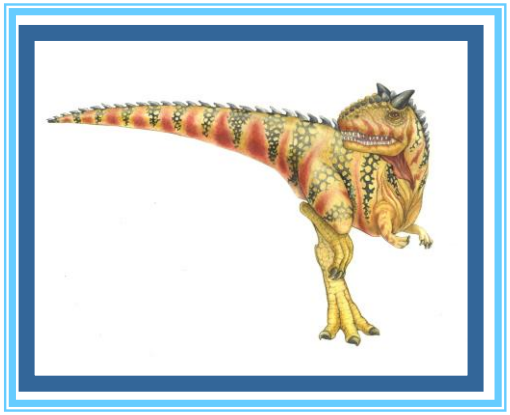
Chapter 6: Process Synchronization

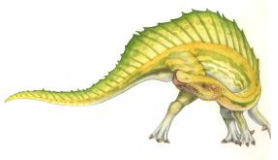


Module 6: Process Synchronization

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

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
- Concurrent access to shared data may result in data inconsistency, Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes



Producer/Consumer

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.

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

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

Race Condition

- count++ *not* atomic operation. Could be implemented as

  register1 = count

  register1 = register1+ 1

  count = register1

- count-- *not* atomic operation. Could be implemented as

  register2 = count

  register2 = register2-1

  count = register2

- 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}

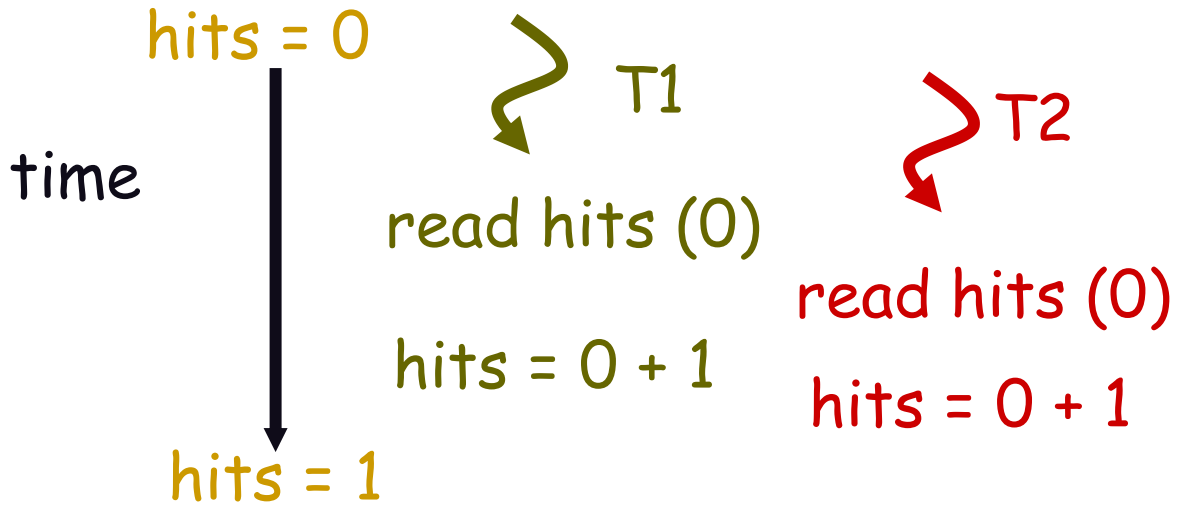
Two threads, one counter, Shared counters

Popular web server

- Uses multiple threads to speed things up. Simple shared state error: each thread increments a shared counter to track number of hits

  hits = hits + 1;

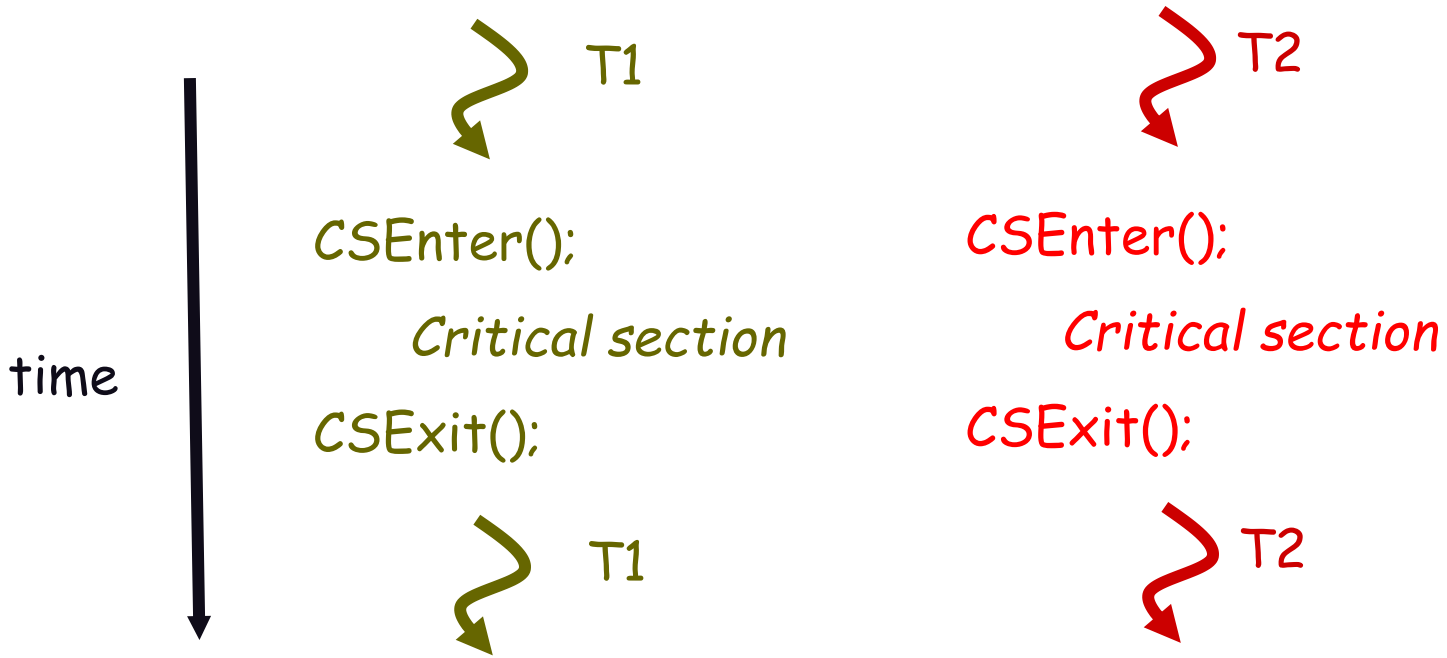
- What happens when two threads execute concurrently?
- Possible result: lost update!



- One other possible result: everything works. ⇒ Difficult to debug
- Called a “race condition”

Critical Section Goals

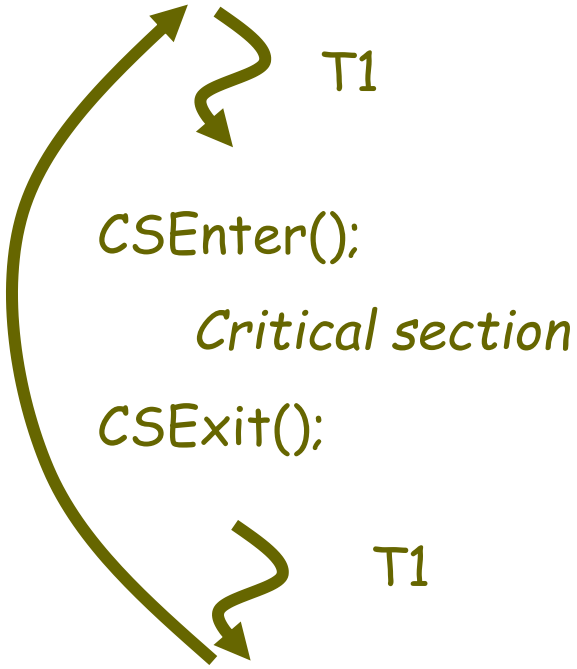
- Threads do some stuff but eventually *might* try to access shared data

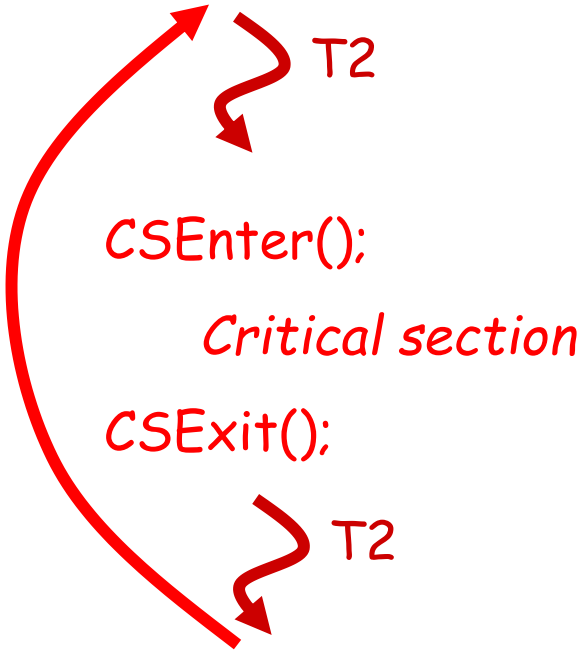


Critical Section: 访问共用资源（例如：共用设备或是存储器单元等）的程序片段，这些共用资源无法同时被多个线程访问的特性.

Critical Section Goals

- Perhaps they loop (perhaps not!)





Critical Section Goals

- We would like
  - Safety (aka mutual exclusion)
    - ✓ No more than one thread can be in a critical section at any time.
  - Liveness (aka progress)
    - ✓ A thread that is seeking to enter the critical section will eventually succeed
  - Bounded waiting
    - ✓ A bound must exist on the number of times that other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted
- Ideally we would like fairness as well
  - If two threads are both trying to enter a critical section, they have equal chances of success
  - … in practice, fairness is rarely guaranteed

Solving the problem

- A first idea:

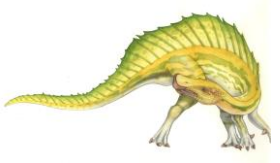
  Have a boolean flag, *inside*. Initially false.

```
CSEnter()
{
    while(inside) continue;
    inside = true;
}
```

```
inside = false;
}
```

Code is unsafe: thread 0 could finish the while test when inside is false, but then 1 might call CSEnter() before 0 can set inside to true!

- Now ask:
  - Is this Safe? Live? Bounded waiting?



Solving the problem: Take 2

- A different idea (assumes just two threads):

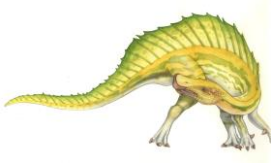
  Have a boolean flag, *inside[i]*. Initially false.

Code isn't live: with bad luck, both threads could be looping, with 0 looking at 1, and 1 looking at 0

```
CSEnter(int i)
{
    inside[i] = true;
    while(inside[i^1]) continue;
}
```

```
{
    Inside[i] = false;
}
```

- Now ask:
  - Is this Safe? Live? Bounded waiting?



Solving the problem: Take 3

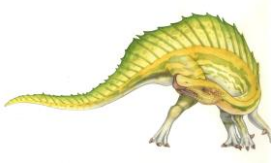
- Another broken solution, for two threads

  Have a turn variable, *turn*, initially 1.

```
CSEnter(int i)
{
    while(turn != i) continue;
}

    turn = i ^ 1;
}
```

Code isn't live: thread 1 can't enter unless thread 0 did first, and vice-versa. But perhaps one thread needs to enter many times and the other fewer times, or not at all

- Now ask:
  - Is this Safe? Live? Bounded waiting?

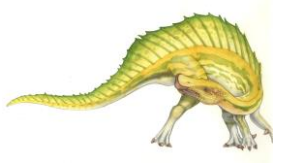


Peterson's Solution

- Two process solution: The two processes share two variables:
  - Int turn;
  - Boolean flag[2]
- 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!

```
CSEnter(int i)
{       flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
}
```

```
CSExit(int i)
{
    flag[i] = FALSE;
}
```



Synchronization Hardware

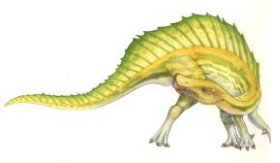
- Many systems provide hardware support for critical section code
- Uniprocessors–could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable

① Either test memory word and set value

② Or swap contents of two memory words

Solution to Critical-section Problem Using Locksdo

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



TestAndndSet Instruction

Definition:

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

Solution using TestAndSet

```
while TS(&lock);
  critical section
lock = FALSE;
  remainder section
```

Shared boolean variable lock., initialized to false.

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

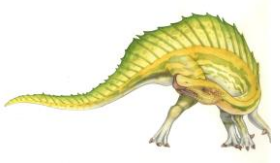
Swap Instruction

Definition:

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

例如X86的交换指令: XCHG OPR1，OPR2

功能：交换操作数OPR1和OPR2的值，操作数类型为字节、字或双字；允许通用寄存器之间，通用寄存器和存储器之间交换数据。

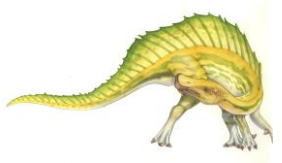


X86 cmpxchg指令

CMPXCHG r/m,r; 比较并交换操作数

将累加器AL/AX/EAX/RAX中的值与首操作数（目的操作数）比较，如果相等，第2操作数（源操作数）的值装载到首操作数，zf置1。如果不等,首操作数的值装载到AL/AX/EAX/RAX并将zf清0. 操作伪代码：

```
IF accumulator == DEST
THEN    ZF <- 1;
        DEST <- SRC;
ELSE    ZF <- 0;
        accumulator <- DEST;
FI;
```

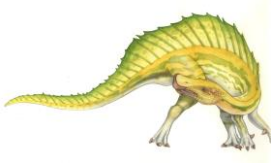


Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true) {
        key = TRUE;
        while ( key == TRUE)   Swap (&lock, &key );
          //    critical section;
        lock = FALSE;
          //    remainder section;
}
```

```
key = TRUE;
do
{
   SWAP(&lock, &key);
} while (key);
   critical section
lock = FALSE;
   remainder section
```



Semaphore

- W.Dijkstra1965年提出信号量

  Semaphore *S*–integer variable,Two standard operations modify S: wait() and signal(),Originally called P() and V()

Can only be accessed via two indivisible (atomic) operations

- wait(S){

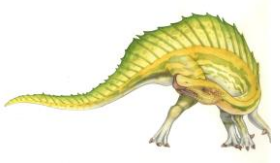
  while S <= 0; // no-op

  S--;

  }
- signal(S){

  S++;

  }



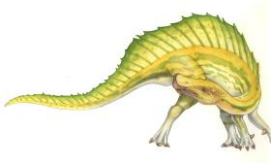
Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – 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

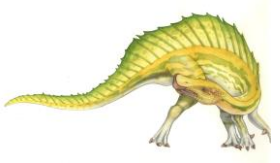
  Provides mutual exclusion

```
Semaphore S;    //  initialized to 1
wait (S);
        Critical Section
signal (S);
```



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
- 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.



Semaphore Implementation with no Busy waiting (Cont.)

Implementation of wait:

```
wait (S){
        S--;
        if (S < 0) {
            block(); / add this process to waiting queue}
}
```

Implementation of signal:

```
Signal (S){
        S++;
        if (S <= 0) {
            wakeup(P);/ remove a process P from the waiting queue }
}
```

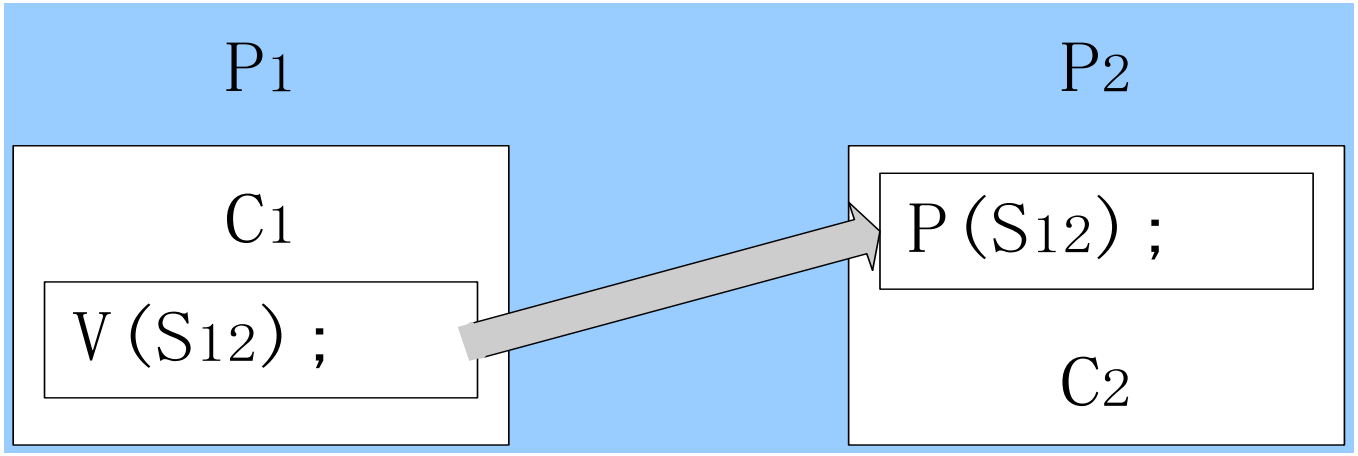
Mutual exclusion using semaphore

Semaphore mutex ; // initialized to 1

```
P(mutex);
    critical section
V(mutex);
    remainder section
```

Synchronization using semaphore

Semaphore S12 ; // initialized to 0



Example 1

Struct

smaphore a,b,c,d,e,f,g,h,I,j=0,0,0,0,0,0,0,0,0,0

cobegin

{S1;V(a);V(b);V(c);}

{P(a);S2;V(d);}

{P(b);S3;V(e);V(f);}

{P(c);S4;V(g);}

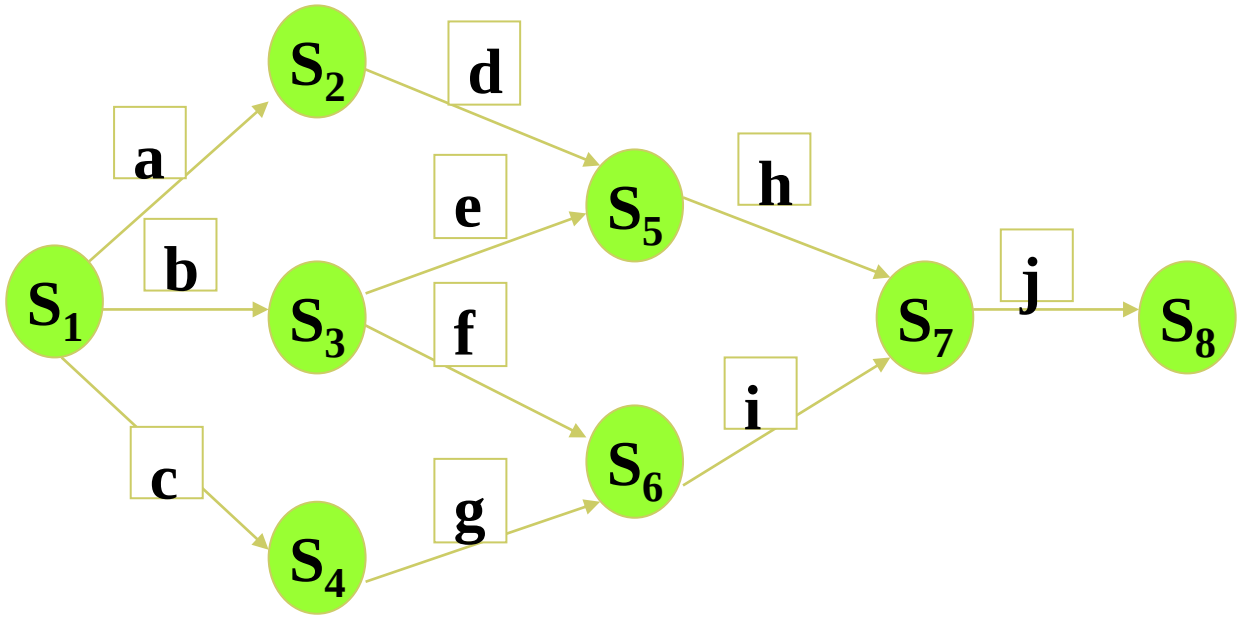
{P(d);P(e);S5;V(h);}

{P(f);P(g);S6;V(i)}

{P(h);P(i);S7;V(j);}

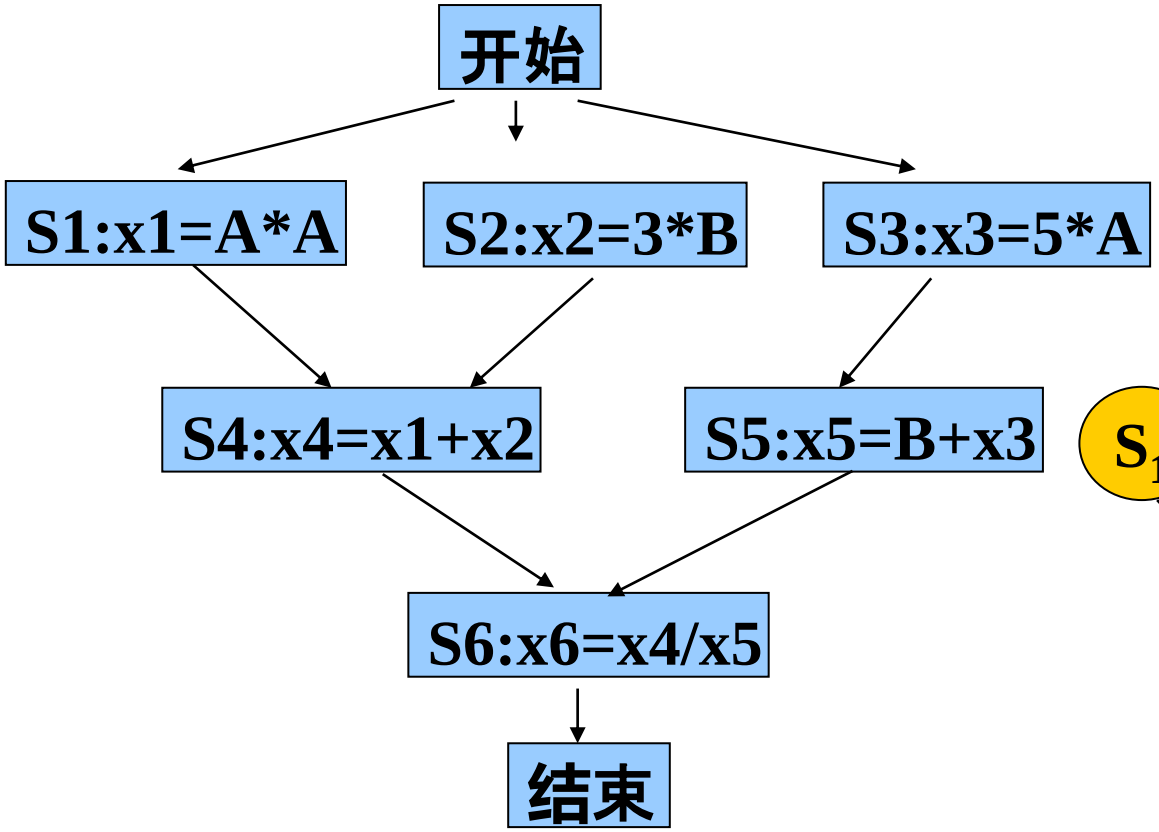
{P(j);S8;}

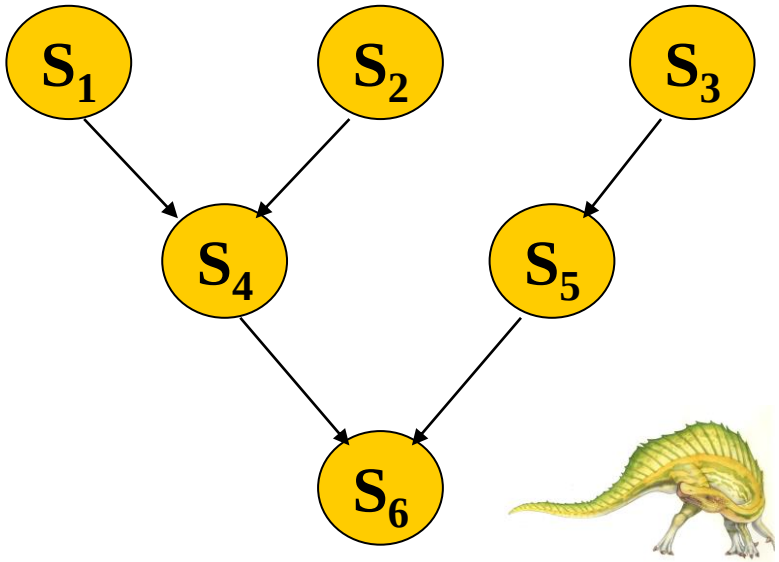
coend



Example 2

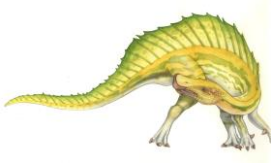
（A^2+3B）/(B+5A), 试画出该公式求值过程的前趋图





Classical Problems of Synchronization

- Bounded-Buffer Problem(Producer-Consumer Problem)
- Readers and Writers Problem
- Dining-Philosophers Problem



Producer-Consumer Problem

Problem:

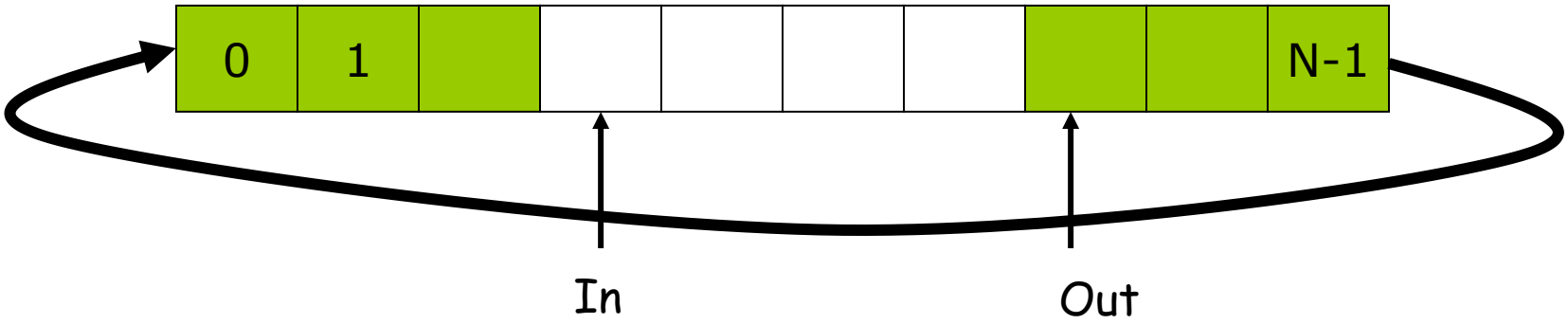
- producer puts things into a shared buffer
- Consumer takes them out
- Need synchronization for coordinating producer and consumer

Coke machine example:

- delivery person (producer) fills machine with cokes
- students (consumer) feed cokes and drink them
- coke machine has finite space (buffer)

Producer-Consumer Problem

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



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.

Producer	Consumer
P(empty); P(mutex); //进入区	P(full); P(mutex); //进入区
one unit --> buffer;	one unit <-- buffer;
V(mutex); V(full); //退出区	V(mutex); V(empty); //退出区

The structure of a Producer/Consumer process

The structure of the producer and Consumer process(con.)

```
Init: Semaphore mutex = 1;    /* for mutual exclusion*/
      Semaphore empty = N;  /* number empty buf entries */
      Semaphore full = 0;       /* number full buf entries */
      any_t buf[N];
      int tail = 0, head = 0;
```

Producer

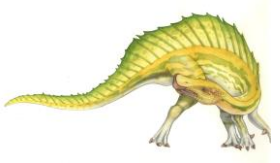
```
void put(char ch)  {

    wait(empty);
    wait (mutex);

    // add ch to buffer
    buf[head%N] = ch;
    head++;

    signal(mutex);
    signal (full );
}
```

Consumer

```
char get() {

    wait (full );
    wait (mutex);

    // remove ch from buffer
    ch = buf[tail%N];
    tail++;

    signal (mutex);
    signal (empty);
    return ch;
}
```

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 count, in, out;
    private Object[] buffer;
}
```



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);
    }
```

enter()/ remove() Method

```
public void enter(Object item) {
    empty.P();
    mutex.P();
    // add an item to the buffer
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    mutex.V();
    full.V();
}
```

```
public Object remove() {
    full.P();
    mutex.P();
    // remove an item from the buffer
    - - count;
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    mutex.V();
    empty.V();
    return item;
}
```

Discussion about Bounded Buffer Solution

Why asymmetry?

- Producer does: wait (empty), signal(full)
- Consumer does: wait (full), signal(empty)

Is order of wait 's important?

- Yes! Can cause deadlock

Is order of signal's important?

- No, except that it might affect scheduling efficiency

What if we have 2 producers or 2 consumers?

- Do we need to change anything?

What's wrong?

```
Init: Semaphore mutex = 1;    /* for mutual exclusion*/
      Semaphore empty = N;    /* number empty buf entries */
      Semaphore full = 0;     /* number full buf entries */
      any_t buf[N];
      int tail = 0, head = 0;
```

Oops! Even if you do the correct operations, the *order* in which you do semaphore operations can have an incredible impact on correctness

Producer

```
void put(char ch) {

  wait(mutex);
  wait (empty);

    // add ch to buffer
    buf[head%N] = ch;
    head++;

   signal(mutex);
   signal (full );
}
```

What if buffer is full?

Consumer

```
char get() {

  wait (full );
  wait (mutex);

    // remove ch from buffer
    ch = buf[tail%N];
    tail++;

   signal (mutex);
   signal (empty);
   return ch;
}
```

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.

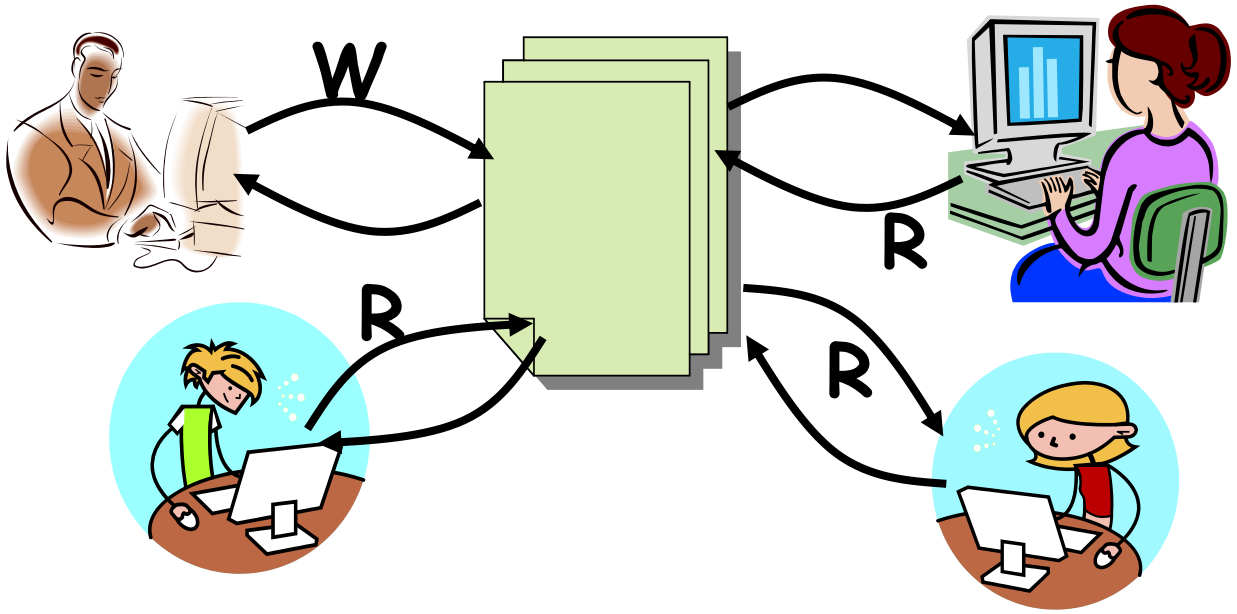
AND型信号量集

```
Swait(S1, S2, …, Sn)  //P原语;
{
 while (TRUE)
   {
      if (S1 >=1 && S2 >= 1 && … && Sn >= 1) //满足资源要求时的处理;
      {
        for (i = 1; i <= n; ++i)  --Si;
           //全部资源要求可满足时，才进行减1操作;
        break;
       }
       else
      {    //某资源不够时进程进入第一个小于1信号量的等待队列Sj.queue;
        阻塞调用进程;
       }
   }
 }
```

AND型信号量集

```
Ssignal(S1, S2, …, Sn)
{
  for (i = 1; i <= n; ++i)
  {
    ++Si;       //释放占用的资源;
    for (each process P waiting in Si.queue)
          //检查每种资源的等待队列;
    {
      从等待队列Si.queue中取出进程P;
      if (判断进程P是否通过Swait中的测试)
           //注：与signal不同，这里要进行重新判断;
          {    //通过检查（资源够用）时的处理;
          进程P进入就绪队列;
          }
      else
          {    //未通过检查（资源不够用）时的处理;
          进程P进入某等待队列;
          }
    }
  }
}
```

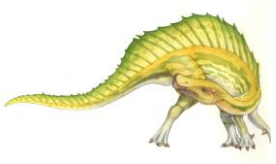
Readers-Writers Problem



- Motivation: Consider a shared database

  Two classes of users:

  - Readers – never modify database
  - Writers – read and modify database



Readers-Writers Problem

- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Is using a single lock on the whole database sufficient?
  - Like to have many readers at the same time
  - Only one writer at a time
- Shared Data
  - Data set
  - Semaphore Rmutex initialized to 1.
  - Semaphore Wmutex initialized to 1.
  - Integer Rcount initialized to 0.

Readers-Writers Problem

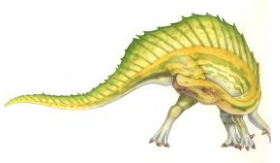
The structure of the Writer/Reader process

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);
```



The structure of the Writer/Reader process(con.)

Shared variables:

```
        Semaphore Rmutex, Wmutex;
        integer Rcount;
Init: Rmutex = 1, Wmutex = 1, Rcount = 0;
```

Writer

```
do {
      wait(Wmutex);
      . . .
      /*writing is performed*/
      . . .
      signal(Wmutex);
}while(TRUE);
```

Reader

```
do { wait (Rmutex);
      readcount++;
      if (readcount == 1)  wait (wrl);
      signal(Rmutex);
      . . .
      /*reading is performed*/
      . . .
      wait (Rmutex);
      readcount--;
      if (readcount == 0) signal(wrl);
      signal(Rmutex);
}while(TRUE);
```

Readers-Writers Problem: Reader/Writer

```
public class Reader extends Thread {
   public Reader(Database db) {
            server = db;
   }
   public void run() {
     int c;
     while (true) {
            c = server.startRead();
            // now reading the database
            c = server.endRead();
     }
   }
   private Database        server;
}
```

```
public class Writer extends Thread {
   public Writer(Database db) {
            server = db;
}
   public void run() {
     while (true) {
            server.startWrite();
            // now writing the database
            server.endWrite();
     }
   }
   private Database  server;
}
```

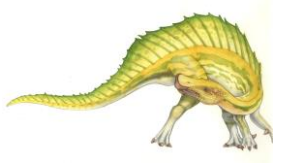
Readers-Writers Problem (cont)

```
public class Database
{
   public Database() {
      readerCount = 0;
      mutex = new Semaphore(1);
      db = new Semaphore(1);
   }
     public int startRead() { /* see next slides */ }
     public int endRead()  { /* see next slides */ }
     public void startWrite() { /* see next slides */ }
     public void endWrite()  { /* see next slides */ }

     private int readerCount; // number of active readers
     Semaphore mutex;   // controls access to readerCount
     Semaphore db;      // controls access to the database
}
```

startRead()/endRead() Method

```
public int startRead() {
    mutex.P();
    ++readerCount;

 // if I am the first reader tell all others
 // that the database is being read
    if (readerCount == 1)
       db.P();
    mutex.V();
    return readerCount;
 }
```

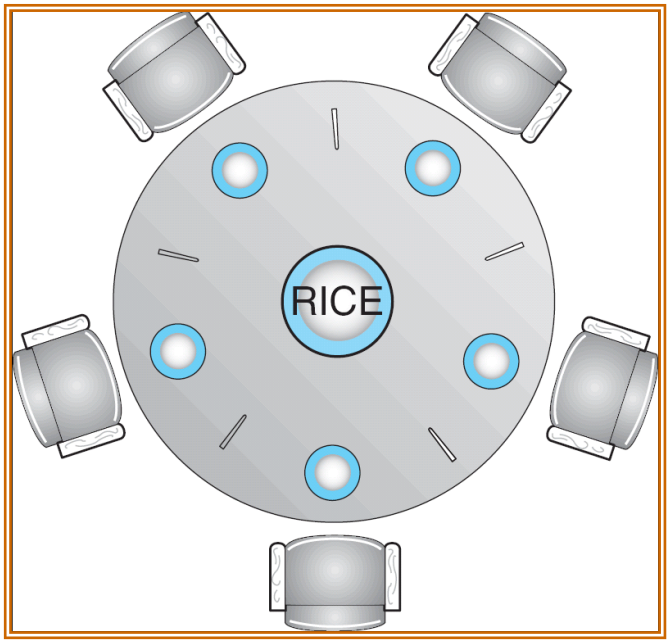
```
public int endRead() {
    mutex.P();
    --readerCount;

// if I am the last reader tell all others
    // that the database is no longer
    being read
     if (readerCount == 0)
        db.V();
     mutex.V();
     return readerCount;
  }
```



Writer Methods

```
public void startWrite() {
    db.P();
}
public void endWrite() {
    db.V();
}
```

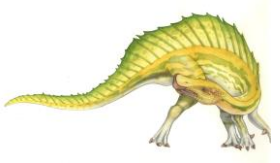
Dining-Philosophers Problem



Shared data

Bowl of rice (data set)

Semaphore chopstick [5] initialized to 1

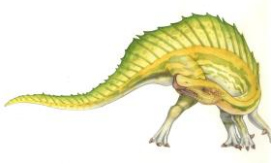


Dining-Philosophers Problem (Cont.)

The structure of Philosopher *i*:

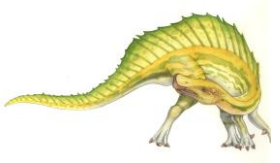
```
While (true)  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
            //  eat
        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
            //  think
}
```

Deadlock?



解法2：AND型信号量

```
Var chopstick array[0,…,4] of semaphore :=(1,1,1,1,1);
Process i
    Reeat
        think;
        Swait (chopstick[(i +1) mod 5],chopstick[i]);
        Eat;
        Ssignal (chopstick[(i +1) mod 5],chopstick[i]);
 Until false;
```



信号量小结

1、信号量的含义

S>0：信号量所表示资源的可用个数；

S=0：信号量所表示资源的可用个数为0；

S<0：在S信号量上因等待该类资源而阻塞的进程个数；

2、P()/V()操作的使用

使用S所表示的资源时用P(S)；

释放S所表示的资源时用V(S)；

3、互斥问题：设置一个公共信号量S且初值为1，表示只有一个互斥资源可用；

同步问题：根据每个进程使用的资源类型情况分别设置各自的私有信号量和初值。

Exercise

- 吃水果问题：桌上有一只盘子，每次只能放一个水果，爸爸向盘中放水果(苹果或桔子)，儿子专等吃盘里的桔子，女儿专等吃盘里的苹果。只要盘子空，爸爸可向盘中放水果，仅当盘中有自己需要的水果时，儿子或女儿可从中取出，请给出三人之间的同步关系，并用P、V操作实现三人正确活动的程序。

吃水果问题

- 分析:该问题是生产者-消费者问题的一个变形
  - 生产者:爸爸
  - 消费者：女儿、儿子
  - 缓冲区：盘子（SIZE=1）
- 设置3个信号量：mutex=1,实现盘子的互斥使用

  S1=0,表示盘中是否有苹果，实现爸爸与女儿的同步

  S2=0,表示盘中是否有桔子，实现爸爸与儿子的同步

爸爸：

P（mutex）;

将水果放入盘中；

IF 桔子 V（S2）

else V（S1）

儿子：P（S2）；

从盘中取桔子；

V（ mutex ）；

吃桔子

女儿：P（S1）；

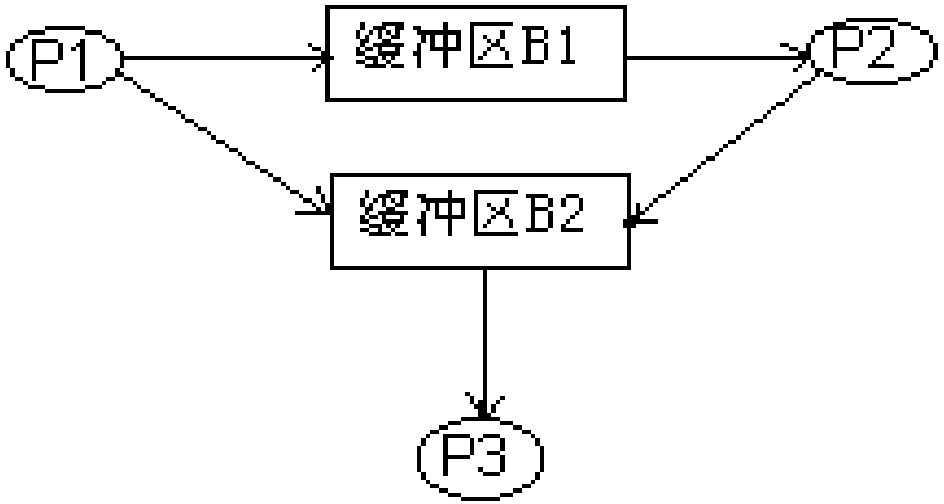
从盘中取苹果；

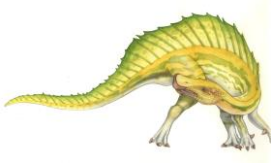
V（ mutex ）；

吃苹果

缓冲区问题

假设有一下图所示的工作模型：有三个并发进程P1、P2和P3，两个单缓冲B1和B2。进程P1负责不断从输入设备读数据，若读入的数据为正数，则直接送入B2，否则应先将数据送入B1，经P2取出加工后再送入B2，P3从B2中取信息输出。请用信号量和P、V操作描述进程P1、P2、P3实现同步的算法。





Problems with Semaphores

- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

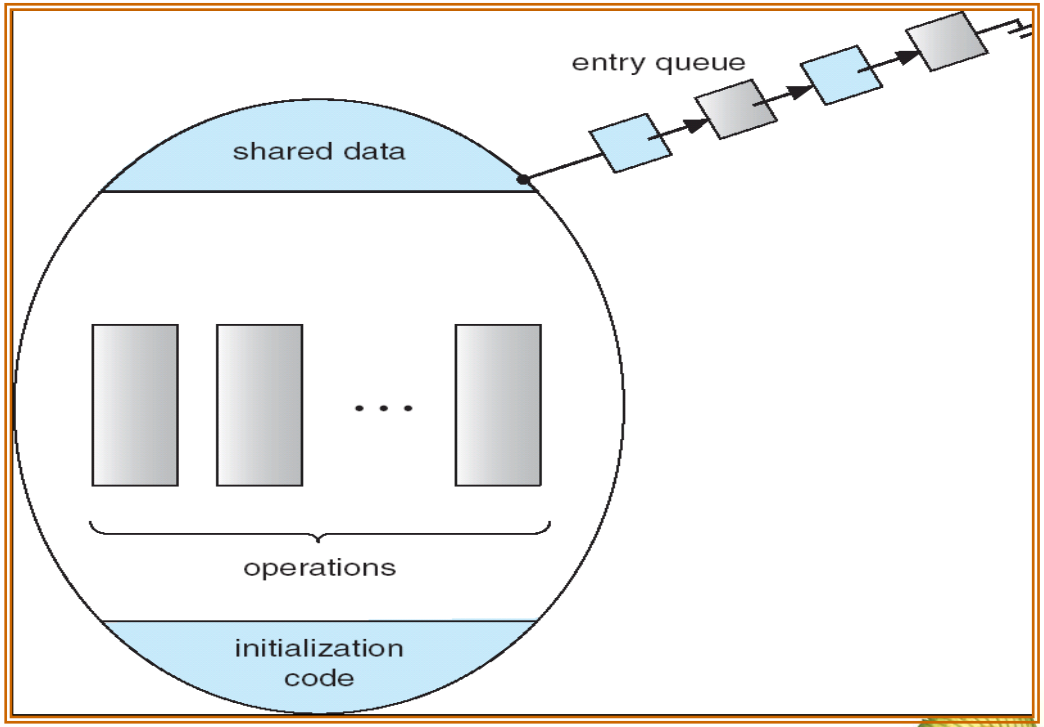
postcript semaphores

- Semaphores are very “low-level” primitives
  - Users could easily make small errors
  - Similar to programming in assembly language
    - Small error brings system to grinding halt
  - Very difficult to debug
- Also, we seem to be using them in two ways
  - For mutual exclusion, the “real” abstraction is a critical section
  - But the bounded buffer example illustrates something different, where threads “communicate” using semaphores
  - 同步问题特点及信号量应用?
- Simplification: Provide concurrency support in compiler
  - Monitors

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time
- monitor monitor-name

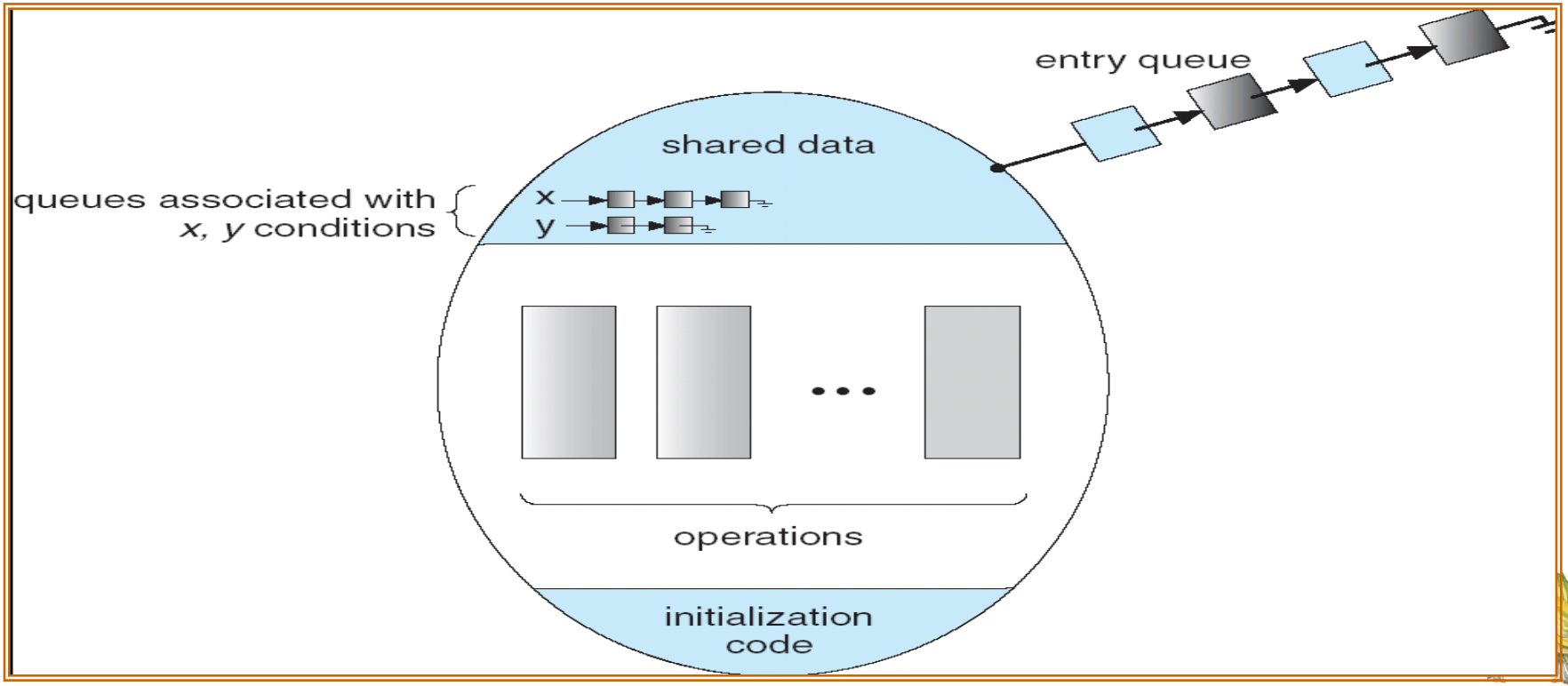
```
{ // shared variable declarations
    procedure P1 (…) { …. }
        …
    procedure Pn (…) {……}
    Initialization code ( ….)
    { … }
}
```



Schematic view of a Monitor

Monitor with 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 ()



Solution to Dining Philosophers

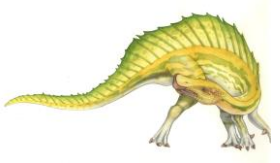
```
monitor DP
 {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];
    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)   self [i].wait;
    }
     void putdown (int i) {
          state[i] = THINKING;
           // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
     }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ; }
    }
initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```

Each philosopher invokes the operations in the following sequence:

```
dp.pickup (i)
    EAT
dp.putdown (i)
```



Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    any_t buf[N];
    int n = 0, tail = 0, head = 0;
    condition full, empty;
    void put(char ch) {
        if(n == N)  empty.wait();
        buf[head%N] = ch;
        head++;
        n++;
        full.signal();
    }
    char get()  {
        if(n == 0)   full.wait();
        ch = buf[tail%N];
        tail++;
        n--;
        empty.signal();
        return ch;
    }
}
```

What if no thread is waiting when signal is called?

Signal is a "no-op" if nobody is waiting. This is very different from what happens when you call signal () on a semaphore – semaphores have a "memory" of how many times signal() was called!

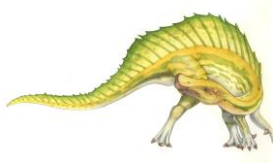
Producer Consumer using Monitors

```
Process_producer :
begin
      repeat
            produce an  item  in nextp ;
            Producer_Consumer.put ( item) ;
      until false   ;
end
Process_consumer :
begin
      repeat
            Producer_Consumer.get (item) ;
            consume the item in nextc ;
      until false ;
end
```

Monitor Implementation?

Synchronization Examples

- Java
- Solaris
- Windows XP
- Linux
- Pthreads



Java Synchronization

- Synchronized, wait(), notify() statements
- Multiple Notifications（多重声明）
- Block Synchronization
- Java Semaphores
- 有界缓冲区方案存在的问题：

1. Busy wait

解决：当缓冲区满了，生产者调用Thread.yield()将CPU让给具有相同优先级的线程——可能导致死锁。

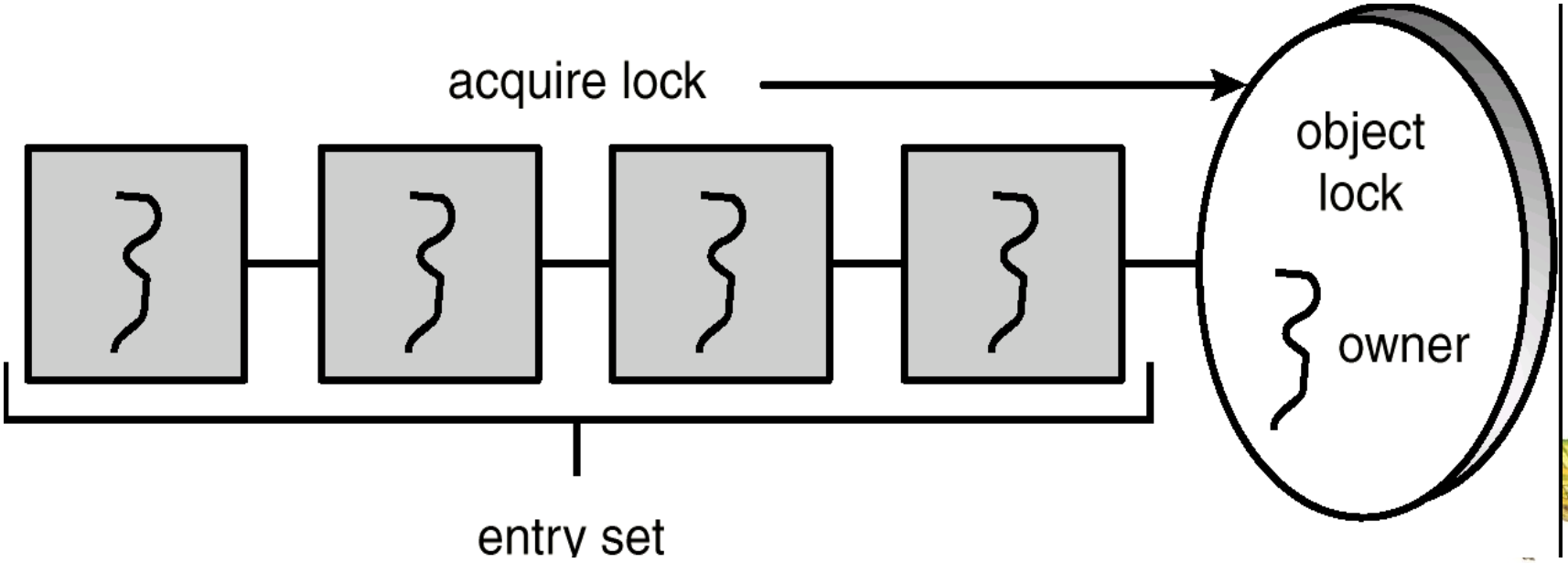
假设有多个生产者且优先级高于消费者，使消费者就得不到CPU，生产者将产品放入缓冲区就可能导致生产者和消费者陷入死锁。

Java Synchronization

2, Race condition

解决：synchronized----when a method is declared as synchronized, calling the method requires owning the lock for the object.

JAVA中每个对象都与一把锁相关联。当方法声明为“synchronized”，那么调用该方法就需要拥有对象的锁。如果锁已被别的线程拥有，那么调用同步方法的线程进入该对象锁的入口等待队列中。当锁被释放后，等待的线程就可以拥有这把锁，调用同步方法。当线程退出这个方法时，释放锁。



synchronized enter()/ remove() Method

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)
        Thread.yield();
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
public synchronized Object remove() {
    Object  item;
    while (count == 0) Thread.yield();
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

- 对象有界缓冲区有一把锁，生产者先获得锁才能调用enter()修改count的值；
- 当生产者拥有这把锁时，调用remove()的消费者阻塞，直到生产者退出enter()时释放锁；
- 保证任何时刻只能一个线程进入enter()和 remove()

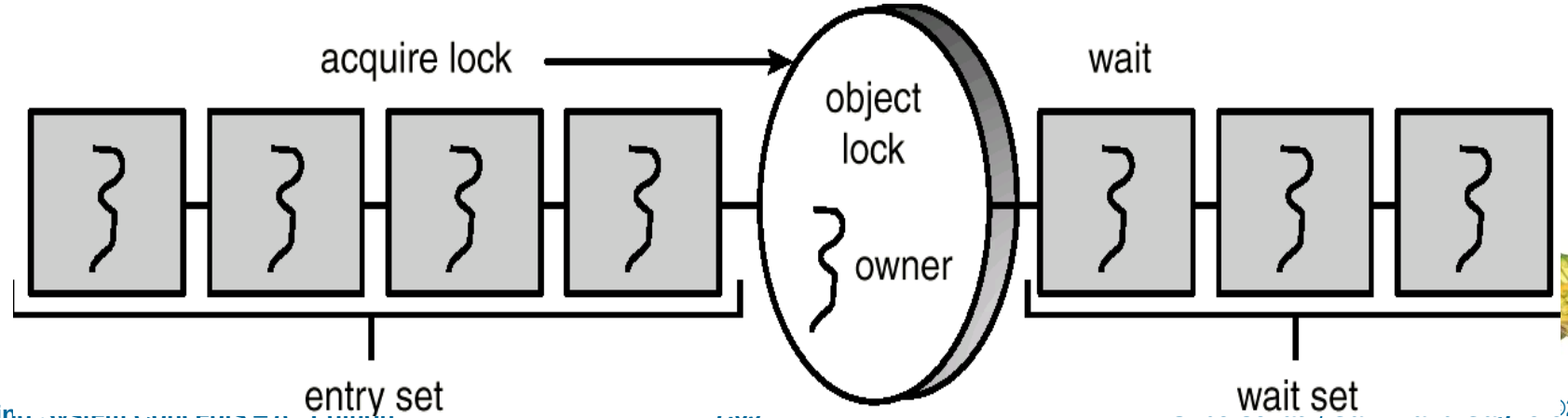
Java Synchronization

- 锁带来问题：如果缓冲区满且消费者处于sleeping状态，生产者可以获得锁进入enter()，但发现缓冲区是满的，于是它调用yield()，让出CPU，但它仍然拥有锁。之后，如果消费者醒来要调用remove()，由于锁被生产者拥有而无法调用，这也形成了死锁。

- 解决：引入wait()和notify()方法。每个对象除了锁再设置一个等待集。
- 例如：当生产者拥有锁进入enter()方法，发现缓冲区已满，则调用wait()方法，释放锁并进入等待集。直到消费者调用remove()方法后，再调用notify()方法将生产者唤醒，使其变为runnable状态，进入入口队列，就可以与其它线程去竞争锁的使用。

The wait()/ notify() Method

- When a thread calls wait(), the following occurs:
  - the thread releases the object lock.
  - thread state is set to blocked.
  - thread is placed in the wait set.
- When a thread calls notify(), the following occurs:
  - selects an arbitrary thread T from the wait set
  - moves T to the entry set.
  - sets T to Runnable.
- T can now compete for the object's lock again.



enter()/remove() with wait/notify Methods

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)
        try { wait();
        } catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    notify();
}
```

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)
      try {wait();
       }catch (InterruptedException e) { }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return item;
}
```

Multiple Notifications

- notify()从等待集中任意选取一个线程。如果等待集中有多个线程，那么就可能出现新的死锁。
- 例如：有5个线程T1--T5，1个共享变量turn，线程都需要调用dowork()，规定只有编号与turn相同的线程才能进入dowork()。假定turn=3， T1，T2，T4在等待集中，T3调用dowork()， 当T3完成工作，把turn设为4，并调用notify()。 notify()从等待集中任意选取一个线程，假设选中T2，当T2运行时，发现turn不等于它，T2再次调用wait()进入等待集。接下来，如果T3，T5调用dowork()，也会进入等待集。这样，5个线程都进入了等待集，形成了死锁。

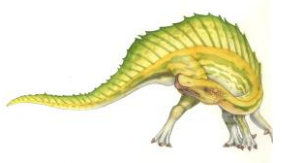
- notifyAll() removes ALL threads from the wait set and places them in the entry set

The Readers-Writers Problem

- 用JAVA同步机制解决第一类读者-写者问题：
  - readerCount：读者数目
  - dbReading：若有读者在读，则设为true
  - dbWriting：若有写者在写，则设为true
  - startRead()、endRead ()、startWrite ()、endWritet ()都声明为同步方法，以保证对共享变量的互斥访问

Reader Methods with Java Synchronization

```
public class Database {
   public Database() {
      readerCount = 0;
      dbReading = false;
      dbWriting = false;
   }
     public synchronized int startRead() { /* see next slides */ }
     public synchronized int endRead()  { /* see next slides */ }
     public synchronized void startWrite() { /* see next slides */ }
     public synchronized void endWrite()  { /* see next slides */ }

     private int readerCount;
     private boolean dbReading;
     private boolean dbWriting;
}
```



Read() Method

```
public synchronized int startRead() {
    while (dbWriting == true) {
        try {      wait();
        }
        catch (InterruptedException e) { }
        ++readerCount;
        if (readerCount == 1)
            dbReading = true;
        return readerCount;
}
```

endRead() Method

```
public synchronized int endRead() {
        --readerCount
        if (readerCount == 0)
            dbReading = false;
        notifyAll();
        return readerCount;
}
```

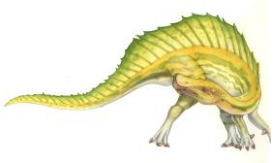
Writer Methods

```
public void synchronized startWrite() {
    while (dbReading == true || dbWriting == true)
        try {
            wait();
        }
        catch (InterruptedException e) { }
        dbWriting = true;
}

public void synchronized endWrite() {
    dbWriting = false;
    notifyAll();
}
```

Block Synchronization

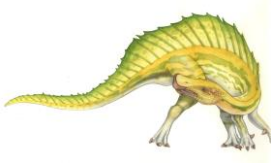
- Blocks of code – rather than entire methods – may be declared as synchronized（同步化程序块——为方法中的一部分，细化加锁的粒度）
- This yields a lock scope that is typically smaller than a synchronized method（锁定范围更小）.

```
Object mutexLock = new Object();
public void someMethod() {
    // non-critical section
    synchronized(mutexLock) {
        // critical  section
    }
        // non-critical section
}
```



Java Synchronization rules

- 一个对象有一把锁，一个线程如果获得了该对象的锁，就可以调用该对象的其他同步方法
- 一个线程可以嵌套调用不同对象的同步方法，故一个线程可以同时拥有不同对象锁
- 没有声明为同步方法的调用不用考虑锁的问题;
- 如果对象的等待集为空，则对notify() 和notigyall() 的调用不起作用;



Solaris Synchronization

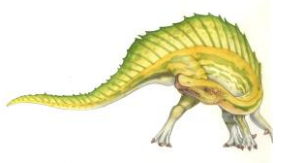
Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Solaris Synchronization

- adaptive mutex：根据占有锁线程的状态选择循环等待还是阻塞。
- 在多CPU系统中，当一个线程要使用共享数据时，如果锁被另一个CPU上的线程拥有，则分两种情况
  - 若拥有锁的线程正在执行，则申请锁的线程循环等待（因为它认为使用共享数据的线程会很快执行结束）；
  - 若拥有锁的线程处于阻塞状态，则申请锁的线程不再等待，而是进入阻塞状态，直到锁被释放才把它唤醒
- 在单CPU系统中，只存在第2种情况。

  适用于短代码保护，因为存在忙等，代码长则CPU利用率低。



Solaris Synchronization

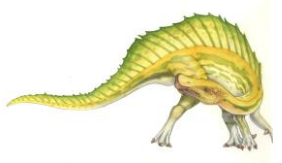
- 在内核级线程和用户级线程都实现了锁机制;
- reader-writer locks用来保护频繁访问的只读数据。可实现并发读，而信号量只能实现顺序访问，故读写锁要比信号量更高效。
- 但由于reader-writer locks的实现成本相对要高，所以通常只用于长代码段的锁定
- 十字转门（turnstile）:to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstile是一个队列结构，包含阻塞在锁上的线程。当释放该锁时，内核会从十字转门中选择一个线程作为锁的下一个拥有者。

Solaris Synchronization

Solaris线程同步API

(1) 互斥:只允许一个已经获取互斥锁的线程进行运行，其他线程被阻塞。创建互斥锁的线程必须释放该锁。Solaris提供的互斥锁的操作原语：

- mutex_enter( )　-获取互斥锁
- mutex_exit( )　-释放互斥锁
- mutex_tryenter( )-如果互斥锁可用，就获取该互斥锁
- 获取锁的线程需调用mutex_enter()，如果失败，是因为另一个线程已经获取了对该对象的互斥访问，则该线程可调用mutex_tryenter()原语进行等待。一旦获取该锁，线程就可以执行必要的操作。最终，当释放该锁的时候，线程执行mutex_exit()。



Solaris Synchronization

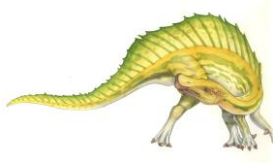
(2) 信号量

- Solaris没有提供与信号量锁机制相关的原语。它提供了递减和递增信号量计数以及执行等待操作的原语如下：
- sema_p（） —信号量递减，申请获取互斥锁
- sema_v（） —信号量递增，释放互斥锁
- sema_tryp（）—执行等待操作

Solaris Synchronization

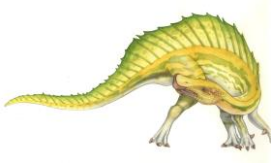
(3) 多读取器，单写入器

- 顾名思义，同经典的读-写问题提供的机制一样，允许多个Solaris线程同时读取一个对象，只允许一个线程可以写入数据，Solaris提供了以下原语：
- rw_enter（）一试图获取如读取器/写入器这样的锁;
- rw_exit（） 一释放前面获取的锁(作为读者/写者);
- rw_tryenter（）一执行等待操作;
- rw_downgrade（）一写入器线程使用的操作，用于从写模式切换到读模式;
- rw_upgrade（）一读取器线程使用的操作，用于从读模式切换到写模式;



Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems;
- Uses spinlocks on multiprocessor systems(忙等);
- Also provides dispatcher objects which may act as either mutexes and semaphores;



Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent

  It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:

  read-write locks

  spin locks

Exercise

- **6.9**
- **6.10**
- **6.11**
- **6.19**
- 进程get把从读卡机读取的数据放进buffer1；进程copy把buffer1的信息处理后放到buffer2；进程put从buffer2中取数并打印输出。请用PV操作协调上述3个进程的同步关系。

End of Chapter 6