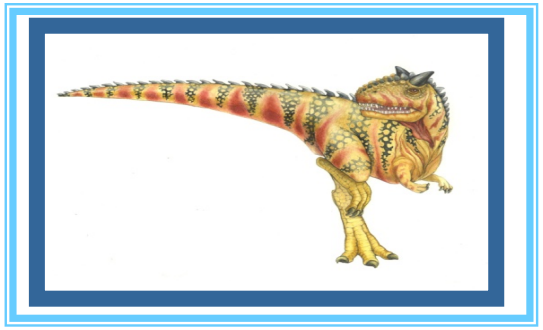
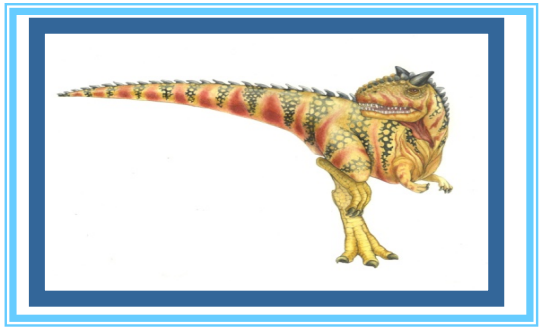
Chapter 6: Process Synchronization
进程同步



Process Synchronization

- 6.1 Background
- 6.2 The Critical-Section Problem
- 6.3 Synchronization Hardware
- 6.4 Semaphores
- 6.5 Classical Problems of Synchronization
- 6.6 Critical Regions
- 6.7 Monitors
- 6.8 OS Synchronization
- 6.9 Atomic Transactions

6.1 Background



6.1 Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Background (Cont.)

- Shared-memory solution to bounded-buffer problem (Chapter 3) allows at most *n*–1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer 有界缓冲区

- Shared data：

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

Bounded-Buffer (Cont.-1)

- Producer process(thread):

算法描述

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

Bounded-Buffer (Cont.-2)

- Consumer process (thread):

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

Bounded Buffer (Cont.-3)

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

原子操作-重要概念

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer (Cont.-4)

- The statement "count++" may be implemented in machine language as:

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

- The statement "count—" may be implemented as:

```
register2 = counter
register2 = register2 – 1
counter = register2
```

Bounded Buffer (Cont.-5)

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer (Cont.-6)

注意：单个CPU

- Assume counter is initially 5. One interleaving of statements is:
  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}
- The value of count may be either 4 or 6, where the correct result should be 5.
- Example： *CriticalSection.cpp*

例题分析

- 有两个进程P1、P2，它们分别执行下面的程序体，其中total是两个进程都能访问的共享变量，初值为0（可理解为共享存储段中的存储单元），count是每个进程的私有变量。假设这两个进程并发执行，并可自由交叉（interleave），则这两个进程都执行完后，变量total可能得到的最小取值是

  A. 50　B. 1　　C. 2　　D. 3

```
P1:{
    int count;
    for ( count =1; count <= 50; count++ )
        total = total + 1;  }

P2:  {
    int count;
    for ( count =1; count <= 50; count++ )
        total = total + 2;    }
```

例题分析

- 分析：total=total+1和total=total +2经过编译后各为三条指令:

total=total +1:	**total=total +2:**
register1= total	register2 = total
register1 = register1 + 1	register2 = register2 + 2
total = register1	total = register2

P1和P2并发执行过程中，这些指令会交替运行。如果两个进程按如下顺序执行：

P1第1次循环：register1 = total (*register1 = 0*)

P1第1次循环：register1 =register1+1 (*register1 =1*)

P2：循环49次

P1第1次循环：total =register1 (*total =1*)

P2第50次循环：register2 = total (*register2 = 1*)

P2第50次循环：register2 = register2 + 2 (*register2 = 3*)

P1：循环48次

P1第50次循环：register1 = total (*register1 = 49*)

P1第50次循环：register1 =register1+1 (*register1 =50*)

P1第50次循环：total =register1 (*total =50*)

P2第50次循环：total =register2 (*total =3*)

两个进程运行结束后，变量total的值为3。

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. Outcome of execution depends on the particular order in which the access takes place.

- To prevent race conditions, concurrent processes must be synchronized.

进程同步的概念

不是同一时刻！！

- 进程之间竞争资源面临三个控制问题：
  - 互斥（mutual exclusion）指多个进程不能同时使用同一个资源
  - 死锁（deadlock）指多个进程互不相让，都得不到足够的资源。永远得不到资源
  - 饥饿（starvation）指一个进程长时间得不到资源（其它进程可能轮流占用资源）。资源分配不公平

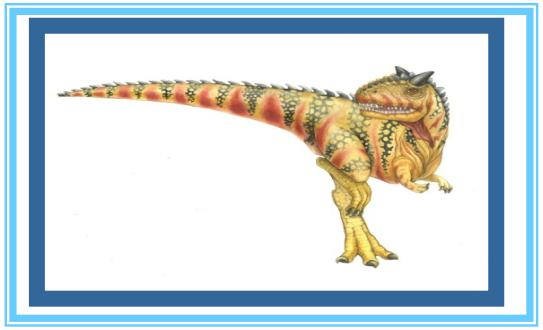
重要概念：进程并发执行

进程两个最基本特性：并发行、独立性

并发执行 → 竞争资源 → 互斥

并发执行 → 合作 → 同步

6.2 The Critical-Section Problem



6.2 The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section（临界区）*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Producer process(thread):

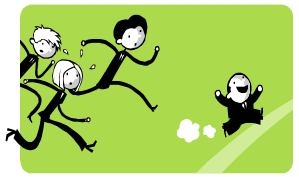
```
while (1) {
        while (counter == BUFFER_SIZE) ;
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
        }
```

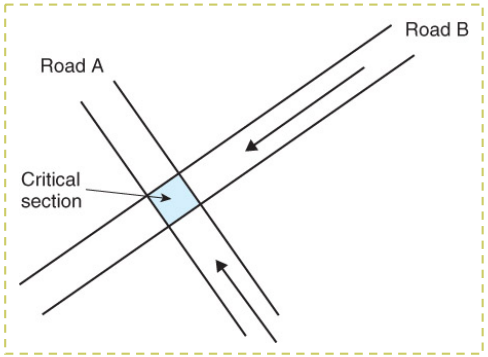
Consumer process (thread):

```
while (1) {
        while (counter == 0);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        }
```

Critical-Section

- 临界资源：一次只允许一个进程使用(访问)的资源。如：硬件打印机、磁带机等，软件的消息缓冲队列、变量、数组、缓冲区等。
- 临界区：访问临界资源的那段代码





- General structure of process P_i (other process P_j)

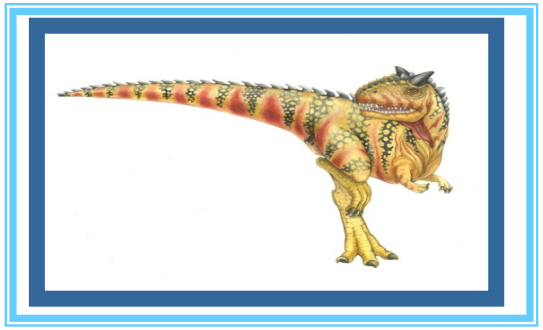
```
do {
        entry section          //进入区
        critical section       //临界区
        exit section           //退出区
        reminder section       //剩余区
} while (1);
```

Solution to Critical-Section Problem must satisfy

1. Mutual Exclusion(互斥): If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress（空闲让进）: 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.
3. 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

- 让权等待——不是必须的

6.3 Peterson's Solution



6.3 Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

Software Solutions
软件方法解决进程互斥—软件机制

Algorithm 1

- Only 2 processes, P_0 and P_1
- Shared variables:
  - int turn;
    initially turn = 0
  - turn = i ⇒ P_i can enter its critical section, i=0,1
- Process P_0, P_1

Two-Process Solutions

```
P0:
do {
    while (turn != 0) ;
          critical section
     turn = 1;
          remainder section
    } while (1);
```

```
P1:
do {
    while (turn != 1) ;
          critical section
     turn = 0;
          remainder section
    } while (1);
```

entry section

exit section

- Satisfies mutual exclusion, but not progress
- 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Pi出让临界区之后，Pj使用临界区之前，Pi不可能再次使用临界区；

Algorithm 2

Two-Process Solutions

- Shared variables
  - boolean flag[2];
    initially flag [0] = flag [1] = false.
  - flag [i] = true ⇒ P_i ready to enter its critical section
- Process P_0、P_1

P_0:

```
do {
        flag[0] = true;
        while (flag[1]) ;
            critical section
        flag [0] = false;
            remainder section
    } while (1);
```

P_1:

```
do {
        flag[1] = true;
        while (flag[0]) ;
            critical section
        flag [1] = false;
            remainder section
    } while (1);
```

- Satisfies mutual exclusion, but not progress requirement.
- P_0和P_1可能都进入不了临界区。当P_0执行了flag[0] := true后， 然后P1执行了flag[1] := true，这样两个进程都无法进入临界区

Algorithm 2-1

Two-Process Solutions

- Shared variables
  - boolean flag[2];
    initially flag [0] = flag [1] = false.
  - flag [i] = true $\Rightarrow P_i$ enter its critical section
- Process P_0、P_1

P_0:	P_1:
do {	do {
while (flag[1]) ;	while (flag[0]) ;
flag[0] = true;	flag[1] = true;
critical section	critical section
flag [0] = false;	flag [1] = false;
remainder section	remainder section
} while (1);	} while (1);

Satisfies progress, but not mutual exclusion requirement.

- P0和P1可能同时进入临界区。当flag [0] = flag [1] = false时， P0执行了while (flag[1])后， P1执行while (flag[0])， 这样两个进程同时进入了临界区

Algorithm 3(Peterson)

- Combined shared variables of algorithms 1 and 2.
- The two processes share two variables:
  - int turn;
  - boolean flag[2]

Two-Process Solutions

- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!
- Process P_i

```
P0:
do {  flag[0]= true;
      turn = 1;
      while (flag[1] and turn = 1) ;
            critical section
      flag[0] = false;
          remainder section
} while (1);
```

```
P1:
do {  flag[1]= true;
      turn = 0;
      while (flag[0] and turn =0) ;
            critical section
      flag[1] = false;
          remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Multiple-Process Solutions
Bakery Algorithm (面包房算法)，Lamport

- Algorithm of solving the Critical section for n processes
- Before entering its critical section, process receives a number. 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

- Define the following Notation

  order : (a,b) -- (ticket #, process id #)

  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0, \ldots, a_{n-1})$ is a number, k

    such that $k \geq a_i$ for $i = 0, \ldots, n-1$

- Shared data

  ```
  boolean choosing[n];
  int number[n];
  ```

  Data structures are initialized to false and 0 respectively

  - choosing[n]为真，表示进程i正在获取它的排队登记号；
  - number[i]的值，是进程i的当前排队登记号。如果值为0，表示进程i未参加排队，不想获得该资源。

The structure of process P_i in the Bakery Algorithm

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++)
     { while (choosing[j]) ;
       while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;
     }
       critical section
    number[i] = 0;
       remainder section
} while (1);
```

保证计算number[i]值的原子操作

6.4 Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Hardware Solutions
硬件方法解决进程互斥—硬件机制

Test-and-Set（测试与设置）

- Hardware features can make the programming task easier and improve system efficiency.
- Hardware instruction: Test and modify the content of a word atomically.

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

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
while(1) {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
};
```

The definition of Swap instruction

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to false):

```
boolean lock;
```

- Process P_i

```
while(1) {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock,&key);
    critical section
    lock = false;
    remainder section
}
```

- 硬件方法的优点
  - 适用于任意数目的进程，在单处理器或多处理器上
  - 简单，容易验证其正确性
  - 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量
- 硬件方法的缺点
  - 等待要耗费CPU时间，不能实现"让权等待"
  - 可能“饥饿”：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
  - 可能死锁：单CPU情况下，P1执行特殊指令(swap)进入临界区，这时拥有更高优先级P2执行并中断P1，如果P2又要使用P1占用的资源，按照资源分配规则拒绝P2对资源的要求，陷入忙等循环。然后P1也得不到CPU，因为P1比P2优先级低。

讨论

如何解决可能产生的死锁？

Bounded-waiting mutual exclusion with TestAndSet

Boolean waiting[n], lock ; to be initialize to false

```
while(1) {
    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 ;
}
```

解决“饥饿”问题

选择下一个进入临界区的进程，
选择下一个waiting[j]=false

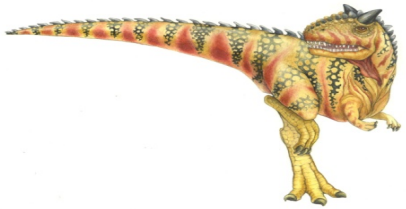
Bounded-waiting mutual exclusion with TestAndSet

- This algorithm satisfies all the critical section requirement .
- To prove that the mutual-exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i]== false or key == false. The value of key can become false only if the TestAndSet is executed. The first process to execute the TestAndSet will find key== false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement .

自旋锁 spinlock

- Windows、Linux内核用来达到多处理器互斥的机制“自旋锁”，它类同于TS指令机制。自旋锁是一个与共用数据结构有关的锁定机制。
- 自旋锁像它们所保护的数据结构一样，储存在共用内存中。为了速度和使用任何在处理器体系下提供的锁定机构，获取和释放自旋锁的代码是用汇编语言写的。例如在Intel处理器上，Windows使用了一个只在486处理器或更高处理器上运行的指令。
- 当线程试图获得自旋锁时，在处理器上所有其它工作将终止。因此拥有自旋锁的线程永远不会被抢占，但允许它继续执行以便使它尽快把锁释放。内核对于使用自旋锁十分小心，当它拥有自旋锁时，它执行的指令数将减至最少。

6.5 Semaphores



信号量(Semaphores)

- 1965年，荷兰学者Dijkstra提出的信号量机制是一种卓有成效的进程同步工具，在长期广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量机制发展到记录型信号量机制，进而发展为“信号集”机制。现在信号量机制已广泛应用于操作系统了同步互斥中。
- 信号量：
  - 整型信号量（integer semaphore）
  - 记录型信号量（record semaphore）
  - AND型信号量，信号量集
  - 二值信号量(binary semaphore)

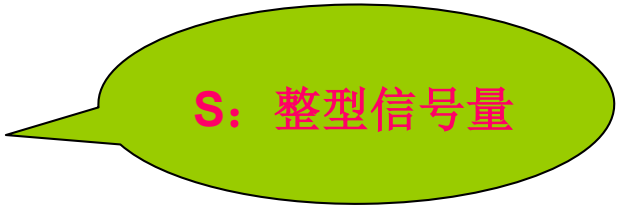
Semaphore（信号量）

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated

一种数据类型，仅初始化、wait、signal操作（运算）

Semaphore（信号量）

- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }



Semaphore Solutions
信号量方法解决进程同步与互斥—
信号量机制

Semaphore as General Synchronization Tool

- Counting semaphore （计数信号量）– integer value can range over an unrestricted domain
- Binary semaphore（二值信号量） – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore

Semaphore as General Synchronization Tool

- Provides mutual exclusion

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

spinlock

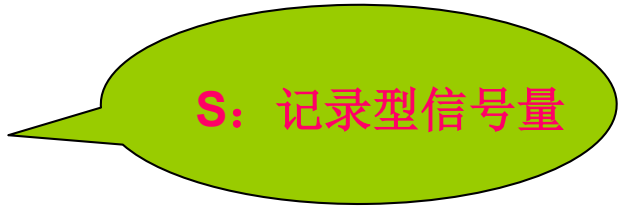
Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- 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
- Define a semaphore as a record

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



Semaphore Implementation with no Busy waiting

- Two operations:
  - block() – place the process invoking the operation on the appropriate waiting queue.
    - running → waiting
  - wakeup() – remove one of processes in the waiting queue and place it in the ready queue.
    - waiting→ready

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

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

记牢wait、signal操作

- Implementation of signal:

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

wait、signal操作讨论

- 通常用信号量表示资源或临界区
- 信号量的物理含义
  - S.value >0 表示有S.value个资源可用；
  - S.value=0 表示无资源可用或表示不允许进程再进入临界区；
  - S.value<0 则|S.value|表示在等待队列中进程的个数或表示等待进入临界区的进程个数。

- wait(S)≡P(S) ≡ down(S) ： 表示申请一个资源
- signal(S)≡V(S) ≡ up(S) ： 表示释放一个资源

wait、signal操作讨论

- wait、signal操作必须成对出现，有一个wait操作就一定有一个signal操作。一般情况下：当为互斥操作时，它们同处于同一进程；当为同步操作时，则不在同一进程中出现。
- 如果两个wait操作相邻，那么它们的顺序至关重要，而两个相邻的signal操作的顺序无关紧要。一个同步wait操作与一个互斥wait操作在一起时，同步wait操作在互斥wait操作前。
- wait、signal操作的优缺点
  - 优点：简单，而且表达能力强
  - 缺点：不够安全；wait、signal操作使用不当会出现死锁；实现复杂。

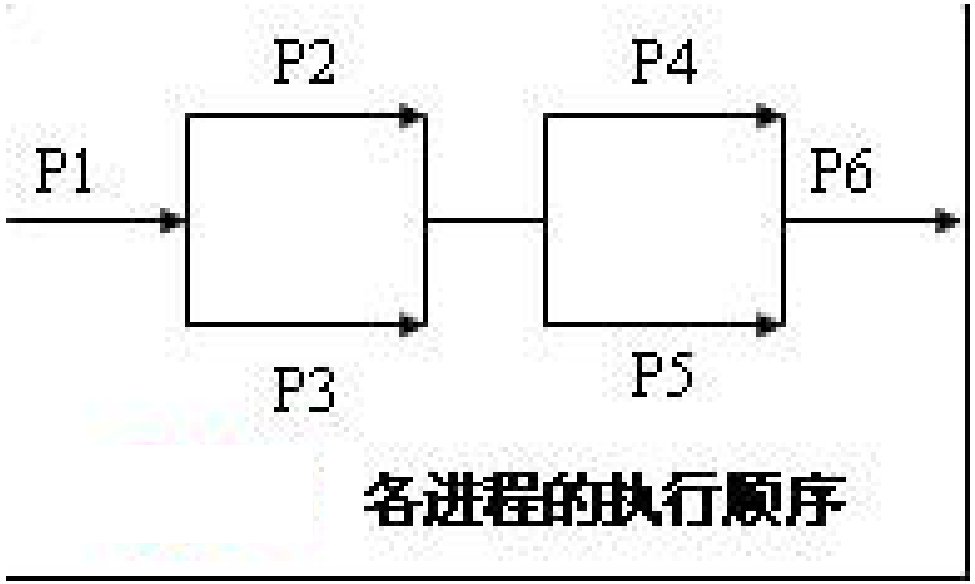
Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
…	…
A	*wait*(*flag*)
signal(*flag*)	B
…	…

实例

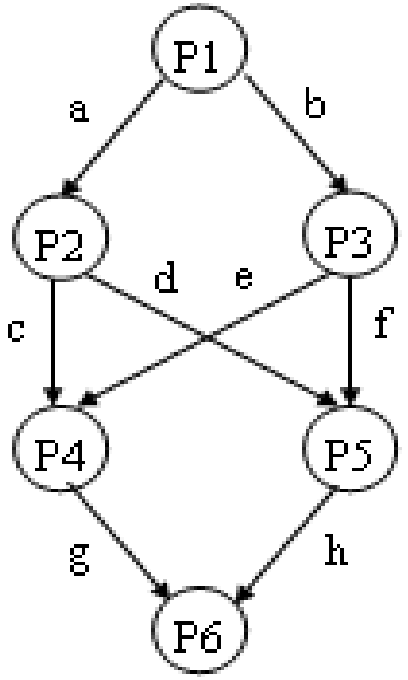
- 一组合作进程，执行顺序如图。请用P（wait）、V（signal）操作实现进程间的同步操作。



参考答案

- 图示出了上述并发进程之间的前趋关系，为了使上述进程同步，可设置8个信号量a、b、c、d、e、f、g、h，它们的初值均为0，而相应的进程可描述为（其中“…”表示进程原来的代码）：

```
main( )
cobegin{
    P1( ) { …; V(a); V(b); }
    P2( ) { P(a); …; V(c); V(d); }
    P3( ) { P(b); …; V(e); V(f); }
    P4( ) { P(c); P(e); …; V(g); }
    P5( ) { P(d); P(f); …; V(h); }
    P6( ) { P(g); P(h); …; }
}coend
```



合作进程的前趋图

mutual exclusion

semaphore mutex=1 ;

cobegin{

P1:
while(1){
Input datd 1 from I/0 1 ;
Compute……;
wait(mutex) ;
Print results1 by printer;
signal(mutex) ;
}

P2:
while(1){
Input datd 2 from I/O 2 ;
Compute……;
wait(mutex) ;
Print results2 by printer;
signal(mutex) ;
}

}coend

What happens when mutex value is initialized to 2 ?

Deadlock and Starvation

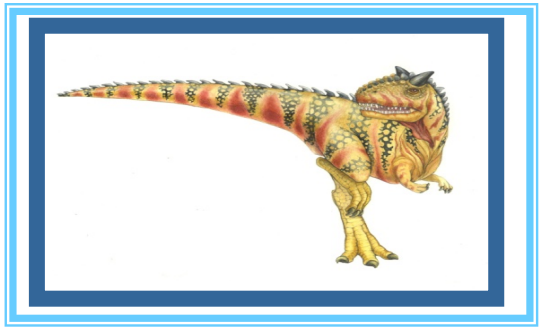
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

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

Deadlock ?

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

6.6 Classical Problems of Synchronization

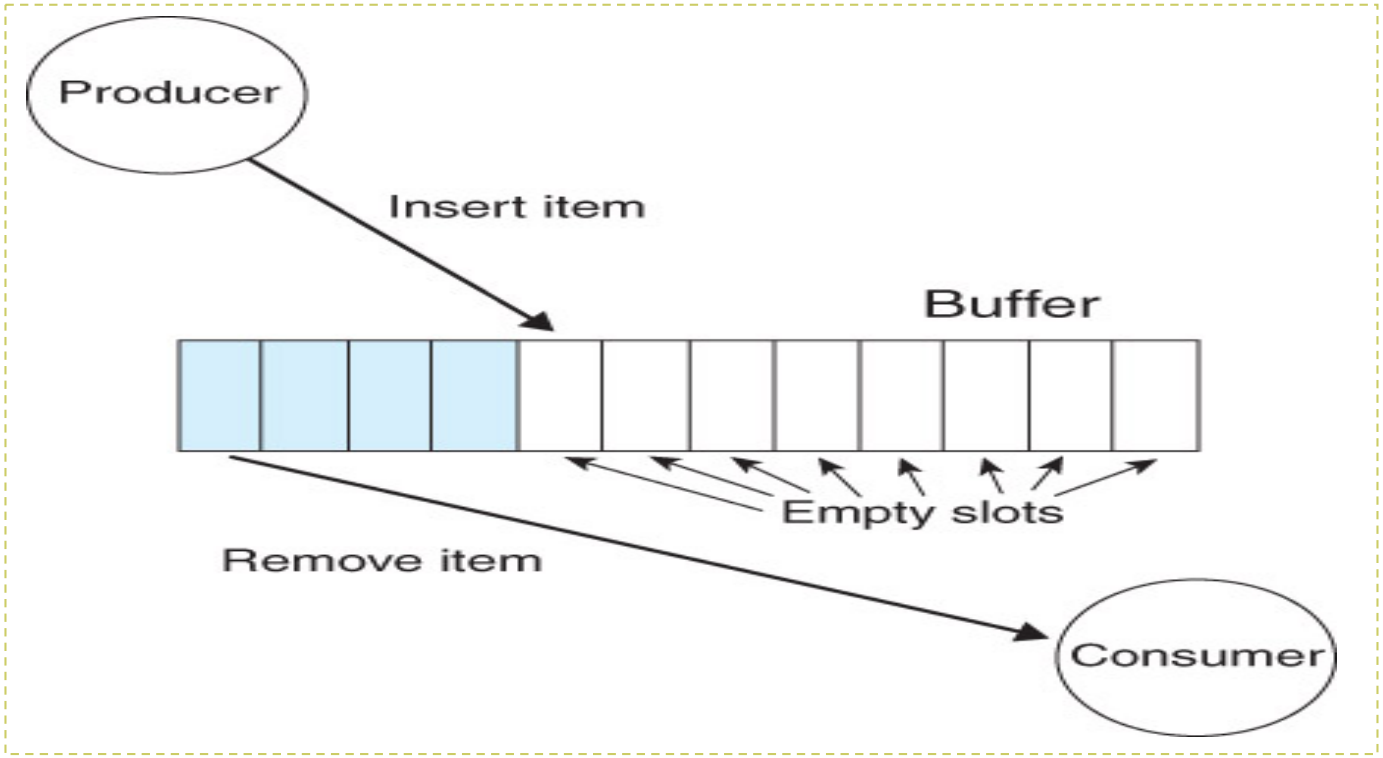


经典同步问题

- 6.6.1 Bounded-Buffer Problem
  有限缓冲区问题（生产者-消费者问题）
- 6.6.2 Readers and Writers Problem
  读者写者问题
- 6.6.3 Dining-Philosophers Problem
  哲学家就餐问题

6.6.1 Bounded-Buffer Problem

- 生产者-消费者问题是最著名的同步问题，它描述一组生产者（P_1 ……P_m）向一组消费者（C_1……C_q）提供消息。它们共享一个有限缓冲池（bounded buffer pool），生产者向其中投放消息，消费者从中取得消息。
- 生产者-消费者问题是许多相互合作进程的一种抽象。



Bounded-Buffer Problem

- *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.

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

The structure of Producer Process

```
do {
        …
    produce an item in nextp
        …

        …
    add nextp to buffer
        …

} while (1);
```

structure of the Consumer Process

```
do {

        …
    remove an item from buffer to nextc
        …

        …
    consume the item in nextc
        …
} while (1);
```

The structure of Producer Process

```
do {
        …
    produce an item in nextp
        …

    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);

} while (1);
```

互斥

structure of the Consumer Process

互斥

```
do {

    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);

        …
    consume the item in nextc
        …
} while (1);
```

The structure of Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (1);
```

structure of the Consumer Process

```
do {
    wait(full);
    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
} while (1);
```

- Example： Simplepc.cpp

6.6.2 Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers （读者）– only read the data set; they do not perform any updates
  - Writers （写者） – can both read and write

- 一个数据集（如文件）如果被几个并行进程所共享：
  - 有些进程只要求读数据集内容，它称读者
  - 一些进程则要求修改数据集内容，它称写者
  - 几个读者可以同时读些数据集，而不需要互斥
  - 一个写者不能和其它进程（不管是写者或读者）同时访问些数据集，它们之间必须互斥。

Problem

- The first readers-writers problem
  - allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
    - writers may starve
- The second readers-writers problem
  - Once a writer is ready the writer performs write as soon as possible
    - readers may starve

The *first* Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

The structure of a Writer Process

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

The structure of a Reader Process

第一个读者到时

wait(wrt);

…

reading is performed

…

最后一个读者离开时

signal(wrt);

The structure of a Reader Process

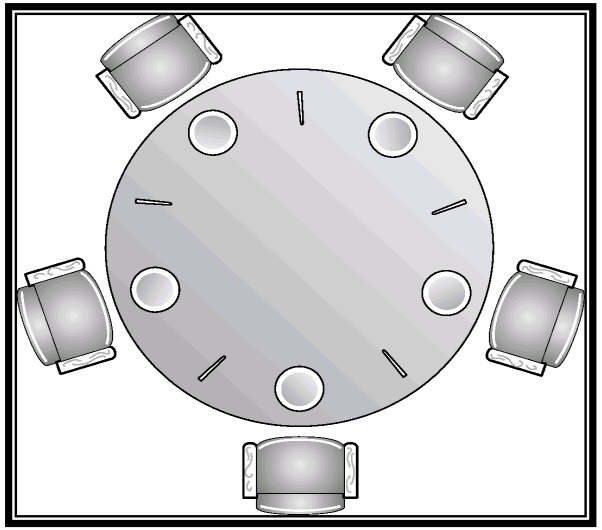
```
readcount++;
if (readcount == 1)
      wait(wrt);
        …
   reading is performed
        …
readcount--;
if (readcount == 0)
    signal(wrt);
```

The structure of a Reader Process

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

6.6.3 Dining-Philosophers Problem

- Problem Statement:
  - N Philosophers sitting at a round table
  - Each philosopher shares a chopstick with neighbor
  - Each philosopher must have both chopsticks to eat
  - Neighbors can't eat simultaneously
  - Philosophers alternate between thinking and eating



Dining Philosophers

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

The structure of Philosophers i

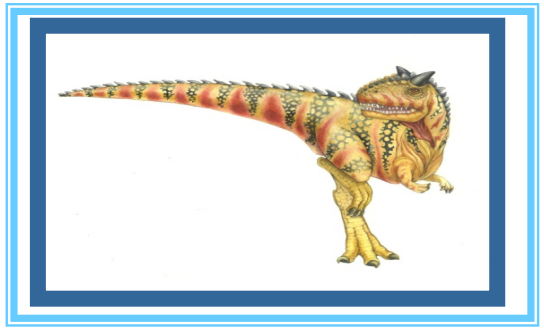
- Philosopher *i*:

```
do {
   wait(chopstick[i])
   wait(chopstick[(i+1) % 5])
      …
   eat
      …
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      …
   think
      …
} while (1);
```

哲学家就餐问题讨论

- Clearly deadlock is rampant (and starvation possible.)
- Several solutions are possible:
  - Allow only 4 philosophers to be hungry at a time.
  - Allow pickup only if both chopsticks are available. (Done in critical section)
  - Odd# philosopher always picks up left chopstick 1st, even# philosopher always picks up right chopstick 1st.
  - 为了避免死锁，把哲学家分为三种状态：思考、饥饿、吃饭，并且一次拿到两只筷子，否则不拿。（Dijkstra）
  - 每个哲学家拿起第1根筷子一定时间后，若拿不到第2根筷子，再放下第1根筷子。

6.7 * Monitors



6.7 * Monitors（管程）

- 管程是一种高级同步机制
- 管程是管理进程间同步的机制，它保证进程互斥地访问共享变量，并方便地阻塞和唤醒进程。
- 管程可以函数库的形式实现。相比之下，管程比信号量好控制。

Monitor Solutions
管程方法解决进程互斥—管程机制

1. 信号量同步的缺点

- 同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如wait、signal操作的次序错误、重复或遗漏）。
- 易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；
- 不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局。
- 正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误。

2. 管程的引入

- 1973年，Hoare和Hanson所提出；其基本思想是把信号量及其操作原语封装在一个对象内部。即：将共享变量以及对共享变量能够进行的所有操作集中在一个模块中。
- 管程的定义：管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块。
- 管程可增强模块的独立性：系统按资源管理的观点分解成若干模块，用数据表示抽象系统资源，同时分析了共享资源和专用资源在管理上的差别，按不同的管理方式定义模块的类型和结构，使同步操作相对集中，从而增加了模块的相对独立性
- 引入管程可提高代码的可读性，便于修改和维护，正确性易于保证：采用集中式同步机制。一个操作系统或并发程序由若干个这样的模块所构成，一个模块通常较短，模块之间关系清晰。

3. 管程的主要特性

- 模块化：一个管程是一个基本程序单位，可以单独编译；
- 抽象数据类型：管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码
- 信息封装：管程是半透明的，管程中的外部过程（函数）实现了某些功能，至于这些功能是怎样实现的，在其外部则是不可见的；

4. 管程的实现要素

- 管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的外部过程（函数）来间接地访问管程中的共享变量；
- 为了保证管程共享变量的数据完整性，规定管程互斥进入；
- 管程通常是用来管理资源的，因而在管程中应当设有进程等待队列以及相应的等待及唤醒操作；

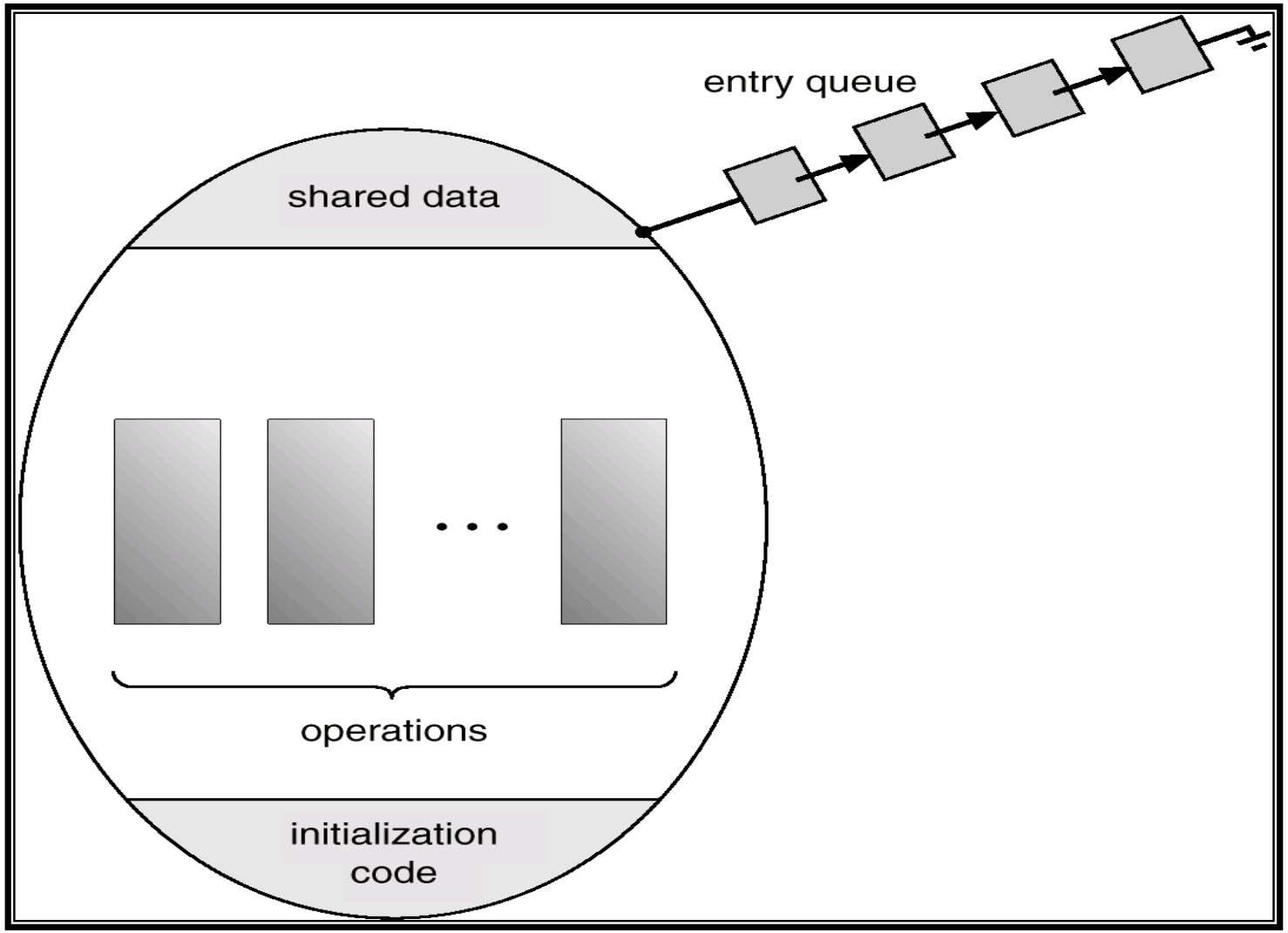
5. 管程的的组成

- 名称：为每个共享资源设立一个管程
- 数据结构说明：一组局部于管程的控制变量
- 操作原语：对控制变量和临界资源进行操作的一组原语过程（程序代码），是访问该管程的唯一途径。这些原语本身是互斥的，任一时刻只允许一个进程去调用，其余需要访问的进程就等待。
- 初始化代码：对控制变量进行初始化的代码

The syntax of a monitor管程的语法

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (…) {  . . .
                          }
    procedure body P2 (…) { . . .
                          }
    procedure body Pn (…) {
        . . .                 }                                  {
       initialization code
    }
}
```

Fig 6.20 Schematic View of a Monitor



6. 管程中的多个进程进入

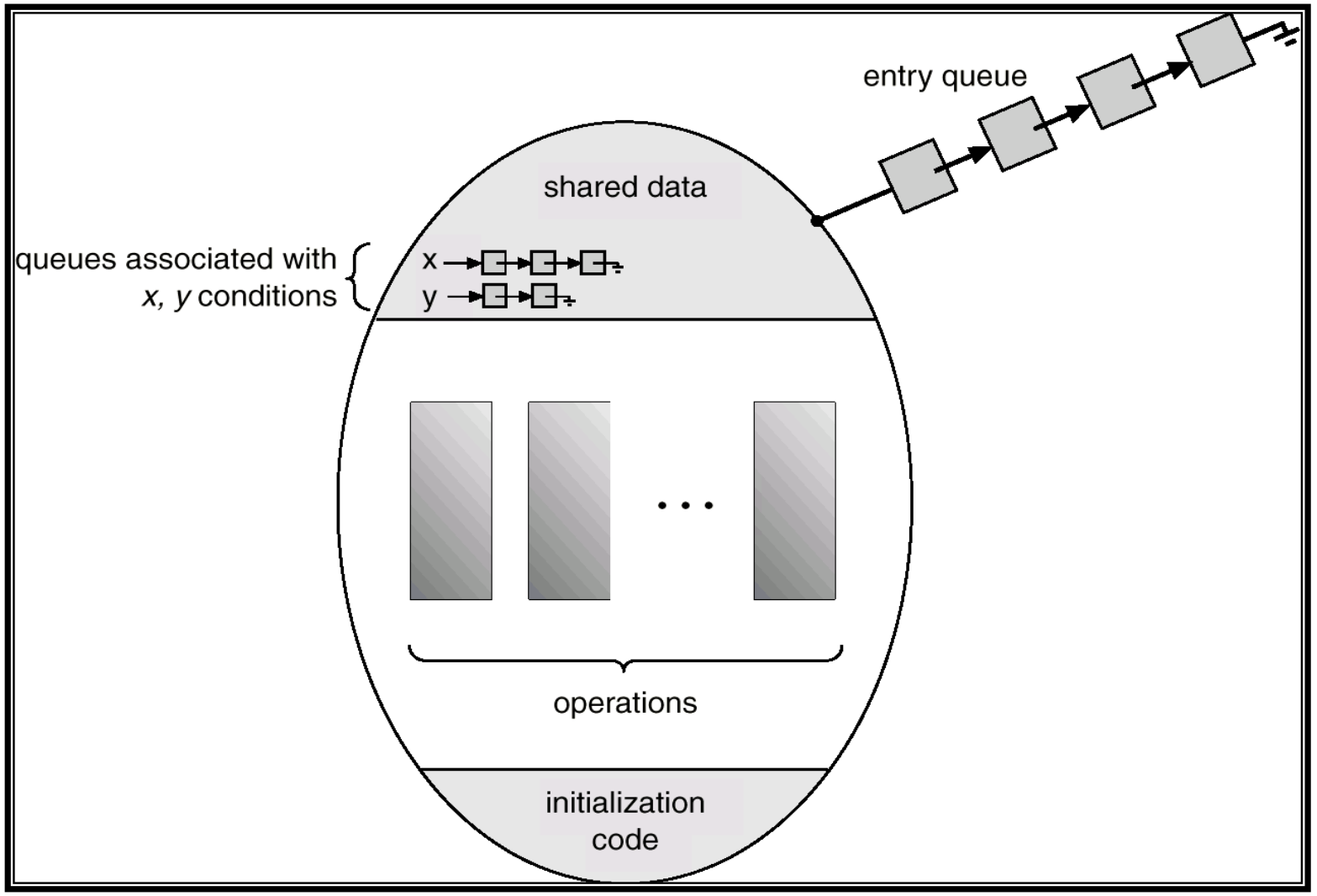
- 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权；当一个进入管程的进程执行唤醒操作时（如P唤醒Q），管程中便存在两个同时处于活动状态的进程，如何处理？。
- 若进程P唤醒进程Q，则随后可有两种执行方式（进程P、Q都是管程中的进程）
  - P等待，直到Q离开管程或下一次等待。Hoare采用。
  - Q等待，直到P离开管程或下一次等待。1980年，Lampson和Redell采用

- 入口等待队列(entry queue)：因为管程是互斥进入的，所以当一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待，因而在管程的入口处应当有一个进程等待队列，称作入口等待队列。
- 紧急等待队列：如果进程P唤醒进程Q，则P等待Q继续，如果进程Q在执行又唤醒进程R，则Q等待R继续，...，如此，在管程内部，由于执行唤醒操作，可能会出现多个等待进程（已被唤醒，但由于管程的互斥进入而等待），因而还需要有一个进程等待队列，这个等待队列被称为紧急等待队列。它的优先级应当高于入口等待队列的优先级。

6. 条件变量(condition)

- 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制。当进入管程的进程因资源被占用等原因不能继续运行时使其等待。为此在管程内部可以说明和使用一种特殊类型的变量----条件变量。
- 每个条件变量表示一种等待原因，并不取具体数值——相当于每个原因对应一个队列。

Fig 6.21 Monitor With Condition Variables



- 同步操作原语wait和signal：针对条件变量x，x.wait()将自己阻塞在x队列中，x.signal()将x队列中的一个进程唤醒。
  - x.wait()：如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程排入x队列尾部（紧急等待队列与x队列的关系：紧急等待队列是由于管程的互斥进入而等待的队列，而x队列是因资源被占用而等待的队列）
  - x.signal()：如果x队列为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者，执行x.signal()操作的进程排入紧急等待队列的尾部

Fig 6.22 A monitor solution to the Dining- Philosophers

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)                    // following slides
  void putdown(int i)        // following slides
  void test(int i)           // following slides
  void init() {
       for (int i = 0; i < 5; i++)
            state[i] = thinking;
  }
}
```

Fig 6.22 A monitor solution to the Dining- Philosophers (Cont.)

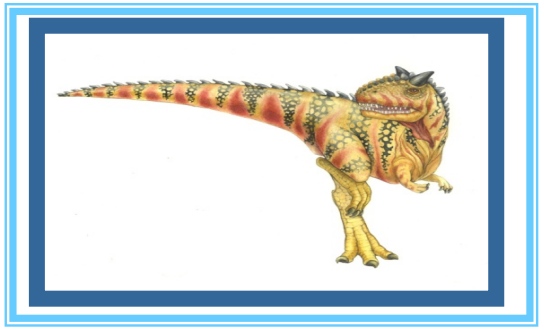
```
void pickup(int i) {
        state[i] = hungry;
        test[i];
        if (state[i] != eating)
                self[i].wait();
}

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

Fig 6.22 A monitor solution to the Dining- Philosophers (Cont.-1)

```
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]:
Do{  dp.pickup(I);
     eat
     dp.putdown(I);
     think
 }while(1)
```

6.8 OS Synchronization



OS Synchronization

6.8.1 Solaris 2.3 *

6.8.2 Windows XP

6.8.1 Solaris Synchronization

- Implements a variety of locksto support multitasking, multithreading (including real-time threads), and multiprocessing
  - Uses adaptive mutexes （适应性互斥）for efficiency when protecting data from short code segments
  - Uses condition variables（条件变量） and readers-writers locks（读-写锁） when longer sections of code need access to data
  - Uses turnstiles （十字转门）to order the list of threads waiting to acquire either an adaptive mutexor reader-writer lock
  - semaphore

6.8.2 Windows XP

- Windows进程线程同步与互斥

6.10 Summary

- shared data make mutual exclusion necessary
  - critical section
- various software solutions –Peterson's algorithm
- hardware solutions
  - atomic operations, like test-and-set
- semaphores
  - mutual exclusion
  - synchronization
- Classical problems of synchronization
  - bounded-buffer, readers-writers, dining-philosophers

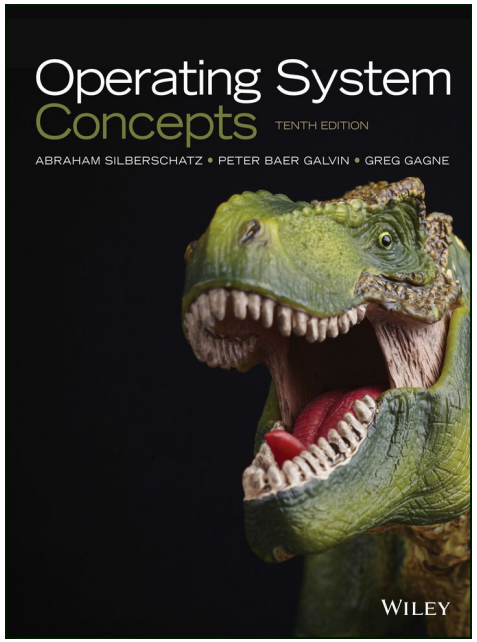
Homework

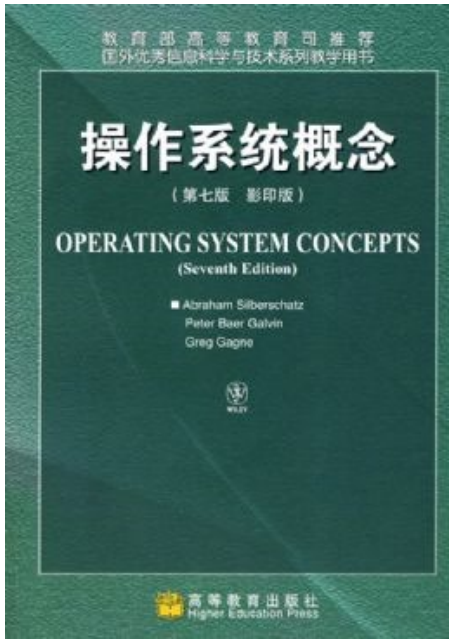
- 学在浙大

- 习题分析

Reading Assignments

- Read for this week:
  - Chapters 6
    of the text book:
- Read for next week:
  - Chapters 7
    of the text book:





End of Chapter