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

6.1 Background

- Processes can execute concurrently
  - May be **interrupted** at any time, ***partially*** completing execution
- **Concurrent access to shared data may result in data inconsistency (数据的不一致性)**
- 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.

Data Inconsistency

```
int value=5;
void *runner1(void *param);
void *runner2(void *param);
int main(int argc, char *argv[])
{
   int pid;
   pthread_t tid1,tid2;
   pthread_attr_t attr1,attr2;

   pthread_attr_init(&attr1);
   pthread_create(&tid1,&attr1,runner1,NULL);
   pthread_attr_init(&attr2);
   pthread_create(&tid2,&attr2,runner2,NULL);

   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
   printf("Child: value=%d\n",value);
}//main
```

```
//threads
 void *runner1(void *param) {
    int value1=value;
    value1=value1+1;
    value=value1;
}
 void *runner2(void *param) {
    int value2=value;
    value2=value2-1;
    value=value2;
}
```

- 两个线程共享进程的变量value；
- 两个线程并发访问变量value时，可能导致数据的不一致性；即value的值不确定，可能的值是4,5,6；但应该是5；

趵突泉公园游人计数系统

- 公园有一个入口和一个出口；
- 系统设置一个记数器count，记录公园中游人的个数；count的初始值为0；
- 该系统由一个中心数据库服务器及两个客户端组成，两客户端可并发访问数据库的数据；
- 每进入一人，入口处系统执行如下操作
  - R1=count; R1++; count=R1;
- 每离开一人，出口处系统执行如下操作
  - R2=count; R2--; count=R2;

趵突泉公园游人计数系统

- count=5; //假定目前园子中有5人

Entry：

R1=count;
R1 +=1; //R1++
count=R1;

Exit：

R2=count;
R2 -=1; //R2--
count=R2;

公园游人的计数系统存在的问题

- 有的时候记数器count不能正确记录游园的人数；(?)

- 例如园中现有5人
  - 此时一个人进来的时候正好有一个人离开
  - 两进程几乎同时访问变量count
  - 此时count的值是多少?

- Four
- Five （should be）
- Six

- 说明了共享变量处理不好可能会导致数据的不一致性

火车票售票系统

- 目前从青岛北→济南高铁尚有100张票
- 有两个人购票，每人购票一张
- 数据库设置一个变量 tickets=100
- Person1：R1=tickets, R1=R1-1, tickets=R1
- Person2：R2=tickets, R2=R2-1, tickets=R2

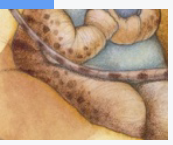
- 问两人购票后tickets的值是多少?

火车票售票系统

- 1. 第一个购票完成后第二个再购票，tickets=98
- 2. 两人几乎同时购票，即两人几乎同时访问变量tickets
  - R1=tickets, R1=R1-1, tickets=R1 与
  - R2=tickets, R2=R2-1, tickets=R2
  - 会交叉执行
  - 则购票后tickets=99

单选题 1分

设置



此题未设置答案，请点击右侧设置按钮

有两个并发进程P1和P2，共享初值为1的变量x。P1对x加1，P2对x减1。加1和减1操作的指令序列分别如下表示：

```
//加1操作

load R1,x   //取x到寄存器R1中
inc R1
store x,R1   //将R1的内容存入x
```

```
//减1操作

load R1,x   //取x到寄存器R1中
dec   R1
store x,R1   //将R1的内容存入x
```

两个操作完成后，x的值是（）。

A 可能为–1或3

B 只能为1

C 可能为0、1或2

D 可能为–1、0、1或2

提交

Bounded Buffer-problems

- 常称为 Producer-Consumer Problem
- 目的：诠释**共享变量处理不好可能会导致数据的不一致性；**
- **问题描述**
  - **多个生产者进程与多个消费者进程共享一个缓冲区buffer[BUFFER_SIZE]；**
  - **生产者将生产的产品依次放入缓冲区buffer中；**
    - **若缓冲区已满，则等待；**
  - **消费者依次从buffer中取出产品消费；**
    - **若缓冲区已空，则等待；**
- **利用变量int count=0记录buffer中产品的个数；**

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++;  //对应三条指令
}
```

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
          count--;    //对应三条指令
      /*  consume the item in nextConsumed
}
```

Bounded Buffer-problems

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

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

Bounded Buffer-problems(cont.)

- 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}
- The value of **count** may be either 4 or 6, where the correct result should be 5

Bounded Buffer-problems

- 如果有多个生产者进程共享变量in，多个消费者共享变量out，则in、out也存在同样的问题

- 打印机等设备共享时也存在同样的问题

Race Condition

- **Race condition**: The situation where several processes access, and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized** in some way.

6.2 The Critical-Section Problem

- 几个术语(terminologies)
  - *Atomic Operation (原子操作)*
  - Primitive （*原语*)
  - Critical-Resources （*临界资源*)
  - Critical-Section （*临界区*)

Atomic Operation

- The statements

  counter++; and

  counter--;

  must be performed ***atomically***.

- ***Atomic operation*** **means an operation that completes in its entirety without interruption.**
- **原子操作、操作的原子性**
- How?

原语（primitive）

- **原子操作（*Atomic Operation*）：**
  - **指一个操作中的所有动作要么全做，要么全不做；**
  - 该操作是一个不可分割的单位；
  - 在执行过程中不允许被中断；
  - 这些原子操作在管态（核心态）下运行；
  - 常驻内存；

- **原语（Primitive）：**
  - 是完成一定功能的一个过程
  - 与一般程序段的不同是，它是原子操作。
  - 原语是一段程序，与一般程序的不同点是，这段程序在执行期间不允许被中断；
  - **这段程序要么全执行，要么全不执行**

Critical-Resources

- Critical-Resources－临界资源
  - 系统中的某些资源，虽然可以提供给多个进程使用，但在一段时间内却只允许一个进程访问该资源。
  - 当一个进程正在访问该资源时，其它欲访问该资源的进程必须等待，仅当该进程访问完并释放该资源后，才允许另一进程对该资源进行访问。
  - *临界资源*：在一段时间内只允许一个进程访问的资源
  - 临界资源要求互斥地共享，或互斥地访问。
  - 许多物理设备、某些共享变量、表格等都属于临界资源；

The Critical-Section Problem (cont.)

- *n* processes all competing to use some shared data, or other critical resources;
- Each process has a code segment, called *critical section*, in which the shared data or other critical resources are accessed.
- **临界区(*critical section*)**
  - **在程序中访问临界资源的那段代码称为临界区(*critical section*)**
  - **进程对临界区必须互斥地进行访问**
- In order to access the critical resources exclusively – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- **将对临界资源的互斥访问转化为对临界区的互斥访问。**

Critical Section Problem(cont.)

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - **When one process in critical section, no other may be in its critical section**

- *Critical section problem* is to design a protocol to solve the issues above

Entry section and exit section

do {

Entry section

critical section

Exit section

reminder section

} **while (1)**;

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Discussion

现有两个并发执行的进程。

问：

它们能正确执行（结果是唯一、正确的）吗？why?

```
int x;

Process p1 {
    int y,z;
    x=1;
    y=0;
    if x>=1 then y++;
    z=y;  }

Process p2 {
    int t,u;
    x=0;
    t=0;
    if x<1 then t=t+2;
    u=t;  }
```

Discussion

现有两个并发执行的进程。

问：

它们能正确执行（结果是唯一、正确的）吗？why?

```
int x;

Process p1 {
    int y,z;
    x=1;
    y=0;
    if x>=1 then y++;
    z=y;  }

Process p2 {
    int t,u;
    x=0;
    t=0;
    if x<1 then t=t+2;
    u=t;  }
```

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； (**互斥--忙则等待**，保证临界区互斥访问)
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； (**前进--有空让进**，当无进程在临界区执行时，若有进程进入应允许；否则可能会出现“饥饿”现象)
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 (**有限等待**--当一个进程申请进入临界区，应限制其它进程进入临界区的次数，以便申请的进程有机会进入临界区）(otherwise starvation maybe occurs)
4. No busy waiting（让权等待,等待的时候释放CPU的执行权 ）(不是必须的)

类比

- 信号灯
  - 正常：(一边红，一边绿)
    - 满足：**Mutual Exclusion，Progress**
  - 红绿灯有限时间内交替
    - 满足：**Bounded Waiting**
  - 都为绿
    - 违反：**Mutual Exclusion**
  - 都为红
    - 违反：**Progress**
  - 一边长绿，一边长红
    - 违反：**Bounded Waiting**

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- *Critical section problem* is to design a protocol to solve ensure,
  - When one process in critical section, no other may be in its critical section, i.e Mutual Exclusion
- And the protocol should satisfy :
  - Progress, Bounded Waiting
- Solutions:
  - Peterson's Solution
  - Synchronization Hardware
  - Semaphore
  - Monitor

6.3 Peterson's Solution

- A classic software-based solution to the critical-section problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
        entry section
        critical section
        exit section
        reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Peterson's Solution Algorithm 1
(将教材给出的算法分解介绍)

- Shared variables:
  - **int turn**;
    initially **turn = 0 ////suppose i=0,j=1;**
  - **turn = i ⇒ P_i can enter its critical section**
- Process P_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        reminder section
} while (1);
```

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0; //suppose i=0,j=1;**
  - **turn = i** $\Rightarrow$ P_i can enter its critical section

- Process P_i

```
do {
while (turn != i) ;
   critical section
turn = j;
   reminder section
} while (1);
```

- Process P_j

```
do {
while (turn != j) ;
   critical section
turn = i;
   reminder section
} while (1);
```

Algorithm 1 - problem

- 变量turn相当于一个门票或token，由两进程轮流使用；
  - 获得令牌，进入；没有获得令牌，等待；
  - 退出时，移交令牌；
  - 两进程轮流地访问临界资源
- It requires strict **alternation of processes** in the execution of the critical section;
- 进程P_i与P_j交替使用临界资源；
- Satisfies mutual exclusion and bounded waiting, but not progress

Peterson's Solution Algorithm 2
(将教材给出的算法分解介绍)

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

```
do {
    flag[i] := true;   // P_i ready to enter CS
    while (flag[j]) ;  // P_j still possess CS?
        critical section  //no
    flag [i] = false;    //P_i release CS
        remainder section
} while (1);
```

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section

- Process P_i

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

- Process P_j

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

Algorithm 2 - Problem

- flag相当于门口的一个登记簿，当进程进门时登记，然后查看其它进程是否已经登记，如果其它进程已经登记，则等待；
- 退出时清除登记信息。
- 问题：
  - 第一个到来，登记，临时有事走开；
  - 第二个到来，登记，然后发现另一个已经登记，等待；
  - 第一个回来，发现第二个已经登记，等待；
  - 互相谦让

- When two processes arrive almost at the same time, both of them are looping forever in their respective *while* statements.
- 当两进程几乎同时到达，P_i与P_j可能会在各自的while循环中无穷循环、无限等待；（互相谦让）
- Satisfies mutual exclusion, but not progress requirement.

Peterson's Solution Algorithm 3

- 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

}
```

Algorithm 3- Peterson's Solution

- Combined shared variables of algorithms 1 and 2.

- Process P_i

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

- Process P_j

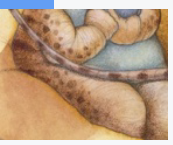
```
do {
    flag [j]:= true;
    turn = i;
    while (flag [i] and turn = i) ;
        critical section
    flag [j] = false;
        remainder section
} while (1);
```

Algorithm 3- Peterson's Solution

- 登记+令牌
- 在门口登记（欲访问资源），并把令牌移交给另一个进程
- 退出时消除登记信息；
- 如果双方都已经登记，但拥有令牌的进程会进入；
  - progress
- 如果对方尚未登记，但拥有令牌，自己会进入；
  - progress
- 如果对方已经登记，且拥有令牌，对方进入，自己等待；
  - mutual exclusion
- 进入时令该令牌转交
  - bunded waiting
- 满足三个条件
- 需要证明 参见P195－196

单选题 1分

设置



此题未设置答案，请点击右侧设置按钮

Algorithm 3- Peterson's Solution

A 不能保证进程互斥进入临界区，会出现“饥饿”现象

B 不能保证进程互斥进入临界区，不会出现“饥饿”现象

C 能保证进程互斥进入临界区，会出现“饥饿”现象

D 能保证进程互斥进入临界区，不会出现“饥饿”现象

提交

解题思路

- 1、说明同步机制应满足的三（四）个准则；
- 2、分析给出的算法，找出一个*特例*说明该同步机制违反了哪个或哪几个准则；
  - 一般是边界条件容易出现问题；
  - E.g. 2k year problem，财务系统中月末转账
  - 考虑多个进程（至少两个）几乎同时访问临界资源
- 如果认为正确，需要证明。

+Exp2 10

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- *Critical section problem* is to design a protocol to solve ensure,
  - When one process in critical section, no other may be in its critical section
- And the protocol should satisfy :
  - Mutual Exclusion, Progress, Bounded Waiting
- Solutions:
  - Peterson's Solution
  - **Synchronization Hardware**
  - Semaphore
  - Monitor

6.4 Synchronization Hardware

- The critical-section problem could be solved simply in a **uniprocessor** environment if we could prevent interrupts from occurring while a shared variable was being modified. （单处理机，关中断）
- **Disabling interrupts**
  - Uniprocessors – **could disable interrupts**
    - Currently running code would execute without preemption
  - Generally **too inefficient** on **multiprocessor** systems
    - Time consuming
    - Operating systems using this not broadly scalable

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Modern machines provide special atomic hardware instructions
  - **Atomic = non-interruptible**
  - Test memory word and set value
    - **TestAndSet Instruction**
  - Swap contents of two memory words
    - **Swap** Instruction
- ***All solutions based on idea of locking***
  - ***Protecting critical regions via locks***

Lock机制

- 讨论
  - UNIX or Linux中，有一个类似Windows中注册表的结构，记录已安装软件的相关信息。
  - 当系统安装或更新软件时，该表是要互斥访问的。
  - 问：
    4. 系统采取什么措施保证这类操作的原子性？ (互斥)

Lock机制

- 讨论
  - 如运行命令：sudo apt-get install xxxx，或

    sudo apt-get update
  - 系统会在目录 /var/lib/dpkg下**试图创建一个lock文件**，即/var/lib/dpkg/lock
    - 如果创建成功，则继续软件的安装或更新操作；
    - 如果创建失败，说明该lock文件已经存在，也说明有软件正在进行安装或更新操作，则本次安装或更新操作不能进行，需要等待
  - 当软件安装或更新操作完成后，系统会将该lock文件删除
  - 如果安装过程中非正常终止或掉电，该lock文件尚未被删除，则运行上述命令会提示：”无法获得锁/var/lib/dpkg/lock 资源暂时不可用...”等字样，操作无法继续
  - How to solve this problem?

Lock 机制

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

Figure 6.3 Solution to the critical-section problem using locks.

- acquire lock与release lock需要解决的问题
  - acquire lock：
    - ✔ 需要测试lock的当前状态 (test)
      - ➤ 如果lock当前处于**开锁状态**，**则关锁**，并访问临界区； (set)
      - ➤ 否则等待，直至锁被打开，然后关锁并继续执行；
  - release lock：开锁；

Lock 机制

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

Figure 6.3 Solution to the critical-section problem using locks.

- Lock机制是一个通用的临界区解决方案
- Lock可以有很多种实现方案
- **后面介绍的方法总体是基于锁机制，是一种锁机制的具体实现，体现test与set的思想**

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
 {
     boolean rv = *target;     //取锁的状态(test)
     *target = TRUE;           //加锁(set)
     return rv:                //返回锁原来的状态
 }
```

- 分别就 lock=false及true两种情况，讨论下述语句的执行结果
  - boolean **a=TestAndSe**t(lock));
  - while (**TestAndSe**t(lock));

Solution using TestAndSet

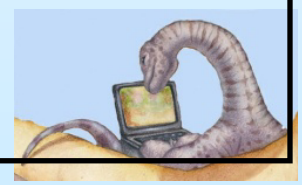
- Shared boolean variable **lock**, **initialized to false. (开锁状态).**

n Solution: process1

```
while (true) {
        while ( TestAndSet (&lock )) ;
        //返回 lock的值，并置 lock为true
        //如果原来已经上锁，自己也上锁，
        //并等待

        //lock is true(已经加锁)
        //critical section

          lock = FALSE;   //开锁

       // remainder section
}
```

n Solution: process2

```
while (true) {
        while ( TestAndSet (&lock )) ;
        //返回 lock的值，并置 lock为true
        //如果原来已经上锁，自己也上锁，
        //并等待

        //lock is true(已经加锁)
        // critical section

          lock = FALSE;   //开锁

        // remainder section
}
```



Swap Instruction

- Definition: //互换两个变量的值

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

- 分别就 lock=false及true两种情况，讨论下述语句的执行结果
  - key=true; **Swap (&lock, &key) ;** //key获取(返回)了lock的状态
  - **key = TRUE; while (key == TRUE) Swap (&lock, &key) ;**

Solution using Swap

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

- Solution: process1

```
while (true) {
    key = TRUE; //自己加锁
    while ( key == TRUE)
        Swap (&lock, &key ) ;
        // key 与lock的值互换，lock now is true,
        // if lock原值为false，then key now is false;
        //如果原来已经加锁，自己也加锁，并等待

        //    critical section

        //    lock is TRUE, key is FALSE

    lock = FALSE;
    //      remainder section
}
```

- Solution: process2

```
while (true) {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key ) ;

            // critical section

        // lock is TRUE, key is FALSE

    lock = FALSE;
    // remainder section
}
```

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- *Critical section problem* is to design a protocol to solve ensure,
  - When one process in critical section, no other may be in its critical section
- And the protocol should satisfy :
  - Mutual Exclusion, Progress, Bounded Waiting

- Solutions:
  - Peterson's Solution
  - Synchronization Hardware
  - *Semaphore*
  - Monitor

6.5 Semaphore

- A synchronization tool that does not require busy waiting
- Semaphore *S* – **integer variable**
  - *是一个受保护的量，对其只能进行*
    - ***初始化，即赋初值***
    - ***然后，对其wait()及signal()操作 (P、V操作)***
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()　(dijkstra提出信号量的概念)
    - P是荷兰语Proberen (test)
    - V是荷兰语Verhogen (increment)
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait(s) and signal(s)

Semaphore

- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

```
while S <= 0 ;  // no-op   //test
 S--;                      //set
}
```

  - signal (S) {

```
S++;    //unlock
}
```

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互斥锁**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

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

例：实现互斥

- Shared data
- **semaphore S;**
- Initially: **S = 1;**

例：实现互斥

P1:

```
do {
        ...
        wait(S);
        critical section;
        signal(S);
        reminder section;
} while (1);
```

P2:

```
do {
        ...
        wait(S);
        critical section;
        signal(S);
        reminder section;
} while (1);
```

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

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

例：实现互斥

- 假定开始时临界资源空闲，S=1
- 当P1欲使用该临界资源，首先执行wait(S)
  - while S <= 0 ； //条件S <= 0不满足
    S--;　//使S=0 (加锁)
  - 当P1在其临界区内执行时，S保持为0
- 若此时P2欲访问临界资源
  - while S <= 0 ；//条件满足，一直循环等待；直到P1退出其临界区
- 当P1执行完其临界区，退出时，执行signal (S);
  - S++　//使S=1 (开锁)
  - 之后若调度P2执行，退出while S <= 0 ，然后执行S--，重新使S=0 (加锁)
- 当P2退出临界区时，执行signal (S);
  - S++　//使S=1 (开锁)，使信号量S恢复到原来状态

Semaphore Implementation

- Must guarantee that **no two processes** can execute wait () and signal () on the same semaphore at the same time;
  - wait() and signal() share and modify semorphore S
  - so, semorphore S is a critical resource
  - ***wait() and signal() should be two atomic operations (primitive)***
- Thus, implementation becomes the critical section problem where the wait() and signal() code are placed in the critical 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.**
  - **busy waiting**

spinlock(自旋锁)

- **While a process is in its critical section, any other process that tries to entry its critical section must loop continuously;(busy waiting)**
  - This type of semaphore is also call "**spinlock(自旋锁)**";
  - 自旋锁的**缺点**是循环等待，占用cpu
  - **在单处理器系统中该问题尤为突出**

因为 wait() and signal() --primitive

    - 循环等待的这段时间其它不访问临界区的进程应该可以执行(但无法执行,why?);
    - 当一个进程等待一个事件，而该事件需要其它进程产生，而其它进程无法执行，也就无法产生该事件
  - 自旋锁的**优点**是进程在循环等待，不需要进行上下文切换，减少了系统开销;
  - 当等待锁的时间较短时，自旋锁是有效的;
  - **一般在多处理器系统中使用（更适合于多处理器）**
    - 一个线程在一个处理器上等待，而另一个线程可以在另一个处理器上访问临界区;
    - 当一个进程等待另一个进程产生的事件，另一个进程可以在其它处理机上执行从而产生该事件
- **后面将要介绍的非忙等信号量机制可解决自旋锁在单处理器中存在的问题;**

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

- Define a semaphore as a record (struct)

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

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(P)** resumes the execution of a blocked process **P**.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait：　（P操作，P原语）

```
wait (S)  {  //P(S)
        value--;
        if (value < 0) {
            add this process to waiting queue
            block();  }  //进程主动阻塞
    }
```

- Implementation of signal:　（V操作，V原语）

```
signal (S)  {   //V(S)
        value++;
         if (value <= 0) {
            remove a process P from the waiting queue
            wakeup(P);  } //进程被动唤醒
}
```

以进程的互斥为例简要说明信号量及两个操作的含义

- 假定系统中有1台打印机，三个进程共享；
- 假定系统中有2台打印机，三个进程共享；

- 观察以上情况下，信号量的值的变化，以及信号量等待队列中的进程的个数的变化情况；
- 总结出信号量以及wait、signal的含义；

三个进程共享1台打印机

- Shared data

  semaphore mutex;

  Initially:

  mutex = 1;

三个进程共享1台打印机

P1:

```
do {
        …
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

P2:

```
do {
        …
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

P3:

```
do {
        …
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

三个进程共享2台打印机

- Shared data

  semaphore mutex;

  Initially:

  mutex = 2;

三个进程共享2台打印机

P1:

```
do {
        …
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

P2:

```
do {
        …
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

P3:

```
do {
        …
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

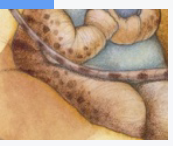
信号量及两个操作的物理含义

- S.value的初值(>=0)代表系统中某类资源的数目，因而信号量又称为*资源信号量*;
- 当S.value<0，其绝对值代表在信号量链表队列S.L的长度，即在该信号量下等待的进程数；

- 每次wait操作，意味着进程*申请*一个单位的资源，表示为S.value:=S.value-1;
- 每次signal操作，意味着进程*释放*一个单位的资源，表示为S.value:=S.value+1;

单选题 1分

设置



此题未设置答案，请点击右侧设置按钮

设与某资源相关联的信号量初始值为3，当前值为1，则目前该资源的可用个数与等待资源的进程数分别是（）。

A 0,1

B 1,0

C 1,2

D 2,0

提交

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. （另一种现象）

信号量的应用

- 实现互斥；
- 描述前趋关系（实质上是同步的一种表示方法）；
- 实现同步；

信号量的应用－互斥

- Shared data

  semaphore mutex;

  Initially:

  mutex = 1;

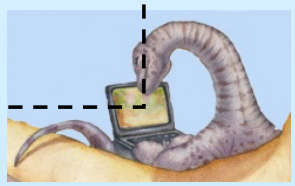
信号量的应用－互斥

P1:

```
do {
        ...
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```

P2:

```
do {
        ...
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
} while (1);
```



例题

现有两个并发执行的进程。

问：

它们能正确执行吗?

若不能，请说明原因，并改正之。

```
int x;

Process p1 {
    int y,z;
    x=1;
    y=0;
    if x>=1 then y++;
    z=y;  }

Process p2 {
    int t,u;
    x=0;
    t=0;
    if x<1 then t=t+2;
    u=t;  }
```

例题

现有两个并发执行的进程。

问：

它们能正确执行吗?

若不能，请说明原因，并改正之。

semaphore mutex=1;

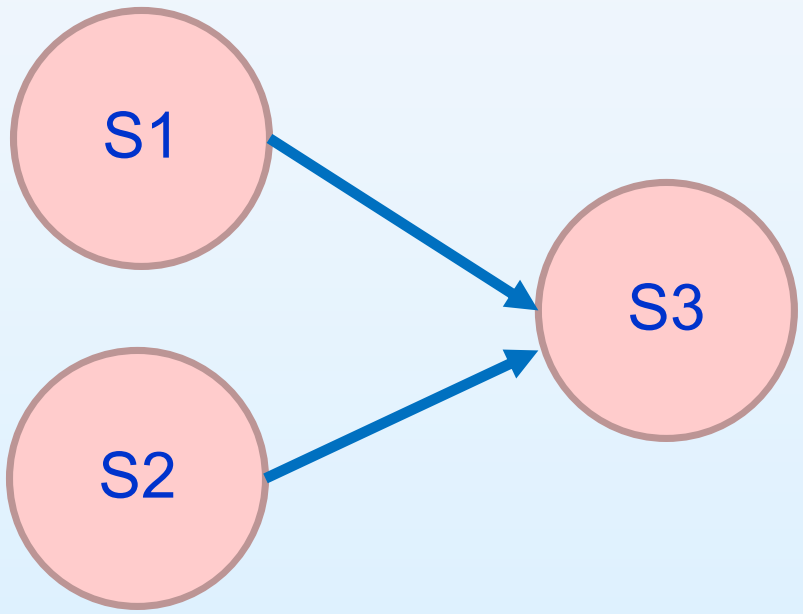
```
int x;
Process p1 {
    int y,z;
   wait(mutext);
    x=1;
    y=0;
    if x>=1 then y++;
   signal(mutex);
     z=y;  }

Process p2 {
    int t,u;
    wait(mutext);
    x=0;
    t=0;
    if x<1 then t=t+2;
   signal(mutex);
    u=t;  }
```

信号量的应用－前趋关系

- 问题描述
  - 三个进程P1、 P2、 P3中分别执行了下述语句：
    - s1: x=a+b;
    - s2: y=c+d;
    - s3: z=x+y;
- 问题：如何协调它们的执行?

前趋图

- 上述问题的前趋图
  - **结点**—进程、线程、语句
  - **有向边**—结点之间的依赖关系



Solution

- Shared data

  semaphore a, b;

  Initially:

  a,b = 0;

P1:

```
do {
        …;
     s1;
      signal(a);
       …;
} while (1);
```

P2:

```
do {
        …;
     s2;
      signal(b);
       …;
} while (1);
```

P3:

```
do {
        …
     wait(a);
     wait(b);
     s3;
     …;
} while (1);
```

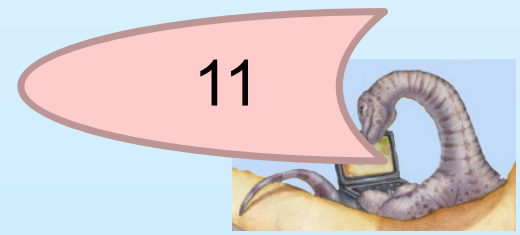
复杂的前趋图举例

- 问题描述
  - 复杂前趋图
  - 前趋图的变种
- 算法描述

该类问题的解题思路

- 将相关问题抽象成前趋图
  - 节点是进程所对应的需要协调的操作（语句）
  - 边是进程之间的依赖关系
- 对应每条边设置一个信号量
- 根据问题设定信号量的初值
- 算法描述
  - 对于某一节点
    - 如果有入边，需要在协调的操作前边加上一个wait()操作，其中的信号量与其前驱中signal()使用的相同；
      - wait()操作的个数与其入边的数目相同；
    - 如果有出边，需要在协调的操作后面加上一个signal()操作，其中的信号量与其后继中wait()使用的相同；
      - signal()操作的个数与其出边的数目相同；

+HW2



Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- Sleeping Barber Problem
- The Cigarette's Problem

Bounded-Buffer Problem
生产者－消费者问题（P-C）

两进程共享1个缓冲区

- 一个生产者进程，一个消费者进程，共享一个缓冲区
- 问题描述
  - 一个输入进程向一个缓冲区中输入数据，另一个输出进程从缓冲区中取出数据输出。
  - 缓冲区中每次只能存放一个数。
  - 假定开始时缓冲区为空；
- 这两个进程构成一对生产者与消费者；
- 它们之间的制约关系如下：

两进程共享1个缓冲区之间的制约关系

- 生产者（输入进程）
  - 如果缓冲区空，则放入数据；同时检查如果有消费者因缓冲区空未取到数据而进入阻塞状态，则唤醒之；
  - 若缓冲区非空，因无法放入数据，也可以说申请一个空缓冲区而未申请到，则进入阻塞状态；当消费者取走数据后，由消费者将其唤醒；

- 因为当消费者取走数据后，缓冲区变空，生产者可以放入数据，也就是说生产者等待的事件（等待缓冲区变空）已经发生，可以由阻塞状态转为就绪状态；

- 消费者（输出进程）
  - 如果缓冲区非空，则取出数据，同时检查如果有生产者因缓冲区非空无法放入数据而进入阻塞状态，则唤醒之；
  - 若缓冲区为空，因无法取出数据，也可以说申请一个非空缓冲区而未申请到，则进入阻塞状态；当生产者放入数据后，由生产者将其唤醒；
- 因为当生产者放入数据后，缓冲区有数据可供消费，消费者可以取走数据，也就是说消费者等待的事件（等待缓冲区非空）已经发生，可以由阻塞状态转为就绪状态；

两进程共享1个缓冲区解决方案算法描述

- 三要素
  - 信号量设置
  - 信号量赋初值
  - 算法描述

两进程共享1个缓冲区解决方案算法描述

- Shared data

semaphore empty, full; //分别表空缓冲区的个数
//与满缓冲区的个数；

Initially:

empty= 1，full＝0；//初始时，缓冲区时空的

两进程共享1个缓冲区同步算法描述

- 生产者（输入进程）结构

```
do {
        生成出一件产品；
            …
       wait(empty);
            …
        add data to buffer
            …
        signal(full);
} while (1);
```

n 消费者（输出进程）结构

```
do {

      wait(full);
            …
      remove data from buffer；
            …
      signal(empty);
            …
       消费取走的产品；
} while (1);
```

例：事件、制约

- 司机与售票员问题
  - 在公共汽车上，司机和售票员的工作流程如下所示。为保证乘客安全，司机和售票员应密切配合协调工作。请用wait、signal操作来实现司机与售票员之间的同步
  - 司机->（loop）{ 启动车辆->正常行车->到站停车 }
  - 售票员->（loop）{ 上乘客->关车门-> 售票-> 开车门->下乘客 }

P(start);

V(open);

V(start);

P(open);

例：事件、制约

- Semaphore start=0, open=0;

- 司机
- while(true)
  {
    P(start);
    启动车辆;
    正常行车;
    到站停车;
    V(open);
  }

- 售票员
- while(true)
  {
    上乘客;
    关车门;
    V(start);
    售票;
    P(open);
    开车门;
    下乘客;
  }

例：事件、制约

```
司机的活动：
P1：do{
                    —①；
          启动车辆；
          正常行车；
          到站停车；
                    —④
}while (1);
```

```
售票员的活动：
P2：do：{
                关车门；
                    —②
          售　票；
                    —③
          开车门；
}while (1);
```

例：事件、制约

semaphore start=0;
semaphore open=0;

司机的活动：

```
P1: do{
        P(start);
        启动车辆;
        正常行车;
        到站停车;
        V(open);
    }while (1);
```

售票员的活动：

```
P2: do:{
        关车门;
        V(start);
        售　票;
        P(open);
        开车门;
    }while (1);
```

Bounded-Buffer Problem
生产者－消费者问题（P-C）

- 一个生产者进程，一个消费者进程，共享N个缓冲区
- 问题描述
  - 缓冲池中的N个缓冲区，一个输入进程依次向缓冲池中的缓冲区输入数据，另一个输出进程依次从缓冲池的缓冲区中取出数据输出。
  - 每个缓冲区中每次只能存放一个数。
  - 假定开始时每个缓冲区为空；
  - 缓冲区编号从0~N-1
    - 设下标in跟踪生产者的送数过程，初始为0，每送入一个数据，in=(in+1)%N
    - 设下标out记录消费者的取数过程，初始为0，每取走一个数据，out=(out+1)%N
- 这两个进程构成一对生产者与消费者；

两进程共享N个缓冲区之间的制约关系

- 生产者（输入进程）
  - 如果缓冲池中尚有空缓冲区，则将该数据放入该空缓冲区中，in=(in+1)%N；同时检查如果有消费者因缓冲区空未取到数据而进入阻塞状态，则唤醒之；
  - 若缓冲池中已无空缓冲区，因无法放入数据，也可以说申请一个空缓冲区而未申请到，则进入阻塞状态；当消费者取走数据后，由消费者将其唤醒；
- 消费者（输出进程）
  - 如果缓冲池中尚有满缓冲区，则取出数据，out=(out+1)%N，同时检查如果有生产者因缓冲区非空无法放入数据而进入阻塞状态，则唤醒之；
  - 若缓冲池中已无满缓冲区，因消费者无法取出数据，也可以说申请一个满缓冲区而未申请到，则进入阻塞状态；当生产者放入数据后，由生产者将其唤醒；

两进程共享N个缓冲区解决方案算法描述

- Shared data

  int in, out;

  semaphore empty，full; //缓冲池中空、满缓冲区的个数

- Initially:

  empty= N，full＝0；

  in=0, out=0;

两进程共享N个缓冲区

- 生产者（输入进程）结构

```
do {
    生成出一件产品
        …
    wait(empty);
        …
    add data to buffer(in);
    in=(in+1)%N;
        …
    signal(full);
} while (1);
```

n 消费者（输出进程）结构

```
do {

    wait(full);
        …
    remove data from buffer(out);
    out=(out+1)%N;
        …
    signal(empty);
        …
    消费取走的产品;
} while (1);
```

Bounded-Buffer Problem

- 问题描述
  - 多个生产者，多个消费者，共享缓冲池中的N个缓冲区
- 生产者进程
  - 需要互斥访问共享变量in，需要设定一个互斥信号量，实现多个生产者之间互斥访问共享变量in；
- 消费者进程
  - 需要互斥访问out，需要设定一个互斥信号量，实现多个消费者之间互斥访问共享变量out；
- 制约关系

Bounded Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex1=1, mutex2=1, empty=N ,full=0;
- int in=0, out=0;

- The structure of the producer process

```
while (true)  {
   //   produce an item
   wait (empty);
   wait (mutex1);
  //add the item to the  buffer[in]
   in=(in+1)%N;
  signal (mutex1);
   signal (full);
 }
```

- The structure of the consumer process

```
while (true) {
   wait (full);
   wait (mutex2);
   //  remove an item from  buffer[out]
    out=(out+1)%N;
   signal (mutex2);
   signal (empty);
   //  consume the removed item
 }
```

- 为简化P-C模型的实现，将将缓冲池视为一个临界资源；

Bounded-Buffer Problem

- 为简化P-C模型的实现，将缓冲池视为一个临界资源；
- 生产者与消费者均互斥访问缓冲池；
- 生产者之间、消费者之间也互斥访问缓冲池；

- *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.
- int in=0, out=0 //index of the buffer for the produces and consumers

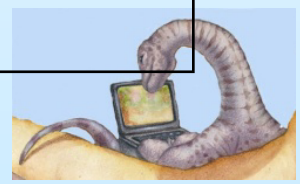
Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {
   //   produce an item
   wait (empty);
   wait (mutex);
   //  add the item to the  buffer
   //buffer[in]=item;
   //in=(in+1)%N;
   signal (mutex);
   signal (full);
}
```

- The structure of the consumer process

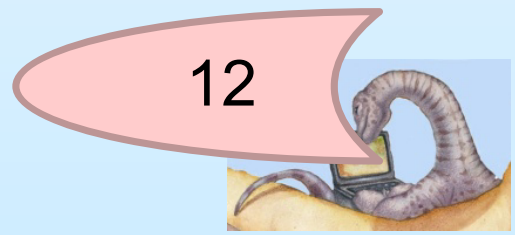
```
while (true) {
   wait (full);
   wait (mutex);
   //  remove an item from  buffer
   //item=buffer[in];
   //out=(out+1)%N;
   signal (mutex);
   signal (empty);
   //  consume the removed item
}
```



Bounded-Buffer Problem

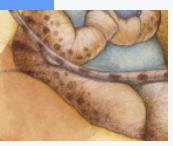
- 讨论
  - 对于同一个信号量的wait与signal操作必须成对出现（实际执行的次数必须一致）
    - 对于**互斥信号量**，成对出现在同一个程序中；
    - 对于**资源信号量（同步信号量）**，出现在不同的程序中；
  - 在每个程序中的多个wait操作，顺序是：
    - **同步在前，互斥在后**。Why? 否则，有死锁的可能
    - 即，执行资源信号量的wait操作在前，执行互斥信号量的wait操作在后
  - wait()与signal()操作必须是原语操作；（OS保证其原子性）

+Makefile
+习题



主观题 10分

设置



某博物馆最多可容纳500人同时参观，有一个出入口，一次仅允许一个人通过。参观者的活动描述如下：

```
cobegin
    参观者进程i:
    {
        进门;
        参观;
        出门;
    }
coend
```

请添加必要的信号量和P、V操作，以实现上述操作过程中的互斥与同步。

正常使用主观题需2.0以上版本雨课堂

作答

续上页

```
semaphore   empty=500;    //博物馆中能容纳的人数
semaphore   mutex=1;      //实现出入口的互斥
cobegin
    参观者进程i:
    {
        P(empty);
        P(mutxt);
        进门;
        V(mutxt);
        参观;
        P(mutxt);
        出门;
        V(mutxt);
        V(empty)
    }
coend
```

思考：

- “进门”之后的V(mutxt)与“出门”之前的P(mutxt)能不能去掉？Why?
- 去掉之后也能保证出入口一次仅允许一个人通过。

思考：

- 为什么不能去掉?
- 原因：并发粒度的问题
- 原则上，参观者进门与出门可以同时进行
- 去掉之后，进程不能并发，只能允许一个人参观
- 要保证进程并发粒度的最大化

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.

Readers-Writers Problem

- 数据库中的有关并发机制
  - 在DBMS中，
    - 允许同时进行多个查询操作；
    - 但不允许查询操作与更新操作同时进行；
    - 更新操作之间也必须互斥；
    - 原因：可能会导致数据的不一致性及不完整性等问题；
  - 对于数据库对象，允许多个读进程同时操作，但读进程与写进程必须互斥；写进程之间也必须互斥；
  - 把进行查询操作的读进程称为读者；把进行更新操作的写进程称为写者；

Readers-Writers Problem

- 有一个共享对象，允许多个读者同时访问，但必须与写者互斥；写者之间也必须互斥；
  - 读者之间不需互斥
  - 读者与写者之间互斥
  - 写者之间互斥
- 当一个读者欲访问该对象，若此时对象空闲，则可以访问；在访问期间，若有其它读者也要访问该对象，则允许访问；若有读者在访问对象期间，拒绝任何写者的访问；
- 若所有的读者均完成读操作，则释放该对象，使对象变为空闲，允许读者、写者（再次）访问；
- 当对象空闲时，写者可以访问该对象，在写操作完成之前，不允许读者及其它的写者同时访问；

Readers-Writers Problem—分析

- 对于写者进程，实现比较简单－互斥；

- 对于读者，其中的两个*特殊的读者*比较重要：*第一个进入的读者*与*最后一个离开的读者*；（e.g. 教室的使用，第一个进入教室的同学与最后一个离开的同学）；

- 第一个进入的读者，应该拒绝写者，但不能拒绝其它的读者；

- 最后一个离开的读者，应该释放对象；如果需要的话，还要唤醒等待的写者；

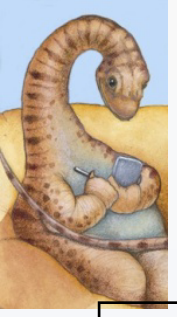
Readers-Writers Problem算法描述

- 读者计数器－对象内读者的个数，主要是指明两个特殊的读者；
- 计数器被读者共享，因此计数器是一个临界资源，需要互斥访问；
- 信号量的设置；
- 信号量的初值；
- 算法描述；

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 wrt initialized to 1.
  - Integer readcount initialized to 0.



Readers-Writers Problem (分析)

n The structure of a writer process

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

n The structure of a reader process

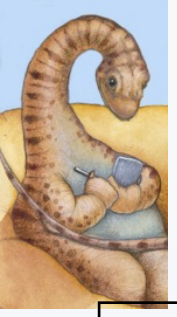
```
while (true) {
     readcount ++ ;
     if (readcount == 1)  wait (wrt) ;
          // reading is performed
     readcount  - - ;
     if (readcount  == 0)  signal (wrt) ;
 }
```

n 对于写者
n 对于读者
  - 第一个
  - 最后一

1、计数器readcount需要互斥；
2、如果目前有写者正在访问，第一个到来的读者会进入等待。但后续的读者仍然可以访问对象。

设置一个互斥信号量可以解决上述两个问题。

n **思考：该实现存在什么问题?**



Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

        // writing is performed

        signal (wrt) ;
}
```

- The structure of a reader process

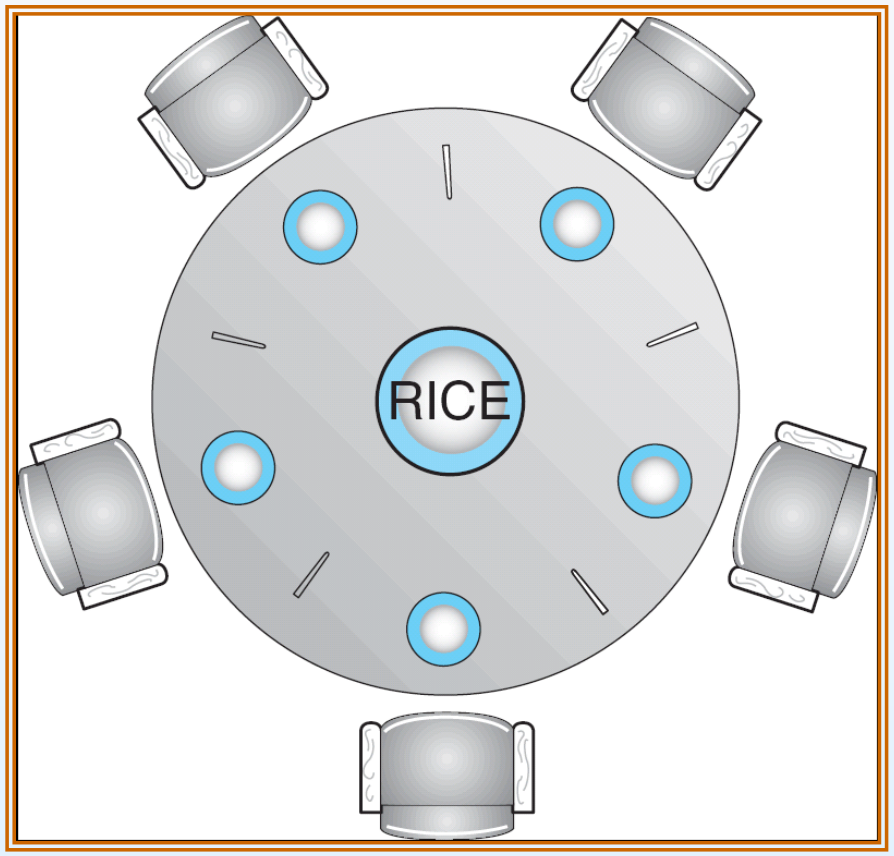
```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)  wait (wrt) ;
    signal (mutex)

        // reading is performed

    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)  signal (wrt) ;
    signal (mutex) ;
}
```

1、若有写者，等待；
2、若无写者，封锁写者；

Readers-Writers Problem (Cont.)

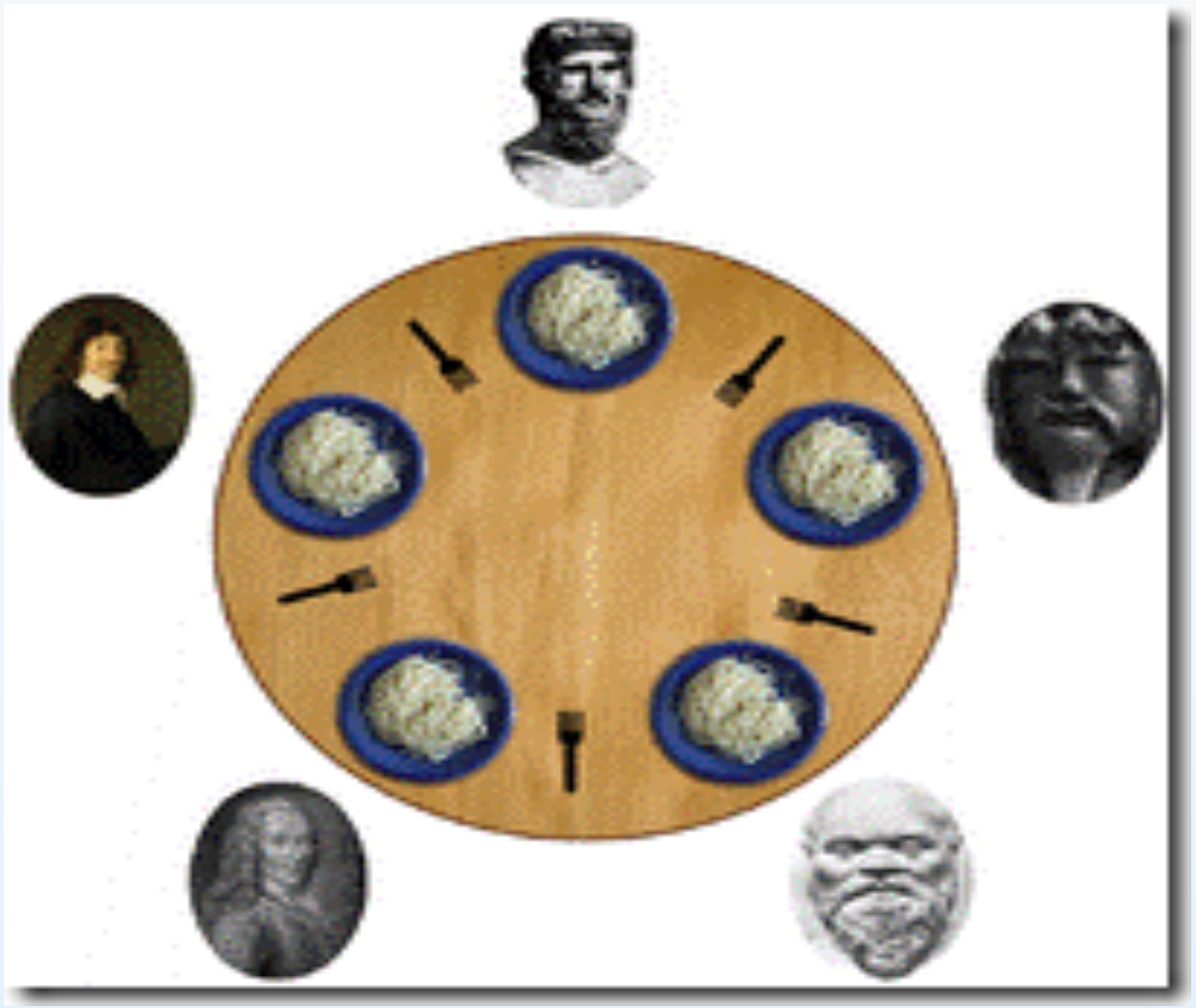
- 讨论：
  - 该方案中，若有写者访问对象，读者在哪些信号量的等待队列中?
  - 该方案存在的问题
    - Problem: 该方案读者优先，可能导致写者出现“饥饿”现象；
    - Solution：当有写者到来时，该写者阻止后续的读者访问该对象；（写者优先，后面课件中，添加一个信号量就能实现）
- Tips：
  - 读者优先
    - 若有读者R与写者W同时到达
    - 若此时有其它读者在访问对象，则读者R和后续的读者会优于读者W访问对象
  - 写者优先
    - 若有读者R与写者W同时到达
    - 写者W阻止读者R及后续的读者访问对象，等正在访问对象的所有读者都离开，写者进入

Dining-Philosophers Problem



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

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

}
```

Problems and Solution

- Problem: probably create a deadlock
- Solutions:
  - Allow at most four philosopher to be sitting simultaneously at the table;
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available;
  - Use a asymmetric solution; an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick;
  - The last philosopher pick up her right chopstick first and the her left; the others pick up her left chopstick first and the her right;

Dining-Philosophers Problem-1

- Allow at most four philosopher to be sitting simultaneously at the table;
- Semaphore seat＝4, chopstick[i]=1 (i=0..4);
- Philosopher *i*:

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

Dining-Philosophers Problem-2

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available;
  - 信号量集（请参阅汤子赢教材）
  - 管程（monitor）（后面讲介绍）

Dining-Philosophers Problem-3

- Semaphore chopstick[i]=1 (i=0..4);
- Philosopher *i*:

```
do {
        if (i%2==0) {
            wait(chopstick[i])
            wait(chopstick[(i+1) % 5])  }
         else {
            wait(chopstick[(i+1) % 5])
            wait(chopstick[i])    }
            …
            //eat
            …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
            …
            //think
            …
} while (1);
```

- Use a asymmetric solution;
- An odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick;

Dining-Philosophers Problem-4

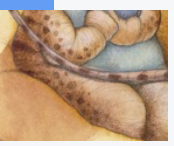
- Semaphore chopstick[i]=1 (i=0..4);
- Philosopher *i*:

```
do {
        if (i<4) {
            wait(chopstick[i])
            wait(chopstick[(i+1) % 5])  }
         else {
             wait(chopstick[(i+1) % 5])
             wait(chopstick[i])    }
            ...
          //eat
            ...
       signal(chopstick[i]);
       signal(chopstick[(i+1) % 5]);
            ...
         //think
            ...
} while (1);
```

The last philosopher pick up her right chopstick first and the her left; the others pick up her left chopstick first and the her right;

主观题 10分

设置



某银行提供1个访问窗口和10个顾客等待座位。顾客到达银行时，若有空座位，则到取号机领取一个号，坐在座位上等待叫号。取号机每次仅允许一个顾客使用。当营业员空闲时，通过叫号选取一位顾客，并为其服务。顾客和营业员的活动过程描述如下：

```
process 顾客i
{
        从取号机获取一个号码;
        等待叫号;
        获得服务;
}
```

```
process 营业员
{
        while(true)
        {
            叫号;
            为顾客服务;
        }
}
```

请添加必要的信号量和P、V（signal、wait）操作实现上述过程的互斥和同步。
要求写出完成的过程，说明信号量的含义并赋初值。

正常使用主观题需2.0以上版本雨课堂

作答

续上页—分析受到制约的事件

银行提供1个访问窗口和10个顾客等待座位。顾客到达银行时，若有空座位，则到取号机领取一个号，坐在座位上等待叫号。取号机每次仅允许一个顾客使用。当营业员空闲时，通过叫号选取一位顾客，并为其服务。顾客和营业员的活动过程描述如下：

```
process 顾客i
{
    从取号机获取一个号码;

    等待叫号;

    获得服务;
}
```

```
process 营业员
{
    while(true)
    {
        叫号;

        为顾客服务;
    }
}
```

有空座位顾客才能取号

取号机需要互斥

顾客等待叫号，若营业员睡眠，唤醒之

有顾客等待营业员才能叫号

释放座位；若有顾客等待，唤醒之

续上页—解决方案

semaphore mutex=1; //互斥使用取号机

semaphore seats=10; //空座位数

semaphore customer=0; //等待的顾客数

semaphore customer=0; //等待叫号

```
process 顾客i
{
        P(seats);
        P(mutex);
        从取号机获取一个号码;
        V(mutex);
        V(customer);
        P(sevice)
        等待叫号;
        获得服务;
}
```

```
process 营业员
{
        while(true)
        {
            P(customer);
            V(seats);
            V(sevice);
            叫号;
            为顾客服务;
        }
}
```

Sleeping Barber Problem

- 问题描述
- 参见P233 题6.11
- 某理发店有一个接待室和一个理发室组成。理发室中有一把理发椅，接待室中有n把椅子。若没有顾客等待理发，则理发师睡眠等待。当一个顾客到达理发店后，若发现座位已满，则**选择离开**；若发现理发师忙而接待室中有空座位，顾客则坐在椅子上等待；若发现理发师正在睡眠，则将理发师唤醒。试写一个程序协调理发师和顾客之间的活动。
- **课后练习：**如果将“**选择离开**”改为“**等待**”，如何解决该问题?

Sleeping Barber Problem

- 类比：诊所中医生与顾客的协调问题。
- **顾客**：
  - 到达诊所后，若没有空座位，则离开；否则，提交病例，然后等待医生呼叫；
    - 提交病例的目的是告诉医生，有顾客等待；
- **医生**：
  - 如果发现还有病例，则呼叫下一个顾客；否则等待。

Sleeping Barber Problem

- 问题分析

- Customer
  - if (接待室有空座位)
    - 进入接待室
    - 顾客数+1
      - 若理发师睡眠，唤醒之
    - 等待理发师呼叫，如果理发师正忙，则睡眠等待
  - else
    - 离开
  - 重复上述步骤

- Barber
  - 检查接待室是否有顾客
    - 若无，则睡眠等待顾客，来顾客后被唤醒
  - 呼叫一个顾客
  - 顾客数-1
  - 为顾客理发
  - 重复上述步骤

Sleeping Barber Problem

常量：CHAIRS //椅子的个数

变量：int waiting=0; //记录等待服务的顾客数

信号量：

customers=0;　　//等待的顾客数

barberReady=0;　//理发师是否可以对顾客提供服务

waitingMutex=1;　//互斥信号量（实现waiting 的互斥访问 ）

给出两个信号量的等待队列中的进程，讨论理发师与顾客之间的协作关系。

Sleeping Barber Problem（分析）

Customer:

```
While (true) {
    if (waiting < CHAIRS)  {
        waiting=waiting +1;
        signal(customers);
        wait(barberReady);   2
    }
    else  {
        leaving;
    }
}
```

Barber:

```
while (true)  {
    wait(customers);   1
    waiting=waiting-1;
    signal(barberReady);
    cut-hair;
}
```

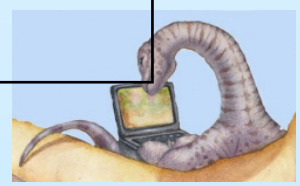
Sleeping Barber Problem(算法)

Customer:

```
While (1) {
      wait(waitingMutex)
      if (waiting < CHAIRS)  {
          waiting=waiting +1;
          signal(waitingMutex);
          signal(customers);
          wait(barberReady);
      }
      else   {
          signal(waitingMutex);
          leaving;
      }
}
```

Barber:

```
 while (true)  {
      wait(customers);
      wait(waitingMutex);
      waiting=waiting-1;
      signal(waitingMutex);

      signal(barberReady);
      cut-hair;
}
```



Sleeping Barber Problem

Customer:

```
While (1) {
    wait(waitingMutex);          //实现对waiting的互斥访问
    if (waiting < CHAIRS)  {     //如果有座位空闲
      waiting=waiting +1;        //
      signal(waitingMutex);
      signal(customers);         //通知理发师（相当于在诊所中交上病例）
      wait(barberReady);         //等待理发师呼叫（相当于等待医生）
    }
    else  {                      // 理发店已满，离开
      signal(waitingMutex);
      leaving;
    }
}
```

Sleeping Barber Problem

```
Barber:
  while (true) {
     wait(customers);  //检查有无顾客（医生查检查是否还有无病例）
                       //如果没有顾客，睡眠（等待顾客）
     wait(waitingMutex);        // 实现对waiting的互斥访问）
     waiting=waiting-1;
     signal(waitingMutex);      //释放waiting的访问权

     signal(barberReady); //理发师准备好可以服务（呼叫顾客）  （呼叫病人）
     cut-hair;            //理发（看病）
  }
```

The Cigarette's Problem-1

- 简单吸烟者问题：
  - 有三个抽烟者坐在桌子边，每个抽烟者不断地卷烟并抽烟。
  - 抽烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和火柴。
  - 三个吸烟者中，每个吸烟者有两种材料，各自缺少烟草、纸和火柴。
  - 有一个供应者，无限地供应所有三种材料中的一种，但每次仅提供三种材料中的一种。
    - 供应者将提供的一种材料放到桌子上
  - 得到缺失一种材料的抽烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供一种材料。
    - 在材料被相应的吸烟者取走之前，不允许供应者供应新的材料。
  - 这一过程重复进行。

The Cigarette's Problem-1

- 分析
  - 吸烟者之间没有关系
  - 每个吸烟者和供应者之间都是一种单个生产者-单个消费者-单个缓冲区的同步关系

```
semaphore P=1,A=0,B=0,C=0;
```

```
供应者:
do {
    wait(P);
    取材料x;
    将材料x放到桌面;
    if (x==A)
        signal(A);
    if (x==B)
        signal(B);
    if (x==C)
        signal(C);
} while (true)
```

```
抽烟者A
do {
    wait(A);
    取材料;
    signal(P);
    制烟;
    吸烟;
} while(true)
```

```
抽烟者B
do {
    wait(B);
    取材料;
    signal(P);
    制烟;
    吸烟;
} while(true)
```

```
抽烟者C
do {
    wait(C);
    取材料;
    signal(P);
    制烟;
    吸烟;
} while(true)
```

The Cigarette's Problem-2

- 简单吸烟者问题：
  - 有三个抽烟者坐在桌子边，每个抽烟者不断地卷烟并抽烟。
  - 抽烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和火柴。
  - 三个吸烟者中，每个吸烟者有两种材料，各自缺少烟草、纸和火柴。
  - 有两个供应者，无限地供应所有三种材料中的一种，但每次仅提供三种材料中的一种。
    - 供应者将提供的一种材料放到桌子上
  - 得到缺失一种材料的抽烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供一种材料。
    - 在材料被相应的吸烟者取走之前，不允许供应者供应新的材料。
  - 这一过程重复进行。

The Cigarette's Problem-2

- 分析
  - 吸烟者之间没有关系
  - 两个供应者之间要互斥

```
semaphore P=1,M=1,A=0,B=0,C=0;
```

```
供应者i:
do {
    wait(P);
    取材料x;
    wait(M);
    将材料x放到桌面;
    signal(M)
    if (x==A)
        signal(A);
    if (x==B)
        signal(B);
    if (x==C)
        signal(C);
} while (true)
```

```
抽烟者A
do {
    wait(A);
    取材料;
    signal(P);
    制烟;
    吸烟;
} while(true)
```

```
抽烟者B
do {
    wait(B);
    取材料;
    signal(P);
    制烟;
    吸烟;
} while(true)
```

```
抽烟者C
do {
    wait(C);
    取材料;
    signal(P);
    制烟;
    吸烟;
} while(true)
```

The Cigarette's Problem—原始描述

- The scenario for the cigarette smokers problem consists of four threads:
  - Three smokers and one agent.
- In order to smoke, the smoker needs to acquire three items: tobacco, paper, and a match.Once the smoker has all three, they combine the paper and tobacco to roll a cigarette and use the match to light it.
- Each of the three smokers has an infinite supply of exactly one item and needs the other two.
- The agent supplies two of three items one time.
- The below Code shows a sample outline for the smoker threads and one agent.
  - The smoker shown there is assumed to have an infinite supply of tobacco but needs the match and paper.
  - The smoker sends a signal to request more paper and matches when she is finished smoking.
- The other two threads are similar, but one would wait on semaphores for a match and tobacco and the third would wait on semaphores for tobacco and paper.

The Cigarette's Problem-原始描述

```
semaphore more_needed =1, match =0, paper =0, tobacco =0;
```

```
void * agent (void)
{
  while (true)   {
      int number = rand() % 3;
      switch (number)   {
          case 0:  signal(match); /*match and paper*/
                   signal(paper);
                   break;
          case 1: signal(match); /*match and tobacco*/
                   signal(tobacco);
                   break;
          case 2: signal(paper); /*tobacco and paper*/
                  signal(tobacco_sem);
                  break;
       }
      wait (more_needed); /*wait for request for more */
   }
}
```

The Cigarette's Problem-原始描述

```
semaphore more_needed =1, match =0, paper =0, tobacco =0;
```

```
void * smoker_with_tobacco (void) {
 while (true)   {
      wait (match); /*grab match from table*/
      wait (paper); /*grab paper from table*/
       /* roll cigarette and smoke */
      signal(more_needed); /*signal to agent*/
    }
}
```

```
void * smoker_with_paper (void) {
 while (true)   {
      wait (match); /*grab match from table*/
      wait (tobacco); /*grab tobacco from table*/
       /* roll cigarette and smoke */
      signal(more_needed); /*signal to agent*/
    }
}
```

```
void * smoker_with_match (void) {
 while (true)   {
    wait (tobacco); /*grab tobacco from table*/
    wait (paper); /*grab paper from table*/
     /* roll cigarette and smoke */
    signal(more_needed); /*signal to agent*/
    }
}
```

思考：
该算法描述可能存在什么问题?

可能导致死锁。

如何解决?

The Cigarette's Problem-3

- 抽烟者问题。(无死锁解决方案)
- 有三个抽烟者，每个抽烟者不断地卷烟并抽烟。
- 抽烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和火柴。
- 三个吸烟者中，一个抽烟者有烟草，一个有纸，另一个有火柴。
- 有一个供应者，无限地供应三种材料，但每次仅提供三种材料中的两种。
- 得到缺失的两种材料的抽烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供三种材料中的两种材料。
  - 在两种材料被相应的吸烟者取走之前，不允许供应者供应新的材料。
- 这一过程重复进行。

The Cigarette's Problem-3

semaphore tobacco_paper = 0 // waiting for tobacco and paper
semaphore tobacco_matches = 0 // waiting for tobacco and matches
semaphore paper_matches = 0 // waiting for paper and matches
semaphore doneSmoking = 1

```
agent:
      while( true ) {
            wait( doneSmoking );  //等待吸烟者取走材料
            pick a random number from 1-3;
            if random number is 1   {
        // put tobacco and paper on table
        signal( tobacco_paper ) }
            else if random number is 2  {
                 // put tobacco and matches on table
                  signal( tobacco_matches )
            else if random number is 3 {
                // put paper and matches on table
                signal( paper_matches ) }
}   /* while */
```

The Cigarette's Problem-3

```
smokers:
    // the smoker that has matches
    while( true )  {
        wait( tobacco_paper );  /* picks up tobacco and paper */
        // roll cigarette and smoke
        signal( doneSmoking );  }
    // the smoker that has paper
    while( true ) {
        wait(tobacco_matches ) ; /* picks up tobacco and match */
        // roll cigarette and smoke
        signal( doneSmoking );  }
// the smoker that has tobacco
 while( true ) {
        wait(paper_matches ) ; /* picks up paper and match */
        // roll cigarette and smoke
        signal( doneSmoking );   }
```

信号量进一步讨论

- **Why should wait and signal be primitive?**

Why should wait and signal be primitive?

- wait()
  - 分析两个进程共享一个临界资源，当s＝1时，阻塞了不应阻塞的进程，违反了Progress；(分析两个进程同时访问wait)
- signal()
  - 分析三个进程共享两个资源，当s＝-1时，应该唤醒的进程而没有被唤醒，违反了Bounded waiting；(分析三个进程同时访问signal())

注意wait操作的顺序

- 在生产者－消费者问题中，交换两个wait操作的次序会出现什么结果？交换两个signal操作呢？说明理由。

注意wait操作的顺序（续）

- 如果交换生产者进程中的两个wait操作
  - 考虑当缓冲池为满的情况
- 如果交换消费者进程中的两个wait操作
  - 考虑当缓冲为空的情况

- 可能导致死锁

- 交换signal的顺序，不会出现问题；

两个缓冲池

- 设有三个进程A、B、C，其中
  - A与B构成一对P－C问题，共享一个由n个缓冲区组成的缓冲池；
  - B与C构成一对P－C问题，共享一个由m个缓冲区组成的缓冲池；
  - 进程B从第一个缓冲区中取出A送入的数据,接着送入第二个缓冲区供C消费
- 试用记录型信号量机制及wait与signal操作实现它们的同步。

生产者-消费者问题实例化

- 某媒体播放器由一组循环使用的缓冲区及两个并发的播放进程与接收进程组成，其中，
  - (1) 8个缓冲区构成一个循环链表，用于缓存要播放的媒体流；
  - (2) 接收进程负责从服务器端接收欲播放的媒体流，并依次放入缓冲区中；
  - (3) 播放进程依次从缓冲区中取出媒体流播放；

请利用信号量机制和wait、signal操作解决这两个进程的同步问题，写出相应的算法描述；

三进程共享一个缓冲区

- 三个进程共享一个缓冲区。
  - 一个计算进程送数；
  - 一个加工进程取出加工，然后将加工结果再送回缓冲区；加工结果送回缓冲区之前，不允许计算进程送数；
  - 一个输出进程将加工后的数据取出打印。缓冲区中每次只能存放一个数。

奇偶数问题--三进程共享一个缓冲区

- 三个进程共享一个缓冲区。
  - 一个负责向缓冲区送数；
  - 一个取偶数输出
  - 另一个取奇数输出。
- 缓冲区中每次只能存放一个数。

奇偶数问题—四进程共享一个缓冲区

- 四个进程共享一个缓冲区
  - 一个送偶数
  - 一个送奇数
  - 一个取偶数
  - 一个取奇数
- 缓冲区中每次只能存放一个数。假定开始时缓冲区是空的。

奇偶数问题—四进程共享N个缓冲区

- 四个进程共享一个由N个缓冲区构成的缓冲池
  - 一个送偶数
  - 一个送奇数
  - 一个取偶数
  - 一个取奇数
- 缓冲区中能存放N个数。
- 对于取奇偶数进程的控制顺序较为复杂

幸福家庭问题

- 幸福家庭问题
  - 桌子上有一只盘子，每次只能放入一个水果。
    - 爸爸向盘中放苹果
    - 妈妈向盘中放桔子
    - 女儿吃盘中的苹果
    - 儿子吃盘中的桔子。
  - 试用P、V操作写出他们能同步的程序。

前趋图

- 要求下列四条语句正确执行

  s1：a:=x+y;

  s2：b:=z+1;

  s3：c:=a-b;

  s4：w:=c+1;

围棋拣子问题

- 围棋拣子问题：
  - 数量相等的黑子与白子混在一起，利用两个进程分开。一个进程拣白子，另一个进程拣黑子。
  - 要求：

（1）两个进程不能同时拣子；

（2）一个进程拣了一个子，必须让另一个进程拣子；即两个进程应交替拣子；

（3）假定先拣黑子。

围棋拣子问题—参考答案

- 如果black=1，white=0，

- 捡黑子进程

```
While (true)
{
    wait(black);
    捡一个黑子；
    signal(white);
}
```

- 捡白子进程

```
While (true)
{
    wait(white);
    捡一个白子；
    signal(black);
}
```

两产品入库问题

- 有一个仓库，可以存放A与B两种产品，仓库的存储空间足够大，但要求：

  （1）每次只能存入一种产品（A或B）；

  （2）-N<A产品数量 – B产品数量<M；

  其中，M和N是正整数。

试用“存放A”和“存放B”和wait、signal描述产品A与产品B的入库过程。

两产品入库问题（Cont.）

- 分析：表达式“-N<A产品数量 – B产品数量<M”可以分解成两个表达式：
  - “A产品数量 – B产品数量<=M-1”
  - “B产品数量 – A产品数量<=N-1”

- 即“存放A”的操作次数比“存放B”的次数不能超过M-1次；
- “存放B”的操作次数比“存放A”的次数不能超过N-1次

- 将产品的数量之间的关系转换为进程执行次数的关系

回顾：围棋拣子问题

- 如果black=1，white=1，思考两进程之间执行次数的制约?

如果先捡白子1次，则黑子可捡2次，比捡白子可多1次；

- 捡黑子进程

```
While (true)
{
    wait(black);
     捡一个黑子;
    signal(white);
}
```

- 捡白子进程

```
While (true)
{
    wait(white);
     捡一个白子;
    signal(black);
}
```

如果先捡黑子1次，则白子可捡2次，比捡黑子可多1次

- 捡黑子的次数比捡白子的次数最多多1次；　捡黑子的次数－捡白子的次数<=1
- 捡白子的次数比捡黑子的次数最多多1次；　捡白子的次数－捡黑子的次数<=1

回顾：围棋拣子问题

- 如果black=M，white=N，思考两进程之间执行次数的制约?

如果先捡白子N次，则黑子可捡M+N次，比捡白子可多(M+N)-N=M次

- 捡黑子进程

```
While (true)
{
    wait(black);
     捡一个黑子;
    signal(white);
}
```

- 捡白子进程

```
While (true)
{
    wait(white);
     捡一个白子;
    signal(black);
}
```

如果先捡黑子M次，则白子可捡N+M次，比捡黑子可多(N+M)-M=N次

- 捡黑子的次数比捡白子的次数最多多M次；　捡黑子的次数–捡白子的次数<=M
- 捡白子的次数比捡黑子的次数最多多N次；　捡白子的次数–捡黑子的次数<=N

两产品入库问题（Cont.）

- semorphore
  - mutex=1　//每次只能存入一种产品
  - sa=M-1
  - sb=N-1;

两产品入库问题（Cont.）

```
Process Input _A：
        while (true) {
              Get a product A;
               wait(sa);
               wait(mutext);
               // put product A into the
               // depository;
               signal(mutext);
               signal(sb); }
            }
```

```
Process Input _B :
        while (1) {
              Get a product B;
                wait(sb);
                wait(mutext);
                // put product B into the
                 // depository;
               signal(mutext);
               signal(sa);
            }
```

题型讲解

- 有一个仓库存放两种零件A和B，最大库容各为m个。
- 有一个车间不断地取A和B进行装配，每次各取一个。
- 为避免零件锈蚀，遵循先入库者先出库的原则。
- 有两组供应商分别不断地供应A和B（每次一个）。
- 为保证齐套和合理库存，当某种零件的数量比另一种的数量超过n（n<m）个时，暂停对数量大的零件的进货，集中补充数量少的零件。
- 试用wait和signal正确实现之。

- 该题的控制关系有4个：
  - A的数量 <= m
  - B的数量 <= m
  - A的数量 – B的数量 <= n
  - B的数量 – A的数量 <= n

- 将产品的数量之间的关系转换为进程执行次数的关系

- 由三个程序组成：
  - 放A
  - 放B
  - 装配：取A与B

- semaphore

  mutex=1; //实现互斥

  Availa=m; //A产品的最大库容

  Fulla=0; //放入A产品的个数

  Sa=n; //放A产品这个操作的次数

  Availb=m; //B产品的最大库容

  Fullb=0; //放入B产品的个数

  Sb=n; //放B产品这个操作的次数

```
process Input_A
        while (true) {
            wait(availa);          //是否超过库容
            wait(sa);              //协调放A与放B的执行次数
            wait(mutext);
            put product A into the depository;
            signal(mutext);
            signal(sb);            //协调放A与放B的执行次数
            signal(fulla);         //协调取产品进程
        }
```

```
Process Input_B {
        while (1) {
            wait(availb);
            wait(sb);
            wait(mutex);
            put product B into the depository;
            signal(mutex);
            signal(sa);
            signal(fullb);
        }
```

```
Process get_product_A_B {
        while (true) {
            wait(fulla);
            wait(fullb);
            wait(mutex);
            get product A and B from the depository;（遵循FIFO原则）
            signal(mutex);
            signal(availa);
            signal(availb);
        }
```

题型讲解

- 有一个活动场地最多可容纳22名同学参与活动。其中一部分同学参与打篮球活动（不妨设为活动A），另一部分同学参与羽毛球活动（不妨设为活动B）。规定如下：
- 若活动场地中同学人数已经超过22人，则申请进入活动场地的同学等待；
- 参与A、B两类活动的同学人数之差不能超过5人；即若参与活动A的同学人数比参加活动B的同学人数多5人，则申请参与活动A的同学等待；同样，若参与活动B的同学人数比参加活动A的同学人数超多5人，则申请参与活动B的同学等待；
- 请用信号量机制及P、V操作描述同学们参加活动A与活动B的过程。

题型讲解

Semaphore enter=22，basketball=5, badminton=5;

活动A：

```
while(true) {
        P(enter)
        P(basketball);
        参与A（加入篮球组）；
        V(badminton);
        离开A（离开篮球组）；
        V(enter);
  }
```

活动B：

```
while(true) {
        P(enter)
        P(badminton);
        参与B（加入羽毛球组）；
        V(basketball);
        离开B（离开羽毛球组）；
        V(enter);
}
```

通过邮箱进行辩论（研统考2014）

- 有A、B两人通过信箱进行辩论，每个人都从自己的信箱中取得对方的问题。
- 然后将答案和向对方提出的新问题组成一个邮件放入对方的信箱中。
- 假设A的信箱最多放M个邮件，B的信箱最多放N个邮件。
- 初始时A的信箱中有x个邮件（0<x<M），B的信箱中有y个邮件（0<y<M）。
- 辩论者每取出一个邮件，邮件数减1。
- A和B两人的操作过程描述如下：

通过邮箱进行辩论（续）

cobegin

```
A{
  while(true) {
     从A的信箱中取出一个邮件;
     回答问题并提出一个新问题;
     新邮件放入B的信箱;
   }
}
```

```
B{
  while(true) {
     从B的信箱中取出一个邮件;
     回答问题并提出一个新问题;
     新邮件放入A的信箱;
   }
}
```

coend

通过邮箱进行辩论（续）

- 当信箱不为空时，辩论者才能从信箱中取邮件，否则等待。当信箱不满时，辩论者才能将新邮件放入信箱，否则等待。
- 请添加必要的信号量和P、V（或wait、signal）操作，以实现上述过程的同步。
- 要求写出完成过程，并说明信号量的含义和初值。

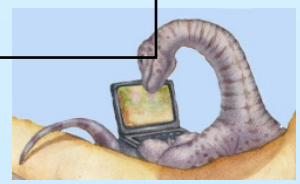
通过邮箱进行辩论（参考答案）

CoBegin

```
A{
  while(true) {
     P(Full_A);
     P(Mutex_A);
     从A的信箱中取出一个邮件；
     V(Mutex_A);
     V(Empty_A);
     回答问题并提出一个新问题；
      P(Empty_B);
      P(Mutex_B);
     新邮件放入B的信箱；
     V(Mutex_B);
     V(Full_B);
   }
}
```

```
B{
  while(true) {
     P(Full_B);
     P(Mutex_B);
     从B的信箱中取出一个邮件；
     V(Mutex_B);
     V(Empty_B);
     回答问题并提出一个新问题；
      P(Empty_A);
      P(Mutex_A);
     新邮件放入A的信箱；
     V(Mutex_A);
     V(Full_A);
   }
}
```

CoEnd



通过邮箱进行辩论（参考答案）

- Semaphore Full_A=x;　　//A的信箱中邮件数量
- Semaphore Empty_A=M-x;　//A的信箱中还可存放的邮件数量
- Semaphore Full_B=y;　　//B的信箱中邮件数量
- Semaphore Empty_B=N-y;　//B的信箱中还可存放的邮件数量
- Semaphore Mutex_A=1;　　//用于对A的信箱互斥访问
- Semaphore Mutex_B=1;　　//用于对B的信箱互斥访问

保证并发度的情况下实现临界资源互斥

- 某进程中有3个并发执行的线程thread1、thread2和thread3，其伪代码如下所示。（2017）

```
typedef struct {
  float a;
  float b;
}  cnum;
cnum x,y,z; //全局变量

//计算两复数之和
cnum add(cnum p, cnum q) {
   cnum s;
   s.a=p.a+q.a;
   s.b=p.b+q.b;
   return s;
}
```

保证并发度的情况下实现临界资源互斥

```
thread1 {
   cnum w;
   w=add(x,y);
   .......
}
```

```
thread2 {
   cnum w;
   w=add(y,z);
   .......
}
```

```
thread3 {
   cnum w;
   w.a=1;
   w.b=1;
   z=add(z,w);
   y=add(y,w);
   .......
}
```

- 请添加必要的信号量和P、V操作，要求确保线程互斥访问临界资源，并且最大程度地并发执行。

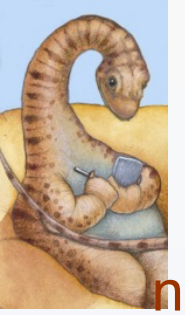
保证并发度的情况下实现临界资源互斥

- 分析：3个线程对全局变量x、y、z的使用场景

变量	Thread1和thread2	Thread1和thread3	Thread2和thread3
x	不共享	不共享	不共享
y	同时读	读写互斥	读写互斥
z	不共享	不共享	读写互斥

- 为提高线程之间的并发度
  - 仅保证相关临界区互斥访问即可

- 如果仅使用一个互斥信号量实现它们的互斥，降低并发度



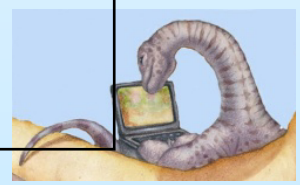
保证并发度的情况下实现临界资源互斥

- Semaphore mutex_y1=1; // 用于thread1与thread3互斥访问y；
- Semaphore mutex_y2=1; // 用于thread2与thread3互斥访问y；
- Semaphore mutex_z=1; // 用于thread2与thread3互斥访问z；

```
thread1 {
   cnum w;
   P(mutex_y1);
   w=add(x,y);
   V(mutex_y1)
   .......
}
```

```
thread2 {
   cnum w;
   P(mutex_y2);
   P(mutex_z);
   w=add(y,z);
   V(mutex_z);
   V(mutex_y2);
   .......
}
```

```
thread3 {
   cnum w;
   w.a=1;
   w.b=1;
   P(mutex_z);
   z=add(z,w);
   V(mutex_z);
   P(mutex_y1);
   P(mutex_y2);
   y=add(y,w);
   V(mutex_y2);
   V(mutex_y1);
   .......
}
```



无限长的消息队列

- 生产者－消费者问题中，考虑无限长的消息队列的同步问题；

阅览室人数控制

- 有一阅览室，共有100个座位。读者进入时必须先在一张登记表上登记，该表为每一座位列一个目录，包括座号与读者姓名。读者离开时要销掉登记内容。试用记录型信号量机制及wait与signal操作描述读者之间的同步。

用同一批流媒体数据比较多个播放器的性能

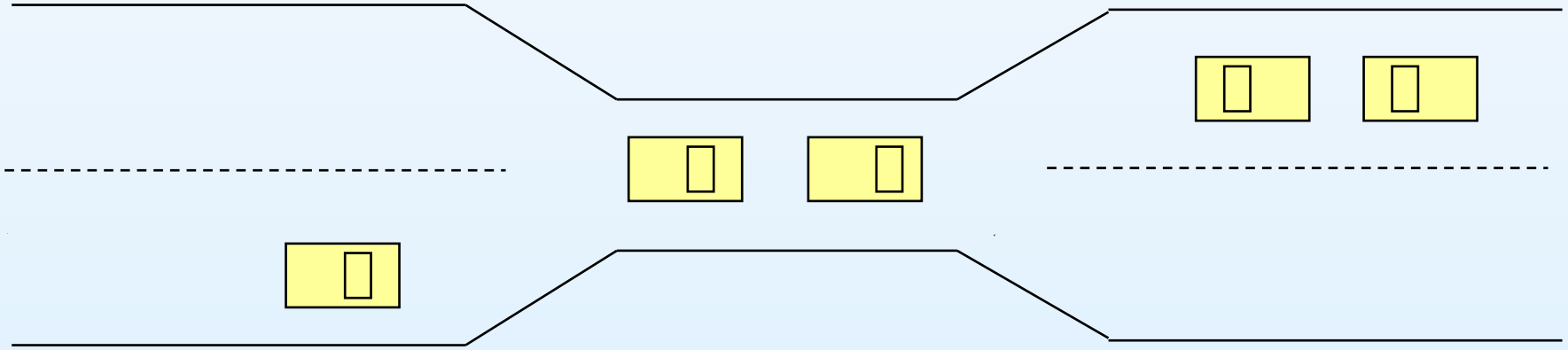
- 为测试两个播放组件的性能，某同学设计了一个媒体播放器。
- 该播放器由一个媒体下载进程P1及两个采用不同播放组件所构成的媒体播放进程P2、P3组成。
- 下载进程P1不断地从网上下载媒体信息并放入一个本地缓冲区Buffer中；
- 两个播放进程P2、P3分别从缓冲区Buffer中取出进程P1下载的信息进行播放。规定如下：
- 下载进程P1 每向缓冲区buffer送入一个媒体信息后, 必须等待两个播放进程P2、P3都取走后，进程P1才可以往buffer中送入下一个信息；
- 进程P2、P3对进程P1 送入的每一信息各取一次。
- 试用信号量及wait、signal操作实现进程P1、P2、P3之间的同步。

网站维护操作与浏览操作协调问题

- 为使浏览客户能够及时获取最新的信息，某网站系统管理员定期对信息页的内容进行更新。对信息页内容的浏览及更新操作，规定如下：
- 对于一个空闲的信息页，可允许多个浏览进程同时对其进行浏览操作；但当浏览进程对其进行浏览的过程中，不允许更新进程对其进行更新操作；
- 对于一个空闲的信息页，更新进程可以对其进行更新操作；但当更新进程对信息页进行更新操作的过程中，不允许其他更新进程对其进行更新操作，也不允许浏览进程对其进行浏览操作。
- 试用信号量及wait、signal操作实现浏览进程及更新进程之间的同步。

汽车过桥问题

- 汽车过桥、火车共享铁路问题



读者写者问题的进一步讨论

- 对于读者写者问题，
  - （1）说明进程间的相互制约关系，应设哪些信号量? (原算法是读者优先)
  - （2）用wait和signal写出其同步算法。
  - （3）修改上述算法，使它对写者优先，即一旦有写者到达，后续的读者都必须等待，而无论是否有已有读者正在访问对象。

读者写者问题的进一步讨论

- 设置一个信号量，当有写者到来时，封锁后续的读者；

- semorphore

  rmutex,wmutex,w =1,1,1;

- int readcount=0;

writer

```
writer:
        while(true) {
                wait(w);
                wait(wmutex);
                perform write operation;
                signal(wmutex);
                signal(w);
        }
```

reader

```
Reader: while(true) {
            wait(w);          //尽管有读者在读，但封锁刚到来的读者
             wait(rmutex);
             if readcount=0 then wait(wmutext);
             readcount:=readcount+1;
             signal(rmutex);
            signal(w);
             perform read operation;
            wait(rmutext);
            readcount:=readcount-1;
            if readcount=0 then signal(wmutext);
            signal(rmutex);
}
```

- 为防止H1N1病毒携带者向社会传播，某飞机场对每个到来航班中的所有乘客都要进行身体检查。
- 机场设置了一个容纳50人的休息室供乘客休息并等候医生检查，开始的时候休息室是空的。当乘客下飞机提取自己的行李后，若休息室中有空座位，则进入休息室等候检查，否则需要在休息室门口等待。医生每次呼叫一个在休息室中等待的乘客进入检查室对其进行检查，无乘客时医生休息。试用信号量及wait、signal操作描述乘客及医生的活动。

- 为防止H1N1病毒携带者向社会传播，某飞机场对每个到来航班中的所有乘客都要进行身体检查。
- 机场设置了一个容纳50人的休息室供乘客休息并等候医生检查，开始的时候休息室是空的。当乘客下飞机提取自己的行李后，若休息室中有空座位，则进入休息室等候检查，否则暂时离开。
- 医生每次呼叫一个在休息室中等待的乘客进入检查室对其进行检查，无乘客时医生休息。试用信号量及wait、signal操作描述乘客及医生的活动。

木兰出征

- 中原与某国战事吃紧，木兰要出征
- 木兰命四个仆人
  - 东市买骏马，西市买鞍鞯，南市买辔头，北市买长鞭。
- 装备齐全后，木兰就出征。

自行车装配问题-1

- 设自行车生产线上有一只箱子, 其中有N个位置(N≥3), 每个位置可存放一个车架或一个车轮; 又设有三个工人, 其活动分别为:

工人1活动:
do {
加工一个车架;
车架放入箱中;
}while(1)

工人2活动:
do {
加工一个车轮;
车轮放入箱中;
}while(1)

工人3活动:
do {
箱中取一车架;
箱中取二车轮;
组装为一台车;
}while(1)

试用信号量与Wait、signal（或P、V）操作实现三个工人的合作 。

信号量与同步问题的进一步讨论

- 为某临界区设置一把锁W，当W＝1时表示关锁，当W＝0时表示锁已经打开。
  - 试写出开锁原语与关锁原语，并利用他们实现互斥。

```
 lock(w): //开锁原语
{
  while (w==0) ;   //Test
   w=0;            //Set
}
```

```
unlock(w):   //关锁原语
    w=1;
```

==========

```
w=1;
do {     …
         lock(w);
         critical section;
         unlock(w);
         reminder section;
    } while (1);
```

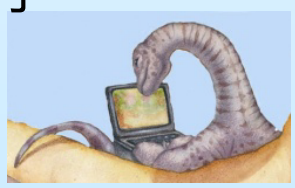
```
do {   …
       lock(w);
       critical section;
       unlock(w);
       reminder section;
     } while (1);
```

比较：整型信号量

```
Semaphore  S;

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

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

- This type of semaphore is also call “**spinlock(自旋锁)**”;
- 自旋锁的缺点是循环等待，占用cpu时间。
  - 在单处理器系统中尤为突出
    - 这段时间其它不访问临界区的进程可以执行；
    - 当 一个进程等该一个事件，而该时间需要其它进程产生，而其它进程无法执行
  - 自旋锁的优点是进程在循环等待，不需要进行上下文切换，减少了系统开销；
  - 当等待锁的时间较短时，自旋锁是有效的；
  - 一般在多处理器系统中使用（更适合于多处理器）
    - 一个线程在一个处理器上等待，而另一个线程可以在另一个处理器上访问临界区；
    - 当一个进程等待另一个进程产生的事件，另一个进程可以在其它处理机上执行从而产生该事件



自行车装配问题-2

- semaphore empty=N;
- semaphore wheel=0;
- semaphore frame=0;

- semaphore s1=N-2; //箱中车架的数量不可超过 N-2个
- semaphore s2=N-1; //车轮的数量不可超过N-1个

自行车装配问题-3

- 工人 1活动：
- do {
- 加工一个车架 ;
- P(s1);
- P(empty);
- 车架放入箱中 ;
- V(frame);
- } while (1)

自行车装配问题-4

- 工人 2活动：

```
do {
    加工一个车轮 ;
    P(s2);
    P(empty);
    车轮放入箱中 ;
    V(wheel);
} while (1)
```

自行车装配问题-5

- 工人 3活动：

```
do {
    P(frame);
    箱中取一车架；
    V(empty);
    V(s1);
    P(wheel);
    P(wheel);
    箱中取二车轮；
    V(empty);
    V(empty);
    V(s2);
    V(s2);
    组装为一台车；
} while (1)
```

和尚取水、饮水问题

- 某寺庙，有小和尚、老和尚若干。庙内有一水缸，由小和尚提水入缸，供老和尚饮用。水缸可容纳 30 桶水，每次入水、取水仅为1桶，不可同时进行。水取自同一井中，水井径窄，每次只能容纳一个水桶取水。现有水桶5个，供小和尚入水及老和尚取水使用。规定入水、取水后，把桶放下，使用时再重新取。试用记录型信号量和wait、signal（或P、V）操作给出老和尚和小和尚的活动。

和尚取水、饮水问题

- semaphore empty=30; // 表示缸中目前还能装多少桶水，初始时能装 30 桶水
- semaphore full=0; // 表示缸中有多少桶水，初始时缸中没有水
- semaphore buckets=5; // 表示有多少只空桶可用，初始时有 5 只桶可用
- semaphore mutex_well=1; // 用于实现对井的互斥操作
- semaphore mutex_bigjar=1; // 用于实现对缸的互斥操作

和尚取水、饮水问题

n **young_monk()** { //到井中打水，然后倒入缸中，供老和尚饮用

```
while(1){
    P(empty);          //水缸是否已满?
    P(buckets);        //申请一个空桶
    get a bucket;  //可以考虑对取桶操作实现互斥
    go to the well;
    P(mutex_well); //实现对井的互斥操作
    get water;
    V(mutex_well); //释放井的使用权
    go to the temple;
    P(mutex_bigjar); //实现对缸的互斥操作
    pour the water into the big jar;
    V(mutex_bigjar); //释放缸的使用权
    V(buckets);  //将桶放下
    V(full);           //老和尚可以取水
   }
}
```

和尚取水、饮水问题

- **old_monk()** { //取水饮用

```
while(1){
    P(full);          //缸中是否有水?
    P(buckets);   //申请一个空桶
    get a bucket; //可以考虑对取桶操作实现互斥
    P(mutex_bigjar); //水缸是否空闲? 申请对缸的使用权
    get water;
    Drink water;
    V(mutex_bigjar); //释放水缸的使用权
    V(buckets);   //放下桶
    V(empty);     //
  } //while(1)
}
```

P233 6.8

- **6.8** Servers can be designed to limit the number of open connections.
  - For example, a server may wish to have only *N* socket connections at any point in time.
  - As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is released.
- Explain how semaphores can be used by a server to limit the number of concurrent connections

P233 6.8

- Semaphore connection=N

- acquireConnection() {
  wait(connection);
  //accept a connection
  }

- releaseConnection() {
  //release a connection;
  signal(connection);
  }

课后阅读：Nachos中的信号量

- code/threads/synch.cc，关于信号量及P、V操作的有关内容
- code/monitor目录下的有关代码：管程

课后阅读：Nachos中Semaphore

```
Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List;
}
```

```
Semaphore::~Semaphore()
{
    delete queue;
}
```

课后阅读：Nachos中Semaphore

```
Void Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);       // disable interrupts

    while (value == 0) {                                    // semaphore not available
            queue->Append((void *)currentThread);           // so go to sleep
            currentThread->Sleep();
    }
    value--;                                                // semaphore available,
                                                            // consume its value

    (void) interrupt->SetLevel(oldLevel);                   // re-enable interrupts
}
```

课后阅读：Nachos中Semaphore

```
Void Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    // make thread ready, consuming the V immediately
    if (thread != NULL)
           scheduler->ReadyToRun(thread);  //
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```