个人简介

齐赛宇，男，工学博士，1986.5.1， 西安交通大学计算机学院副教授

教育背景

学历/学位	毕业学校	时间	专业	导师
本科/学士学位	西安交通大学	2004.09~2008.07	计算机科学与技术	
博士研究生/博士学位	香港科技大学	2008.09~2015.03	计算机工程与技术	**刘云浩教授，倪明选教授**
访问学者	新加坡南洋理工大学	2011.10~2013.11	计算机工程与技术	**李默教授**

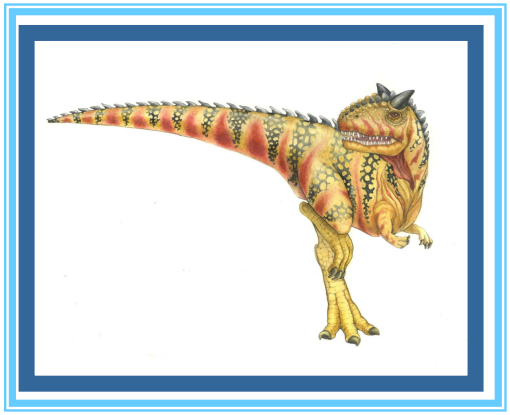
工作经历

副教授	西安交通大学	2021.4~至今	计算机学院
讲师	西安电子科技大学	2019.10.~2021.4.	网络与信息安全学院
博士后（**导师：陈晓峰教授**）	西安电子科技大学	2015.12.~2019.10.	通信工程(密码学)

研究方向

- 分布式系统及安全，隐私计算，应用密码学

Chapter 6: Process Synchronization



Chapter 6: Process Synchronization

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

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

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

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++;
}
```

Consumer

```
while (true) {
        while (count == 0)
            ; // do nothing
        nextConsumed= buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        /* consume the item in nextConsumed


}
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- 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}

Race Condition

- What is a race condition?
  - Several processes access and manipulate the same data concurrently
  - The result of the execution depends on particular access order.
- To avoid race condition, we need to ensure that only one process at a time can be manipulating the same data.----mutual exclusion

Two threads, one counter

Popular web server

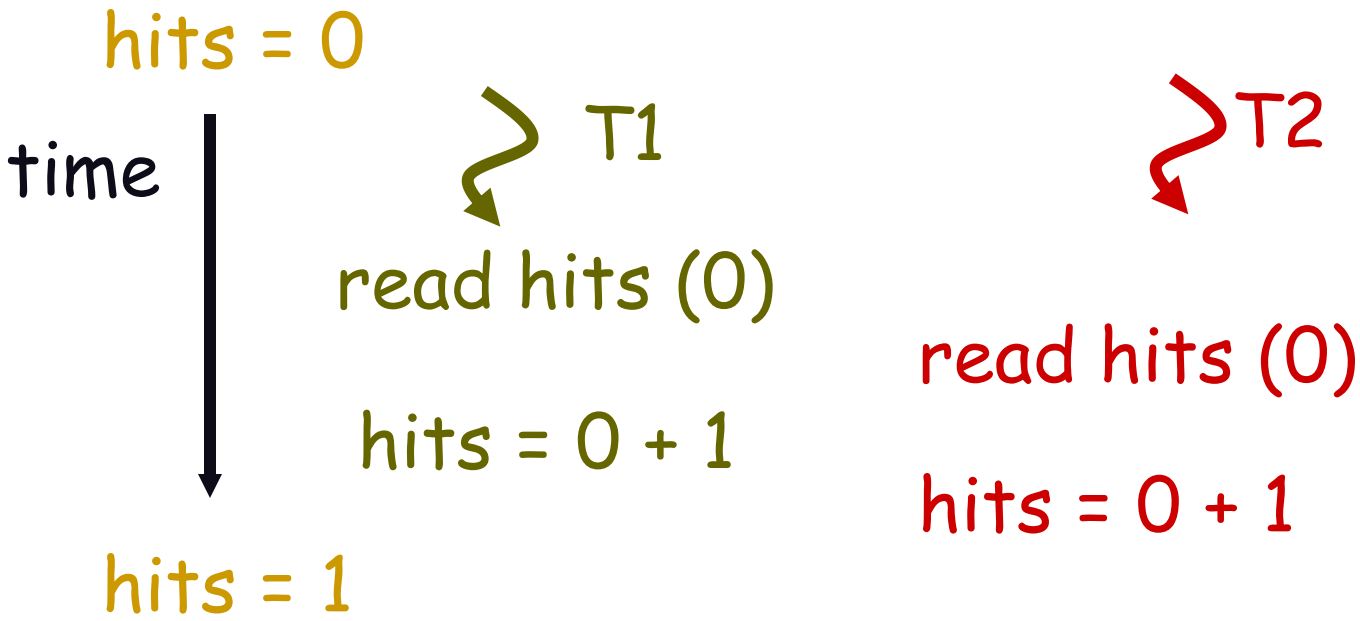
- Uses multiple threads to speed things up.
- Simple shared state error:
  - each thread increments a shared counter to track number of hits

```
...
hits = hits + 1;
...
```

- What happens when two threads execute concurrently?

Shared counters

- Possible result: lost update!



- One other possible result: everything works.

  $\Rightarrow$ Difficult to debug

- Called a “race condition”

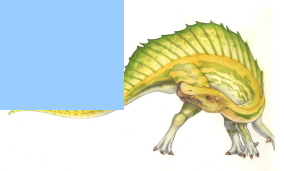
Critical-Section

- Critical resource:
  - The resource that must be accessed mutual excluded.
- Critical section
  - A segment of code in which the process is accessing critical resource
  - Consider a set of concurrent processes{ $P_0, P_1, \ldots, P_{n-1}$}.
  - Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.

Critical-Section

- To ensure the shared data to be modified mutual excluded, no more than one process can execute in its critical section at the same time.
- Each process must request permission to enter its critical section----entry section
- The remaining code is the remainder section.

```
entry section
  critical section
exit section
  remainder section
```



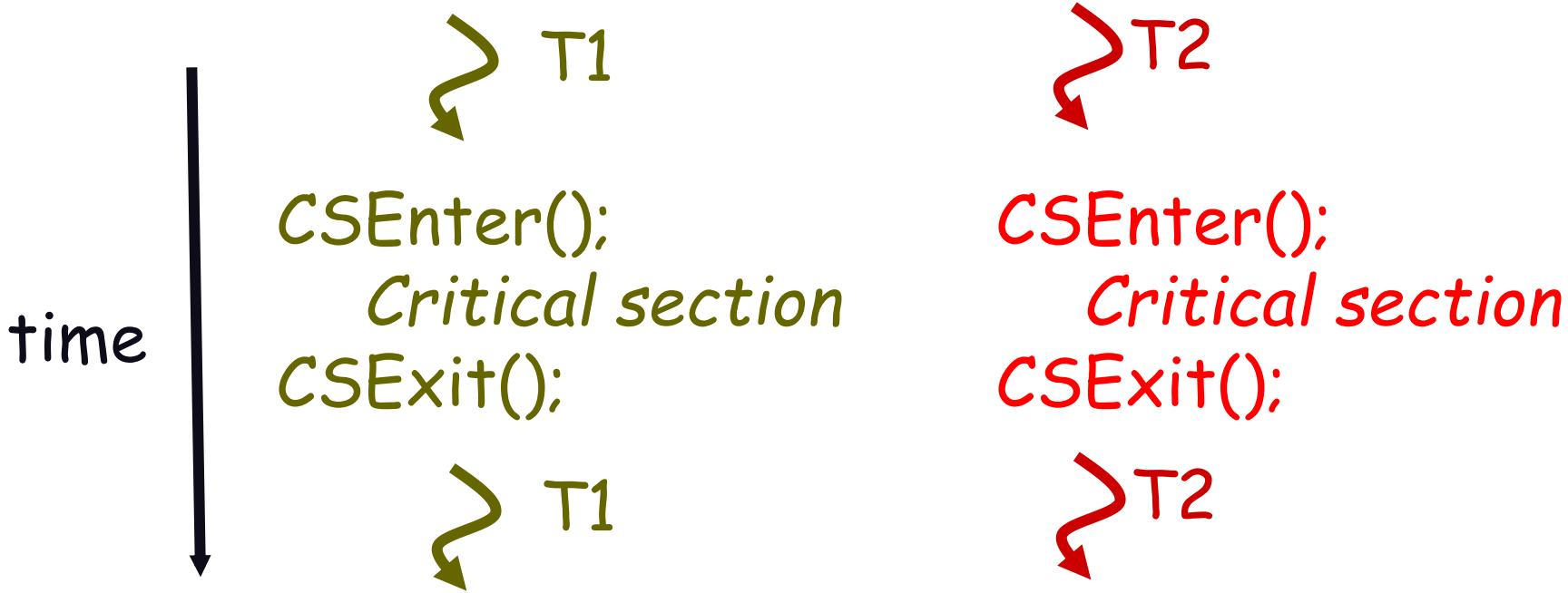
Solution to Critical-Section Problem

1. Mutual Exclusion (safety) - No more than one process can be in a critical section at any time.
2. Progress(liveness) - If no process is executing in its critical section and there exist some processes wishing 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 - 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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

- 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

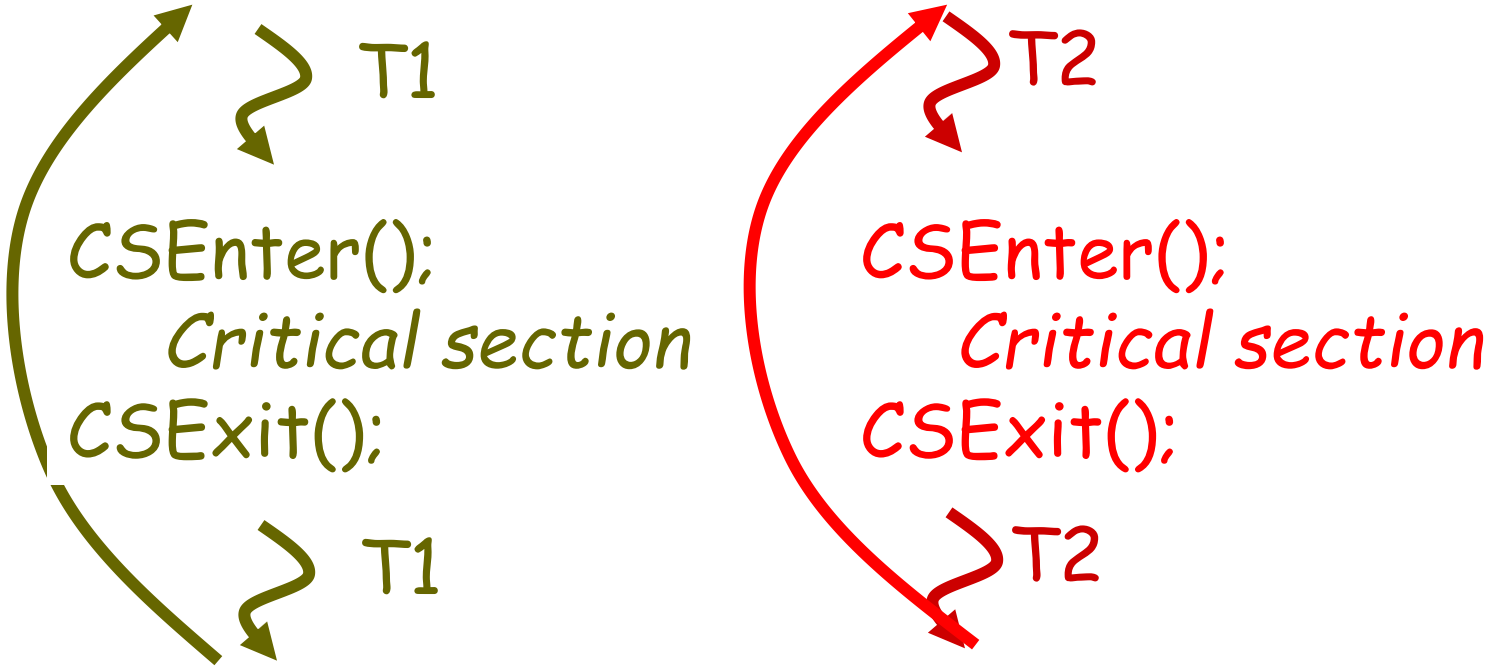
Critical Section Goals

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



Critical Section Goals

- Perhaps they loop (perhaps not!)



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.

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

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

```
    turn = i ^ 1;
}
```

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

Peterson's Solution

- A classic software-based solution to the critical section problem.
- Works for two processes.
- 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!

Peterson's Algorithm

```
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);

  critical section

flag[i] = FALSE;

  remainder section
```

Pi

```
flag[j] = TRUE; turn = i;
while (flag[i] && turn == i);

  critical section

flag[j] = FALSE;

  remainder section
```

Pj

Algorithm for Process Pi

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION

    }
```

Synchronization Hardware

- Many systems provide hardware support for critical section code.
- Hardware features can make programming task easier and improve system efficiency.
- Modern machines provide special atomic hardware instructions to solve critical section problem in a relatively simple manner.
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

- In general, any solution to critical-section requires a lock in order to prevent race condition.

```
do {

        Acquire lock

                critical section

        Release lock

                Remainder section


} while (true)
```

TestAndndSetInstruction

- Definition:

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

Solution using TestAndSet

Shared boolean variable lock, initialized to false.

```
while TS(&lock);

   critical section

lock = FALSE;

   remainder section
```

Solution using TestAndSet

Shared boolean variable lock, initialized to false.

Solution:

```
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:
}
```

Solution using Swap

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

```
key = TRUE;
do
{
   SWAP(&lock, &key);
} while (key);

   critical section

lock = FALSE;

   remainder section
```

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

}
```

Semaphore

- *A* Semaphore is a special integer variable, it can be accessed only through two standard atomic operations:
  - Wait() and Signal()
  - Originally called P() and V()
  - 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
  - Can be used to control access to a given resource consisting of a finite number of instances.
- Binary semaphore
  - integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
  - Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

    Critical Section

    signal (S);

Semaphore Implementation

- The main disadvantage of the above semaphore definition is busy waiting which wastes CPU cycles.
  - Spinlock
  - Process spins while waiting for the lock.
  - In a multiprocessor system, spinlock is often used to protect a short code.
- 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

- To overcome busy waiting, we can modify the implementation of semaphore S and the definition of wait() and signal().
- Now that a semaphore is a struct.
  - a value (of type integer)
  - a waiting queue
- 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){
        value--;
        if (value < 0) {
                add this process to waiting queue
                 block();  }
 }
```

- Implementation of signal:

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

Deadlock and Starvation

- The implementation of a semaphore with a waiting queue may result in one of the two following situations.
- 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 Semaphore

```
Swait(S1, S2, …, Sn)
        {
while (TRUE)
{
     if (S1 >=1 && S2 >= 1 && … && Sn >= 1)
     {
       for (i = 1; i <= n; ++i)  --Si;

       break;
     }
  else
  {    add the process to Sj.queue //no available resource to
use and Sj is the first semaphore of which value is less than 1 ;
       block();
     }
}
}
```

AND Semaphore

```
Ssignal(S1, S2, …, Sn)
{
  for (i = 1; i <= n; ++i)
  {
    ++Si;       //release resources;
    for (each process P waiting in Si.queue)
          //check if there is some process waiting;
    {
      remove a process P from Si.queue;
      if (Swait(S1, S2, …, Sn) )
           //different from  signal, P will be waken up when
it acquires all resources ;
          {    //all needed resources are available;
          wakeup(P);
          }
      else
          {    //otherwise;
          add process P to another waiting queue;
          }
    }
  }
```

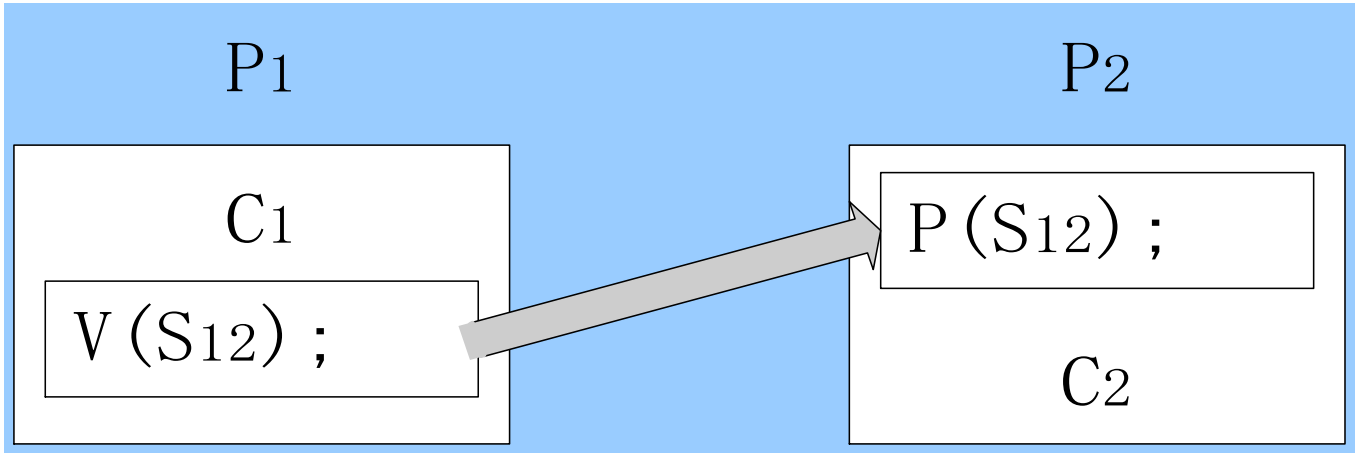
Mutual exclusion using semaphore

Semaphore mutex ; // initialized to 1

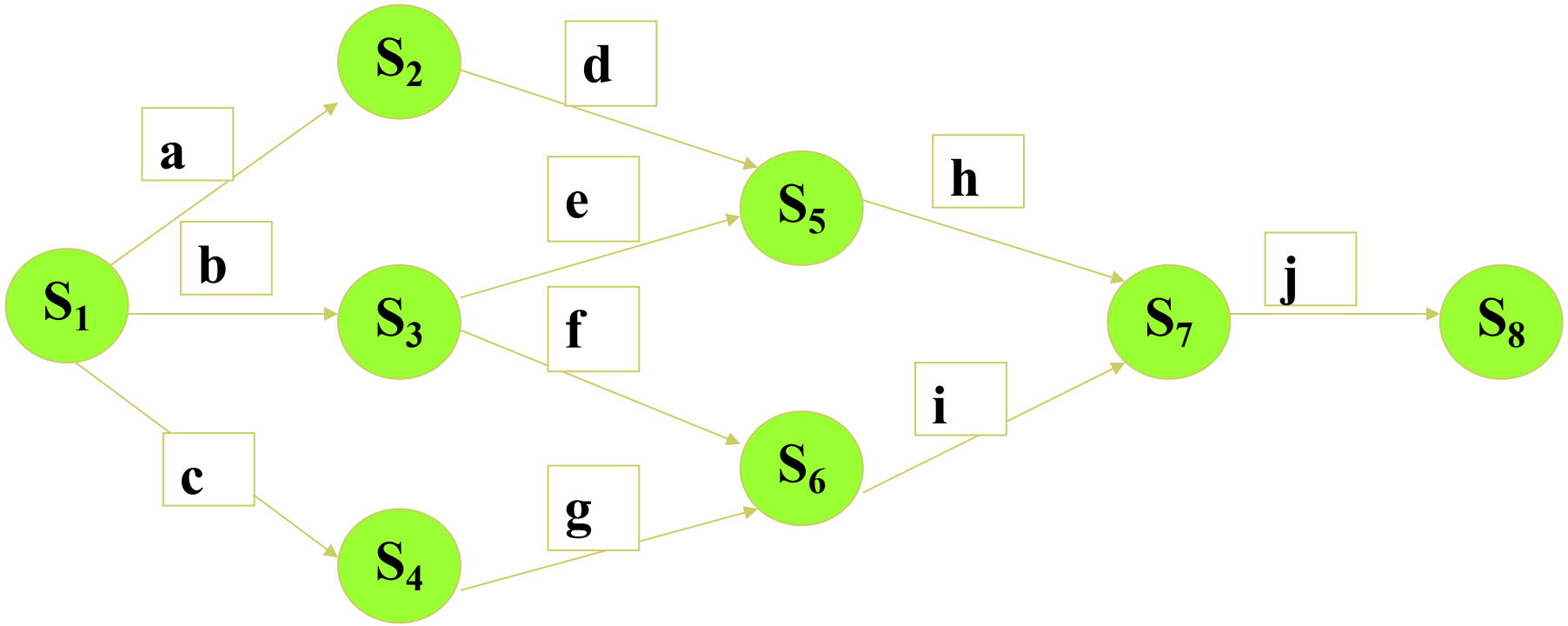
```
P(mutex);
    critical section
V(mutex);
    remainder section
```

Synchronization using semaphore

- Consider two concurrently running processes P1 and P2, P1 with a statement C1 and P2 with a statement C2. suppose we require that C2 be executed only after C1 has completed
- We can define a common 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);}

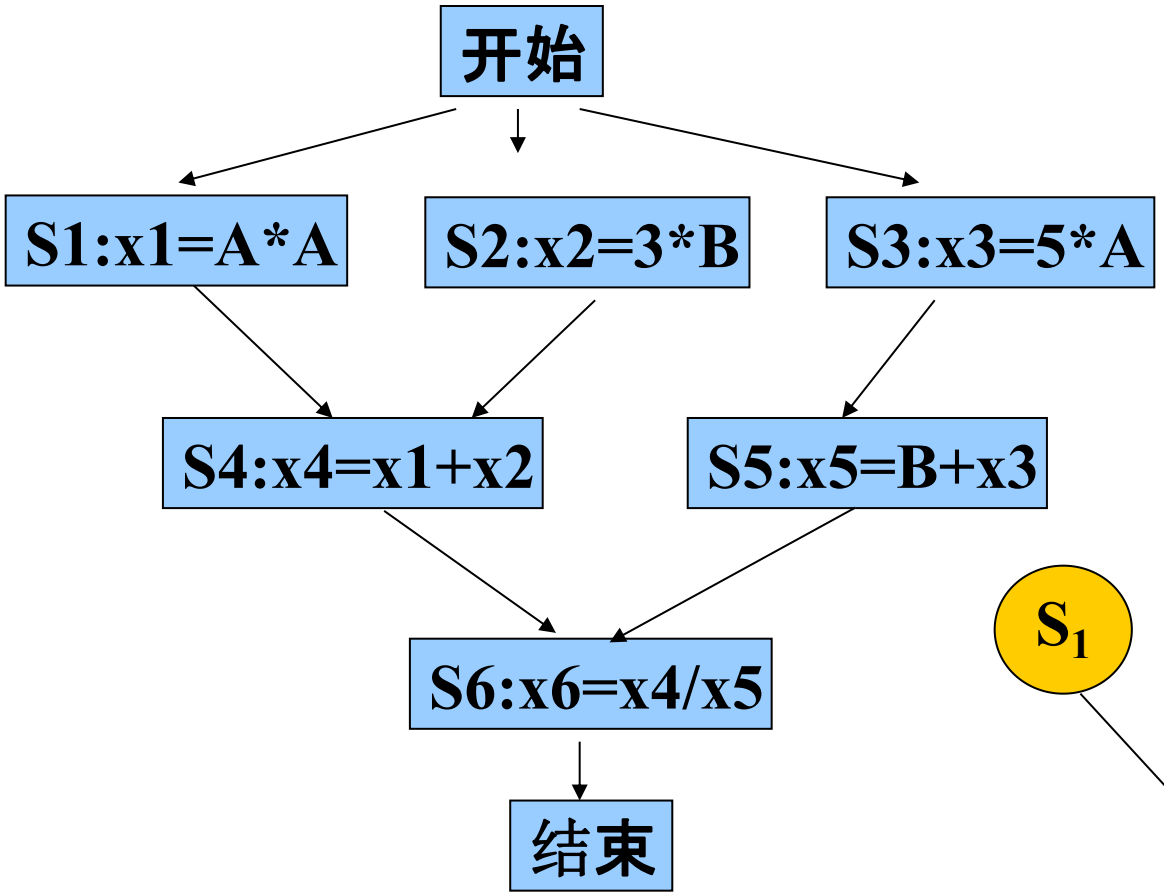
{P(j);S8;}

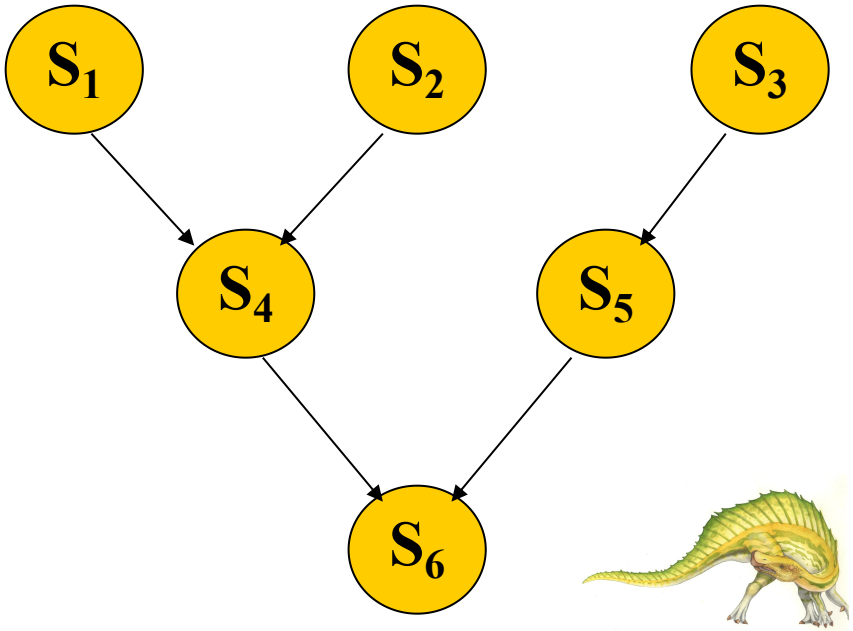
coend

Example 2

- $(A^2+3B)/(B+5A)$,若A,B已赋值,

试画出该公式求值过程的前趋图。


开始
S1:x1=A*A
S2:x2=3*B
S3:x3=5*A
S4:x4=x1+x2
S5:x5=B+x3
S6:x6=x4/x5
结束


S1
S2
S3
S4
S5
S6

（A2+3B）/(B+5A)

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

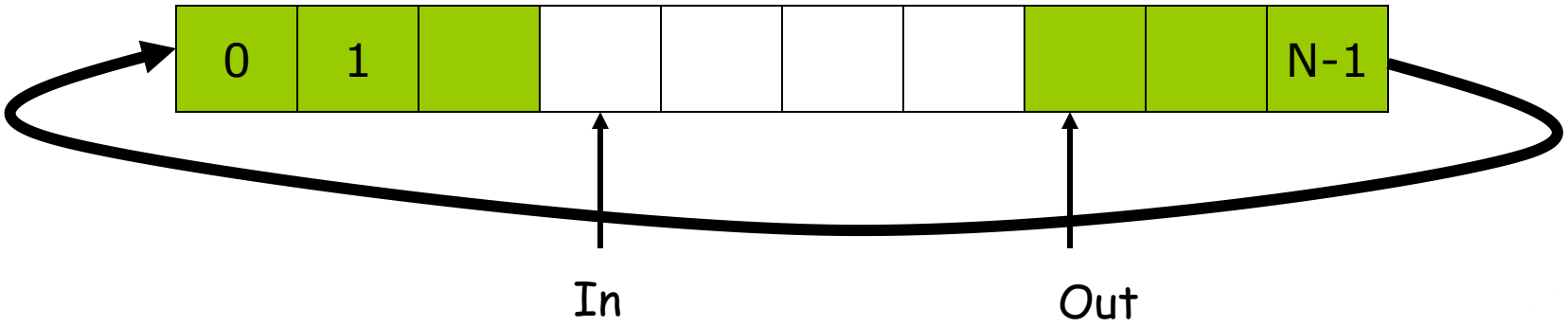
- Problem:
  - producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization for coordinating producer and consumer

Bounded-Buffer Problem

- 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.

The structure of Producer/Consumer process

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

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

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

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

                //  remove an item from  buffer

        signal (mutex);
        signal (empty);

                //  consume the removed item

}
```

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() 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();
}
```

remove() Method

```
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;
  }
```

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

What if buffer is full?

Producer

```
void put(char ch) {

   wait(mutex);
   wait (empty);

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

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

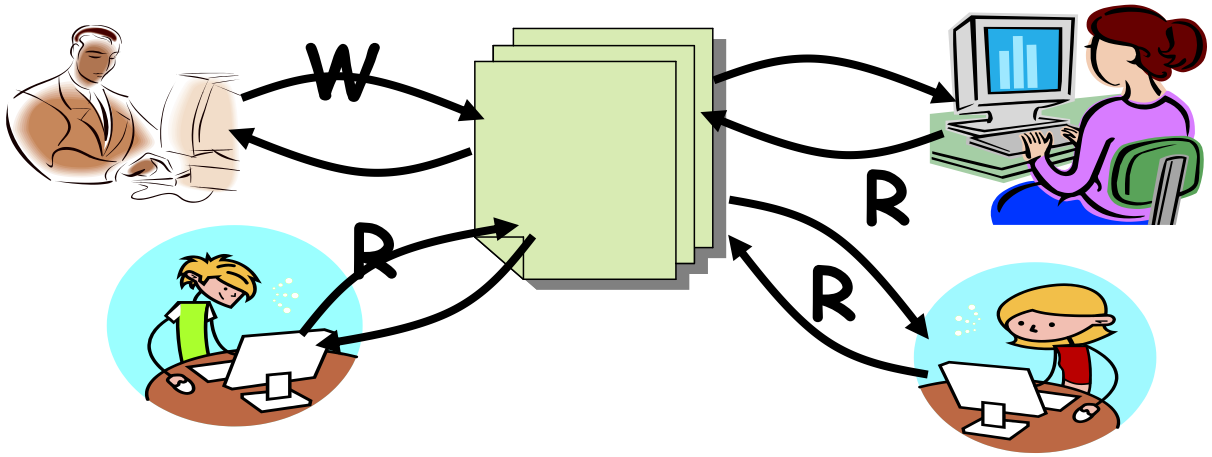
Consumer

```
char get() {

   wait (full );
   wait (mutex);

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

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

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?

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.
- Shared Data
  - Data set
  - Integer readcount initizalized to 0.
  - Semaphore Rmutex initialized to 1.
  - Semaphore Wmutex initialized to 1.

Readers-Writers Problem

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 mutex, wrl;
                  integer readcount;
Init: mutex = 1, wrl = 1, readcount = 0;
```

Writer

```
do {
        wait(wrl);
        . . .
        /*writing is performed*/
        . . .
        signal(wrl);

}while(TRUE);
```

Reader

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

}while(TRUE);
```

Readers-Writers Problem: Reader

```
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;
}
```

Readers-Writers Problem: Writer

```
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() 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;
  }
```

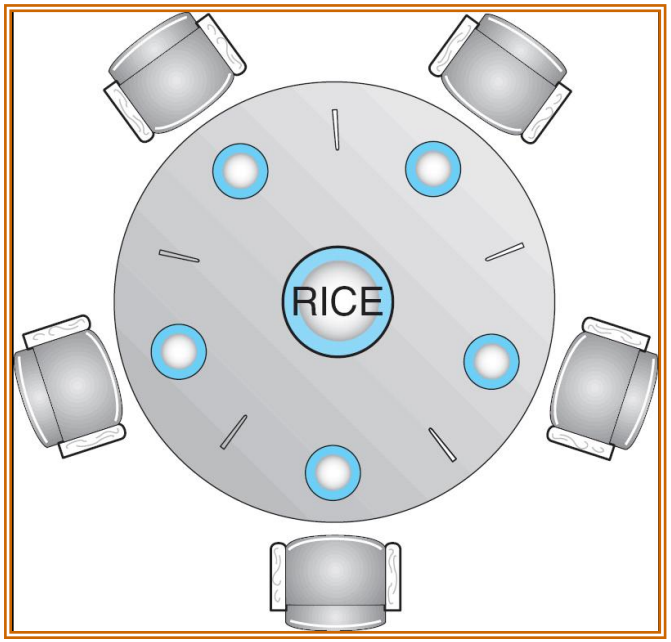
endRead() Method

```
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?

Solution2 : AND semaphore

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

Discussion about Semaphores

- The value of a semaphore represents the number of available resource
  - S>0: there are S available resources
  - S=0: all resources are being used
  - S<0: there are |S| processes waiting for the resource
- Operations can be done to a semaphore S:
  - Initialization: >=0
  - Wait(): request a resource
  - signal (): release a resource

Discussion about Semaphores(Cont.)

- Either wait(S) or signal(S) can not be omitted
  - For mutual exclusion: they appear in the same process
  - For synchronization: they appear in different process respectively
- For two successive wait(S1) and wait(S2) in a process, the order is important. If S1 is for synchronization, and S2 is for mutual exclusion, then
  - The correct order is wait(S1) ; wait(S2)
  - If they are written in the order wait(S2) ;wait(S1) , there would be a deadlock.

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，则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。

- 分析:此问题实际上是生产者-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）
- 解答：设置3个信号量：
  - mutex=1,实现盘子是否可用，实现与女儿/儿子的同步
  - 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

吃水果问题

- 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

Problems with Semaphores

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors.
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

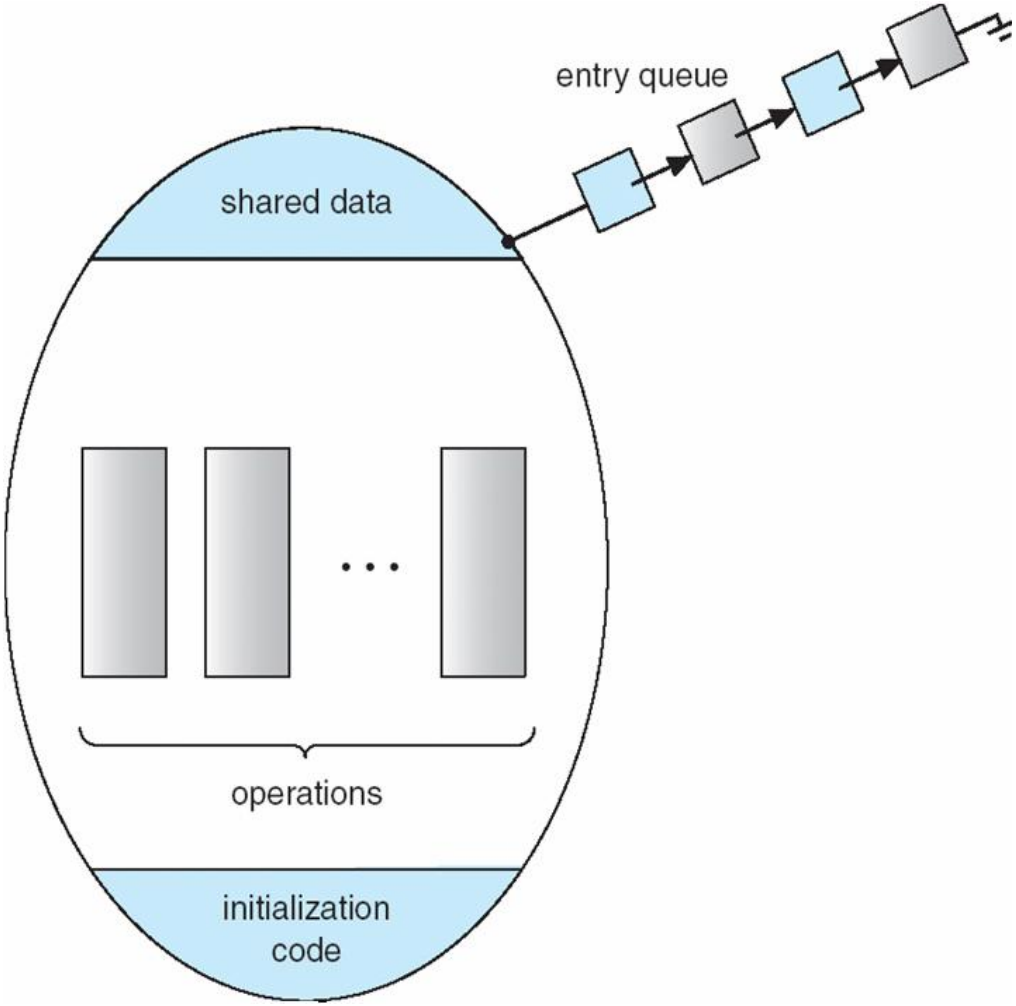
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

    Initialization code ( ….) { … }
            …
   }
}
```

Monitors

- A programming language construct
- A collection of procedures, variables, data structures
- Access to it is guaranteed to be exclusive.
  - By the compiler (not programmer)
- Monitors use separate mechanisms for the two types of synchronization
  - use locks for mutual exclusion
  - use condition variables for ordering constraints
- **monitor = a lock + the condition variables associated with that lock**

Schematic view of a Monitor



Condition Variables

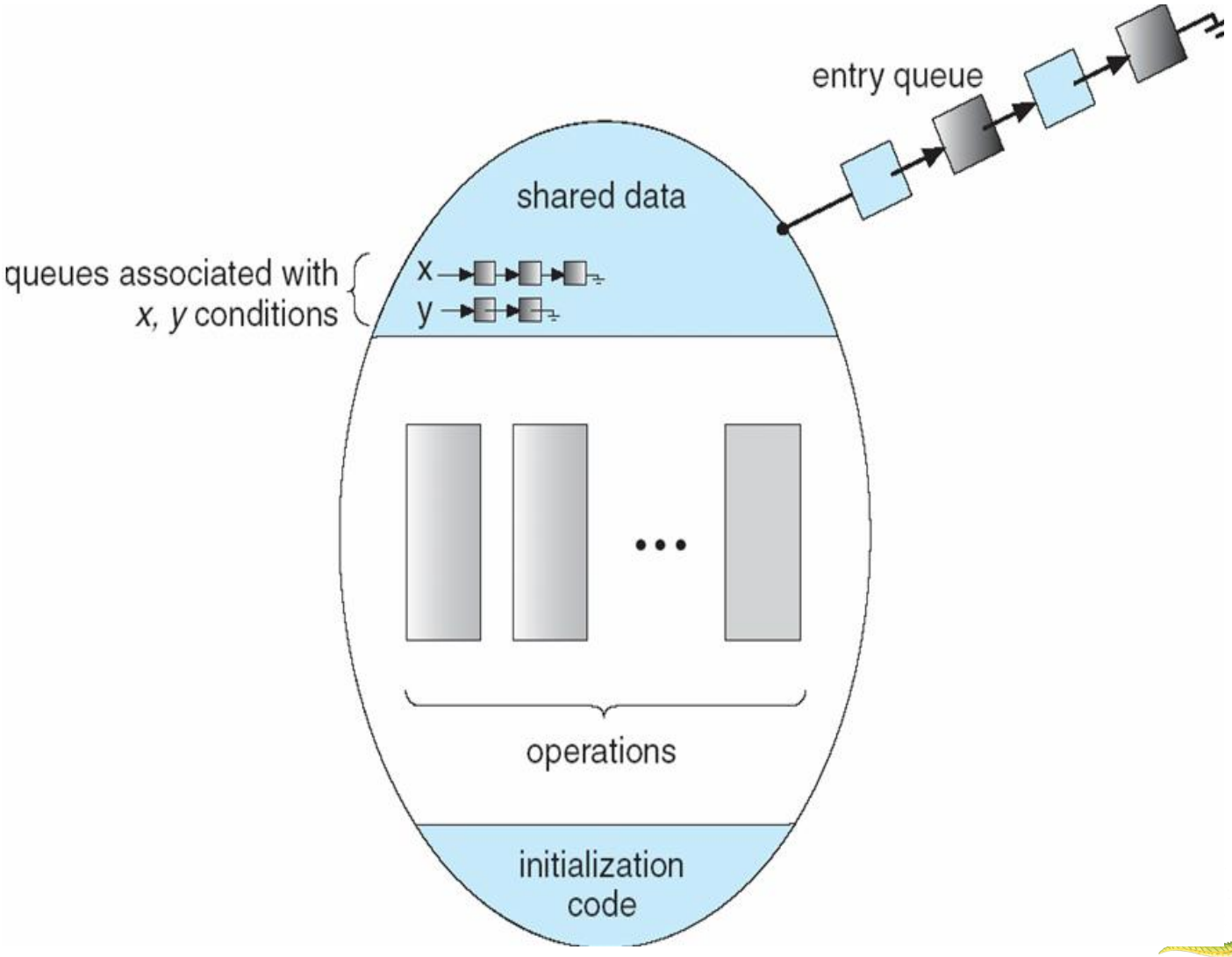
- Main idea:
  - make it possible for thread to sleep inside a critical section
- Approach:
  - by atomically releasing lock, putting thread on wait queue and go to sleep
- Each variable has a queue of waiting threads
  - threads that are sleeping, waiting for a condition

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 ()

Monitor with Condition Variables



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

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

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;
    }
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

Monitor Implementation Using Semaphores

- For a monitor, a semaphore mutex(initially = 1) is necessary to ensure mutual exclusion.
  - A process must execute wait(mutex) before entering the monitor, and execute signal(mutex) after leaving the monitor.
- Another semaphore next(initially =0) is introduced for a signaling process to suspend itself until the signaled process leaves or waits.
- Next-count is an integer variable to count the number of processes suspended on next.

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

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

```
wait(mutex);
    …
        body of F;

    …
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implem

因为唤醒了其他进程,则该进程需要等待直到被唤醒进程离开或等待为止

- For each condition variable ***x***, we have:

// (initially = 0)

调用x.wait而等待的进程数

当前在管程中的进程因为资源不可用而等待,则需要唤醒一个进程进入管程

x.wait can be implemented as:

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

x.signal can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        //只允许一个进程活动
        next-count--;
}
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads
- JAVA

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 a reader-writer lock.

Solaris Synchronization

- adaptive mutex is used to protect critical data
- On a multiprocessor system
  - spinlock
    - If the lock is held by a thread currently running on another CPU, the thread spins while waiting for the lock to become available
  - Not spin
    - If the thread holding the lock is not running, the thread blocks until the releasing of the lock.
- On a single-processor system
  - The thread holding the lock is never running if the lock is tested by another thread, so threads always sleep rather than spin if they encounter a lock.

Solaris Synchronization

- Reader-writer locks are used to protect data that are accessed frequently but are usually accessed in a read-only manner.
- More efficient than semaphore
- Used on only long sections of code.

Windows XP Synchronization

- For kernel thread
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
- For user thread
  - provides dispatcher objects which may act as mutexes, semaphores and events
  - An event acts much like a condition variable
  - An event may notify a waiting thread when a desired condition occurs.

Linux Synchronization

- Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be peempted.
- Now the Linux kernel is fully preempted.
- Linux kernel provides:
  - Semaphores
    - For single processor system
  - spin locks
    - For SMP

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
    - To protect critical section of code
  - condition variables
    - behave much as described in monitor
  - read-write locks
- Non-portable extensions include:
  - semaphores
  - spin locks

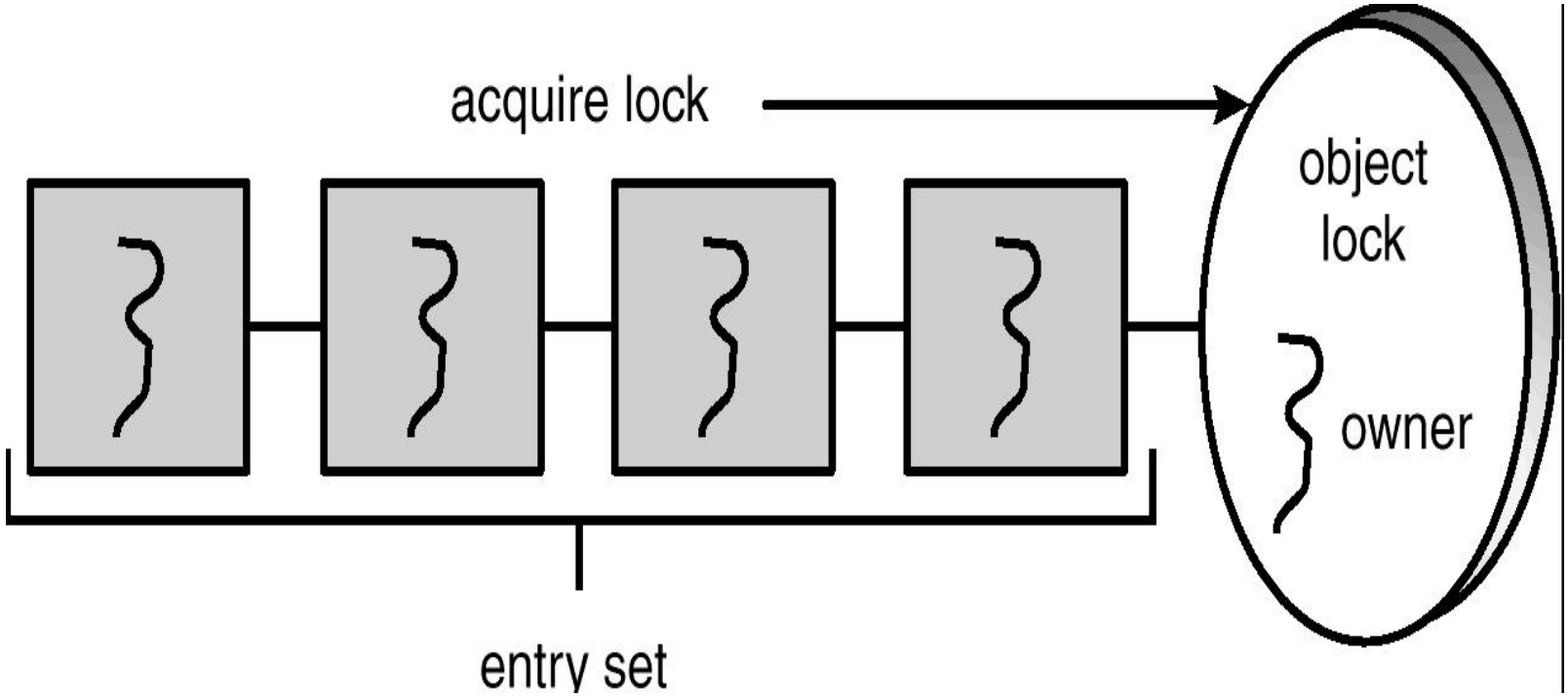
Java Synchronization

- Synchronized, wait(), notify() statements
- Multiple Notifications
- Block Synchronization
- Java Semaphores

Java Synchronization

- Synchronized statements
- Every object in JAVA has associated with it a single lock.
- When a method is declared as synchronized, calling the method requires owning the lock for the object.
- Entry set:
  - the set of threads waiting for the lock to become available.

Entry Set



synchronized enter() Method

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

synchronized remove() Method

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

Java Synchronization

- Producer-consumer problem:
  - The object BoundedBuffer has one lock
  - A producer must acquire the lock before invoking method enter().
  - When the lock is owned by a producer, if a consumer want to invoke remove(), it has to block until the producer exits the method enter() and release the lock.
  - This ensure mutual exclusion.

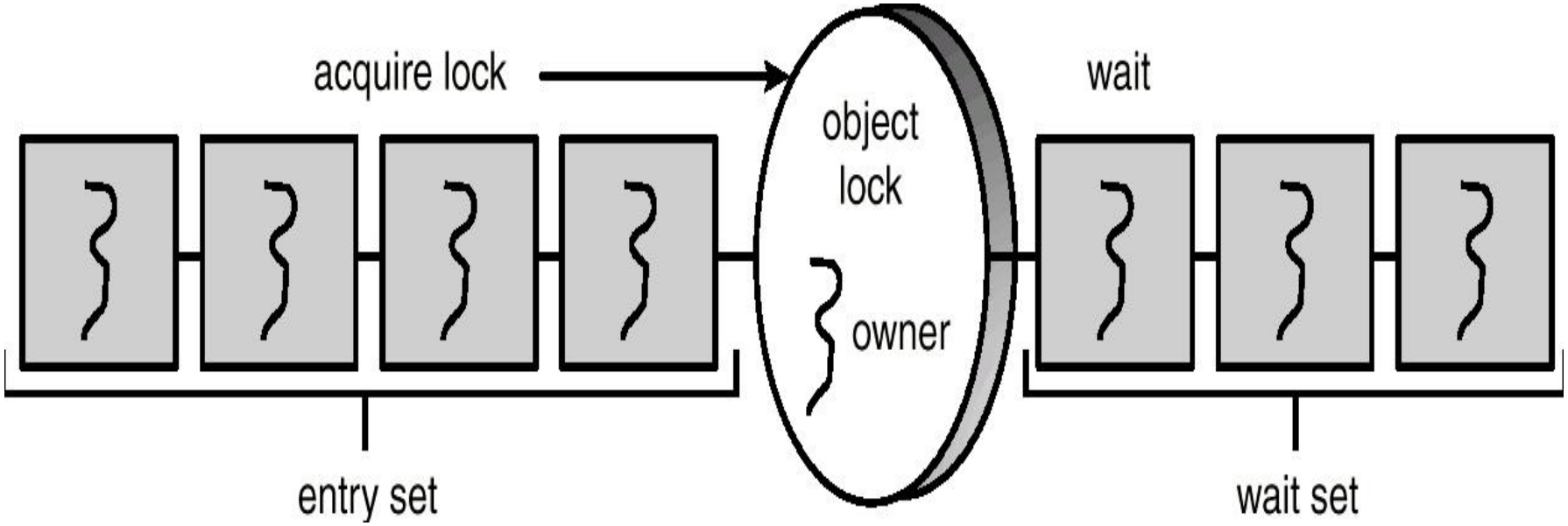
Java Synchronization

- However, lock ownership has led to another problem.
- Assume that the buffer is full and the consumer is sleeping.
- If the producer calls enter(), it will be allowed to continue because the lock is available. When the producer invokes enter(), it sees a full buffer and performs the yield(), but it still owns the lock for the object.
- When the consumer awakens and try to call remove(), it will block because it does not own the lock.
- This leads to deadlock.

Java Synchronization

- Solution:
  - In addition to having a lock, every object also has a wait set.
  - Two new methods: wait() and notify().
- If the producer calls enter() and the buffer is full, it will release the lock by invoking wait(), and enter to the wait set for the object.
- So the consumer can acquire the lock and call remove(), and then awake the producer by notify().

Entry and Wait Sets



The wait() 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

The notify() Method

- 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() 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();
}
```

remove() with wait/notify Methods

```
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() selects an arbitrary thread T from the wait set
- If there are multiple threads in wait set and more than one condition for which to wait, it is possible to have another deadlock.
- 例:有5个线程T1，T2，T3，T4，T5，一个共享变量turn，5个线程都需要调用dowork()方法，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在调用dowork()。当T3完成了它的工作，它把turn设为4，并调用notify()。notify()从等待集中任意选取一个线程，假设选中了T2，但当T2恢复运行后检查条件时，发现turn并不是它的值，于是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。

Multiple Notifications

- notify() selects an arbitrary thread from the wait set. This may not be the thread that you want to be selected.
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.

The Readers-Writers Problem

- JAVA solution to reader-writer problem：
  - readerCount： number of readers
  - dbReading： is set to true if the database is currently being read
  - dbWriting： is set to true if the database is currently being written
  - startRead()、endRead ()、startWrite ()、endWritet (): are all declared as synchronized to ensure mutual exclusion th the shared variables.

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;
}
```

startRead() 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
- This yields a lock scope that is typically smaller than a synchronized method

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

Java Semaphores

- Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism

Semaphore Class

```
public class Semaphore {
   public Semaphore() {
            value = 0;
    }
   public Semaphore(int v) {
            value = v;
    }
   public synchronized void P() { /* see next slide */ }
    public synchronized void V() { /* see next slide */ }
   private int value;
}
```

P() Operation

```
public synchronized void P() {
    while (value <= 0) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }
    value --;
}
```

V() Operation

```
public synchronized void V() {
    ++value;
    notify();
}
```

Java Synchronization rules

- a thread that owns the lock for an object can enter another synchronized method for the same object.
- a thread can nest synchronized method invocations for different objects. thus, a thread can simultaneously own the lock for several different objects

Java Synchronization rules(Cont.)

- if a method is not declared as synchronized, then it can be invoked regardless of lock ownership, even while another synchronized method for the same object is executing.
- if the wait set for an object is empty, then a call to notify() or notifyall() has no effect.

assignment

- P233
  - 6.4
  - 6.9
  - 6.11

End of Chapter 6