请ticket, 等待 $P_j$ 申请完\*/

    while ((number[j] != 0) && {(number[j], j) < (number[i], i)});

        /\*如有票号小于自身的其它进程 $P_j$  正在或等待运行,  
        等待 $P_j$ 访问完\*/

}

$P_i$ 和 $P_j$ 两个进程，采用时间片轮转，申请票号。 $P_i$ 在第一个时间片内读取其他进程的最大number=5，在还没有完成通过max操作将number[i]设置为6的时候，CPU切换到 $P_j$ ， $P_j$ 读取到的其他进程最大值仍然是number=5，通过max获取的票号也将是6

```
critical section    /*进入临界区, 访问临界资源*/  
number[i] = 0;      /*丢弃使用过的票, 退出临界区*/  
remainder section  
} while (1);
```

#### ■ Features

- ❑ Bakery algorithm is correct
- ❑ demerit: busy-waiting, while



## 6.4 Synchronization Hardware (略)

- Three schemes
  - TestAndSet Instruction
  - Swap Instruction
  - Interrupt masking(中断屏蔽)
- Principles of TestAndSet Instruction and Swap Instructions
  - 每个临界资源设置一个布尔变量lock, 初值为false, lock用机器字实现
  - CPU提供专门的硬件指令TestAndSet 或Swap, 允许对一个字的内容进行检测和修正, 或交换两个字的内容
  - 硬件指令可以解决共享变量的完整性和正确性, 防止出现race condition
  - 进程在enter\_section中利用原子操作TestAndSet 指令或 Swap 指令测试lock, 察看临界资源是否空闲, 从而决定是否进入critical section

# Synchronization Hardware (cont.)

---

- Properties

- 简单、有效，特别适用于多处理机
- 缺点：忙等待
- for more details about **TestAndSet** Instruction and **Swap** Instruction, refer to Appendix 6-1

# Synchronization Hardware (cont.)

---

## ■ Interrupt masking based hardware synchronization

- ❑ “开关中断”指令保证用户态下的进程在临界区执行时不被其它进程中断,从而实现多个进程互斥访问临界区
- ❑ 进入临界区前执行: “关中断”指令
- ❑ 离开临界区后执行: 执行“开中断”指令
- ❑ 简单, 有效
- ❑ 较高的代价, 限制CPU并发能力(临界区大小)
- ❑ 不适用于多处理器

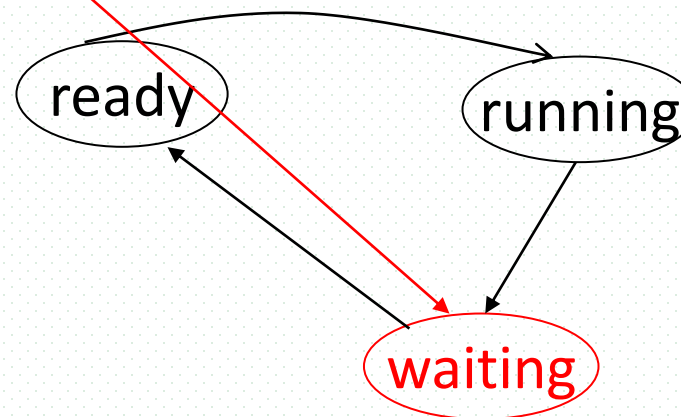
## 6.5 Semaphores (信号量)

### ■ Semaphore definitions

- synchronization tools that may avoid busy waiting
- provided by OS, working in kernel space
  - 由操作系统在内核空间提供的一种用于多个合作进程间同步与互斥的机制
- 信号量也被称为锁(lock)
- 但在大多数情况下，锁专指二元信号量，主要用于进程互斥

### ■ Representation of Semaphore S

- integer variable, *binary* or *non-binary*
- two standard operations modify S:  
    wait() and signal()  
    , originally called P() and V() [早期教科书中]



# Semaphore Types

- **Binary** semaphore (二元信号量)
  - ▣ integer value can range only between 0 and 1, be simpler to implement
    - 用于进程间互斥
  - ▣ binary semaphore 也称为mutex lock (互斥锁)
- **Counting** semaphore (计数、多元、一般信号量)
  - ▣ integer value can range over an unrestricted domain
    - 用于进程间同步、互斥
  - ▣ the value corresponds to the number of shared resources usable
- 同步：相互协作多个进程间，由于需要协调它们的工作而相互等待，或需要相互交换信息而产生的制约关系
- 互斥：进程间因相互竞争使用临界资源而产生的制约

# Semaphore Features

---

- Semaphore with busy waiting
  - ▣ processes pass only ready and running states
- Semaphore without busy waiting
  - ▣ processes pass three states: ready, running, **waiting**
  - ▣ when the resources represented by the semaphores are not available, the processes enter into the waiting state

# Semaphores with Busy Waiting

- Operations on semaphore S

- ▣ **initialization** (S)

- initialize S to a value

- ▣ **wait** (S) {

- while  $S \leq 0$  do no-op;**

- /\* the process that issues*

- this operation is **waiting**\*/*

- S--;**

- }**

- Wait() may results in process busy waiting

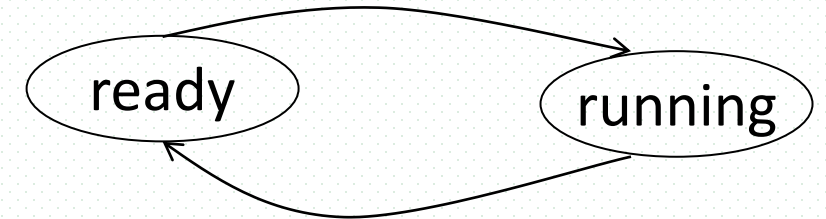
- Example. 自旋锁Spinlock in Linux

- ▣ **signal** (S) {

- S++;**

- }**

processes pass two states:



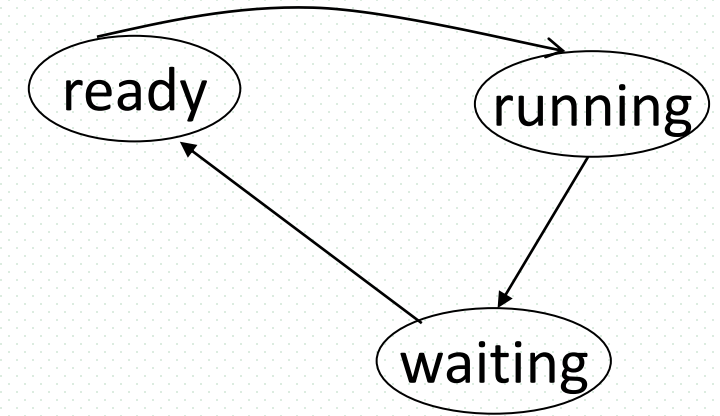
time-sharing scheduling,  
time slots (e.g. 50ms)  
are assigned to  
processes alternatively

# Semaphores without Busy Waiting

- Define a semaphore S as a C struct

```
typedef struct {  
    int    value;  
    struct process *list;  
           /*因s而阻塞/等待的进程队列*/  
} semaphore;
```

processes pass three states:  
ready, running, **waiting**



- Assume two simple operations

- ▣ **block**

suspends (挂起) the process that invokes it, i.e., the process are changed into ***blocked (i.e. waiting) state***

- ▣ **wakeup(P)**

resumes the execution of a blocked process **P**, i.e. from the **waiting** to **ready** state



# Semaphores Without Busy Waiting

- Operations on counting (non-binary) semaphore defined as

- **wait(S){**

```
S.value--; /*申请一个单位的资源/  
if (S.value < 0) { /*无可可用资源*/  
    add this process to S.list;  
    block();  
}  
}
```

- compared with busy-waiting wait() 

- **signal(S) {**

```
S.value++; /*释放一个单位的资源*/  
if (S.value <= 0) /*仍有被阻塞进程*/  
    { remove a process P from S.list;  
      wakeup(P); /*唤醒被阻塞进程*/  
    }  
}
```

- compared with busy-waiting signal() 

# Semaphores Without Busy Waiting

- Operations on binary semaphore defined as

- **waitB(S){**

```
    if S.value=1
    then S.value=0;
    else { add this process to S.L
           block();
         }
}
```

- **signalB(S) {**

```
    if S.L is empty    / there is no process
                        blocked and S.value is 0
    then S.value=1
    else { remove a process P from S.L;
           wakeup(P) /*唤醒被阻塞进程*/
        }
}
```

- compared with busy-waiting wait() 

- A counting semaphore S can be implemented as a binary semaphore and an integer variable

# 信号量S物理意义

- 针对无忙等待的多元信号量
  - processes pass three states, i.e. ready, running and waiting
- S对应于临界/共享资源
  - $S (> 0)$  ——可用共享资源的数目
  - $S (= < 0)$  ——无共享资源可用,  $|S|$ 表示因请求使用S而被等待或阻塞的进程 (注: 针对不带有忙等待的信号量)
- wait()
  - 请求分配一个单位的资源S给执行wait操作的进程
- signal()
  - 进程释放一个单位的资源S
- Wait and signal in practical OS
  - in Windows, WaitForSingleObject()  
ReleaseSemaphore()
  - In Linux, down(), up()

说明:

1. Linux提供了可以对信号量进行加、减  $n(n>1)$  的wait/signal操作
2. 允许进程一次申请、释放多个资源实例, 简化了基于信号量的编程

# Example

- Considering  $n=10$  processes, which mutual exclusively use the resource type  $R$  of  $m=6$  instances
- A semaphore  $S$  is designed to synchronize these processes. The maximum and minimal values of  $S$  are \_\_\_\_\_ respectively
  - A. 10, -4      B. 6, -4
  - C. 10, 6      D. 6, -10

value range of  $S$ :  $[m-n, m]$ , e.g.  $n=10, m=6$

# 信号量三种用法

## ■ 1.资源互斥使用

- 资源只有1个实例，各个进程通过二元互斥信号量mutex互斥地进入临界区，使用资源
  - mutex：代表资源的控制权，初值为1
- e.g. 生产者-消费者问题中的buffer

## ■ 2.资源竞争使用

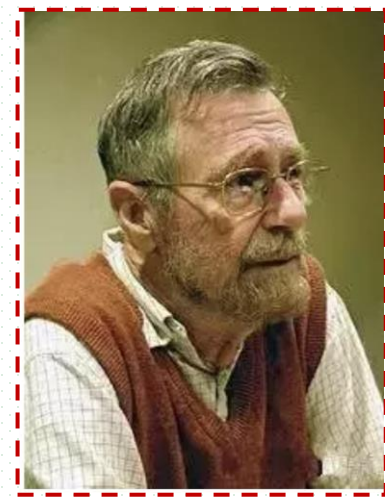
- 资源有多个实例，允许多个进程竞争使用资源
- 多元信号量表示：1)资源可用数目，2) 资源使用权
- e.g. **empty**, **full** in the bounded-buffer problem

## ■ 3.进程间同步

- 进程间的执行步骤需要有先后顺序关系
- 同步二元信号量sync，初值为0

# P/V Operations on Semaphores

- 信号量的概念是由荷兰计算机科学家E.W.Dijkstra在1962、1963年提出的
  - 当时，Dijkstra和他的团队正在为Electrologica X8开发一款操作系统（该系统后来被称为THE多道程序系统），他提出了用P、V操作实现进程的同步与互斥
  - P和V分别取自荷兰语的测试(**P**roberen)和增加(**V**erhogen)的首字母
  - P、V操作也称为wait、signal操作
  - 最初，Dijkstra提出的是二元信号量(互斥)，后来推广到一般信号量(多值，同步)
- Edsger Wybe Dijkstra<sup>9</sup> (1930-2002)
  - 出生于荷兰，美国Texas大学计算机科学和数学教授
  - Dijkstra单源最短路径算法的发明者
  - Turing Prize Winner (1972)
    - 因ALGOL第二代编程语言，获得图灵奖
  - “**Go To** Statement Considered Harmful”(EWD215): 被广为传颂的经典之作



# Usage in Mutual Exclusion

- **Mutual exclusion** for  $n$  processes to concurrently access resource  $R$

- ▣ shared data/resource  $R$  by  $n$  processes

binary semaphore **mutex**; /\* corresponding to shared data/resource,

initially  $\text{mutex} = 1$

- ▣ Process  $P_i$ :

do {

`wait(mutex);`

    critical section

`signal(mutex);`

    remainder section

} while (1);

针对共享资源R,  
设立mutex

# Usage in Synchronization

## ■ Synchronizing of $n$ processes

- concurrent process  $P_1$  with statement  $S_1$  and process  $P_2$  with statement  $S_2$
- requirement:  $S_2$  should be executed only after  $S_1$  has completed
- $P_1$  and  $P_2$  share semaphore **synch** (表示S1是否已经执行完) which is initialized to 0
- $P_1$ :  $M_1$ ;  
     $S_1$ ;  
    signal(synch);
- $P_2$ :  $M_2$ ;  
        wait(synch);  
         $S_2$



# Example One

---

- Consider processes P1, P2, and P3 in the following figures.
  - ▣ in Fig. (a), only after P1 ends, can P2 begin. And only after P2 ends, can P3 begin;
  - ▣ in Fig. (b), only after both P1 and P2 end, can P3 begin;
  - ▣ in Fig. (c), only after P1 ends, can both P2 and P3 begin
- Define semaphores, and synchronize the execution of P1, P2, and P3 by wait() and signal() on these semaphores. Assuming the main bodies of P1, P2, and P3 are as follows

P1:	P2:	P3:
begin	begin	begin
S1;	S3;	S5;
S2;	S4;	S6;
end	end	end

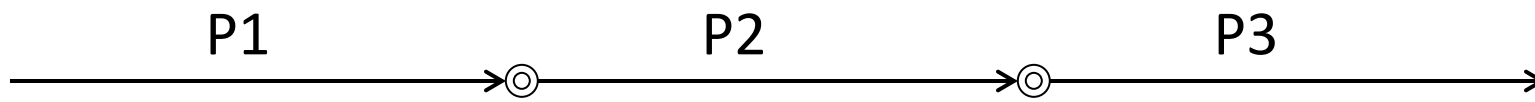


Fig. (a)

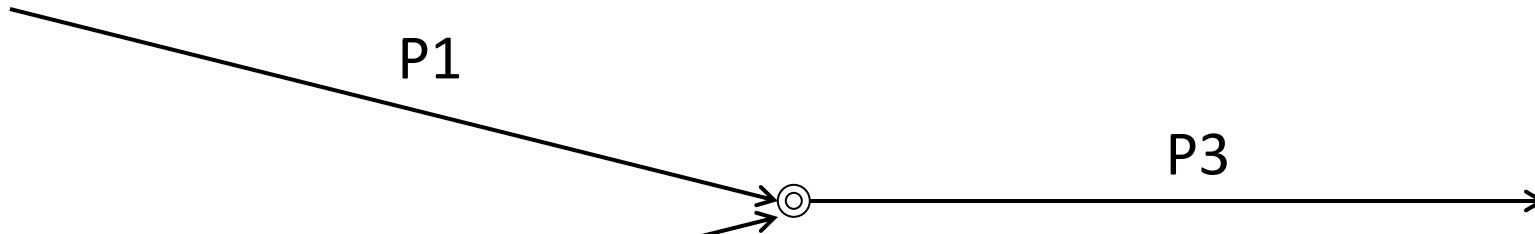


Fig. (b)

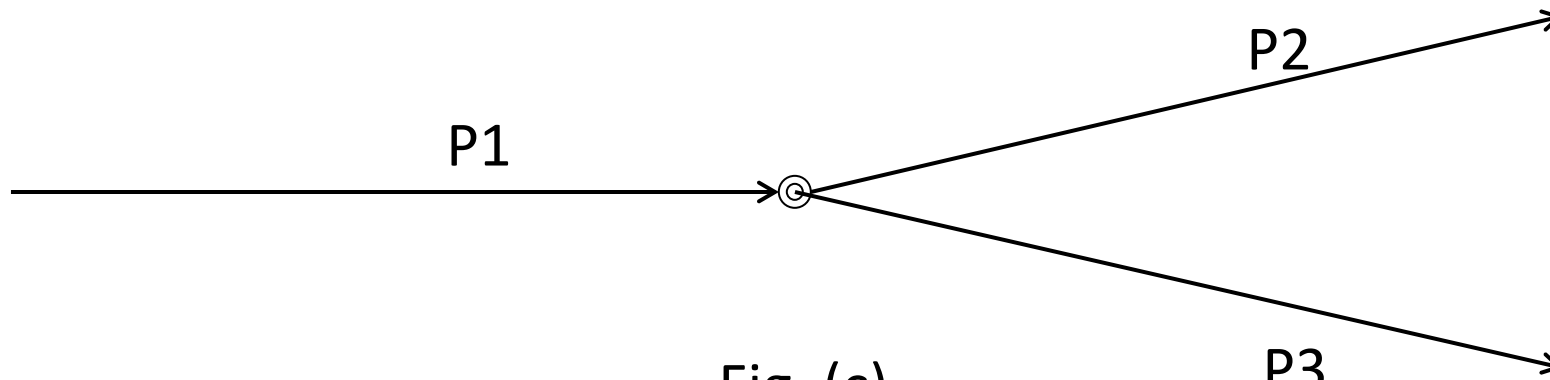


Fig. (c)

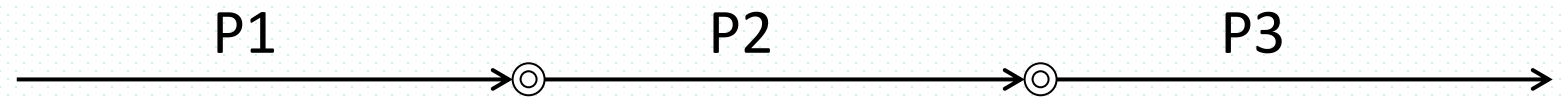
# Example One

- Initially, defining and initializing **binary** semaphores S1 and S2 as:

- (binary) semaphores sync1, sync2;

- sync1:=0, sync2:=0;

- For processes in Fig. (a)



**P1:**

begin

S1;

S2;

signal(sync1);

end

**P2:**

begin

wait(sync1);

S3;

S4;

signal(sync2);

end;

**P3:**

begin

wait(sync2);

S5;

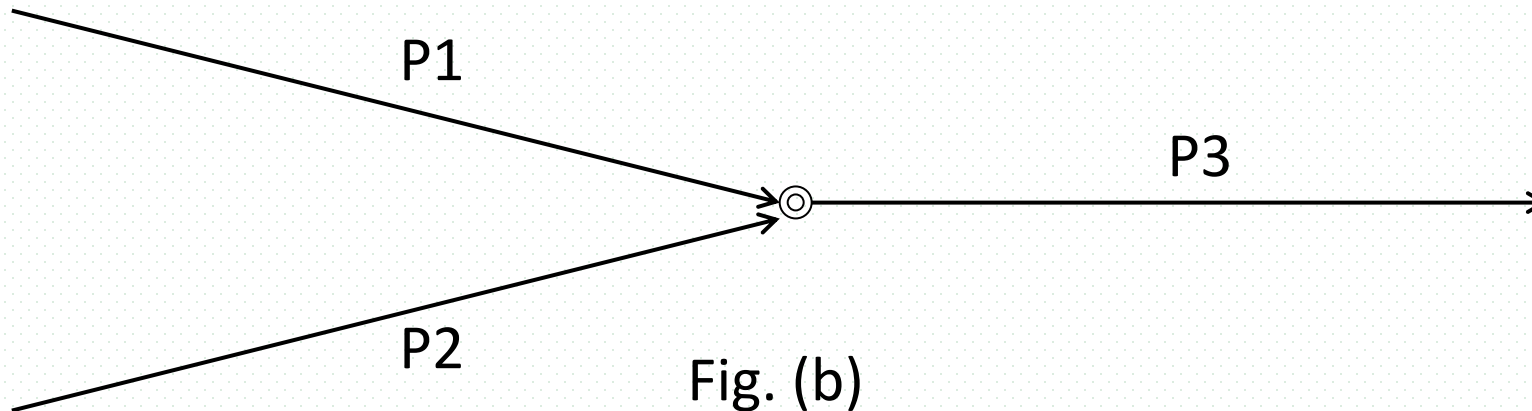
S6;

end

# Example One

- For processes in Fig. (b)

P1:	P2:	P3:
begin	begin	begin
S1;	S3;	wait(sync1);
S2;	S4;;	wait(sync2);
signal(sync1);	signal(sync2);	S5;
end	end	S6:
		end



# Example One

- For processes in Fig. (C)

**P1:**  
begin  
S1;  
S2;  
signal(sync1);  
signal(sync2);  
end

**P2:**  
begin  
wait(sync1);  
S3;  
S4;  
end

**P3:**  
begin  
wait(sync2);  
S5;  
S6;  
end

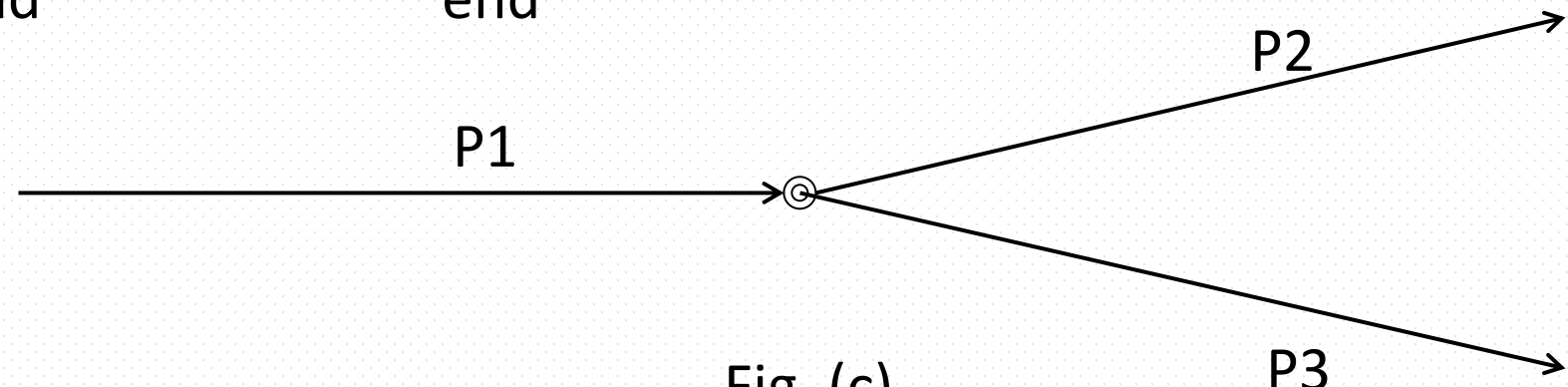


Fig. (c)

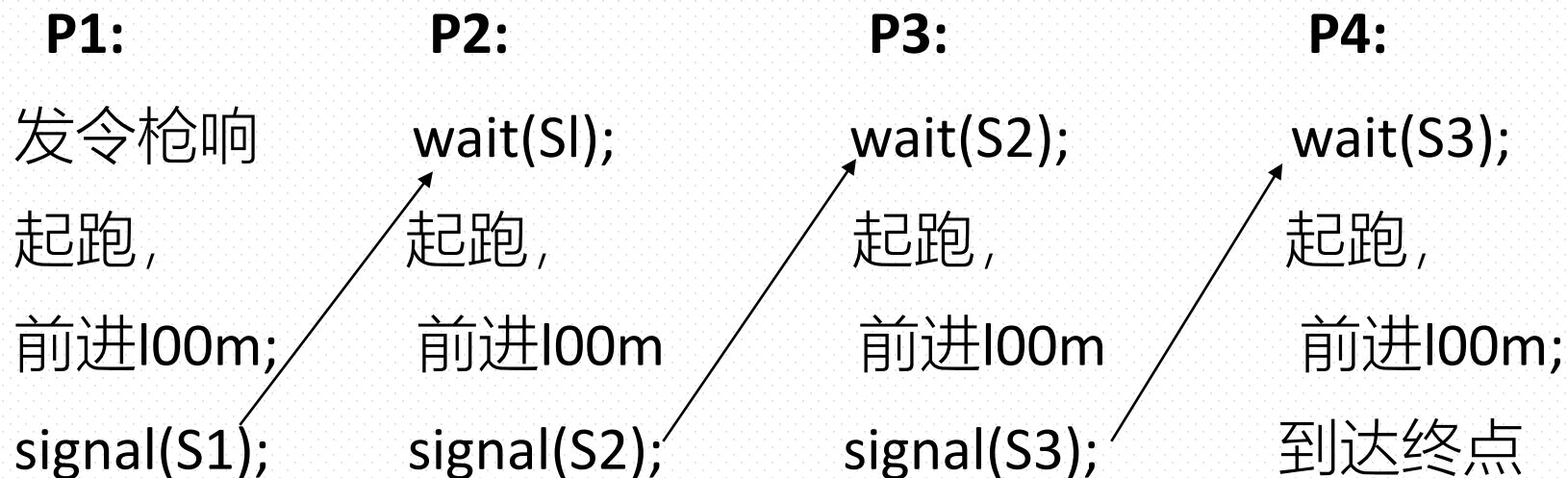
## Example Two

- 用信号量描述4 乘100 米接力过程

- (binary) semaphores S1, S2, S3;

- /\*  $s_i$  代表 第[i]棒是否跑完, 第[i+1]棒是否可开始,  
i= 1, 2, 3

- initially, S1=S2=S3=0



# Deadlock

---

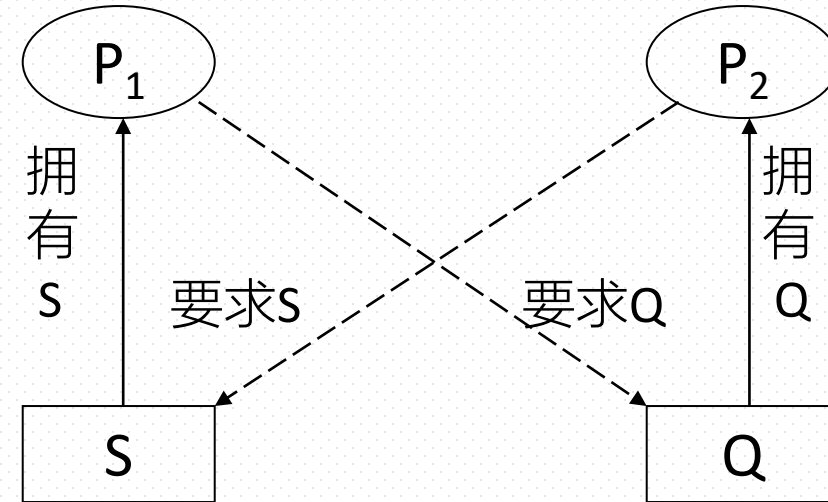
## ■ Definition

- ❑ ***states/situations*** related to more than one process in the system , *in which the processes are waiting indefinitely for an event or events (e.g., resources acquisition and release) that can be caused by only one of the waiting processes.*
- ❑ 系统内多个并发进程彼此互相等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源，造成各个进程都想得到资源而又都得不到资源，各并发进程不能继续执行的状态.

# Deadlock Example

- Let  $S$  and  $Q$  be two semaphores initialized to 1

$P_1$	$P_2$
wait(S);	
	wait(Q);
wait(Q);	
	wait(S);
⋮	⋮
signal(S);	
	signal(Q);
signal(Q);	
	signal(S);



- Semaphore  $S$  for resource  $S$ ,  
Semaphore  $Q$  for resource  $Q$



# Starvation

---

- Starvation

- ❑ *states /situations related to a process (or some processes)*, in which the process may never be removed from the semaphore queue in which it is suspended, or a process waits indefinitely within the semaphore.
- ❑ *infinite* waiting or blocking

- Another example of process starvation

- ❑ the process with lower priority may be starved in priority based CPU scheduling

# 不允许用户应用程序在用户态下直接对信号量操作

- 信号量是OS提供的一种内核空间进程同步互斥机制
- 在用户态下运行的用户应用程序只能通过wait()、signal()等系统调用，**间接地**操纵修改信号量
- **不允许用户应用程序直接在用户态下修改信号量**

- 反例。 defining semaphore S

```
main{  
    ...  
    S := S - 1  
    if S > 0 then ...  
        else ...  
}
```

# Priority Inversion (优先级反转)

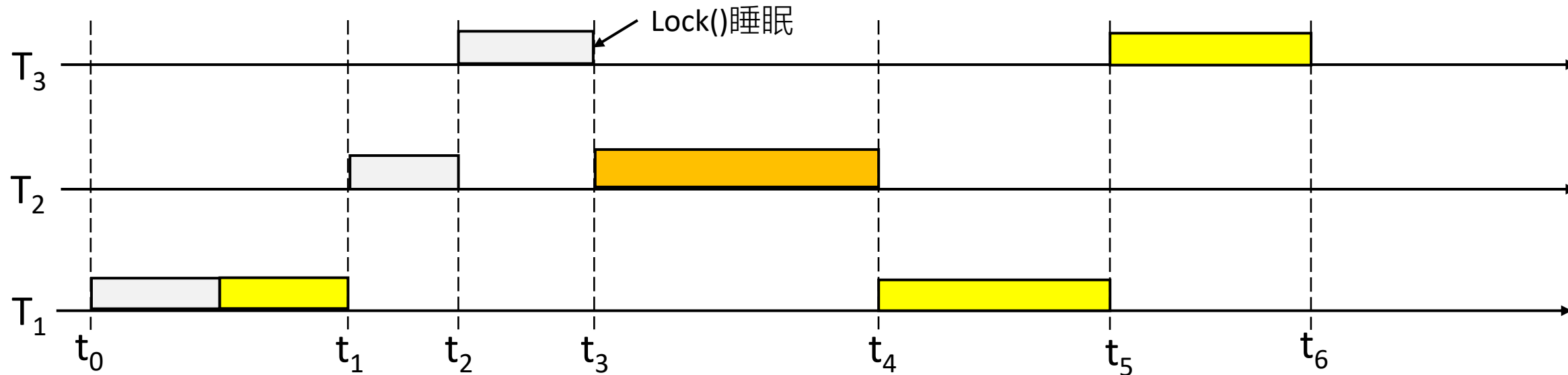
- Priority Inversion

- ▣ 由于线程间同步互斥，导致线程执行顺序违反预设的优先级规定的执行顺序
- 实时系统中，优先级反转导致被调度的线程无法按照优先级顺序执行，某些线程可能无法在deadline之前，完成任务，造成严重后果

- 假设：单核实时系统中

- ▣ 线程 $T_1$ 、 $T_2$ 、 $T_3$ 优先级由低到高，OS采用抢占式调度策略
  - ▣ 低优先级线程 $T_1$ 与高优先级线程 $T_3$ 竞争同一把锁

正常执行 临界区 优先级反转



- ❑ 低优先级 $T_1$ 创建、执行，获得锁，进入临界区， $[t_0, t_1]$
- ❑ 中优先级 $T_2$ 创建，抢占临界区中 $T_1$ 的CPU，执行， $[t_1, t_2]$
- ❑ 高优先级 $T_3$ 创建，抢占 $T_2$ ，执行， $[t_2, t_3]$ ； $t_3$ 时刻申请加锁，失败，进入睡眠等待
- ❑ 与 $T_1$ 无资源竞争关系的 $T_2$ 获得CPU，执行，优先级反转，直至结束， $[t_3, t_4]$
- ❑  $T_1$ 获得CPU，在临界区继续执行，直至退出临界区，释放锁并激活 $T_3$ ，结束， $[t_4, t_5]$
- ❑  $T_3$ 获得锁，进入临界区中执行，直至退出临界区，释放锁，结束， $[t_4, t_5]$

# Priority Inversion

## ■ 解决方案1

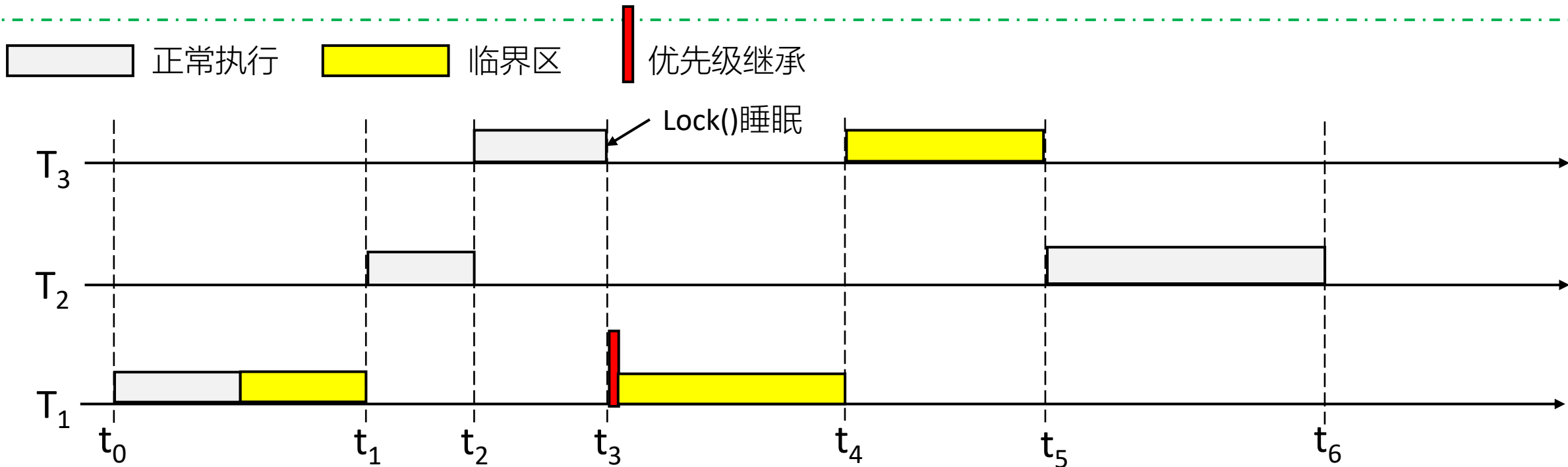
不可抢占临界区协议(Non-preemptive Critical section Protocol, NCP)

- 不允许线程在临界区中被抢占
- e.g. 低优先级 $T_1$ 进入临界区,  $[t_0, t_1]$ , 不允许后创建的高优先级 $T_2$ 、 $T_3$ 抢占,  $T_1$ 可以在临界区中正常执行完

## ■ 解决方案2

优先级继承协议(Priority Inheritance Protocol, PIP)

- 当高优先级线程申请锁得不到满足, 而处于阻塞态时, 使锁的持有者继承其优先级, 避免低优先级线程在临界区中被其它线程抢占CPU资源



- ❑ 低优先级 $T_1$ 创建、执行，获得锁，进入临界区， $[t_0, t_1]$
- ❑ 中优先级 $T_2$ 创建，抢占临界区中 $T_1$ 的CPU，执行， $[t_1, t_2]$
- ❑ 高优先级 $T_3$ 创建，抢占 $T_2$ ，执行， $[t_2, t_3]$ ； $t_3$ 时刻申请加锁，失败，进入睡眠等待，同时将高优先级传递给临界区中的 $T_1$ ；
- ❑  $T_1$ 继承高优先级，继续在临界区中执行，直至退出临界区，释放锁，结束， $[t_3, t_4]$
- ❑  $T_3$ 获得锁，进入临界区中执行，直至退出临界区，释放锁，结束， $[t_4, t_5]$
- ❑ 与资源竞争无关的 $T_2$ 获得CPU，执行，直至结束， $[t_5, t_6]$

$T_1$	正常执行区：50ms	临界区执行：50ms	正常执行区：30ms
-------	------------	------------	------------

$T_2$	正常执行区：100ms
-------	-------------

正常执行区：30ms
------------

$T_3$	正常执行区：50ms	临界区执行：50ms
-------	------------	------------

$T_1$	正常执行区：30ms	临界区执行：50ms	正常执行区：40ms
-------	------------	------------	------------

$T_2$	正常执行区：80ms
-------	------------

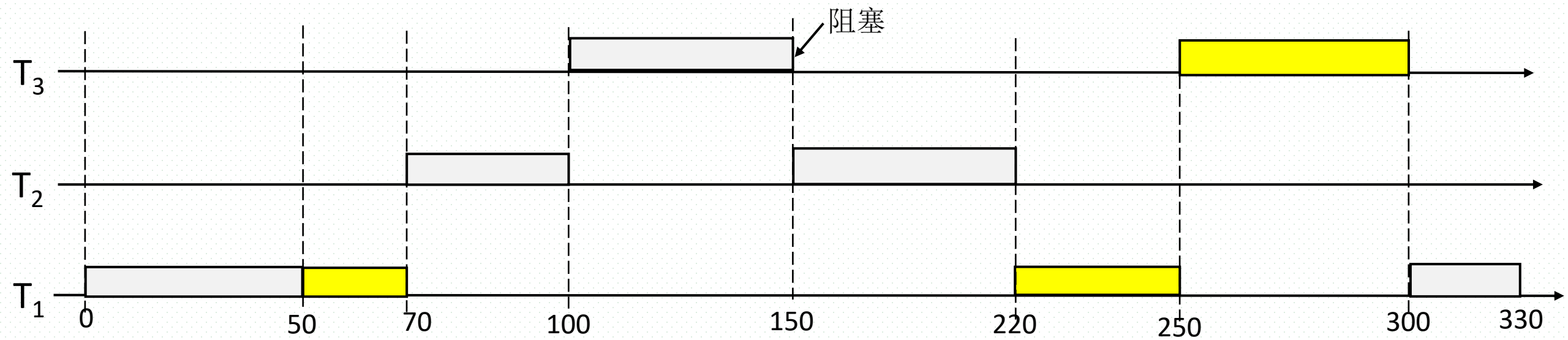
$T_3$	正常执行区：50ms	临界区执行：30ms
-------	------------	------------

$T_3$  正常执行区: 50ms 临界区执行: 50ms

$T_2$  正常执行区: 100ms

$T_1$  正常执行区: 50ms 临界区执行: 50ms 正常执行区: 30ms

$T_1$ 、 $T_2$ 、 $T_3$ 进入系统的时间分别为:  
0ms, 70ms, 100ms

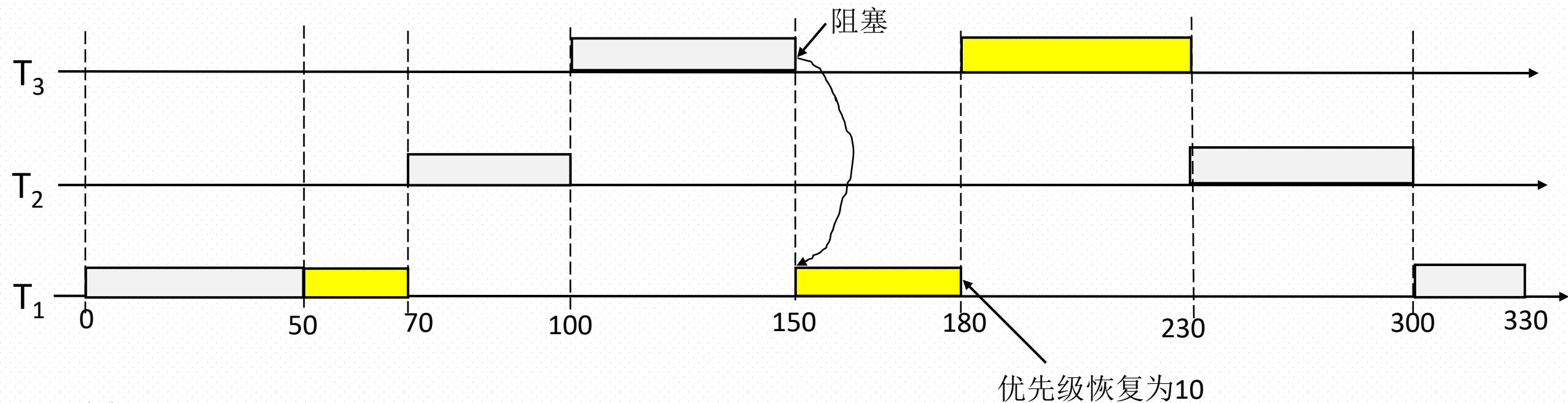


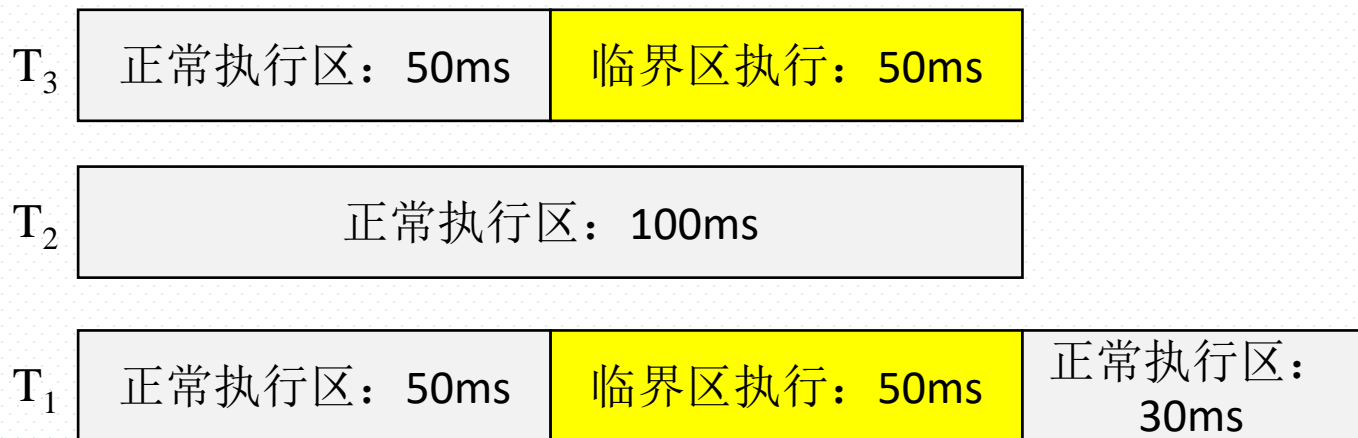




$T_1$ 、 $T_2$ 、 $T_3$ 进入系统的时间分别为:  
0ms, 70ms, 100ms

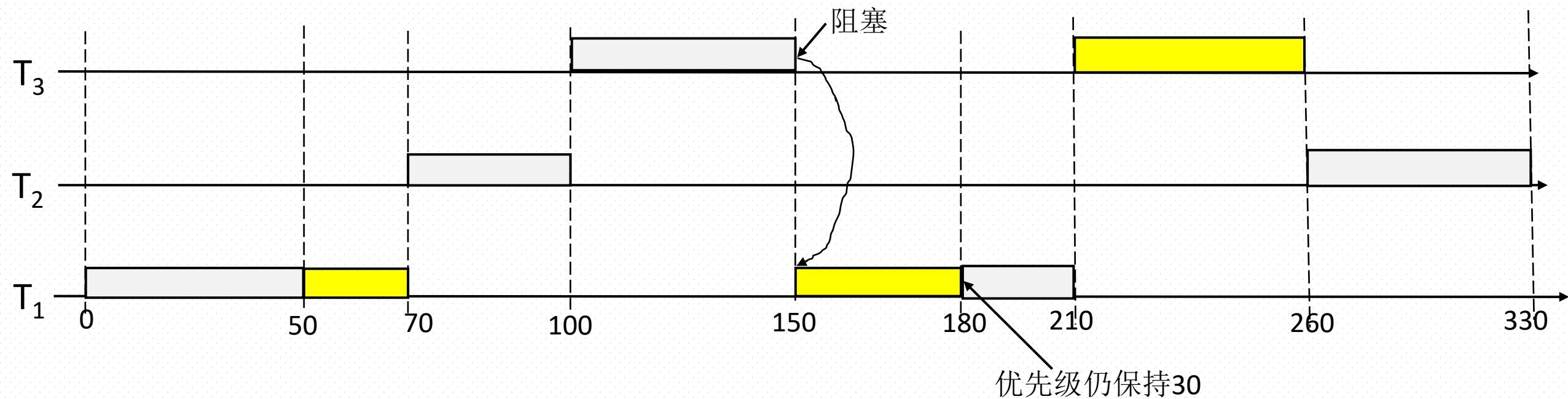
优先级继承协议: 答案1  
 $T_3$ 在进入阻塞前, 将其高优先级30传递给处于临界区中的 $T_1$ , 使得 $T_1$ 继续执行临界区代码;  $T_1$ 退出临界区后, 其优先级恢复为原有值10

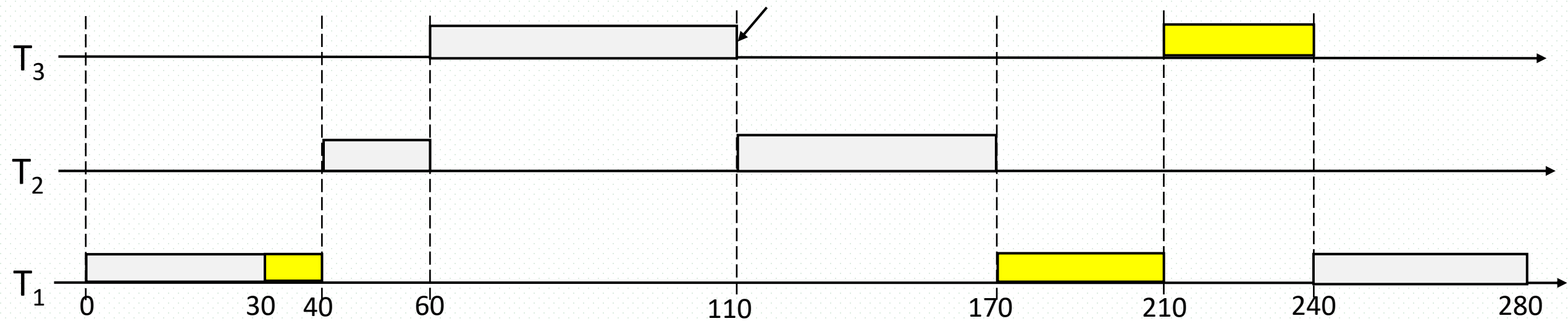
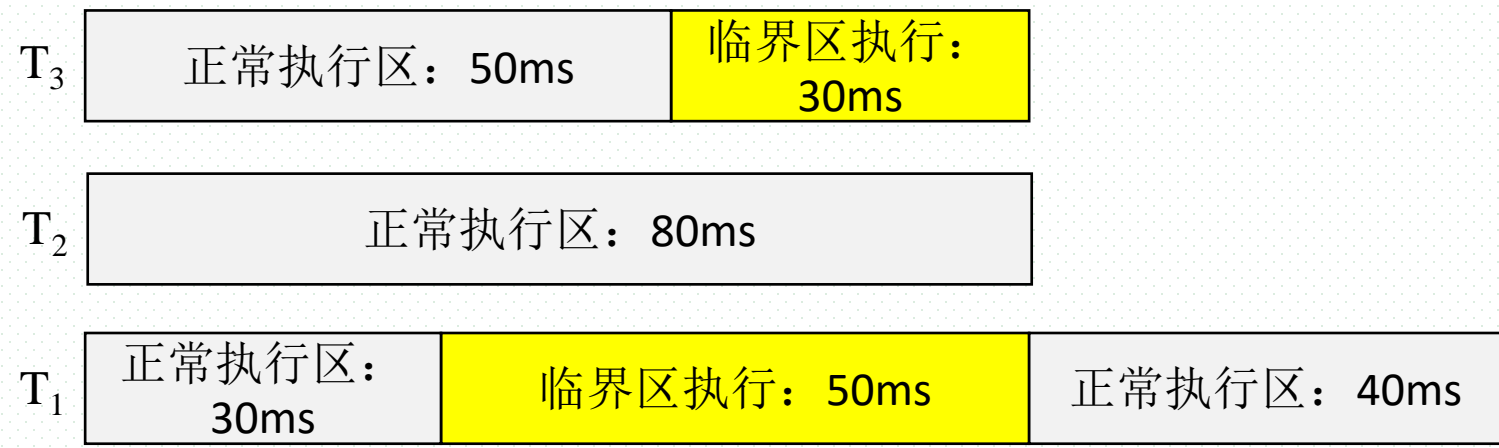




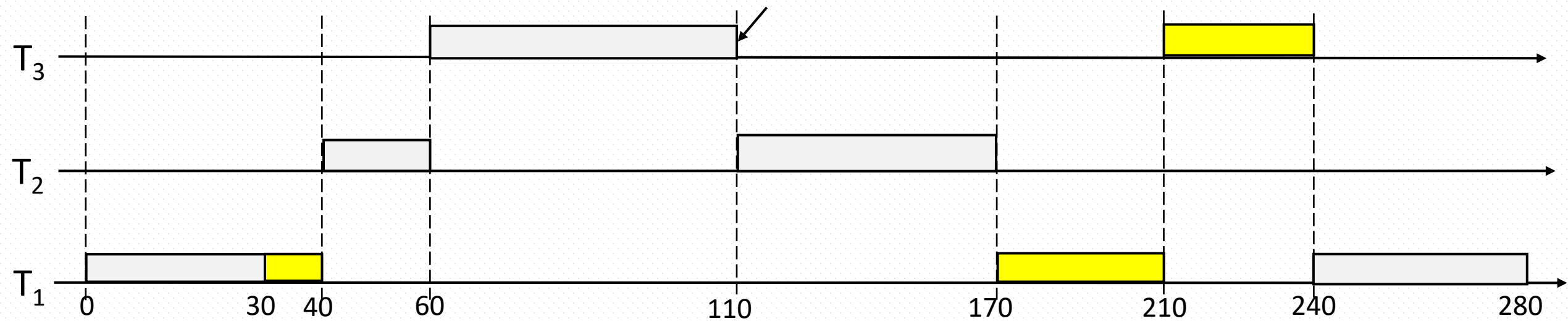
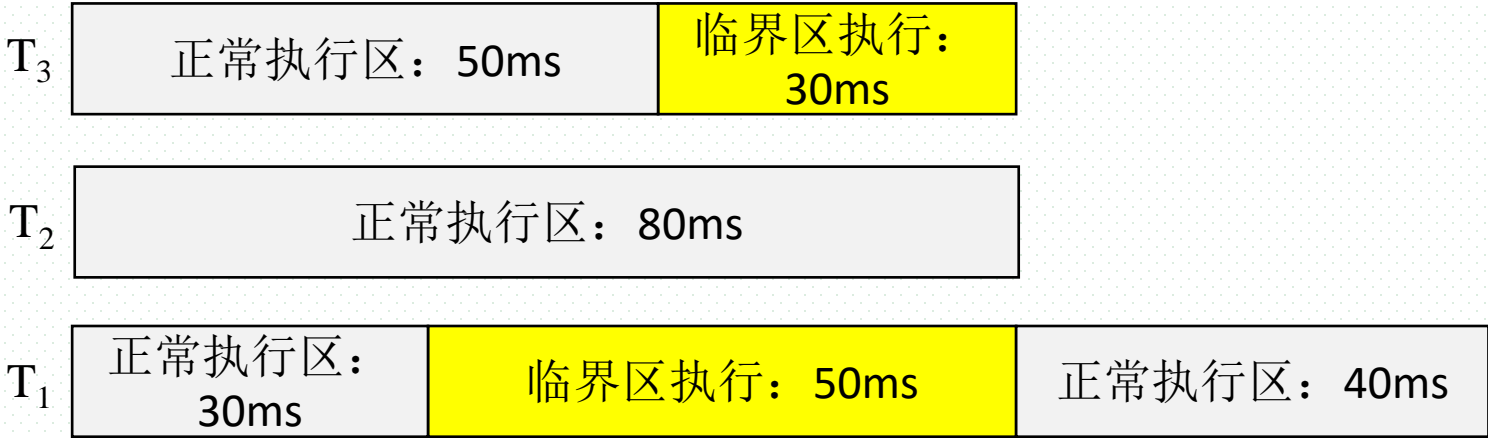
$T_1$ 、 $T_2$ 、 $T_3$ 进入系统的时间分别为:  
0ms, 70ms, 100ms

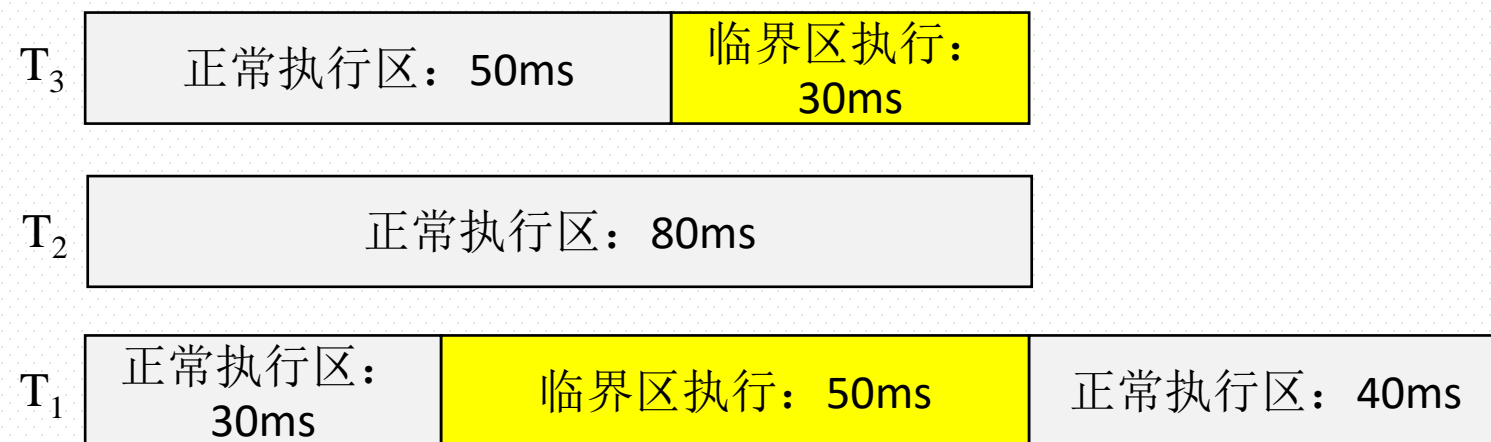
优先级继承协议: 答案2  
 $T_3$ 在进入阻塞前, 将其高优先级30传递给处于临界区中的 $T_1$ , 使得 $T_1$ 继续执行临界区代码, 但 $T_1$ 退出临界区后, 其优先级仍然保持30, 继续运行



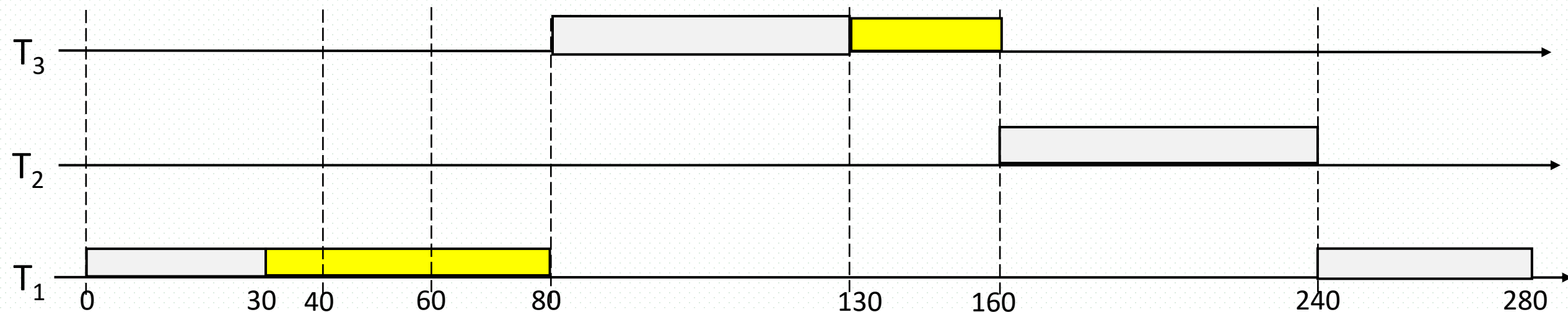


用不可抢占临界区协议





采用不可抢占临界区协议



## 6.6 Classical Problems of Synchronization

---

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Sleeping-barber problem

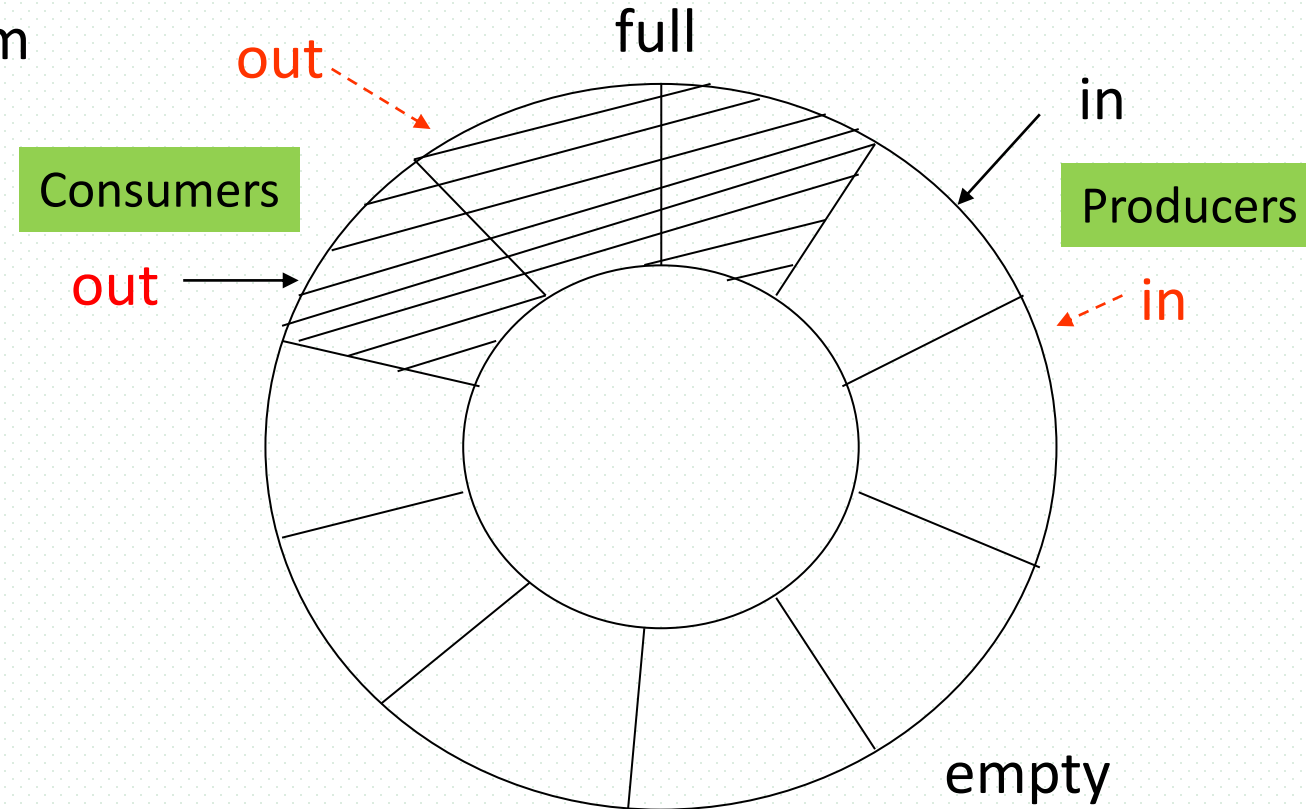
# Bounded-Buffer Problem

---

- Bounded-buffer problem
- Extended bounded-buffer problem
  - 例3. Extended Bounded-Buffer Problem 1: 同时具有生产者/消费者角色
    - 老和尚-小和尚挑水, 流水生产线
  - 例4. Extended Bounded-Buffer Problem 2: 生产者/消费者同时进入buffer
  - 例5. Extended Bounded-Buffer Problem 3: 多类产品 in buffer
  - 例6. Extended Bounded-Buffer Problem 4:
    - 每次从buffer中取出 $m > 1$ 个产品, 或者向buffer中放入 $n > 1$ 个产品
  - 期末试题/实验编程题
  - $\pm n (n > 1)$ 的信号量wait()、signal()操作

# Bounded-Buffer Problem

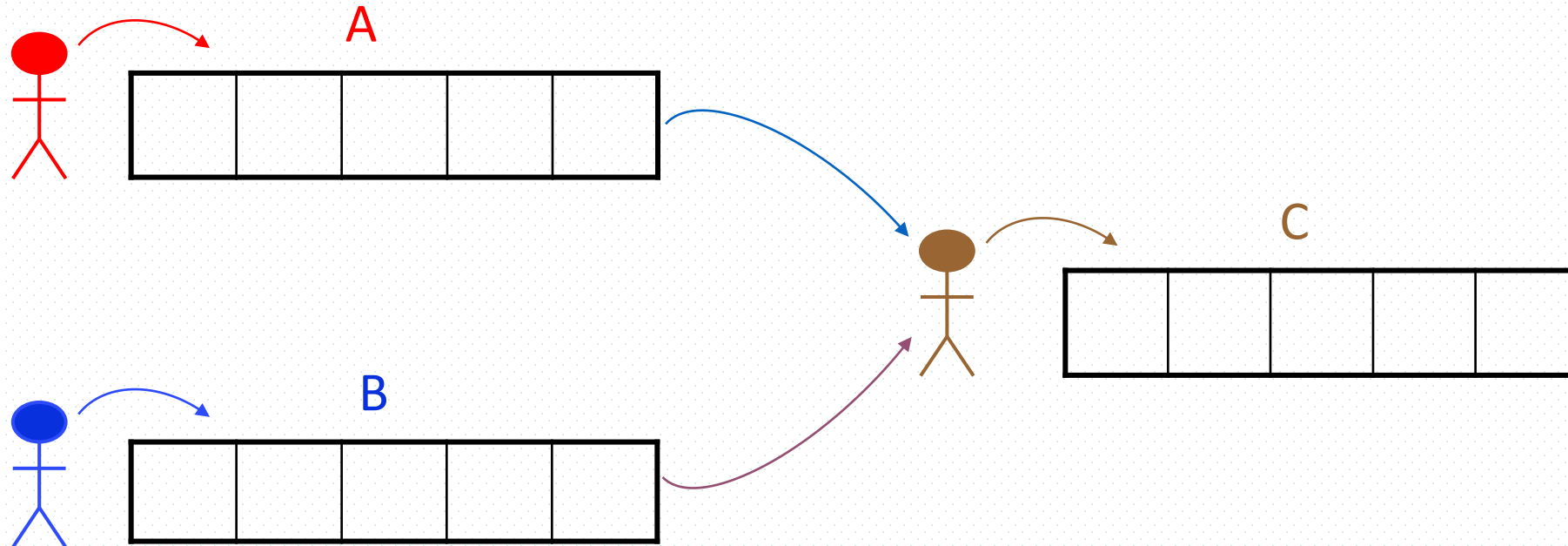
- Also known as Producer-consumer problem
- Mutual exclusion among
  - producers
  - consumers
  - producers and consumers
- Mutual exclusion objects
  - empty units, full units in the buffer





# Bounded-Buffer Problem

- E.g.1 mail-based inter-process communications
- E.g.3 A production pipeline consisting of 3 steps, i.e. A, B, and C, and each step holds a bounded buffer to store the components produced



# Bounded-Buffer Problem

## ■ Semaphores

### □ mutex

- *binary semaphore*, guaranteeing mutual exclusively operating on the buffer, i.e. only one process is allowed to operate on the buffer

### □ full

- full-buffer-units available
- 作为counting semaphore, 计数缓冲区中产品数量

### □ empty

- empty-buffer-unit available

## ■ Initially

□ **full = 0,**

□ **empty = n**

□ **mutex = 1**

```
do {  
    ...  
    produce an item in nextp  
    ...  
    { wait(empty);    /*是否有空单元*/  
      wait(mutex);    /*是否可向buffer/空单元写, 或是否有其它进程在操作buffer*/  
      ...  
      add nextp to buffer  
      ...  
      { signal(mutex); /*允许其它进程访问*/  
        signal(full); /*满单元数加1, 唤醒被阻塞的消费者*/  
      }  
    }  
} while (1);
```

do { ...

**producer**

produce an item in **nextp**

...

**wait(empty);**

/\*是否有空单元

**wait(mutex);**

/\*是否可向**buffer**/空单元写  
，或是否有其它进程在操作**buffer**

...

add **nextp** to **buffer**

...

**signal(mutex);**

/\*允许其它进程访问

**signal(full);**

/\*满单元数加1，唤醒被阻塞消费者

**} while (1);**

do {...

**consumer**

**wait(full)**

/\*是否有满单元

**wait(mutex);**

/\*是否可从**buffer**/满单元读

...

**remove an item from buffer to nextc**

...

**signal(mutex);**

/\*允许其它进程访问

**signal(empty);**

/\*空单元数加1，唤醒被阻塞的生产者

...

**consume the item in nextc**

...

**} while (1);**

# 上机作业/期末试题

- An assembly line is to produce a product C with four part As, and three part Bs 【 $C = 3A + 4B$ 】
- The worker of machining(加工) A and worker of machining B will produce two part As and one part B independently each time. Then the two part As or one part B will be moved to the station(工作台), which can hold at most 12 of part As and part Bs altogether
- Two part As must be put onto the station simultaneously
  - worker A一次生产2个A, 必须一次性放入工作台
- The workers must exclusively put a part on the station or get it from the station. In addition, the worker to make C must get all part of As and Bs for one product once.
  - worker C必须一次性获得所需的3个A、4个B
- Using semaphores to coordinate the three workers who are machining part A, part B and the product C to manufacture the product without deadlock. It is required that
  - (1) definition and initial value of each semaphore, and
  - (2) the algorithm to coordinate the production process for the three workers

should be given

# Cautions



- To avoid deadlock,
  - ▣ for the producer processes, the operation `wait(empty)` should be executed before `wait(mutex)`
    - 先判断有无空单元（是否有访问buffer的必要），再执行`wait(mutex)`，获取对buffer的访问权
  - ▣ for the consumer processes, the operation `wait(full)` should be executed before `wait(mutex)`
    - 先判断有无满单元（是否有访问buffer的必要），再执行`wait(mutex)`，获取对buffer的访问权
- 如果wait操作执行顺序不当，有可能产生死锁
- In exit sections, producers/consumers release the buffer at first by `signal(mutex)`, to permit other producers/consumers to access the buffer as early as possible

# Readers-Writers Problem

- Problem description

- ▣ *readers, writers*, concurrently access on shared data, e.g. files or database systems
- ▣ constraints
  - more than one reader can simultaneously enter their critical sections to access the data item
  - when a reader is accessing the data item, the other readers are also allowed to access, but no writer is permitted

$R-R$	W-R	W-W
Yes	No	No

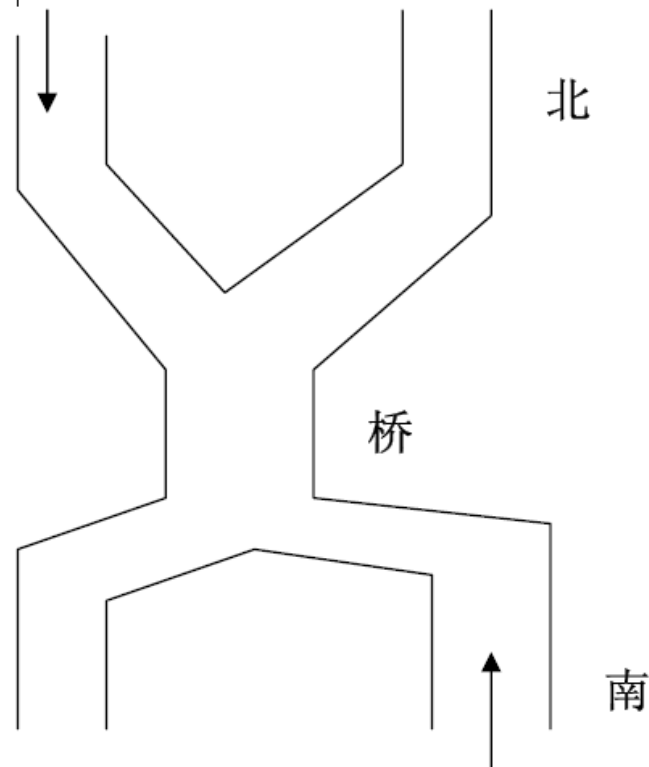
# Extended Readers-Writers Problem

- E.g. Soldiers in two queues, passing a bridge
  - ▣ **two types of readers**, and each queue can be viewed as a type of readers



例题 11. (北京大学 1992 年试题)

有桥如图 4.15 所示。车流如箭头所示。桥上不允许两车交会，但允许同方向多辆车依次通行(即桥上可以有多个同方向的车)。用 P, V 操作实现交通管理以防桥上堵塞。



例. Bupt2020 软件工程考研题



# Solution to Readers-Writers Problem

---

- Data structures

- semaphore wrt      /\*共享数据写操作的互斥信号量
- readcount:          /\*正在读的reader数目
- semaphore mutex  
                         /\*访问共享变量readcount的互斥信号量

- Initially wrt = 1, mutex = 1, readcount = 0

```
wait(mutex);      /*互斥访问readcount
readcount++;      /*读者数加1
if (readcount == 1) /*如果是第1个读者, 判断是否有写者、存在读写冲突
    wait(wrt);     互斥/禁止写操作
signal(mutex);    /*恢复对readcount访问*/
...
reading is performed
...
wait(mutex);      /*互斥访问readcount*/
readcount--;      /*读者数减1*/
if (readcount == 0) /*如果已经无readers,
    signal(wrt);    恢复写操作许可*/
signal(mutex);    /*恢复对readcount访问*/
```

# Solution to Readers-Writers Problem

---

- **Writer** Process: `wait(wrt);`

...

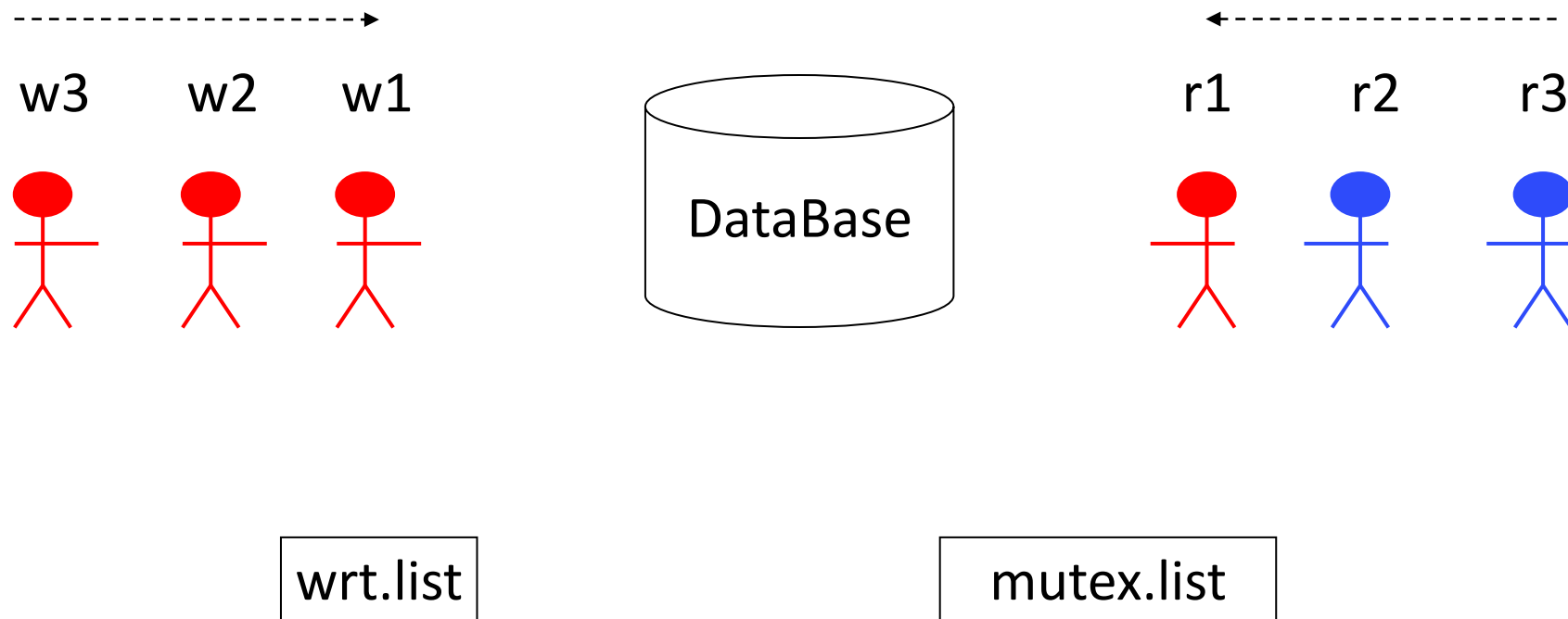
writing is performed

...

`signal(wrt);`

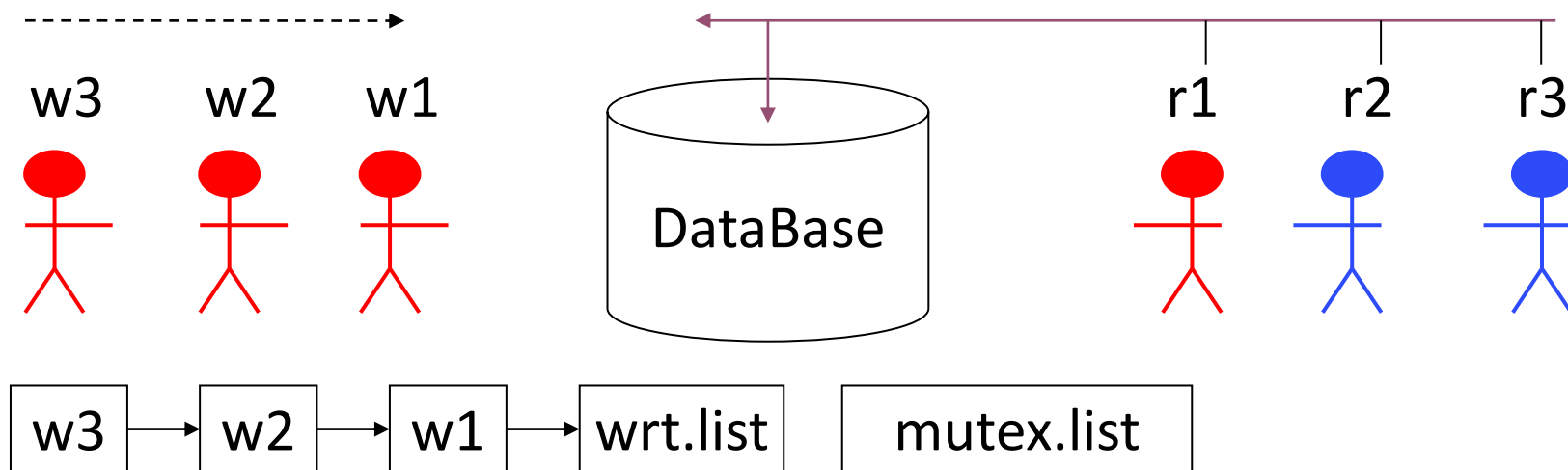
- **Note:** if a *writer* is in its critical section while  $n$  *readers* are waiting,
  - ▣ the first reader is queued on **wrt**
  - ▣ the other  $n-1$  readers are queued on **mutex**

第一个读者执行wait(wrt)，负责与写者争夺控制权



initially, readcount=0, mutex=1, wrt=1

# 先读后写



initially, readcount=0,  
mutex=1,  
wrt=1

先读后写：3个读者依次进入，在临界区内并行读；当读者未完成时，稍后3个writers提出进入申请，均被阻塞：

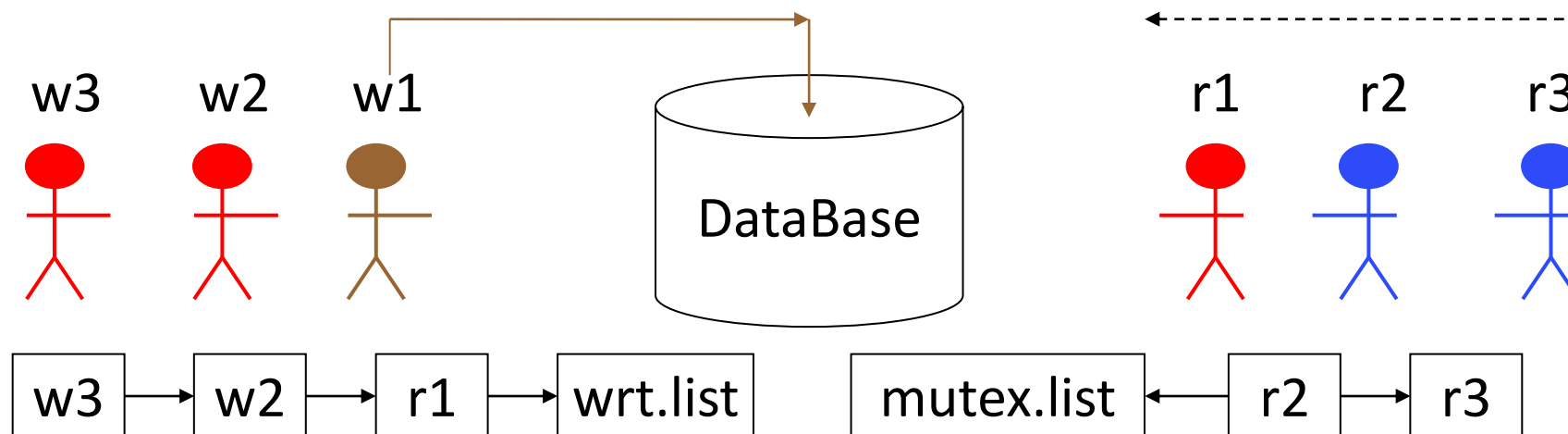
Readers	Writers	wrt	mutex	readcount
r1		1→0	1→0→1	0→1
	w1	×		
r2			1→0→1	1→2
	w2	×		
r3			1→0→1	2→3
	w3	X		

```
wait(mutex);  
readcount++;  
if (readcount == 1)  
    wait(wrt);  
signal(mutex);
```

```
wait(wrt);  
...  
writing is performed  
signal(wrt);
```

# 先写后读

initially, readcount=0, mutex=1, wrt=1



先写后读：3个写者依次申请进入临界区内写；当第1个写者未完成时，稍后writers、reader提出进入申请：

Readers	Writers	wrt	mutex	readcount
	w1	1→0	1	0
r1		×	1→0	0→1
	w2	×		1
r2			×	1
	w3	×		0
r3			×	1

```
wait(mutex);  
readcount++;  
if (readcount == 1)  
    wait(wrt);  
signal(mutex);
```

```
wait(wrt);  
...  
writing is performed  
signal(wrt);
```

# Extended Readers-Writers Problems

---

- 例7. Extended Reader-Writer Problem 1
  - 多队读者，没有写者
  - e.g. 两队士兵相向过独木桥，两车队相向通过bridge
- 例8. Extended Reader-Writer Problem 2
  - 允许最多 $m$ 个读者同时读
- 例9. Extended Reader-Writer Problem 3
  - $m$ 个读者依次读完，允许写者
- 例10. Extended Reader-Writer Problem 4
  - 读写者公平竞争
- 例11. Extended Reader-Writer Problem 5
  - 写者优先

# Linux内核读写锁

- Linux提供读写锁机制，允许更高层次的进程并行
  - ▣ 允许多个读者同时读
- 读写锁具有三种状态
  - ▣ 读模式下加锁，写模式下加锁，不加锁
- 读写锁简化了编程
  - ▣ 只需要一个读写锁信号量，无需readcount、mutex等，即可实现读写者问题
- 资料：Linux内核读写锁与互斥锁的区别
  - ▣ [https://zhuanlan.zhihu.com/p/494172254?utm\\_source=wechat\\_session&utm\\_medium=social&utm\\_oi=679111968938397696&utm\\_campaign=shareopn](https://zhuanlan.zhihu.com/p/494172254?utm_source=wechat_session&utm_medium=social&utm_oi=679111968938397696&utm_campaign=shareopn)



# Dining-Philosophers Problem

---

- Problem description
  - ▣ five **philosophers** (i.e. processes), five **chopsticks** (i.e. resources), thinking and eating;
  - ▣ pick up two chopsticks that are closest to him to eat, pick up only one chopstick at a time;
  - ▣ once finish eating, put down both of her chopsticks, and start thinking
- As a model applicable in a large class of concurrency control problems
  - ▣ representing the need of **allocating several resources among several processes** in a deadlock-free and starvation-free manner
    - e.g. multiple processes concurrently use shared resources, such as I/O devices

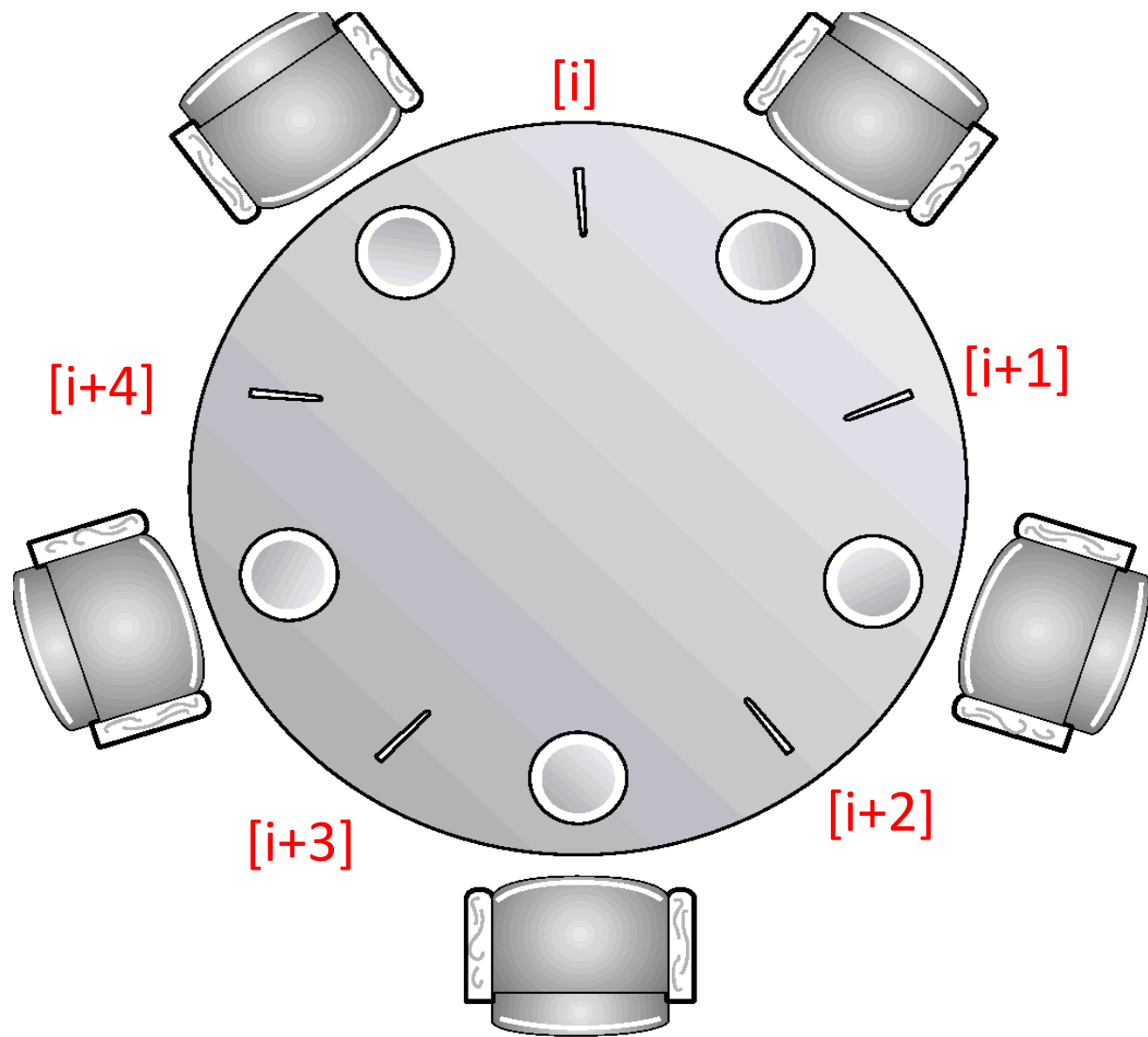


Fig. Dining-Philosophers Problem

特点：  
为防止死锁，进程需要同时获得2类资源：  
1) 左手刀叉 2) 右手刀叉  
2类资源同时申请，不允许只申请一部分资源。

应用：  
1. (第7章) 资源预分配策略，控制死锁；  
2. 课后作业：  
进程执行时需要同时使用1 page 内存资源和1个I/O设备

# An Incorrect Solution

---

- An **incorrect** ( **may resulting in starvation or deadlock**) solution based on semaphores

- shared data

**semaphore chopstick[5];**

*/\*representing that chopstick[i] is available,  $1 \leq i \leq 5$ \*/*

, initially all values are 1

# Philosopher $i$ :

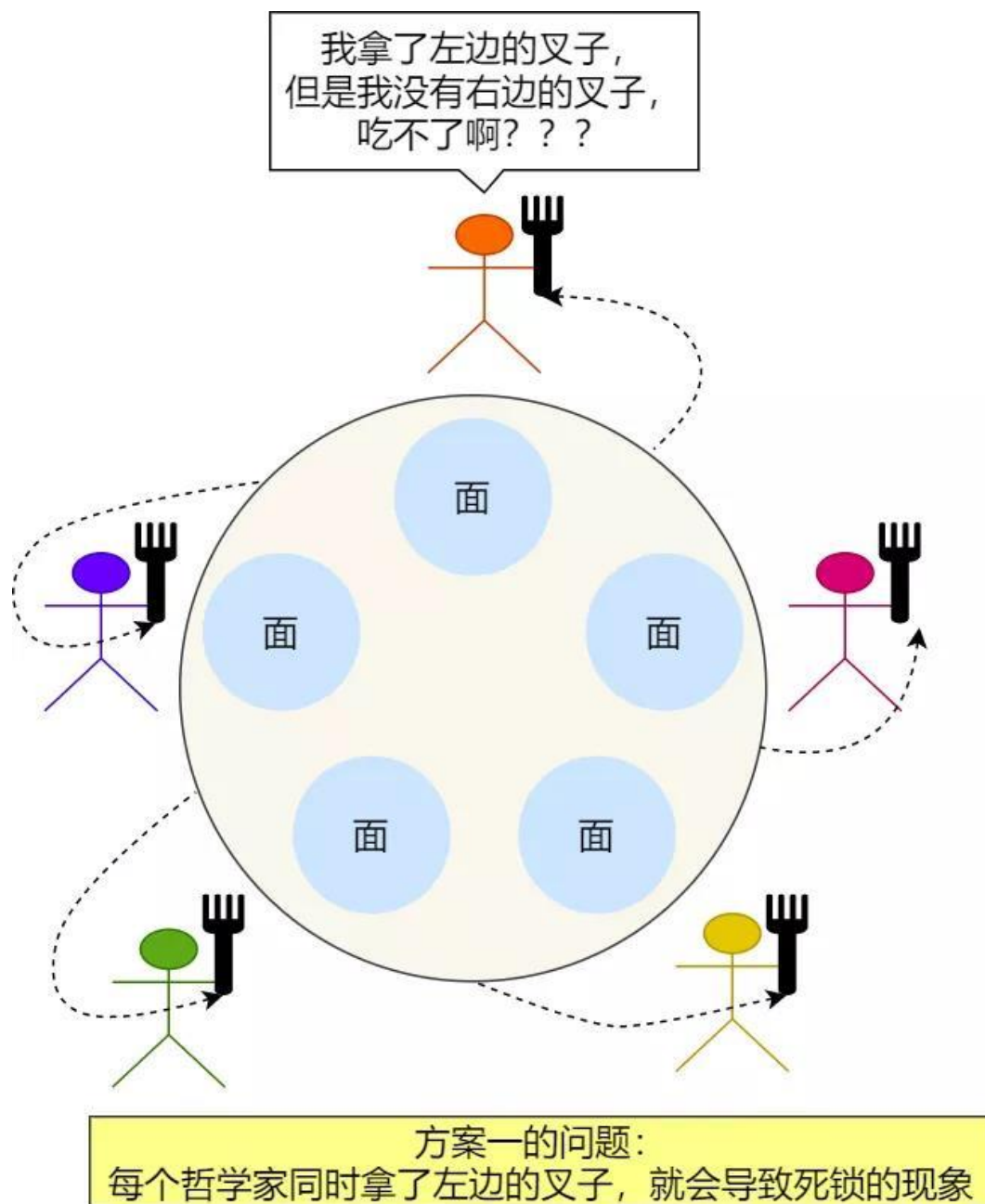
## ▣ Philosopher $i$ :

```
do {  
    wait(chopstick[i])  
        /* right chopstick */  
    pickup the chopstick  
    wait(chopstick[(i+1) % 5])  
        /* left chopstick */  
    pickup the chopstick  
    ...  
    eating  
    ...  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    ...  
    think  
    ...  
} while (1);
```

## ■ Properties

- ▣ guarantee that no two neighbors are eating simultaneously
- ▣ **deadlock** occurs, e.g., when each philosopher picks up her left chopsticks

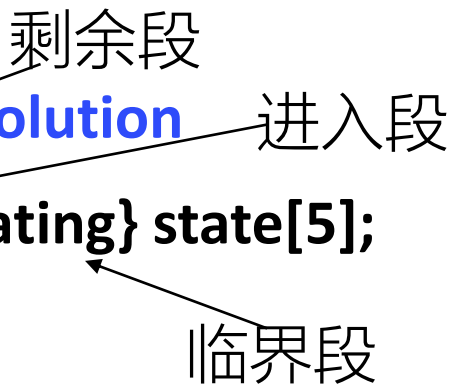
## ■ 方案1及其错误



# Dining-Philosophers Problem

---

- Dining-Philosophers problem can also be resolved by monitors as shown in (§6.7)
- A correct semaphore-based solution to this problem, similar to that in §6.7, is given

- A correct semaphore-based solution
    - enum {thinking, hungry, eating} state[5];  
/\*记录每个人状态\*/
    - semaphore mutex=1;  
/\*临界区互斥\*, 保证每时刻最多只能有一个人执行从餐桌上取或放2把叉子的动作/
    - semaphore self[5];  
/\*哲学家信号量, 表明i是否已获得2把刀叉并可以开始进餐\*/
    - initially, mutex=1, state[i]=thinking, self[i]=0
- 

```
□ void philosopher(int i) {  
    while (true) {  
        thinking();    /*剩余段： 思考*/  
        pickup(i);     /*进入段： 同时获取两把叉子， 或阻塞*/  
        eating();      /*临界段： 进餐*/  
        putdown(i);    /*退出段： 放下两把叉子*/  
  
    };  
}
```



❑ void pickup(int i) {

wait(mutex);

/\*申请进入临界区访问餐桌

state[i] = hungry;

/\*记录为饥饿

test[i];

/\*测试左右刀叉是否可用

signal(mutex);

/\*退出访问餐桌区

wait(self[i])

/\*等待左右空闲刀叉,

如果得不到叉子则被阻塞,

}

❑ void test(int i)

{

if ( (state[(i + 4) % 5] != eating) &&

(state[i] == hungry) &&

(state[(i + 1) % 5] != eating))

{

state[i] = eating;

signal(self[i]); /\*唤醒i, 允许其进餐\*/

}

}

当左右邻居空闲、左右刀叉均可用时, self[i] 0→1

```
void putdown(int i) {  
    wait(mutex);    /*申请访问餐桌*/  
    state[i] = thinking; /*记录为思考*/  
    test[(i+1)%5];  /*测试左邻居是否想进餐*/  
    test[(i+4)%5];  /*测试右邻居是否想进餐*/  
    signal (mutex); /*退出访问餐桌*/  
}
```

signal(self[i+1]%5)

signal(self[i+4]%5)

i进餐完毕，唤醒  
可能被阻塞在self  
对列的左右邻居



# Extended Dining Problems

---

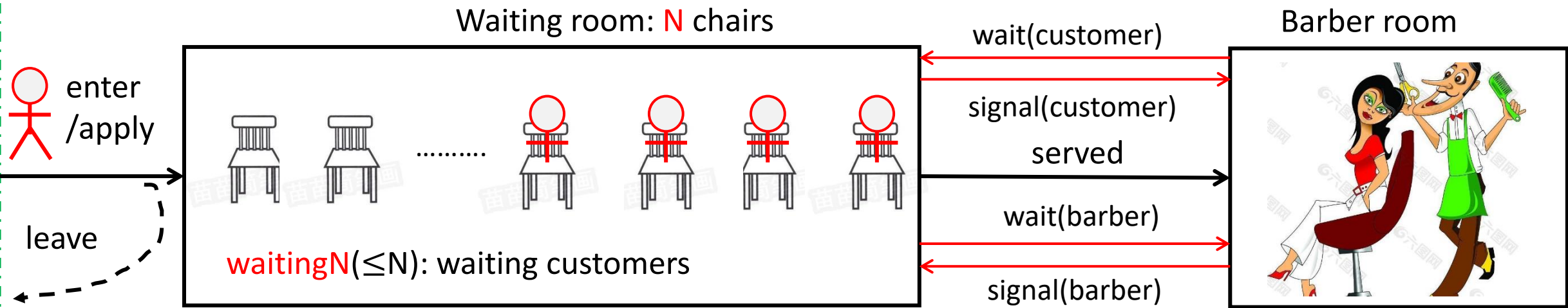
- 例12. 同时申请多类资源
  - ▣ e.g. memory、I/O devices 【习题】
- 例13. 哲学家就餐问题第三种解法
- 例14. 反例1、反例2
  - ▣ 死锁，执行效率低

# Sleeping Barber Problem

---

## ■ Sleeping Barber Problem

- ❑ A barbershop consists of a waiting room with **N chairs** and a barber room with one barber chair
- ❑ If there are no customers to be served, the barber goes to sleep
- ❑ If a customer enters to the barbershop, and all chairs are occupied, then the customer leaves the shop
- ❑ If the barber is busy but the chairs are available, the customer sits in one of free chairs. If the barber is asleep, the customer wakes up the barber
- ❑ 进程间同步 + 利用“用户空间计数变量 + 内核空间互斥信号量”判断buffer容量、顾客数量



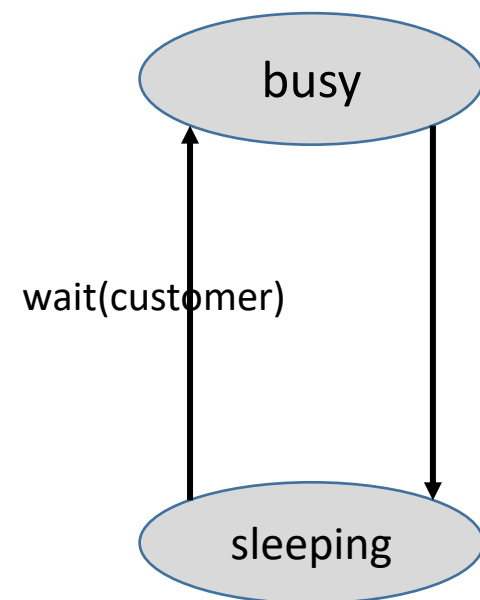
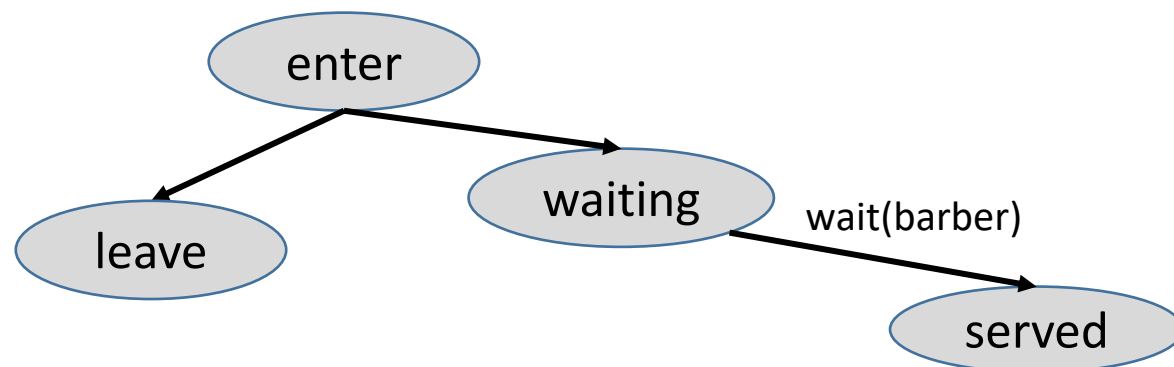
waitingN < CHAIRS = N?  
决定是否进入或离开

要求:  
waiting room 容量为 N,  
当容量已满时 (n 个  
顾客等待), 后续顾  
客离开, 不能被阻塞,  
故引入计数变量  
waitingN

信号量 customer:  
customer-barber 间通知是  
否有顾客, 用于 customer  
唤醒 barber 为顾客理发  
customer: signal(customer)  
barber: wait(customer)

信号量 barber:  
barber-customer 间通知  
barber 是否空闲, 用  
barber 于唤醒等待的顾客  
barber: signal(barber)  
customer: wait(barber)

one chair, one barber



# Sleeping Barber Problem

- Problem description in P233 in the textbook
  - ▣ barbershop: waiting room, barber room
  - ▣ n chairs in waiting room, one barber chair in barber room
  - ▣ customers: enter—wait or leave—severed
  - ▣ barber: sleep—be waken up—be busy/working

特点:

1. 资源竞争

顾客竞争chairs

顾客竞争barbers

2. 同步

customer-barber

- binary semaphore **customer**

- customer-barber间通知是否有顾客，唤醒barber为顾客理发  
customer: signal(**customer**)  
barber: wait(**customer**)

- binary semaphore **barber**

- barber-customer间的同步/唤醒信号量

barber是否可以为顾客理发  
barber: signal(**barber**)  
customer: wait(**barber**)

- int **waitingN**

- the number of waiting customers, a copy of semaphore customers

- binary semaphore **mutex**

- for mutual exclusion on the variable waitingN

- Initially

- customers=0, barber=0, mutex=1
- waitingN=0, CHAIRS=N /\*椅子数目

1. 引入waitingN统计等待的顾客总数，以便与等候室内椅子总数N进行比较，判断有无空椅子
2. 引入mutex，控制互斥访问waitingN

# Barber

```
■ Void Barber(void) {  
    while (true) {  
        wait(customers);  /*如果无等待顾客，则睡觉  
        wait(mutex);      /*服务顾客，将等待顾客数减1  
        waitingN:=waitingN-1;  
        signal(mutex);  
        signal(barber);    /*理发师召唤1个等待顾客，开始理发  
        cut-hair()    }  
    }
```

barber工作前提：有等待顾客， wait(customers)



# Customer

```
■ Void Customers(void) {  
    wait(mutex);  
    if (waitingN < CHAIRS) { /*判断是否有空椅子, 等待  
        waitingN:= waitingN+1;  
        signal(customers); /*通知理发师, 有等待顾客  
        signal(mutex);  
        wait(barber);      /*请求理发师为自己服务*/  
        get-hair()  
    }  
    else /*如果客满无空椅子, 离开  
        { signal(mutex)  
          }  
}
```

顾客被服务前提:

1. 有空椅子, 可以等待,  $\text{waitingN} < \text{CHAIRS} = N$ ;
2. 被理发师召唤, `wait(barber)`

## 6.7 Monitors

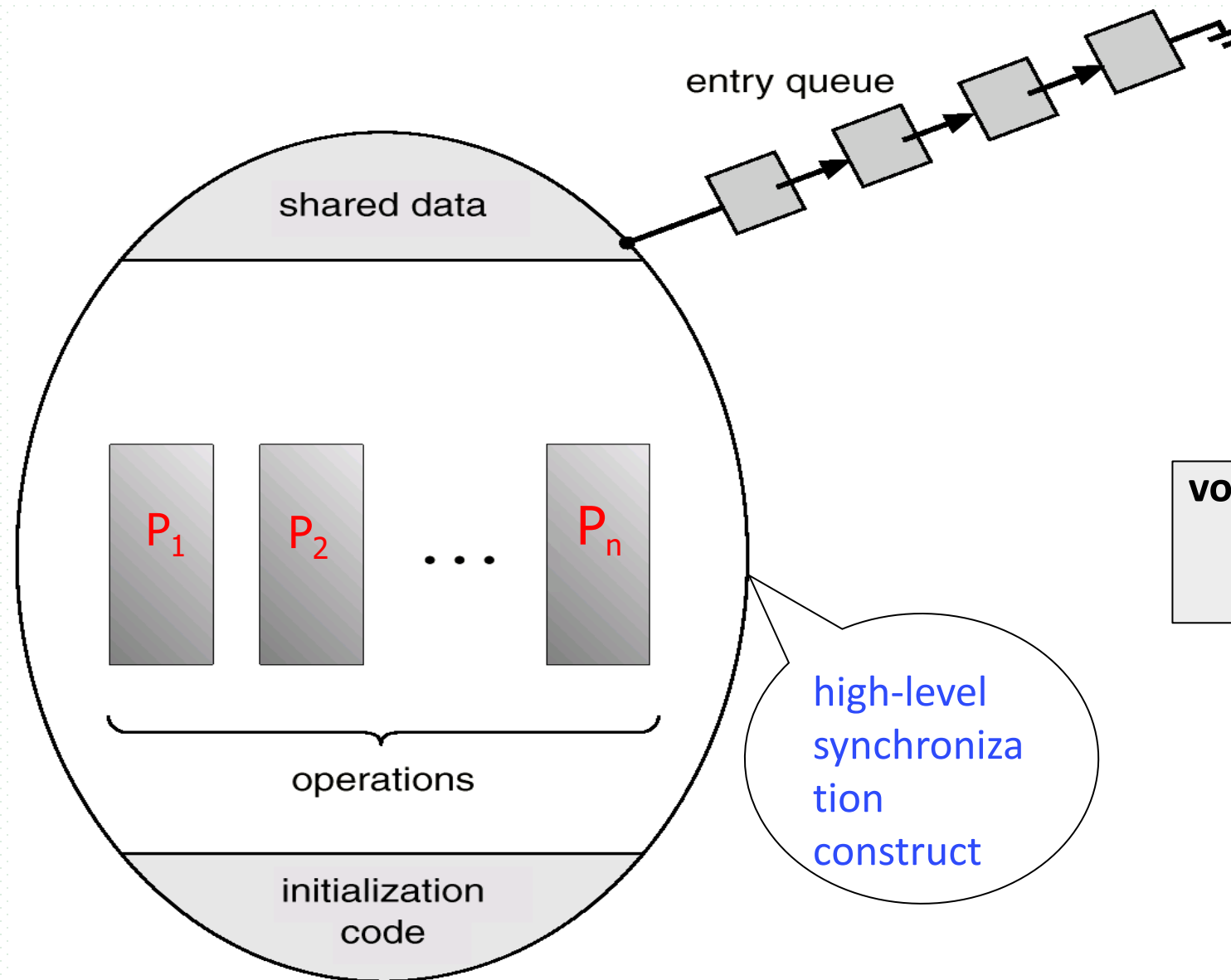
- Monitor(管程)
  - ▣ a type of high-level synchronization construct (i.e. not in kernel space) that allows the safe sharing of an abstract data type among concurrent processes
  - ▣ provided by programming environments ( such as Concurrent C, C++, Java, C#)
- Why the monitor needed ?
  - ▣ process synchronization on the basis of semaphores is somewhat difficult for the application programmers
  - ▣ disorders of the *wait* and *signal* operations in application programs may result in **deadlock**
  - ▣ to avoid this demerit, the critical sections in programs are controlled by a centralized controller, i.e. the monitor, provided by programming languages or middleware
- 管程”定义
  - ▣ 由程序设计语言/环境 (e.g. Java, Concurrent C) 提供的一种进程同步互斥机制
  - ▣ 将分散在应用进程中互斥访问临界/共享资源的临界区代码集中到管程内部, 统一地管理
  - ▣ 管程可以控制共享资源, 为访问这些共享资源提供一种互斥机制
  - ▣ 形式上, 管程 = 局部于管程内部的公共共享变量 + 访问这些变量的过程
    - 公共共享变量代表共享资源, 过程对应于临界区代码

# Monitors Representation

```
monitor monitor-name
{
    shared variable declarations
    procedure body  $P_1$  (...) {
        ...
    }
    procedure body  $P_2$  (...) {
        ...
    }
    procedure body  $P_n$  (...) {
        ...
    }
    {
        initialization code
    }
}
```

- Procedure  $P_1, P_2, \dots, P_n$  operate on **shared variables** declared within the monitor
- Monitor consists of
  - shared variables
  - concurrent procedures, e.g. *producers*, *consumers*
  - initialization codes
- Monitor construct ensures that only one process at a time can be active within the monitor ( access the shared variables by calling  $P_1, P_2, \dots, P_n$ )
  - act as mutual exclusive semaphore ***mutex***

# Fig. Schematic view of a monitor



```
monitor pc //producer-consumer
{
    shared variable buffer
    void add(int i)
    void remove(int i)
    initialization code
}
```

```
void add( int i) {
    add nextp to buffer
}
```

```
void remove( int i) {
    remove from buffer
}
```

替代了对buffer的mutex

**Producer i:**

wait(empty)

**pc.add**

signal(full)

**Consumer i:**

wait(full)

**pc.remove**

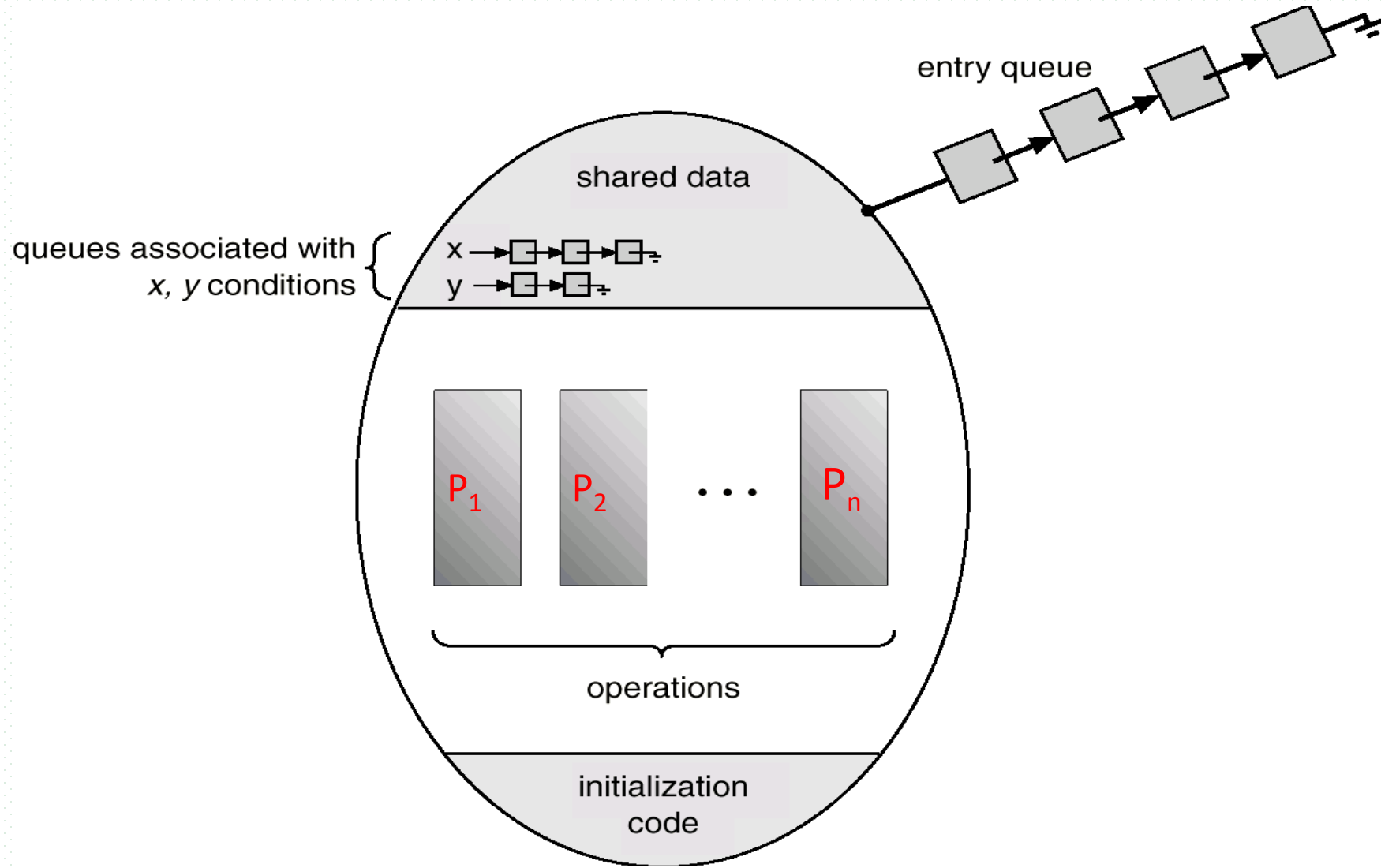
signal(empty)

# Monitor

---

- Improvement
  - to refining the synchronizing ability of the monitor, a process can be allowed to **wait within the monitor**, so **condition variables** 【条件变量】 are introduced into the monitor, which are declared, as:  
**condition x, y**
- **Monitor** consists of
  - shared variables
  - **conditional variables** 【条件变量】
  - concurrent procedures
  - initialization codes
- Fig. 6.18

# Fig. Monitor With Condition Variables 【条件变量】



# Monitor

---

- Condition variable can only be used with the operations **wait** and **signal**. These two operators are in procedures  $P_1, P_2, \dots, P_n$ 
  - ▣ **x.wait()**  
means that the process invoking this operation is suspended until another process invokes
  - ▣ **x.signal()**  
the **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect

# Bounded-buffer Monitor

**monitor** **pc**

{

shared variable **buffer**

condition **empty, full**

void **add**(int i)

void **remove**(int i)

void init() {

empty=n;

full=0

}

}

void **add**(int i) {

**empty.wait**

access and add **nextp** to buffer

**full.signal**

}

void **remove**(int i) {

**full.wait**

access and remove from buffer  
to nextc

**empty.signal**

}

**Producer i:**

**pc.add(i)**

**Consumer i:**

**pc.remove(i)**



# Monitor for Dining Philosophers

## ■ Monitor **dp**

与信号量解法相比，无需  
pickup、putdown中的**mutex**



```
{
1. enum {thinking, hungry, eating} state[5];
2. condition self[5];
3. void pickup(int i)
   void putdown(int i)
   void test(int i)
4. void init() {
    for (int i = 0; i < 5; i++)
        state[i] = thinking;
}
```

----- shared variables  
-----conditional variables  
} concurrent procedures  
} initialization code

# Monitor for Dining Philosophers

```
■ void pickup(int i) {  
    state[i] = hungry;  
    test[i];  
    if (state[i] != eating)  
        self[i].wait();  
}
```

```
■ void putdown(int i) {  
    state[i] = thinking;  
    /* test left and right neighbors  
    test((i+4) % 5);  
    test((i+1) % 5);  
}
```

```
■ void test(int i) {  
    if ( (state[(i + 4) % 5] != eating) &&  
        (state[i] == hungry) &&  
        (state[(i + 1) % 5] != eating)) {  
        state[i] = eating;  
        self[i].signal();  
    }  
}
```

Philosopher i:  
dp.pickup (i)  
eat  
dp.putdown(i)

# 基于条件变量和互斥锁的生产者-消费者问题

C/C++等高级程序设计语言提供了条件变量、锁等同步互斥机制

```
1  int empty=5;
2  int full=0;
3  struct cond empty_cond;
4  lock empty_cnt_lock;
5  struct cond full_cond;
6  lock full_cnt_lock;
7
8  void producer(void)
9  {
10     int new_msg;
11
```

```
11 while (true) {
12     new_msg=producer_new();
13     lock(&empty_cnt_lock);
14     while (empty==0)
15         cond_wait(&empty_cond, &empty_cnt_lock);
16     empty-- ;
17     unlock(&empty_cnt_lock);
18
19     buffer_add_safe(new_mag);
20
21     lock(&full_cnt_lock);
```

## 基于条件变量和互斥锁的生产者-消费者问题

```
21  lock(&full_cnt_lock);  
22  full++;  
23  cond_signal(&full_cond);  
24  unlock(&full_cnt_lock);  
25  }  
26  }  
27
```

# 基于条件变量和互斥锁的生产者-消费者问题

```
27
28 void consumer(void)
29 {
30     int cur_msg;
31     while (true) {
32         lock(&full_cnt_lock);
33         while (full==0)
34             cond_wait(&full_cond, &full_cnt_lock);
35         full--;
36         unlock(&full_cnt_lock);
37
```

```
37
38     cur_msg=buffer_remove_safe();
39
40     lock(&empty_cnt_lock);
41     empty++;
42     cond_signal(&empty_cond);
43     unlock(&empty_cnt_lock);
44     consume_msg(cur_msg);
45 }
46 }
```

# Monitor vs Semaphore

---

- Monitors are implemented by semaphores
- For more details, *refer to* Appendix 6-2
  
- Monitors vs Semaphores
  - ▣ refer to next slide

信号量方式	管程方式
<p>互斥访问的资源/变量,</p> <p>e.g.1 buffer</p> <p>e.g.2 shared data; <b>readcount</b></p> <p>e.g.3 餐桌; <b>state[i]</b></p>	<p>monitor;</p> <p><b>shared variables</b></p>
<p>binary semaphores: 对临界资源/变量互斥访问</p> <p>e.g.1 <b>mutex</b> e.g.2 <b>wrt, mutex</b> e.g.3 <b>mutex</b></p>	<p>说明: 管程自动实现对管程所代表的资源或管程内<b>shared variables</b>的互斥访问, 因此不需要单独定义类似于<b>binary semaphores mutex</b>角色的结构</p>
<p>counting semaphores</p> <p>e.g.1 empty, full;</p> <p>e.g.2 self[i];</p>	<p>conditional variable x</p>
<p>wait(), signal() on counting semaphores</p>	<p>x.wait(), x.signal() on conditional variables</p>
<p>concurrent processes的各个业务逻辑步骤</p>	<p>concurrent procedures</p>
<p>shared variables, <b>binary semaphores</b>, counting semaphores初始化</p>	<p>initialization code:</p> <p>对<b>shared variables</b>, conditional variables 初始化</p> <p>说明: 无<b>binary semaphores</b></p>

# Monitors vs Semaphores

---

- 以管程方式实现的并发进程，其业务逻辑由一系列顺序步骤组成；
- 当其中某些业务逻辑步骤需要访问临界资源时，需要采用定义在管程内的 `procedures`；
- 如果进程在其生命周期内需要顺序访问多个临界资源，每个临界资源通过一个管程来控制对其互斥访问，则进程的业务逻辑步骤中会调用来自多个管程的 `procedures`
  - ▣ e.g. “小和尚-老和尚挑水取水”



# 说 明

---

- 软件同步适用于单CPU/单核系统
  - e.g. 反例：多/双核4线程系统上的Bakery同步
- 信号量/硬件/管程等其它几种机制适用于单、多CPU系统，特别是硬件指令、自旋锁
  - e.g. 多核系统中的随机数发生
  - e.g. “利用流水线和锁提高服务器吞吐量”

## Appendix 6-1 Synchronization Hardware (§6.4 【略】)

- Two schemes of hardware synchronization by machine instructions
  - ▣ Test and Modify
  - ▣ Swap
- Test and modify the content of a word **atomically**, by ***TestAndSet*** instruction
  - ▣ /\* 测试布尔变量target的值, 再将其设置为true

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
    return rv;  
}
```

# Test and Set

---

- If two TestAndSet instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in an arbitrary order
  - To conduct mutual exclusion on a type of resource, declare a Boolean variable *lock*, specific to the resource and initialized to *false*
  - The processes utilize **TestAndSet** instructions on this *lock* to mutually access the resource, as follows
    - ▣ shared data
- boolean lock = false** /\*表示没有加锁，无用户访问

Process  $P_i$ ,

do {

while (TestAndSet(lock)) ;

/\* false  $\rightarrow$  true, 加锁, 阻止其它进程  
critical section

lock = false;

/\*释放锁, 允许其它进程进入

remainder section

}

TestAndSet (lock)

```
{ boolean rv = lock
  lock = true;
  return rv;
}
```

$P_j$

do {

while (TestAndSet(lock)) ;

/\*busy-waiting

while (TestAndSet(lock)) ;

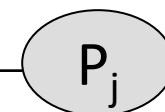
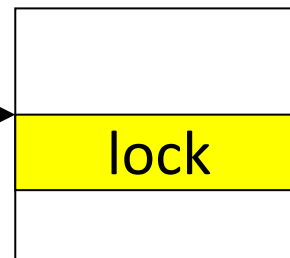
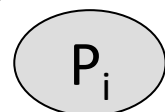
/\* false  $\rightarrow$  true, 加锁, 阻止其它进程)

critical section

lock = false; /\*释放, 允许其它进程进入

remainder section

}



lock

F

T

F

T

F

# Swap

- Atomically swap two variables



```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

- Shared data (initialized to false, 表示资源没被占用):

- **boolean lock;**

## ■ 方案1

- shared data (initialized to **false**, 表示资源没被占用):  
**boolean lock;**

- Process  $P_i$   
do {  
    **key = true;**  
    **while (key == true)**      /\*进入段, lock=true时, 资源被占,  
                                陷入while忙等待  
        **Swap(lock, key);**  
        critical section  
        **lock = false;**      /\*退出段, 释放锁  
        remainder section  
    }

问题:  
busy-waiting, do not meet the  
bounded-waiting requirement

## ■ 方案2

Process  $P_i$

- Shared data  
(initialized to false):

**boolean lock;**  
**boolean waiting[n];**

■ 1

```
do {  
    waiting[i]=true;  
    key = true;  
    while (waiting[i]&&key )  
        key=TestAndSet(lock);  
    waiting[i]=false;  
    critical section  
    j=(i+1) %n;  
    while ((j!=i) && !waiting[j])  
        j = (j+1) %n  
    if (j==i)  
        lock=false  
    else  
        waiting[j]=false;  
    remainder section  
}
```

*bounded-waiting* mutual  
exclusion with TestAndSet

/\*进入段, lock, waiting[i]=true  
时, 资源被占, 陷入while:等待

—————→表明自身不再需要等待

考察并释放  
其它等待进程,使这  
些进程跳出while循环

## Appendix 6-2 Monitor Implementation Using Semaphores 【略】

- Variables

  - semaphore mutex; // (initially = 1)**

  - semaphore next; // (initially = 0)**

  - int next-count = 0;**

- Each external procedure ***F*** will be replaced by

  - wait(mutex);**

  - ...

  - body of *F*;

  - ...

  - if (next-count > 0)**

    - signal(next)**

  - else**

    - signal(mutex);**

- Mutual exclusion within a monitor is ensured.



# Monitor Implementation Using Semaphores

---

- For each condition variable **x**, we have:  
    **semaphore x-sem; // (initially = 0)**  
    **int x-count = 0;**
- The operation **x.wait** can be implemented as:

```
x-count++;  
    if (next-count > 0)  
        signal(next);  
    else  
        signal(mutex);  
    wait(x-sem);  
    x-count--;
```

# Monitor Implementation Using Semaphores

---

- The operation **x.signal** can be implemented as:

```
if (x-count > 0) {  
    next-count++;  
    signal(x-sem);  
    wait(next);  
    next-count--;  
}
```

# Monitor Implementation Using Semaphores

---

- *Conditional-wait* construct: **x.wait(c);**
  - ❑ **c** – integer expression evaluated when the **wait** operation is executed.
  - ❑ value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - ❑ when **x.signal** is executed, process with smallest associated priority number is resumed next
- Check two conditions to establish correctness of system:
  - ❑ User processes must always make their calls on the monitor in a correct sequence.
  - ❑ Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols

## Appendix 6-3 Linux 进程同步互斥机制

---



Linux-7-kern  
nchronizatio



Thanks for your  
attention



北京邮电大学