西安交通大学
软件学院

操作系统原理

Operating System Principle

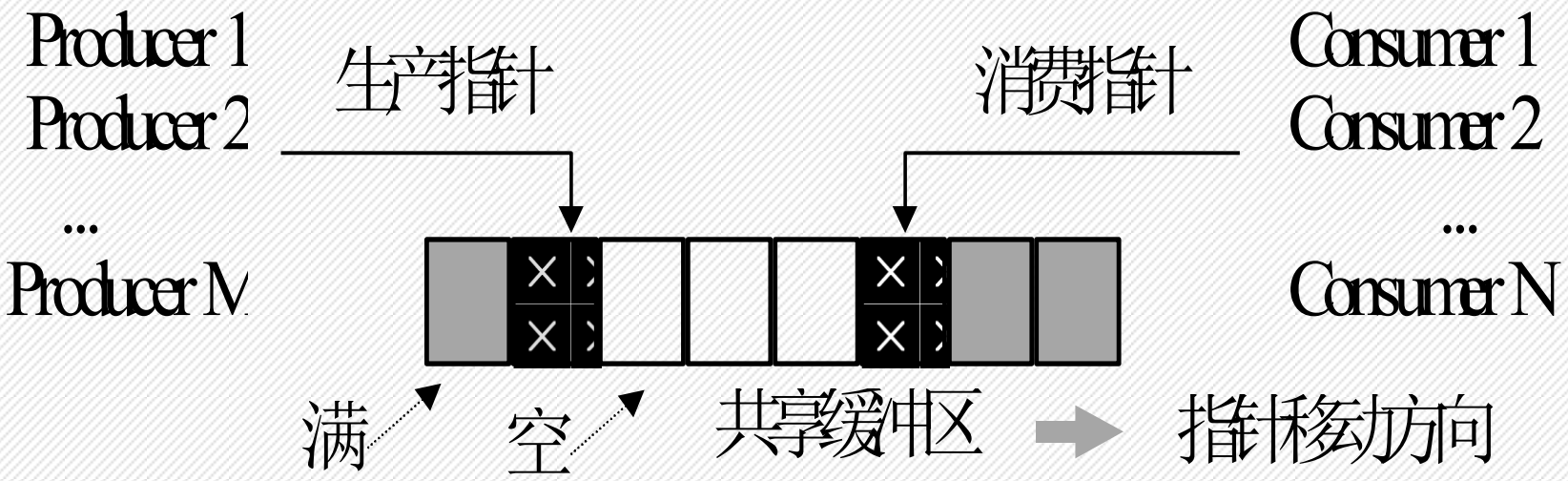
田丽华

6-1 进程同步

生产者消费者问题

采用共享内存解决生产者消费者问题时，N个缓冲区最多只能用N-1个。

如何解决?


Producer 1
Producer 2
...
Producer M
生产指针
消费指针
Consumer 1
Consumer 2
...
Consumer N
满
空
共享缓冲区
指针移动方向

Background
背景

```
// producer calls this method
public void enter(Object item) {
        while (count == BUFFER_SIZE) ; // do nothing
        // add an item to the buffer
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Background
背景

```
// consumer calls this method
   public Object remove() {
      Object item;
            while (count == 0) ; // do nothing
             // remove an item from the buffer
              item = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            count--;
            return item;
   }
```

Background
背景

Count++:

Register1=count;

Register1=register1+1;

Count=register1

Count--:

Register2=count;

Register2=register2-1;

Count=register2

Background
背景

Consider this execution interleaving with "count = 5"initially:

- S0: producer execute register1 = count {register1 = 5}
- S1: producer execute register1 = register1+ 1 {register1 = 6}
- S2: consumer execute register2 = count {register2 = 5}
- S3: consumer execute register2 = register2-1 {register2 = 4}
- S4: producer execute count = register1 {count = 6 }
- S5: consumer execute count = register2 {count = 4}

Background
背景

Concurrent access to shared data may result in data inconsistency（对共享数据的并发访问可能导致数据的不一致性）.

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes（要保持数据的一致性，就需要一种保证并发进程的正确执行顺序的机制）.

Shared-memory solution to bounded-butter problem (Chapter 4) has a race condition on the class data *count*（解决有界缓冲区问题的共享内存方法在类数据count 上存在竞争条件）

race condition竞争条件

若干个并发的进程(线程)都可以访问和操纵同一个共享数据,从而执行结果就取决于并发进程对这个数据的访问次序.

为了保证数据的一致性,需要有同步机制来保证多个进程对共享数据的互斥访问.

Background
背景

进程类型

协作进程
独立进程

进程间资源访问冲突

共享变量的修改冲突
操作顺序冲突

进程间的制约关系

间接制约：有些资源需要互斥使用，因此各进程进行竞争--独占分配到的部分或全部共享资源，进程的这种关系为进程的互斥
直接制约：进行协作--具体说，一个进程运行到某一点时要求另一伙伴进程为它提供消息，在未获得消息之前，该进程处于等待状态，获得消息后被唤醒进入就绪态.进程的这种关系为进程的同步

Background
背景

一个飞机订票系统，两个终端，运行T1、T2进程

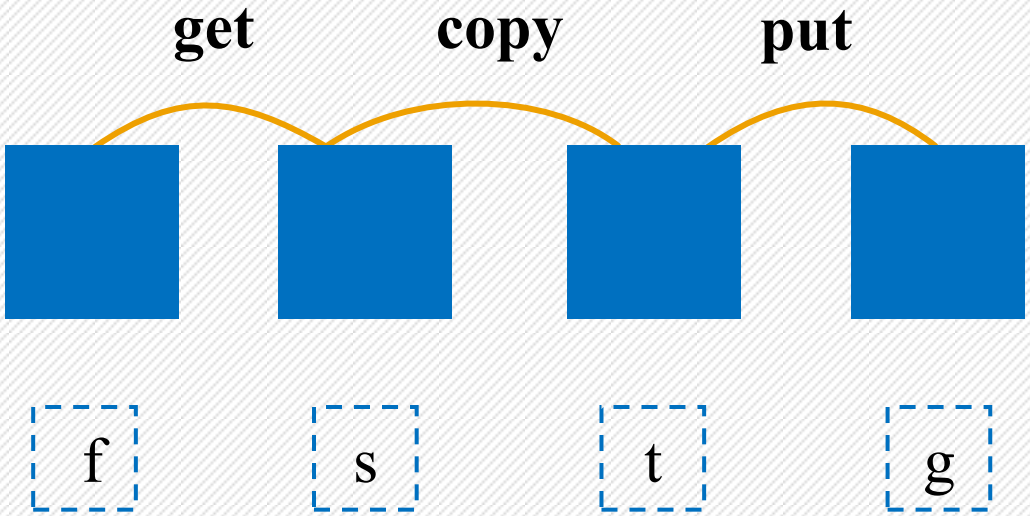
```
T1:
…
Read(x);
if  x>=1 then
    x:=x-1;
write(x);
…
```

```
T2:
…
Read(x);
if  x>=1 then
    x:=x-1;
write(x);
…
```

x = 1

共享变量的修改冲突

Background
背景



有3个进程

get, copy和put，它们对4个存储区域f、s、t和g进行操作。

操作顺序冲突

Interactions between processes
进程间的交互关系

- 互斥，指多个进程不能同时使用同一个资源；
- 同步，进程之间的协作；
- 死锁，指多个进程互不相让，都得不到足够的资源；

西安交通大学
软件学院

操作系统原理

Operating System Principle

田丽华

6-2 临界区

Access to critical areas
临界区的访问过程

- 对于临界资源，多个进程必须互斥的对它进行访问。
- **临界区(critical section)：**进程中访问**临界资源**的一段代码。
- **实现进程对临界资源的互斥访问—各进程互斥的进入自己的临界区**
- 假定一个系统有n个进程{P0,P1,……,Pn-1},每个进程有一个代码段称为临界区,在该区中进程可能修改共享变量\更新一个表\写一个文件等.

  当一个进程在临界区中执行时,其他进程都不能进入临界区

Access to critical areas
临界区的访问过程

临界区的执行在时间上是互斥的,进程必须请求允许进入临界区

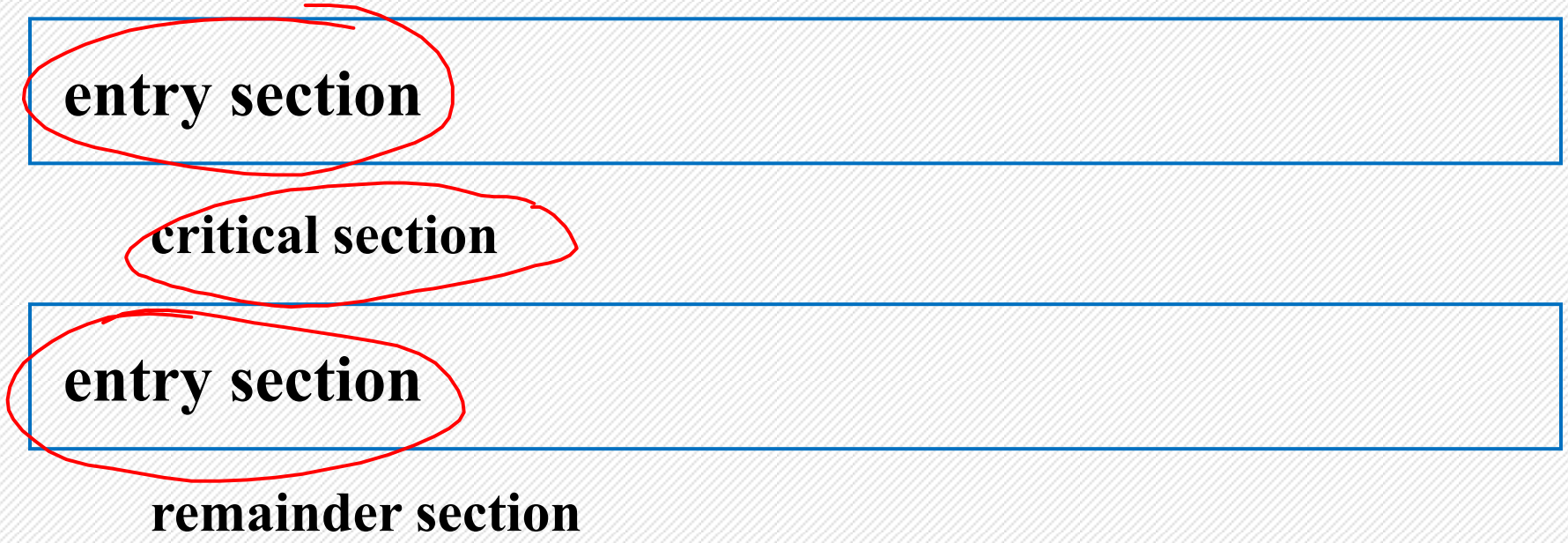
进入区(entry section)：在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应“正在访问临界区”标志。

退出区(exit section)：用于将"正在访问临界区"标志清除。

剩余区(remainder section)：代码中的其余部分。

Access to critical areas

临界区的访问过程



Solution to Critical-Section Problem
临界区问题的解决方案

Mutual Exclusion 互斥

If process Pi is executing in its critical section, then no other processes can be executing in their critical sections（假定进程Pi在其临界区内执行，其他任何进程将被排斥在自己的临界区之外）.

Solution to Critical-Section Problem
临界区问题的解决方案

Progress 有空让进

If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely（临界区虽没有进程执行，但有些进程需要进入临界区，不能无限期地延长下一个要进入临界区进程的等待时间）.

Solution to Critical-Section Problem
临界区问题的解决方案

Bounded Waiting 有限等待

. A bound 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（在一个进程提出进入临界区的请求和该请求得到答复的时间内，其他进程进入临界区的次数必须是有限的）.

如何实现进程间的互斥？

- 轮流？
- 申请？

两进程互斥的软件方法

算法1：

设立一个两进程公用的整型变量 **turn**：描述允许进入临界区的进程标识

有两个进程 **Pi**，**Pj**，如果 **turn==i**，那么进程**Pi**允许在其临界区执行

Pi

```
while (turn != i);
    critical section
turn = j;
    remainder section
```

Pj

```
while (turn != j;)
    critical section
turn = i;
    remainder section
```

- 在进入区循环检查是否允许本进程进入：
- **turn为i时，进程Pi可进入；**
- 在退出区修改允许进入进程标识：
- **进程Pi退出时，改turn为进程Pj的标识j；**

Algorithm 1

缺点：

强制轮流进入临界区，没有考虑进程的实际需要。

容易造成资源利用不充分：在进程1让出临界区之后，进程2使用临界区之前，进程1不可能再次使用临界区；

两进程互斥的软件方法

算法1只记住了哪个进程能进入临界区，没有保存进程的状态

设立一个标志数组**flag[]**：描述进程是否准备进入临界区，初值均为**FALSE**,先申请后检查。可防止两个进程同时进入临界区。

```
flag[i] = TRUE;          〈b〉
while (flag[j]);         〈a〉
  critical section
flag[i] = FALSE;
  remainder section
```

```
flag[j]=TRUE;
while(flag[i]);
  critical section
flag[j]=FALSE;
  remainder section
```

Algorithm 2

优点：

不用交替进入，可连续使用；

缺点：

两进程可能都进入不了临界区

Pi和Pj在切换自己flag之后和检查对方flag之前有一段时间，如果都切换flag，都检查不通过。

Algorithm 3

turn=j;描述可进入的进程（同时修改标志时）

在进入区先修改后检查，并检查并发修改的先后：

- 检查对方**flag**，如果不在临界区则自己进入 - - 空闲则入
- 否则再检查**turn**：保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入 - - 先到先入，后到等待

```
flag[i]=TRUE;turn=j
while(flag[j]&&turn==j);
    critical section
flag[i]=FALSE;
    remainder section
```

```
flag[j]=TRUE;turn=i
while(flag[i]&&turn==i);
    critical section
flag[j]=FALSE;
    remainder section
```

西安交通大学
软件学院

操作系统原理

Operating System Principle

田丽华

6-3 信号量

Semaphore

- 可以用临界区解决互斥问题，它们是平等进程间的一种协商机制
- OS可从进程管理者的角度来处理互斥的问题，信号量就是OS提供的管理公有资源的有效手段。

- 1965年，由荷兰学者Dijkstra提出（所以P、V分别是荷兰语的test(proberen)和increment(verhogen)），是一种卓有成效的进程同步机制。
- 用于保证多个进程在执行次序上的协调关系的相应机制称为进程同步机制。

信号量代表可用资源实体的数量。

Semaphore

- **Semaphore *S* – integer variable**（信号量*S* – 整型变量，代表可用资源实体的数量）
- **can only be accessed via two indivisible (atomic) operations**（除了初始化之外，仅能通过两个不可分割的[原子]操作访问，）

```
P (S):
        while S≤ 0 do no-op;
        S--;

V(S): S++;
```

- 存在忙等——自旋锁:进程在等待锁时自旋

- **Synchronization tool that does not require busy waiting**（一种不需要忙等待的同步工具）.

```
P (S):
        S--;
        if  S< 0 do block;
V(S):
        S++;
        if S<=0 then wakeup;
```

Semaphore Eliminating Busy-Waiting

纪录型信号量

```
semaphore s;
P(s) {
    s.value--;
    if (s.value < 0) {
        add this process to list s.L
        block
    }
}
V(s) {
    s.value++;
    if (s.value <= 0) {
        remove a process P from list
s.L
        wakeup(P);
    }
}
```

```
typedef struct{
    int value;
    struct process *L;
}semaphore
```

Semaphore

S是与临界区内所使用的公用资源有关的信号量

P(S):表示申请一个资源

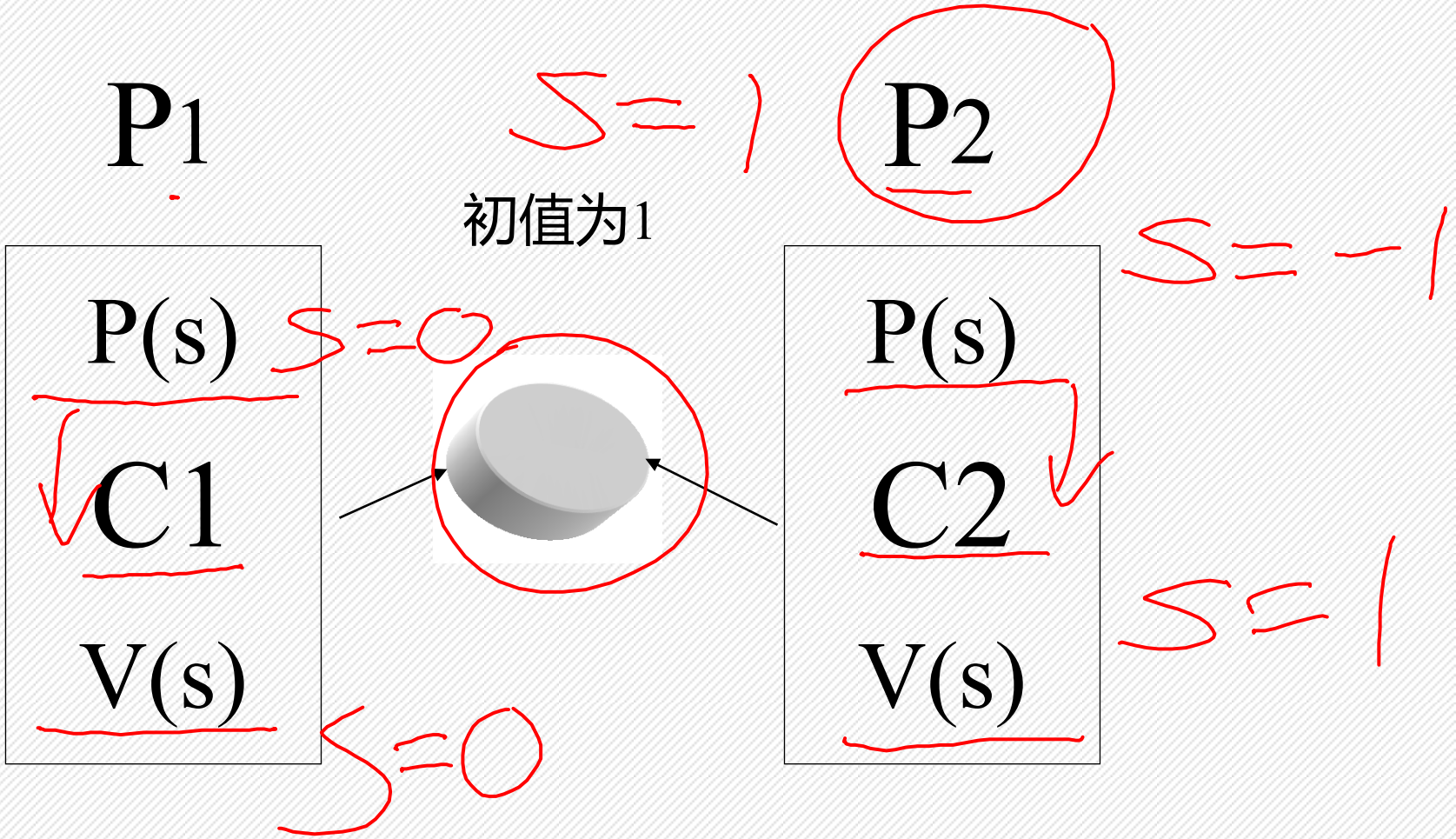
V(S):表示释放一个资源。

- 初始化指定一个非负整数值，表示空闲资源总数

在信号量经典定义下，信号量S的值不可能为负

- S≥0 可供并发进程使用的资源数
- S<0 其绝对值就是正在等待进入临界区的进程数

利用信号量来描述互斥关系



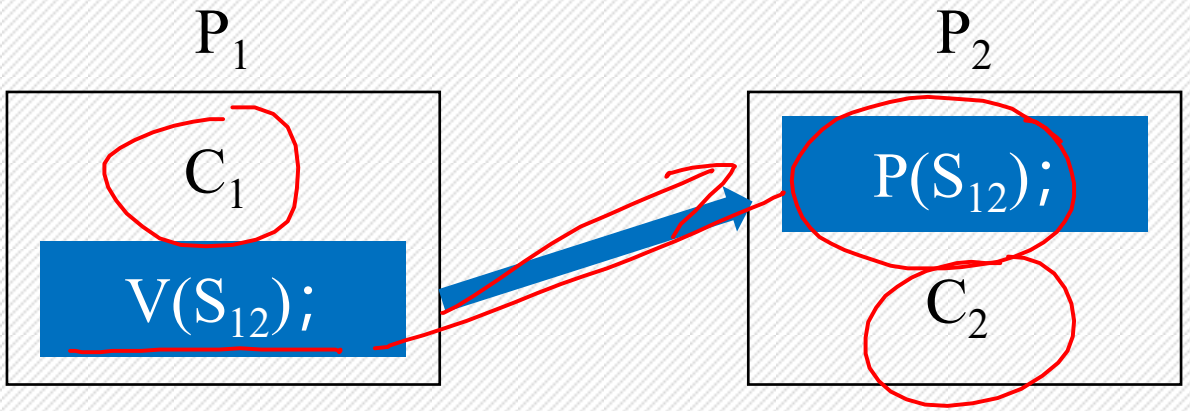
Semaphore as General Synchronization Tool

利用信号量实现互斥：
为临界资源设置一个互斥信号量，其初值为1；
Semaphore S; // initialized to 1

在每个进程中将临界区代码置于P(S)和V(S)原语之间

```
P(S);
    CriticalSection()
V(S);
```

利用信号量来描述前趋关系



- 前趋关系：并发执行的进程**P1**和**P2**中，分别有代码**C1**和**C2**，要求**C1**在**C2**开始前完成；
- 为每个前趋关系设置一个同步信号量**S12**，其初值为**0**

西安交通大学
软件学院

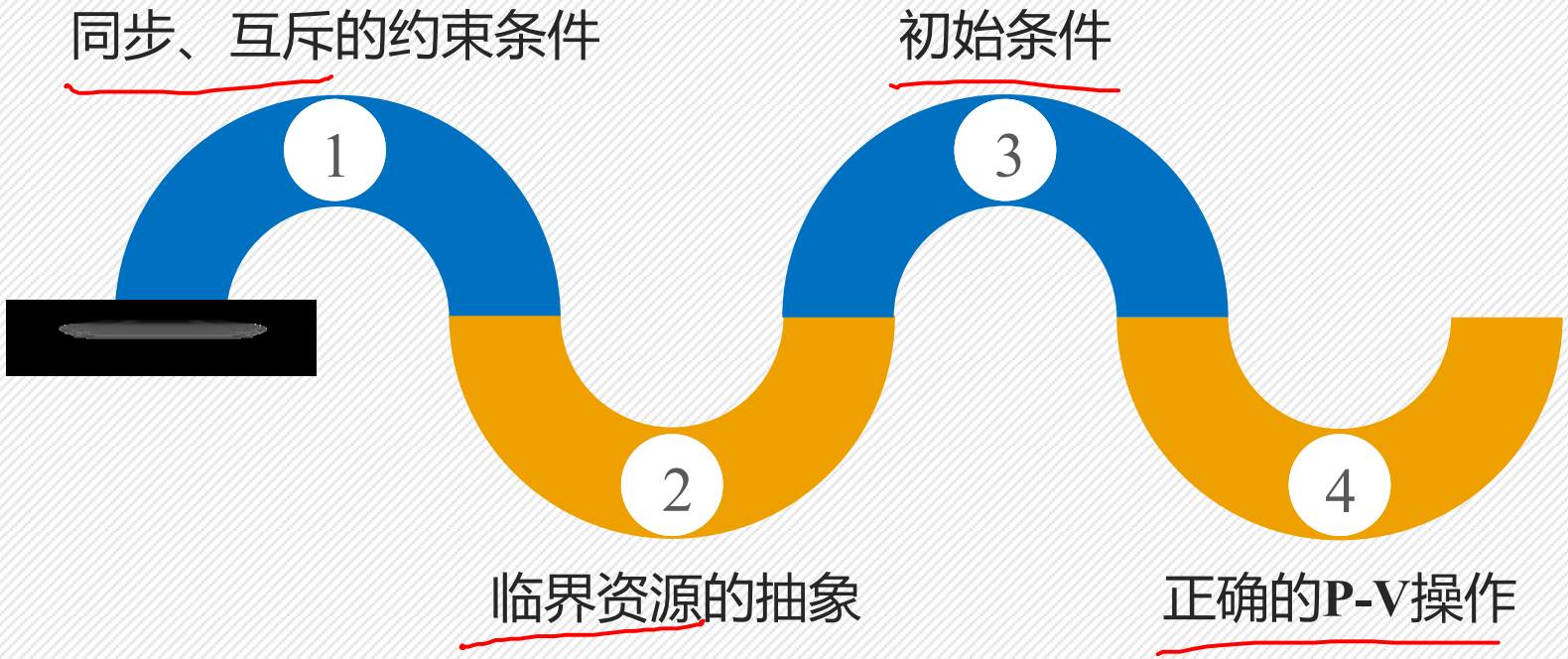
操作系统原理

Operating System Principle

田丽华

6-4 哲学家问题

Semaphore
信号量



Classical Problems of Synchronization

Dining-Philosophers Problem

(哲学家就餐问题)

Bounded-Buffer Problem

(有限缓冲区问题)

Readers and Writers Problem

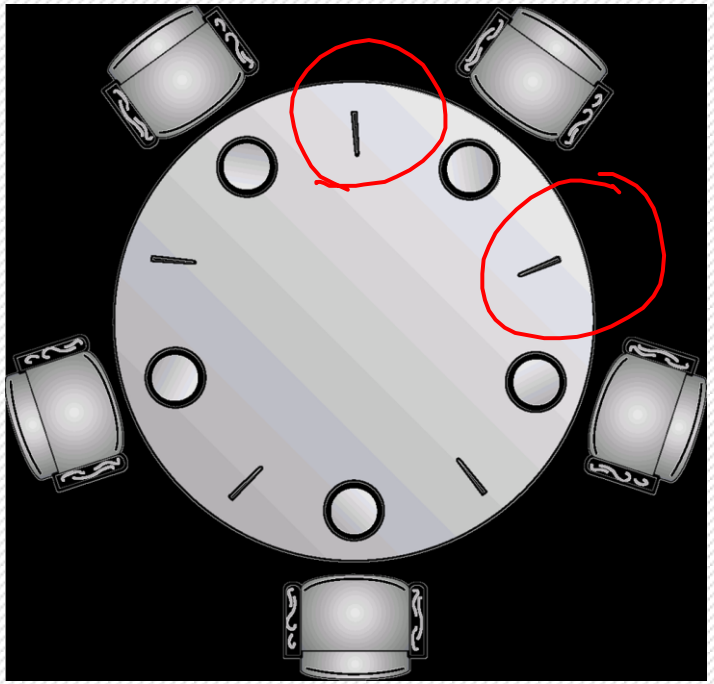
(读者写者问题)

哲学家进餐问题
(the dining philosophers problem)

问题描述：

（由Dijkstra首先提出并解决）5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子，每两个哲学家之间放一支；哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两支筷子，思考时则同时将两支筷子放回原处。如何保证哲学家们的动作有序进行？如：不出现相邻者同时进餐；

Dining-Philosophers Problem



- Shared data
- Semaphore chopStick[] = new Semaphore[5];

Philosopher(i)

Repeat

思考;

取**chopStick[i]**;

取**chopStick[(i+1) mod 5]**;

进餐;

放**chopStick[i]**;

放**chopStick[(i+1) mod 5]**;

Until false;

Dining-Philosophers Problem (Cont.)

Philosopher i:

```
while (true) {
    // get left chopstick
    chopStick[i].P();
    // get right chopstick
    chopStick[(i + 1) % 5].P();

    // eat for awhile

    //return left chopstick
    chopStick[i].V();
    // return right chopstick
    chopStick[(i + 1) % 5].V();

    // think for awhile
}
```

讨论

可能会出现死锁，五个哲学家每人拿起了他左边的筷子

01 最多允许四个哲学家同时就坐

02 同时拿起两根筷子

03 非对称解决

西安交通大学
软件学院

操作系统原理

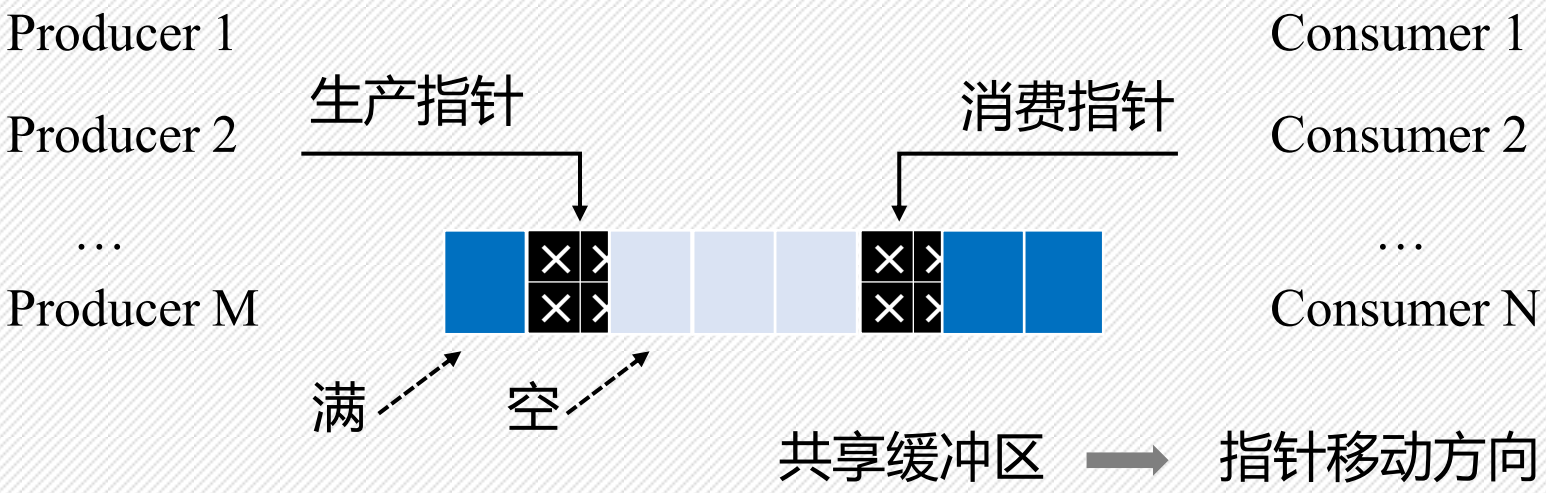
Operating System Principle

田丽华

6-5 有限缓冲区问题

生产者 - 消费者问题

问题描述：若干进程通过有限的共享缓冲区交换数据。其中，"生产者"进程不断写入，而"消费者"进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。



the producer-consumer problem
生产者 - 消费者问题

采用信号量机制：

- full是"满"数目，初值为0，empty是"空"数目，初值为N。实际上，full + empty == N
- mutex用于访问缓冲区时的互斥，初值是1
- 每个进程中各个P操作的次序是重要的：先检查资源数目，再检查是否互斥——否则可能死锁

the producer-consumer problem
生产者 - 消费者问题

Producer

```
P(empty);
P(mutex);      //进入区
One unit → buffer;
V(mutex);
V(full);       //退出区
```

Consumer

```
P(full);
P(mutex);      //进入区
One unit ← buffer;
V(mutex);
V(empty);      //退出区
```

Bounded-Buffer Problem

```
public class BoundedBuffer  {
      public BoundedBuffer() { /* see next slides */  }
      public void enter() { /* see next slides */  }
      public Object remove() { /* see next slides */  }

    private static final int   BUFFER_SIZE = 2;
      private Semaphore mutex;
      private Semaphore empty;
      private Semaphore full;
      private int in, out;
      private Object[] buffer;
}
```

生产者 - 消费者问题

Bounded Buffer Constructor

```
public BoundedBuffer() {
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
        mutex = new Semaphore(1);
        empty = new
Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }
```

the producer-consumer problem
生产者 - 消费者问题

enter() Method

```
public void enter(Object item) {
    empty.P();
    mutex.P();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    mutex.V();
    full.V();
}
```

生产者 - 消费者问题

remove() Method

```
public Object remove() {
    full.P();
    mutex.P();

    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.V();
    empty.V();

    return item;
  }
```

西安交通大学
软件学院

操作系统原理

Operating System Principle

田丽华

6-6 读写问题

the readers-writers problem

读者 - 写者问题

问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个

the readers-writers problem

读者 - 写者问题

如果读者来：

01 无读者、写者，新读者可以读

02 有写者等，但有其它读者正在读，则新读者也可以读

03 有写者写，新读者等

the readers-writers problem
读者 - 写者问题

采用信号量机制：

- 信号量Wmutex表示"允许写"，初值是1。
- 公共变量Rcount表示“正在读”的进程数，初值是0；
- 信号量Rmutex表示对Rcount的互斥操作，初值是1。

the readers-writers problem

读者 - 写者问题

Writer：

```
P(Wmutex);
    write;
V(Wmutex);
```

Reader：

```
P(Rmutex);
    if (Rcount==0)
        P(Wmutex);
    ++ Rcount;
V(Rmutex);
    ….
    read;
    ….
P(Rmutex);
    -- Rcount;
    if (Rcount==0)
        V(Wmutex);
V(Rmutex);
```

PV操作讨论

信号量的物理含义：

S>0表示有S个资源可用

S=0表示无资源可用

S<0则| S |表示S等待队列中的进程个数

P(S):表示申请一个资源

V(S):表示释放一个资源。

信号量的初值应该大于等于0

PV操作讨论

PV操作必须成对出现，有一个P操作就一定有一个V操作

当为互斥操作时 › 它们处于同一进程

当为同步操作时 › 则不在同一进程中出现

对于前后相连的两个P(S1)和P(S2)，顺序是至关重要的:

同步P操作应该放在互斥P操作前