北京邮电大学软件学院
School Of software Engineering Of BUPT

Operating Systems

Lecture 5 Process Synchronization

Jinpengchen
Email: jpchen@bupt.edu.cn

Catalog Description

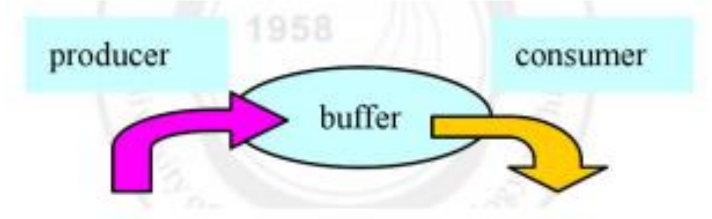
- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

- The processes are cooperating with each other directly or indirectly.
  - Independent process cannot affect or be affected by the execution of another process
  - Cooperating process can affect or be affected by the execution of another process
- Concurrent access (并发访问) to shared data may result in data inconsistency(不一致)
  - for example: printer, shared variables/ tables/ lists
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background

- Producer-Consumer Problem
  - Producer-Consumer Problem (生产者-消费者问题，PC问题): Paradigm for cooperating processes
    - ✓ producer (生产者) process produces information that is consumed by a consumer (消费者) process.
  - Shared-Memory solution
    - ✓ a buffer of items shared by producer and consumer



    - ✓ Two types of buffers
    - ✓ unbounded-buffer places no practical limit on the size of the buffer
    - ✓ bounded-buffer √ assumes that there is a fixed buffer size

Background

- Another solution using counting value
  - A solution to the PC problem that fills all the buffers (not BUFFER_SIZE-1).
  - An integer count: keeps track of the number of full buffers.
    - ✓ Initially, count = 0.
    - ✓ Incremented by the producer after it produces a new buffer, and decremented by the consumer after it consumes a buffer.

Background

- Producer

```
while (true) {
/* produce an item and put in
nextProduced */
while (count == BUFFER_SIZE)
; // do nothing
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
count++;
}
```

Background

- Consumer

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

Catalog Description

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

进程间互斥

- 进程互斥产生的原因
  - 进程宏观上并发执行，依靠时钟中断来实现微观上轮流执行
  - 访问共享资源

进程间互斥

- Example：两个进程，读-修改-写

```
  Process 1                         Process 2
tmp1= count;                      tmp2= count;
tmp1++;                           tmp2= tmp2+2;
count= tmp1;                      count= tmp2;
```

- 请问：如果在这些进程执行之前，count变量的值为1，那么它最后的结果是多少？

进程间互斥

- Example: Case 1

```
  Process 1                      Process 2
  tmp1= count;(=1)
  interrupt…
                               tmp2= count;(=1)
                               tmp2= tmp2+2;(=3)
                               count= tmp2;(=3)
tmp1++;(=2)
count= tmp1;(=2)
```

进程间互斥

- Example： Case 2

```
        Process 1                     Process 2
                                 tmp2= count;(=1)
                                 interrupt…

  tmp1= count;(=1)
 tmp1++;(=2)
  count= tmp1;(=2)

                                   tmp2= tmp2+2;(=3)
                                   count= tmp2;(=3)
```

进程间互斥

- Example： Case 3

Process 1	Process 2
tmp1= count;(=1)	
tmp1++;(=2)	
count= tmp1;(=2)	
	tmp2= count;(=２)
	tmp2= tmp2+2;(=４)
	count= tmp2;(=４)

进程间互斥

- Race condition(竞争状态)
  - 两个或多个进程对同一共享数据同时进行读写操作，而最后的结果是不可预测的，它取决于各个进程具体运行情况。
  - 解决之道：
    - ✓ 在同一时刻，只允许一个进程访问该共享数据，即如果当前已有一个进程正在使用该数据，那么其他进程不能访问。这就是互斥的概念。

上述例子有何问题？

进程间互斥

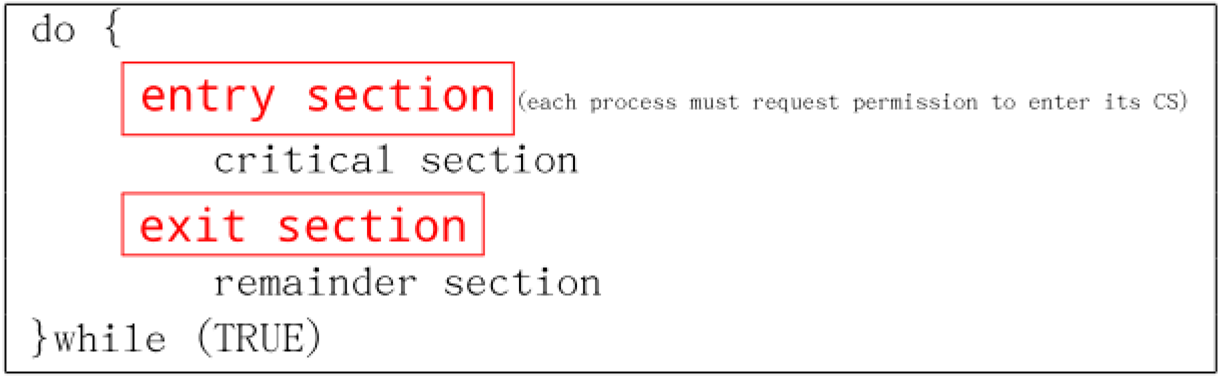
- 竞争状态问题的抽象描述
  - **把一个进程在运行过程中所做的事情分成两类**

✓进程内部的计算或其他的一些事情，肯定不会导致竞争状态的出现

✓对共享内存或共享文件的访问，可能会导致竞争状态的出现。需要一些概念来进行描述、

The Critical-Section Problem (临界区问题)

- Critical-Section (临界区)
  - Critical Resources(临界资源):
    - ✓ 在一段时间内只允许一个进程访问的资源
  - Critical Section (CS,临界区):a segment of code, access and may change shared data (critical resources)
    - ✓ Make sure, that any two processes will not execute in its own CSes at the same time
  - the CS problem is to design a protocol that the processes can use to cooperate.



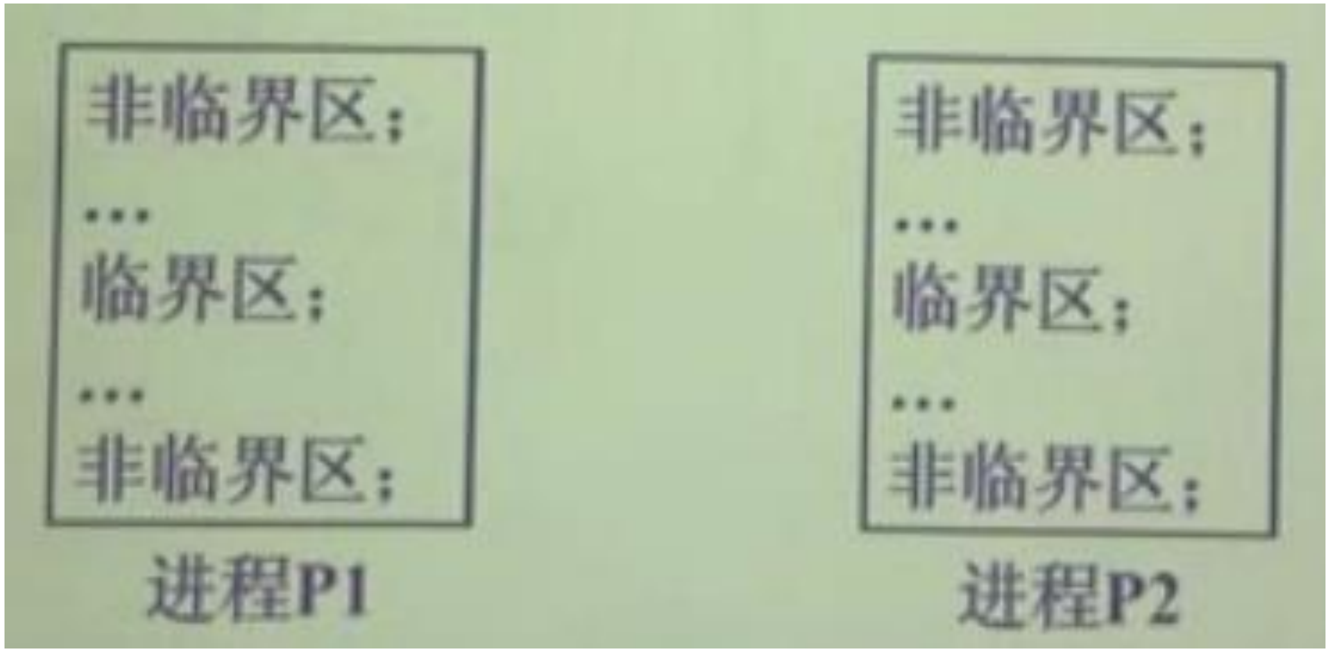
```
do {
        entry section  (each process must request permission to enter its CS)
            critical section
        exit section
            remainder section
}while (TRUE)
```

The Critical-Section Problem (临界区问题)

- A solution to the Critical-Section problem must satisfy:
  - Mutual Exclusion (互斥):
    - ✓ If process Pi is executing in its CS, no other processes can be executing in their CSes.
  - Progress (空闲让进):
    - ✓ If no process is executing in its CS and there exist some processes that wish to enter their CSes, the selection of the processes that will enter the CS next cannot be postponed indefinitely
  - Bounded Waiting (有限等待):
    - ✓ A bound must exist on the number of times that other processes are allowed to enter their CSes after a process has made a request to enter its CS and before that request is granted
      - -Assume that each process executes at a nonzero speed
      - -No assumption concerning relative speed of the N processes

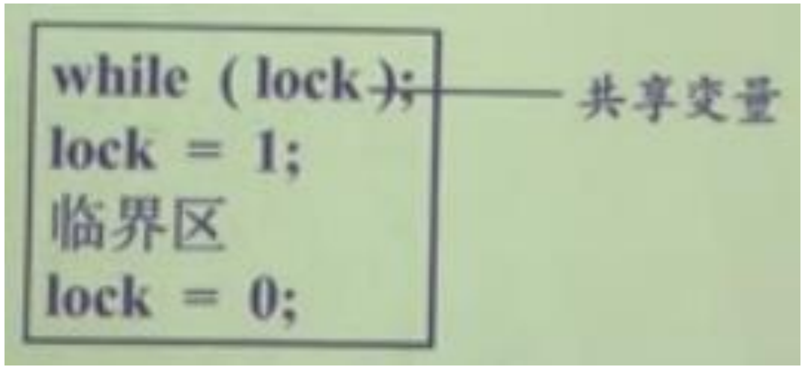
进程间互斥

- 如何实现两个进程之间的互斥访问
  - 问题描述：两个进程，在各自临界区中需要对某个共享资源进行访问



进程间互斥

- 基于繁忙等待的互斥实现
  - Method 1：加锁标志位法



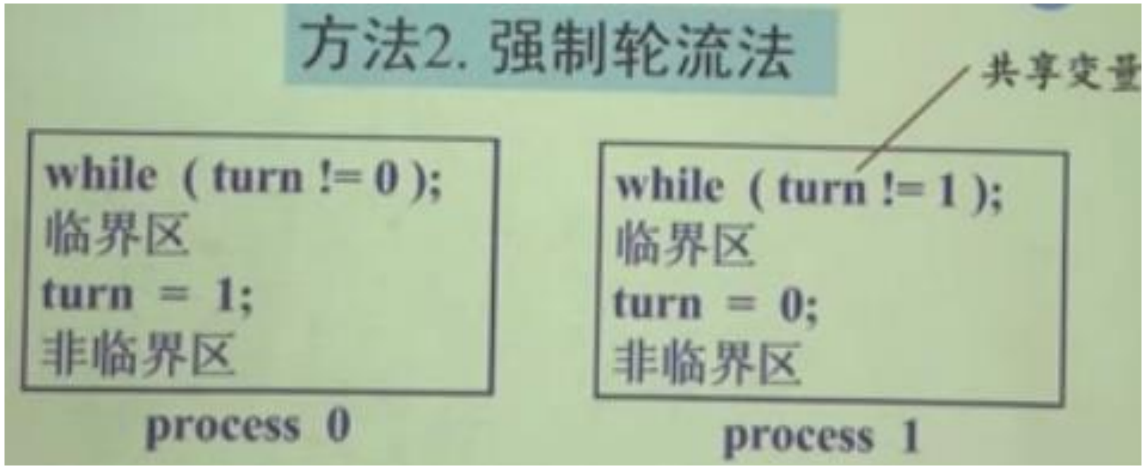
```
while ( lock );   ——— 共享变量
lock = 1;
临界区
lock = 0;
```

Lock的初始值为0，当一个进程想进入临界区时，先查看lock的值，若为1，说明已有进程在临界区内，只好循环等待。等它变成了0，才可进入。每个进程的操作类似。

缺点：可能出现针对lock的竞争状态问题。

进程间互斥

- 基于繁忙等待的互斥实现
  - Method 2：



方法2. 强制轮流法

```
while ( turn != 0 );
临界区
turn = 1;
非临界区
```

process 0

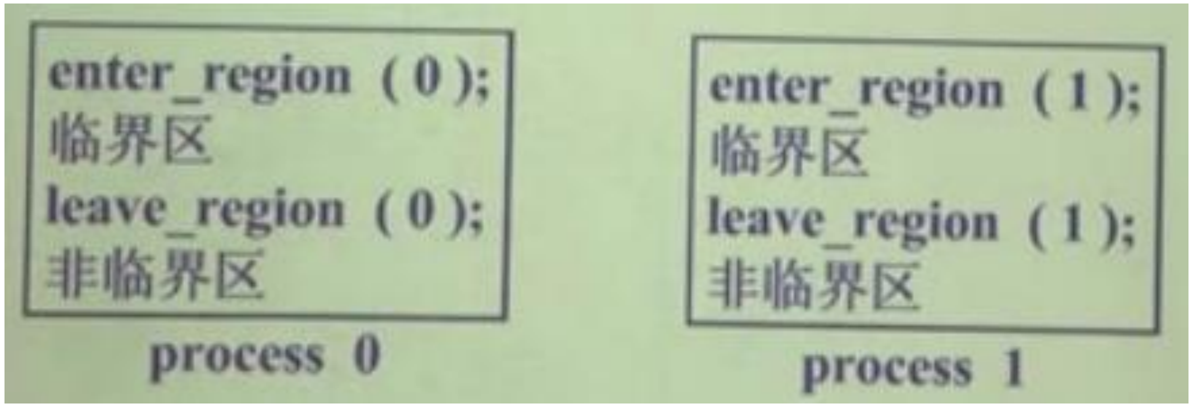
```
while ( turn != 1 );
临界区
turn = 0;
非临界区
```

process 1

基本思想：每个进程严格地按照轮流的顺序来进入临界区。
优点：保证在任何时刻最多只有一个进程在临界区
缺点：违反了互斥访问条件中的第三个条件

进程间互斥

- 基于繁忙等待的互斥实现
  - Method 3： Peterson方法



```
enter_region (0);
临界区
leave_region (0);
非临界区
```
process 0

```
enter_region (1);
临界区
leave_region (1);
非临界区
```
process 1

基本思想：当一个进程想进入临界区时，先调用enter_region函数，判断是否能安全进入，不能的话等待；当它从临界区退出后，需调用leave_region函数，允许其他进程进入临界区。两个函数的参数均为进程号。

进程间互斥

- 基于繁忙等待的互斥实现
  - Enter_region

```
#define FALSE 0;
#define TRUE 1;
#define N 2;  //进程的个数
int turn;   //轮到谁?
int interested[N];//兴趣数组，初始值均为false
void enter_region(int process) //process= 0或者1
{
   int other;  //另外一个进程的进程号
   other = 1- process;  //---中断（1）
   interested[process] = TRUE; //表明本进程感兴趣 ---中断（2）
   turn = process;   //设置标志位---中断（2）
   while (turn = = process && interested[other]= = TRUE）;//(4)
}
```

进程间互斥

- 基于繁忙等待的互斥实现
  - Leave_region

```
void leave_region(int process)
{
interested[process] = False; //本进程已离开临界区
}
```

  - Peterson方法解决了互斥访问的问题，而且不会相互妨碍，可以完全正常地工作。

进程间互斥

- 基于繁忙等待的互斥实现
  - **上述方法都是基于繁忙等待的策略，都可归纳为一种形式：当一个进程想要进入它的临界区，首先检查一下是否允许它进入，若允许，就直接进入；若不允许，就在那里循环地等待，一直等到允许它进入**
  - **缺点：**

✓　浪费CPU时间

✓　可能导致预料之外的结果（如 一个低优先级进程位于临界区中，这时有一个高优先级进程也试图进入临界区）

进程间互斥

- 一个低优先级进程正在临界区中；
- 另一个高优先级进程就绪了；
- 调度器把**CPU**分给高优先级进程；
- 该进程也想进入临界区；
- 高优先级进程将会循环等待，等待低优先级进程退出临界区；
- 低优先级进程无法获得**CPU**，无法离开临界区。
- ----形成死锁

进程间互斥

- 解决之道
  - 当一个进程无法进入临界区的时候，应该被阻塞
  - 当一个进程离开临界区时候，应该被唤醒
  - 克服了繁忙等待方法的两个缺点

进程间互斥

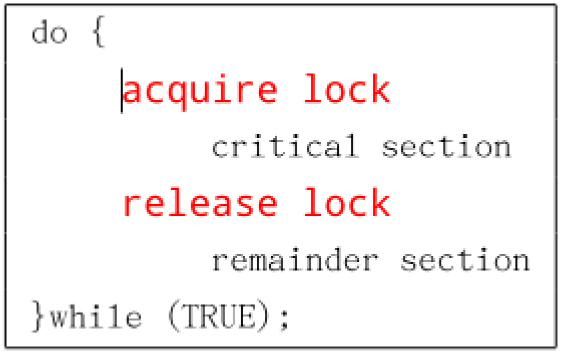
- 现有的进程互斥问题形式：两个或者多个进程都想进入自己的临界区，但在任何时刻，只允许一个进程进去临界区
- 新的进程互斥形式：两个或者多个进程都想进入自己的临界区，但在任何时刻，只允许N个进程同时进入临界区（**N>=1**）.

Catalog Description

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Synchronization Hardware

- Generally, any solution to the CS problem requires a LOCK
  - a process
    - ✓ acquires a lock before entering a CS
    - ✓ releases the lock when it exits the CS

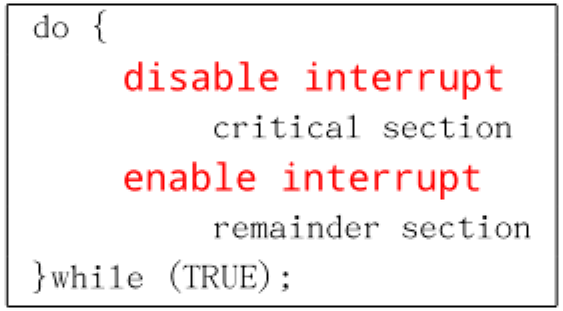


```
do {
        acquire lock
                critical section
        release lock
                remainder section
}while (TRUE);
```

  - CSes are protected by locks
  - Race conditions are prevented

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Current code would execute without preemption



```
do {
        disable interrupt
                critical section
        enable interrupt
                remainder section
}while (TRUE);
```

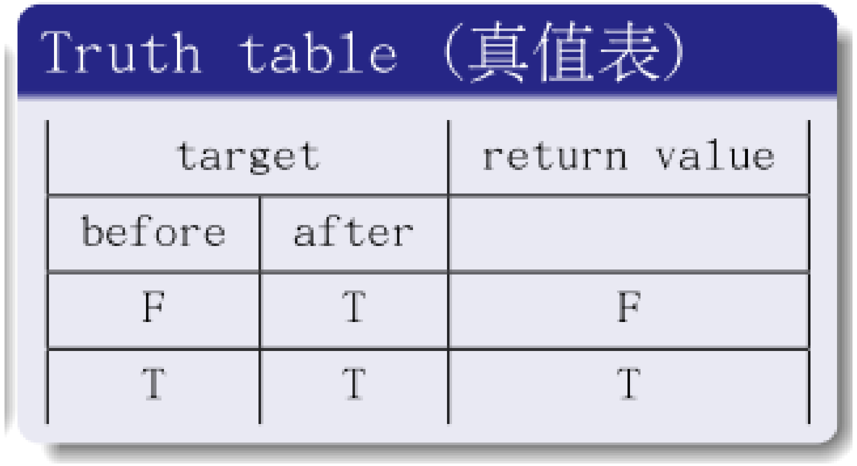
  - Generally too inefficient on multiprocessor systems, Oses using this not broadly scalable
- Modern machines therefore provide special atomic hardware instructions

      Atomic = non-interruptable
  - TestAndSet()   Swap()

Synchronization Hardware

- TestAndSet Instruction

```
Definition:
boolean  TestAndSet  (boolean  *target)  {
boolean  rv  =  *target;
*target  =  TRUE;
return  rv;
}
```



Truth table （真值表）

target		return value
before	after	
F	T	F
T	T	T

Synchronization Hardware

- Mutual-execution solution using TestAndSet
  - Shared boolean variable lock, initialized to false.
  - Solution:

```
while (true) {
while ( TestAndSet (&lock ))
;  //  do  nothing
//  critical  section
lock  =  FALSE;
//  remainder  section
}
```

  - busy-waiting?×
    starvation

Catalog Description

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Semaphores

- Less complicated
- Do not require busy waiting
- Semaphore S －integer variable(整型信号量)
- Two standard operations modify S:
  - wait() and signal()
  - Originally called
    - ✓ P() from the Dutch proberen, “to test”
    - ✓ V() from the Dutch verhogen,” to increment”
  - Can only be accessed via two indivisible (atomic) operations

Semaphores

wait()

```
wait(S) {
while  (S  <=  0)
;  //  no-op
S——;
}
```

signal()

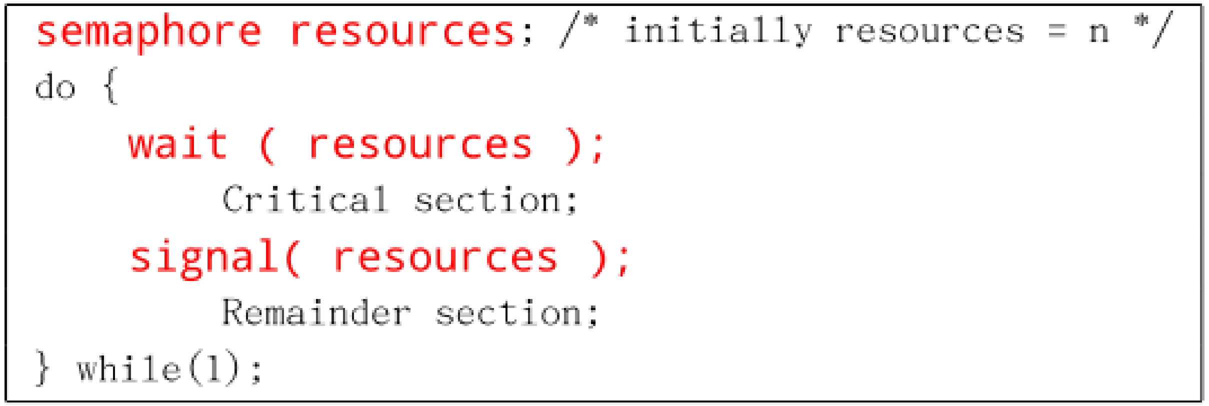
```
signal(S) {
S++;
}
```

Semaphores

- Using as
  - counting semaphore
    - ✓ control access to a given resource consisting of a finite
    - ✓ number of instances
  - binary semaphore
    - ✓ provide mutual exclusion, can deal with the critical-section problem for multiple processes

Semaphores

- Counting semaphore, also named as Resource semaphore
  - Initialized to N, the number of resources available
  - resource requesting: wait()
    - ✓ if the count of resource goes to 0, waiting until it becomes > 0
  - resource releasing: signal()
  - usage

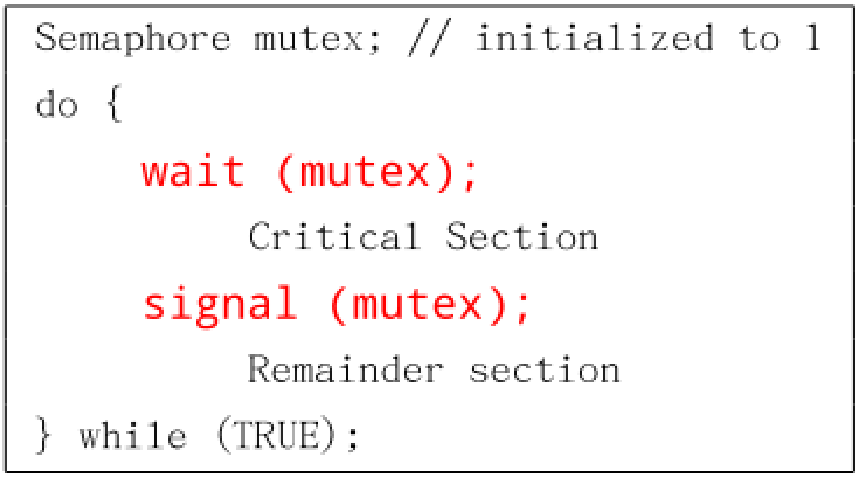


```
semaphore resources; /* initially resources = n */
do {
     wait ( resources );
          Critical section;
     signal( resources );
          Remainder section;
} while(1);
```

Semaphores

- Binary semaphores, also known as mutex locks (互斥锁), provides mutual
  - integer value: 0 or 1;
  - can be simpler to implement;

    Can implement a counting semaphore S as a binary semaphore
  - usage



```
Semaphore mutex; // initialized to 1
do {
      wait (mutex);
            Critical Section
      signal (mutex);
            Remainder section
} while (TRUE);
```

Semaphores

- Semaphore Implementation
  - Disadvantage:
    - ✓ the previous semaphore may cause busy waiting(忙等)
    - – this type of semaphore is also called a spinlock （自旋锁), suitable situation
  - Semaphore implementation with no busy waiting

    Record semaphore(记录型信号量)
    - ✓ With each semaphore there is an associated waiting queue.
    - ✓ Each entry in a waiting queue has two data items:
      - – value (of type integer)
      - – pointer to next record in the list
    - ✓ Two operations:

      block – place the process invoking the operation on the appropriate waiting queue.

      wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphores

- Semaphore Implementation
  - Record semaphore(记录型信号量)

```
typedef struct {
int value;
struct process *list; // a waiting queue
} semaphore;
```

wait()

```
wait(Semaphore *S){
S->value--;
if (S->value<0){
add this process to S->list;
block();
}
}
```

signal()

```
signal(semaphore *S){
S->value++;
if (S->value <= 0){
remove a process P from S->list;
wakeup(P);
}
}
```

Semaphores

- Semaphore Implementation
  - 分析S->value
    - ✓ 对于wait操作：

      当value≥1时，说明有资源剩余；申请资源只需要减1

      当value<1时，说明没有资源剩余；此时，减去1，并等待
    - ✓ 对于signal操作，

      若value ≥0，说明没有等待者，不必唤醒，只需加1释放资源

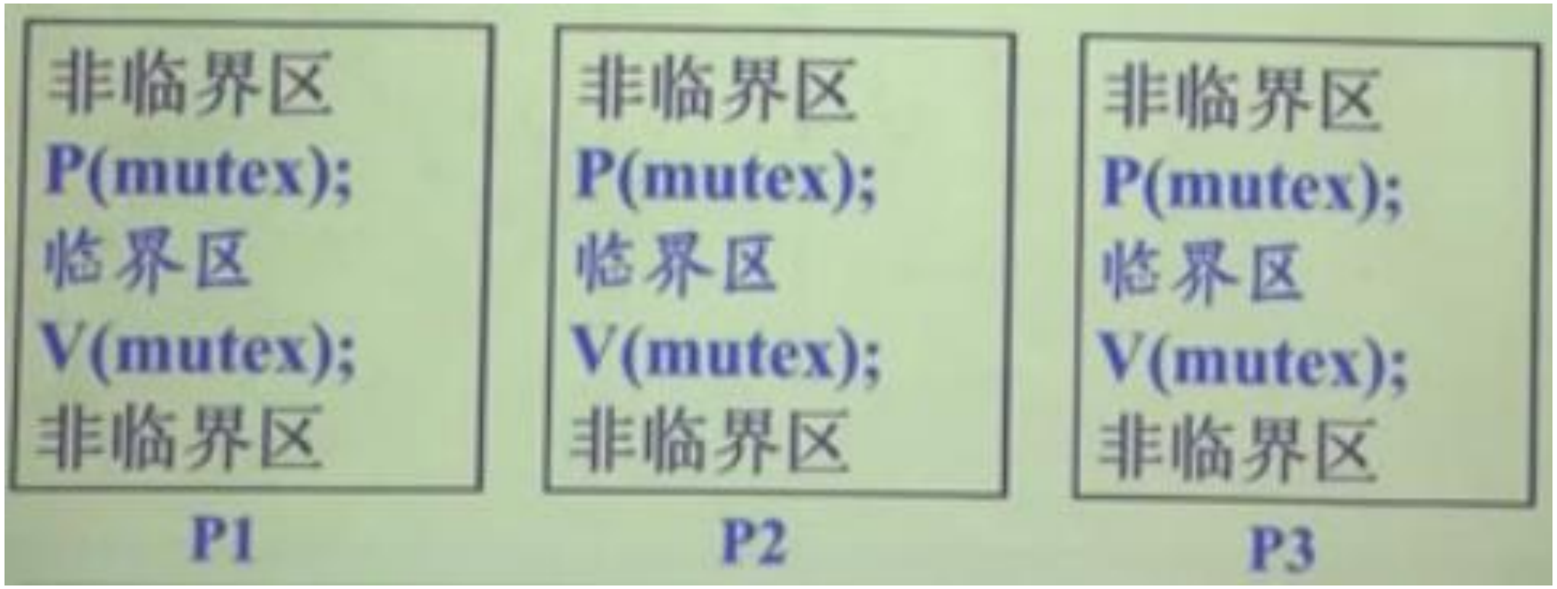
      若value<0，说明有等待者；加1缩短等待队列长度，并唤醒1个进程（资源分配给这个进程）
    - ✓ 查看value

      value ≥0，说明没有等待者，此时，value值表示剩余资源的个数

      value<0，说明有等待者，此时L上有等待进程；此时，value的绝对值表示等待进程的个数

Semaphores

- Semaphore Implementation
  - int value //共享变量，临界资源
  - Semaphore mutex //互斥信号量，初值为?



```
非临界区
P(mutex);
临界区
V(mutex);
非临界区
```
P1

```
非临界区
P(mutex);
临界区
V(mutex);
非临界区
```
P2

```
非临界区
P(mutex);
临界区
V(mutex);
非临界区
```
P3

Semaphores

- Misuse of semaphore: Deadlock and Starvation
  - Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S)	wait(Q)
wait(Q)	wait(S)
...	...
signal(S)	signal(Q)
signal(Q)	signal(S)

  - Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Catalog Description

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Classical Problems of Synchronization

- 进程间的同步是指多个进程中发生的事件存在某种时序关系，因此在各个进程之间必须协同合作，相互配合，使各个进程按照一定的速度执行，以共同完成某一项任务。
- 同步----合作
- 互斥-----竞争
- 只考虑基于信号量的同步

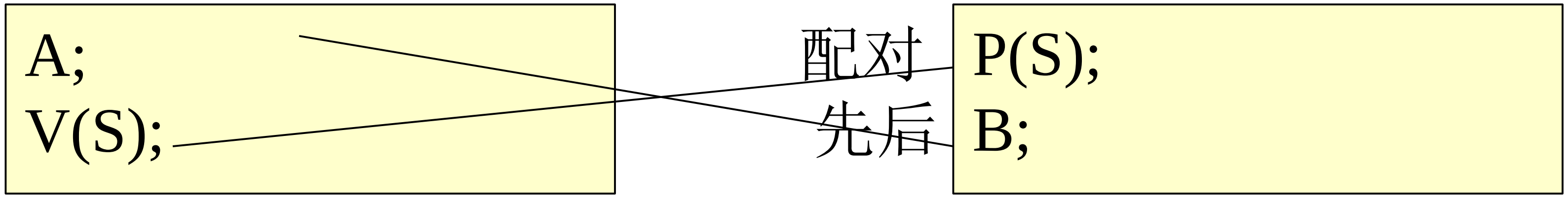
Classical Problems of Synchronization

- 如何实现A先执行，然后B执行？ 信号量S初值？

```
……
A(先);
……
```
P0

```
……
B(后);
……
```
P1

- S初值为0



```
A;
V(S);
```

```
P(S);
B;
```

P0 P1

Classical Problems of Synchronization

- Bounded-Buffer Problem，生产者-消费者问题（PC Problem）
- Readers and Writers Problem，读者-写者问题
- Dining-Philosophers Problem，哲学家就餐问题

Classical Problems of Synchronization

- Solution to Bounded-Buffer Problem (PC problem, 生产者-消费者问题)
  - N buffers, each can hold one item
  - Semaphore mutex initialized to the value 1
  - Semaphore full initialized to the value 0
  - Semaphore empty initialized to the value N.

The structure of the producer process

```
while (true) {
// produce an item
wait (empty);//是否有空闲缓冲区
wait (mutex);//进入临界区
// add the item to the buffer
signal (mutex);//离开临界区
signal (full);//新增一个产品
}
```

The structure of the consumer process

```
while (true) {
wait (full);//缓冲区有无产品
wait (mutex);//进入临界区
// remove an item from buffer
signal (mutex);//离开临界区
signal (empty);//新增一个空闲缓冲区
// consume the removed item
}
```

Classical Problems of Synchronization

- Solution to Readers-Writers Problem(读者-写者问题)
  - A data set is shared among a number of concurrent processes
    - ✓ Readers – only read the data set; they do not perform any updates
    - ✓ Writers – can both read and write
  - Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
  - Shared Data
    - ✓ Data set
    - ✓ Semaphore mutex initialized to 1
    - ✓ Semaphore wrt initialized to 1
    - ✓ Integer readCount initialized to 0

Classical Problems of Synchronization

- Solution to Readers-Writers Problem(读者-写者问题)

The structure of a writer process

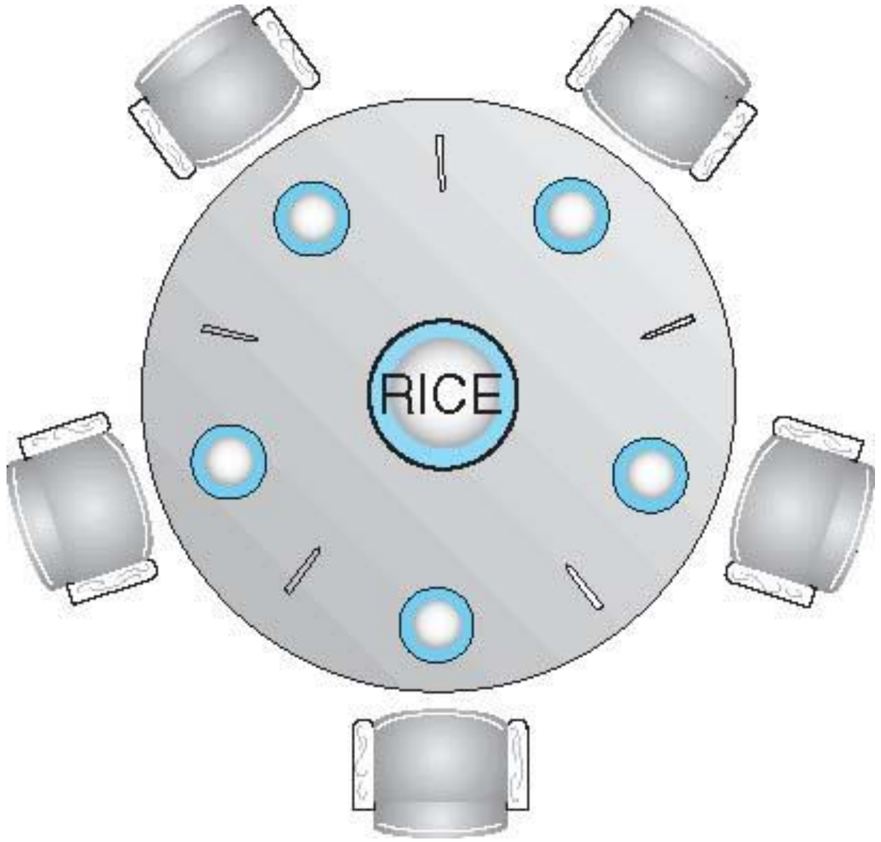
```
while (true) {
wait(wrt);
// writing is performed
signal(wrt);
}
```

The structure of a reader process

```
while (true) {
wait(mutex);
readcount ++;
if (readcount == 1)
wait(wrt);
signal(mutex);
// reading is performed
wait(mutex);
readcount - -;
if (readcount == 0)
signal(wrt);
signal (mutex);
}
```

Classical Problems of Synchronization

- Dining-Philosophers Problem （哲学家就餐问题）



Classical Problems of Synchronization

- Dining-Philosophers Problem （哲学家就餐问题）
  - Shared data
    - ✓ Bowl of rice (data set)
    - ✓ Semaphore chopstick [5] initialized to 1

This solution may cause a deadlock.
WHEN?

The structure of Philosopher i:

```
While (true) {
wait ( chopstick[i] );
wait ( chopStick[ (i + 1) % 5]);
// eat
signal ( chopstick[i] );
signal (chopstick[ (i + 1) %5] );
// think
}
```

Classical Problems of Synchronization

- Dining-Philosophers Problem （哲学家就餐问题）
  - Method 2

互斥访问，正确。
但是每次只允许一人进餐

```
The structure of Philosopher i:
Semaphore mutex;
While (true) {
think(); //哲学家正在思考
wait（mutex）; // 进入临界区
wait ( chopstick[i] ); //去拿左边的叉子
wait ( chopStick[ (i + 1) % 5]); //拿右边的叉子
// eat
signal ( chopstick[i] ); //放下左边叉子
signal (chopstick[ (i + 1) %5] ); //放下右边叉子
signal ( mutex ); //退出临界区
// think
}
```

Classical Problems of Synchronization

- Dining-Philosophers Problem （哲学家就餐问题）
- Several possible remedies
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - Odd philosophers pick up first her left chopstick and then her right chopstick, while even philosophers pick up first her right chopstick and then her left chopstick.

Catalog Description

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Monitors

- Monitor type: A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - encapsulates private data with public methods to operate on that data.
  - Mutual exclusion: Only one process may be active within the monitor at a time

Syntax of a monitor

```
monitor monitor-name {
// shared variable declarations
procedure P1 (…) {…}
…
procedure Pn (…) {…}
Initialization code (….)
{…}
}
```

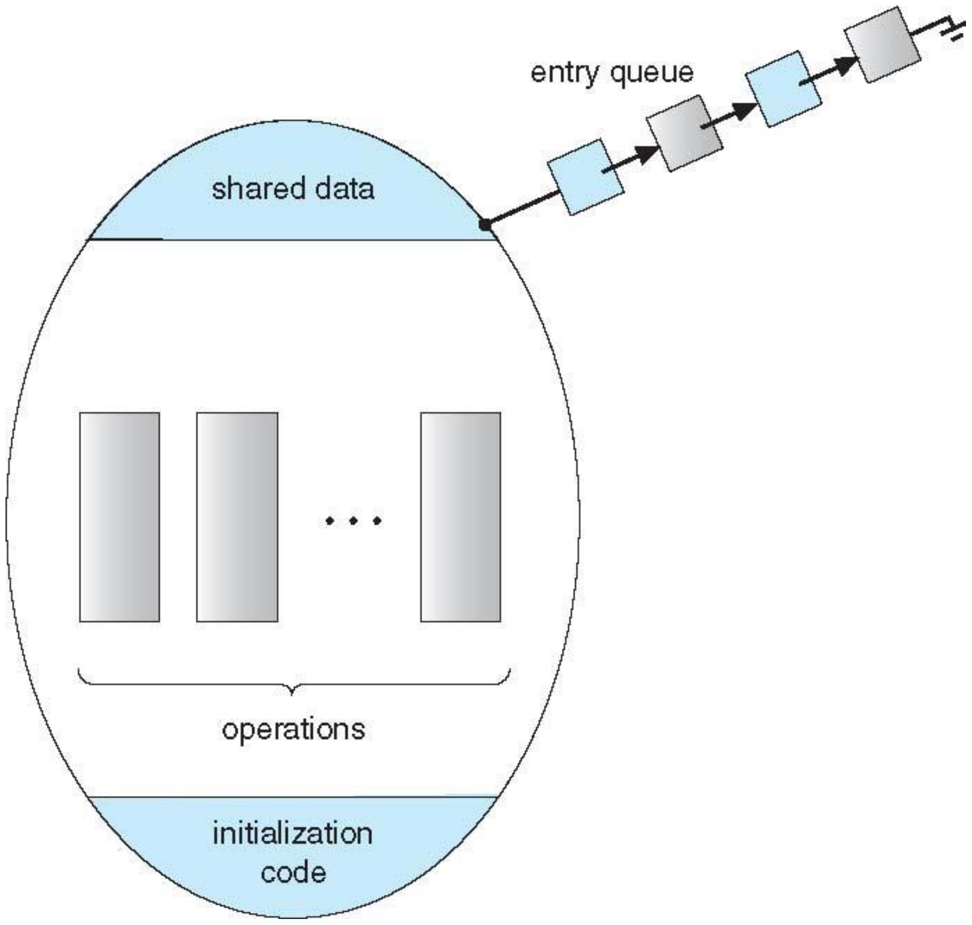
Within a monitor

✓ a procedure can access only local variables and formal parameters

✓ the local variables can be accessed by only the local precedures

Monitors

- Schematic view of a Monitor



Monitors

- Condition Variables
  - the monitor construct is not sufficiently powerful for modeling some synchronization scheme.
  - Additional synchronization mechanisms are needed.
  - Condition variables:

```
        condition  x, y;
```

  - ✓ Two operations on a condition variable:

x.wait()

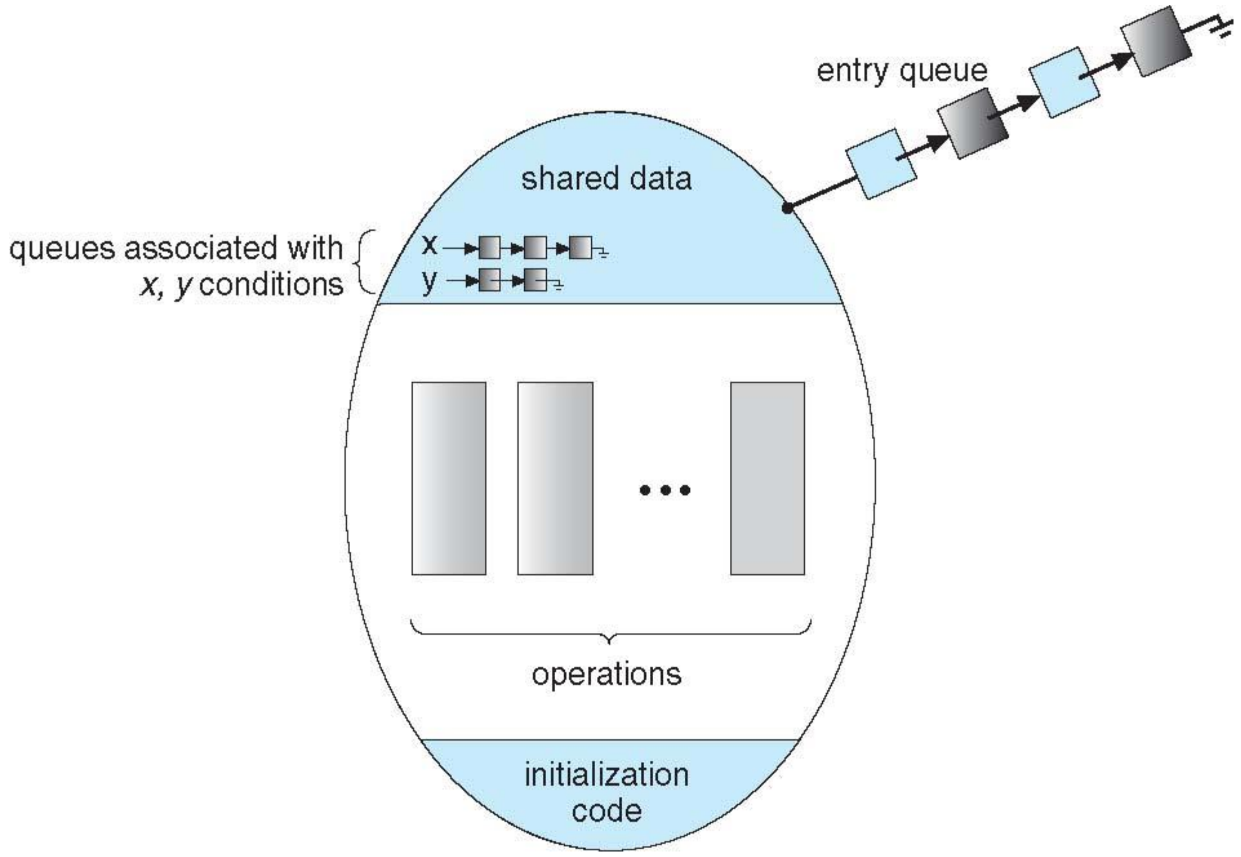
a process that invokes the operation is suspended.

x.signal()

resumes one of processes (if any) that invoked x.wait ()

Monitors

- Monitor with Condition Variables



The sleeping-barber problem

- Barbershop consists of a waiting room with N chairs and one barber room with one barber chair
- If there is no customers, the barber goes to sleep.
- If the barber is asleep, the customer wakes up the barber.

The sleeping-barber problem

- Var empty, full, mutex: Semaphore: n, 0, 1

```
Begin
 Parbegin
  Customer:
  Begin
   repeat
    wait(empty);
    wait(mutex);
     find a seat;
    signal(mutex);
    signal(full);
   until false;
  End
 ParEnd
END
```

```
Begin
 Parbegin
  Barber:
  Begin
   repeat
    wait(full);
    signal(empty);
     cutting;
   until false;
  End
 ParEnd
END
```

The sleeping-barber problem

- Var int waiting =0;

  Semaphore customer,barber,mutex;

  customer=0;barber=0;mutex=1;

```
Parbegin
  Customer:
  Begin
    wait(mutex);
    if(waiting<n)
      {waiting++;
       signal(customer);
       signal(mutex);
       wait(barber);
      }
     else{
       signal(mutex);}
  End
ParEnd
```

```
Parbegin
 Barber:
 Begin
  repeat
   wait(customer);
   wait(mutex);
   waiting--;
   signal(mutex);
   cutting;
   signal(barber);
  until false;
 End
ParEnd
```