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

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 (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Consumer

```
while (count == 0)
   ; // do nothing

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

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:

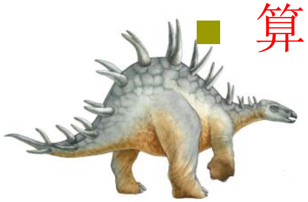
    S0: producer execute register1 = count {register1 = 5}
    S1: producer execute register1 = register1 + 1 {register1 = 6}
    S2: consumer execute register2 = count {register2 = 5}
    S3: consumer execute register2 = register2 - 1 {register2 = 4}
    S4: producer execute count = register1 {count = 6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

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - 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 - 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

Solution to Critical-Section Problem

- 必须强制实施互斥：在具有关于相同资源或共享对象的临界区的所有进程中，一次只允许一个进程进入临界区；
- 一个在非临界区的进程必须不干涉其他进程；
- 不允许一个需要访问临界区的进程被无限延迟；
- 没有进程在临界区时，任何需要进入临界区的进程必须能够立即进入；
- 相关进程的速度和处理器数目没有任何要求和限制；
- 一个进程阻留在临界区中的时间必须是有限的；
  - 有空让进;
  - 无空等待;
  - 择一而入;
  - 算法可行;

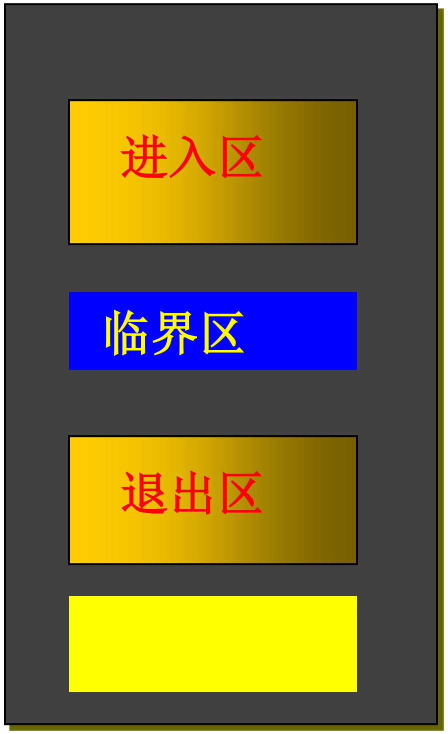


Critical-Section Problem

1. Race Condition - When there is concurrent access to shared data and the final outcome depends upon order of execution.
2. Critical Section - Section of code where shared data is accessed.
3. Entry Section - Code that requests permission to enter its critical section.
4. Exit Section - Code that is run after exiting the critical section

Structure of a Typical Process

```
while (true) {
        entry section
              critical section
        exit section
              remainder section
}
```



Peterson's Solution

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

Algorithm for Process P_i

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

              critical section

        flag[i] = FALSE;

              remainder section

}
```

```
while (true) {
        acquire lock

              critical section

        release lock

              remainder section

}
```

Critical Section Using Locks

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-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

互斥：硬件的支持-专用指令

TS(Test-and-Set)指令

利用TS指令实现的进程互斥算法是：每个临界资源设置一个变量Blot，Blot==1表示正被占用，Blot=0表示空闲，初值为0.

```
While !TS(Blot);
```

临界区

```
Blot=0
```

剩余区

TS指令的处理过程

Test and Set Instruction

```
boolean testset (int i)
{
    if (i == 0)
    {
        i = 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
            /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));
}
```

忙等待/自旋等待

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
            /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

互斥：硬件的支持-专用指令

Exchange（或Swap）指令

利用Exchange指令实现的进程互斥算法是：每个临界资源设置一个公共布尔变量Blot;初值为0;每个进程设置一个私有布尔变量keyi,用于与Blot间的信息交换。

在进入区利用Exchange指令交换Blot和keyi的内容，然后检查keyi的状态；有进程在临界区时，重复交换和检查的过程，直到其它进程退出。

互斥：硬件的支持-专用指令

Excnange（或Swap)指令

```
void exchange (int register, int memory)
 {
int temp;
temp = memory;
memory = register;
register = temp;
          }
```

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
             exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
             exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: acquire() and release()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

    release() {
        value++;
    }
```

信号量

- 前面方法解决临界区调度问题的缺点：
  1) 对不能进入临界区的进程，采用忙式等待测试法，浪费CPU时间。
  2) 将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重了用户编程负担。

- 1965年E.W.Dijkstra(荷兰人）提出了新的同步工具--信号量和P （荷兰语的测试Proberen） semWait、V操作（荷兰语的增量Verhogen） semSignal。

信号量

- 信号量和P、V操作 ，将交通管制中多种颜色的信号灯管理交通的方法引入操作系统，让两个或多个进程通过特殊变量展开交互。

- 信号量：一种软资源
  - 一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(Semaphore)，复杂的进程合作需求都可以通过适当的信号结构得到满足。
- 原语：内核中执行时不可被中断的过程

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

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```

Semaphore Implementation

- Must guarantee that no two processes can execute acquire () and release () 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
- 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 **acquire()**:

```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block;
    }
}
```

- Implementation of **release()**:

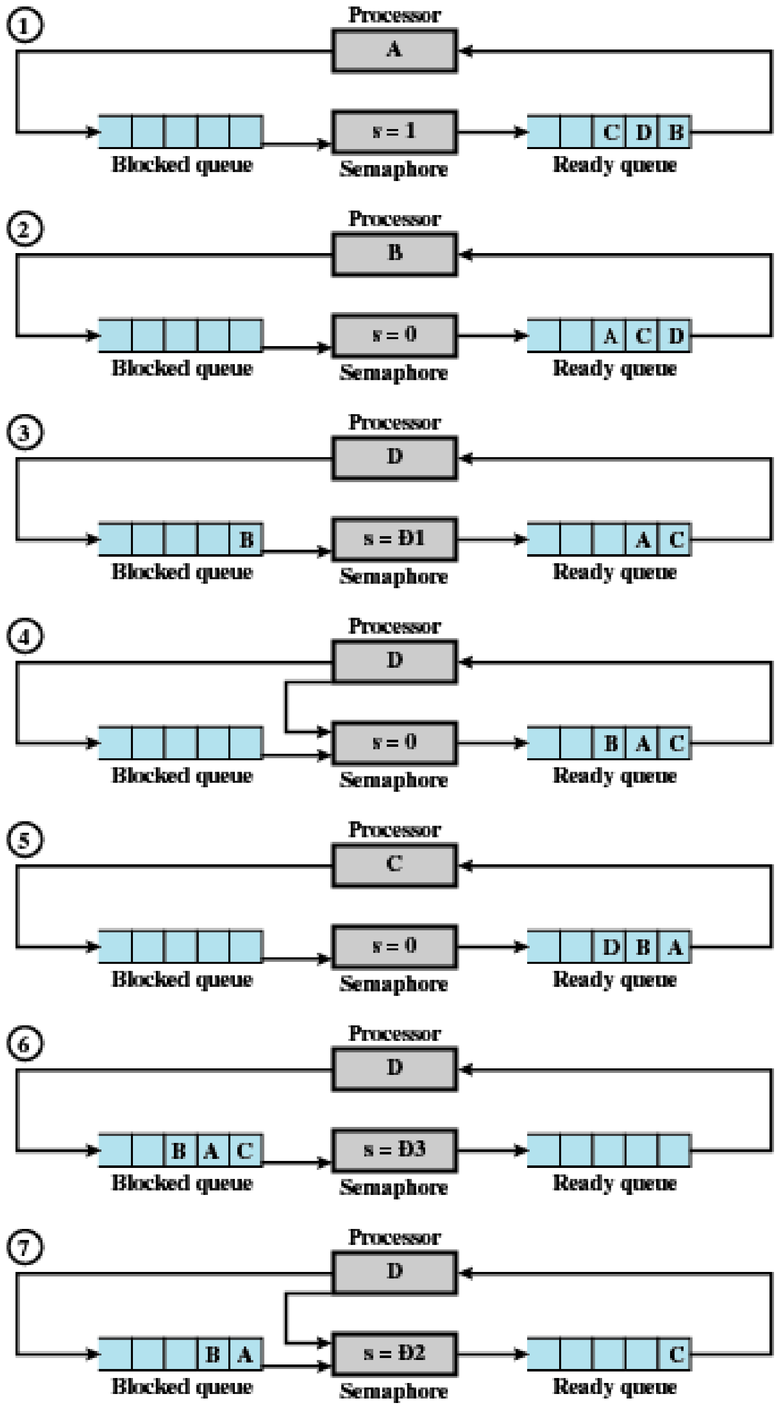
```
release(){
    value++;
    if (value <= 0) {
       remove a process P from list
       wakeup(P);
    }
}
```

二元信号量

- 一个信号量可以初始化成0或1
- semWaitB操作检查信号量的值，若值为0，则执行semWaitB的进程被阻塞，否则，将值改为0，继续执行；
- semSignalB操作检查是否有任何进程在该信号上受阻，若有，受阻的进程就会被唤醒，若没有进程受阻，那么信号量设为1。

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```


1
Processor
A
Blocked queue
s = 1
Semaphore
C D B
Ready queue
2
Processor
B
Blocked queue
s = 0
Semaphore
A C D
Ready queue
3
Processor
D
B
Blocked queue
s = Ð1
Semaphore
A C
Ready queue
4
Processor
D
Blocked queue
s = 0
Semaphore
B A C
Ready queue
5
Processor
C
Blocked queue
s = 0
Semaphore
D B A
Ready queue
6
Processor
D
B A C
Blocked queue
s = Ð3
Semaphore
Ready queue
7
Processor
D
B A
Blocked queue
s = Ð2
Semaphore
C
Ready queue

信号量小结

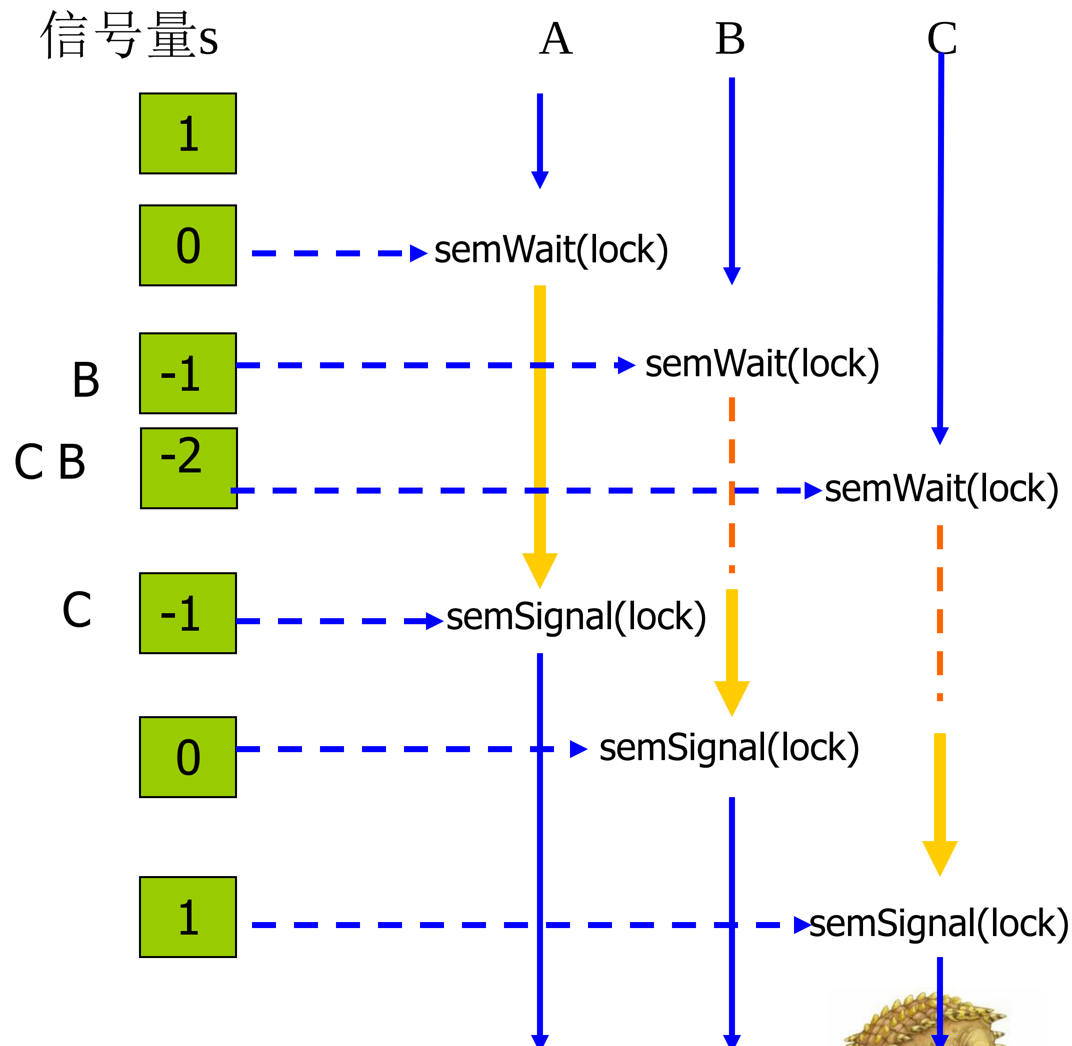
- 若信号量s.count>=0，则该值等于可以执行semWait(s)而不需要挂起的进程数。
- 若信号量s.count为负值，则其绝对值等于登记排列在该信号量s.queue队列之中等待的进程个数、亦即恰好等于对信号量s实施semSignal操作而被封锁起来并进入信号量s.queue队列的进程数。

信号量 s.count
- **>=0 表示可用临界资源的实体数;**
- **<0 表示挂起在s.queue对列中的进程数**

互斥

有三个进程A、B、C需共享一个临界资源，

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```



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

```
S.acquire();          Q.acquire();
Q.acquire();          S.acquire();
. .
. .
. .
S.release();          Q.release();
Q.release();          S.release();
```

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

Classical Problems of Synchronization

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

生产者/消费者问题

有n(>=1)个生产者产生某种类型的数据，并放置在缓冲区，有m(>=1)个消费者从缓冲区取数据，每次取一项；系统保证对缓冲区的重复操作。其中，*P*和*C*都是并发进程，生产者*P*生产的产品投入缓冲区；只要缓冲区不空，消费者进程*C*就可从缓冲区取走并消耗产品。

- 生产进程P
- 生产一个产品
- 送缓冲区

buffer

缓冲区

- 消费进程C
- 从缓冲区取一产品
- 打印

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process

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

Producer Process

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);
```

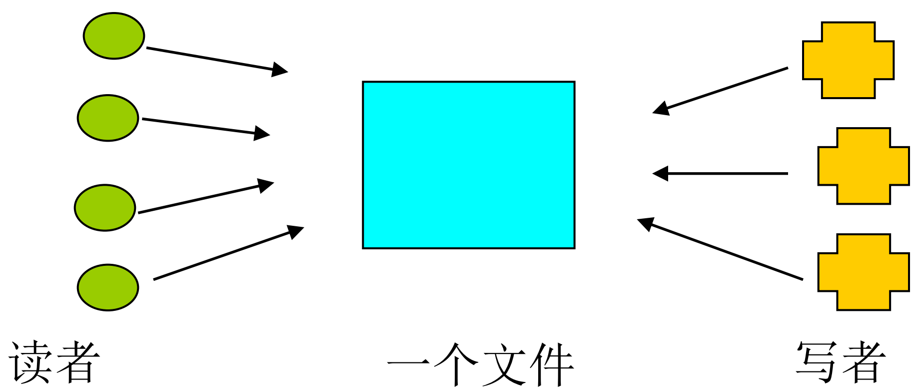
Initially:
full = 0, empty = n, mutex = 1

有两组并发进程：读者和写者，共享一个文件F，要求：

任意多个读者可以同时读文件；

一次只有一个写进程可以往文件中写；

写进程执行写操作前，则禁止任何读进程读文件；



问题：在计算机系统中当若干个并发进程都要访问某个共享文件时应区分是读还是写。

1. 允许多个进程同时读文件（读一读允许）；
2. 不允许在进程读文件时让另外一进程去写文件；有进程在写文件时不让另外一个进程去读该文件（“读-写”互斥）；
3. 更不允许多个写进程同时写同一文件（“写-写”互斥）。

因此读-写进程之间关系为：“读-写”互斥、和“读-读”允许。

如果读者到：

1）无读者、写者，新读者可以读

2）有写者等，但有其它读者正在读，则新读者也可以读

3）有写者写，新读者等

如果写者到：

1）无读者，新写者可以写

2）有读者，新写者等待

3）有其它写者，新写者等待

- 读者优先：当存在读者时，写操作将被延迟，并且只要有一个读者活跃，随后而来的读者都将被允许访问文件。从而，导致了写进程长时间等待，并有可能出现写进程被饿死。

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
    }
```

- 实际的系统为写者优先：即当有进程在读文件时，如果有进程请求写，那么新的读进程被拒绝，待现有的读者完成读操作后，立即让写者运行，只有当无写者工作时，才让读者工作。

Readers-Writers Problem

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

Readers-Writers Problem

```
wait(wrt);
  …
    writing is performed
  …
signal(wrt);
```

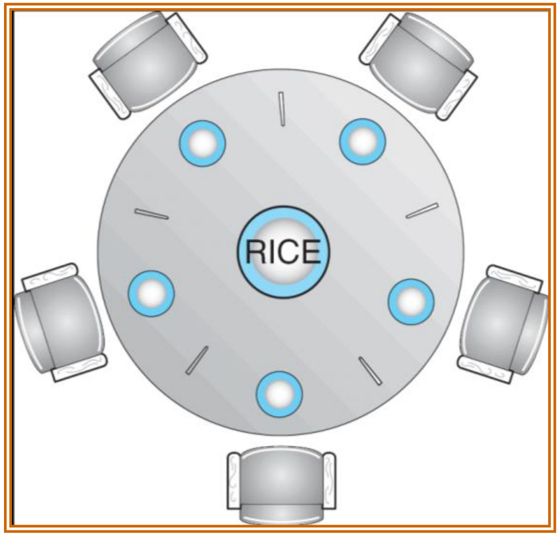
Writer Process

- **Shared data**
  semaphore mutex, wrt;
  Initially
  mutex = 1, wrt = 1, readcount = 0

Reader Process

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

Dining-Philosophers Problem



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

- Philosopher *i*:

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

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

管程

- 背景

  用信号量机制可以实现进程间的同步和互斥，但由于信号量的控制信息分布在整个程序中，其正确性分析很困难，使用不当还可能导致进程死锁。针对信号量的问题，Dijkstra与1971年提出，为每个共享资源设立一个“**秘书**”来管理对它的访问，一切来访者都要通过秘书，而秘书每次仅允许一个来访者（进程）访问共享资源。这样既便于系统管理共享资源，有能保证呼哧访问和进程间同步。1973年，Hansen和Hoare又把“秘书”的概念发展成为管程。

管程-定义

- 管程定义了一个数据结构和能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。
  - 局部数据变量只能被管程的过程访问，任何外部过程都不能访问；
  - 一个进程通过调用管程的一个过程进入管程；
  - 在任何时候，只能有一个进程在管程中执行，调用管程的任何其他进程都被挂起，以等待管程变为可用；

管程-组成

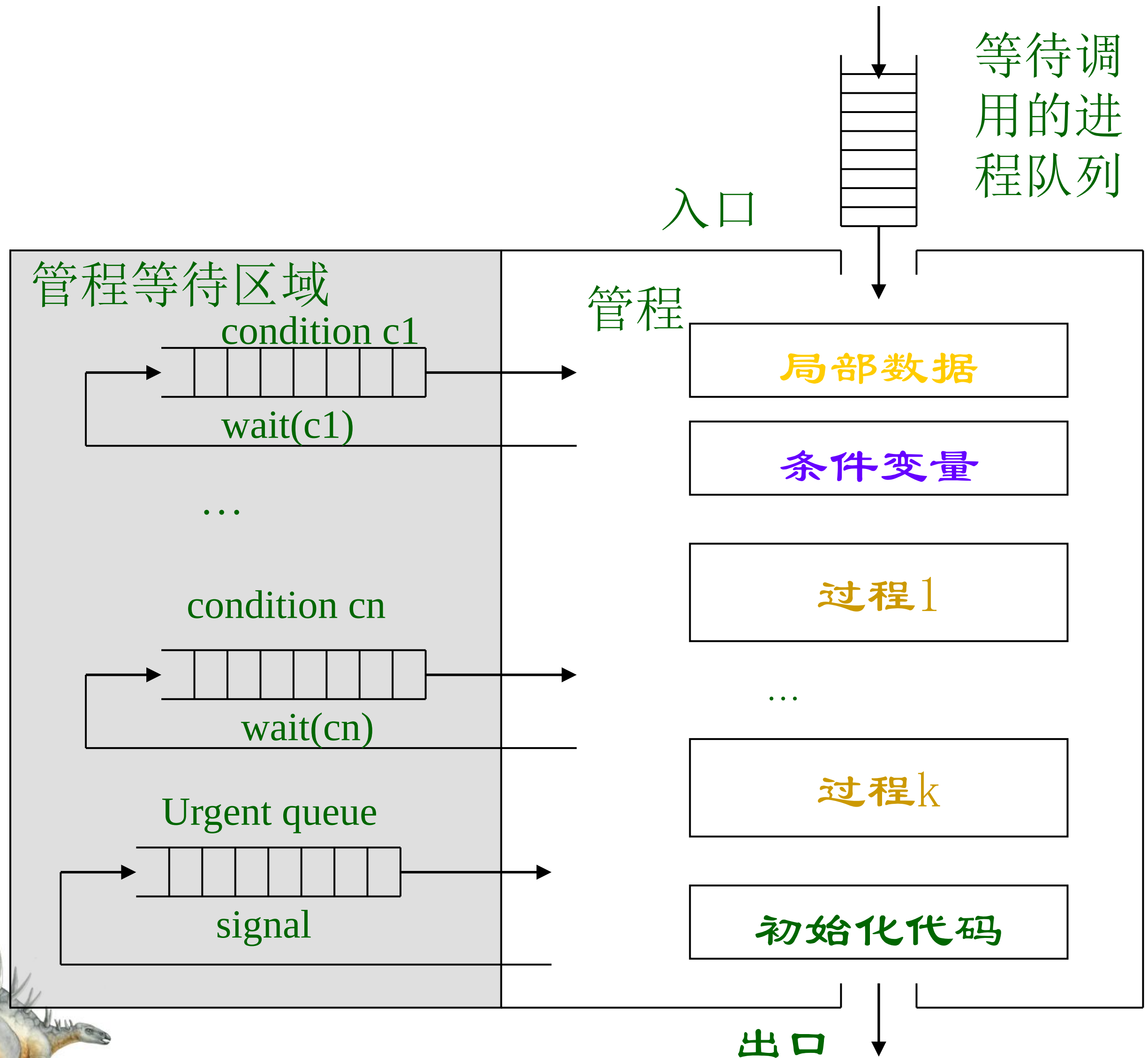
管程的组成部分

－管程的局部数据（共享变量）

－对该数据结构进行操作的一组过程（自己定义别人引用、引用别人）

－初始值

＋管程名字


等待调
用的进
程队列
入口
管程等待区域
condition c1
wait(c1)
…
condition cn
wait(cn)
Urgent queue
signal
管程
局部数据
条件变量
过程1
…
过程k
初始化代码
出口

有界缓冲生产者/消费者问题

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                        /* space for N items */
int nextin, nextout;                                      /* buffer pointers */
int count;                                        /* number of items in buffer */
cond notfull, notempty;               /* condition variables for synchronization */

void append (char x)
{
     if (count == N)
        cwait(notfull);                         /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                          /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0)
        cwait(notempty);                       /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                      /* one fewer item in buffer */
     csignal(notfull);                           /* resume any waiting producer */
}
{                                                              /* monitor body */
     nextin = 0; nextout = 0; count = 0;               /* buffer initially empty */
}
```

End of Chapter 6