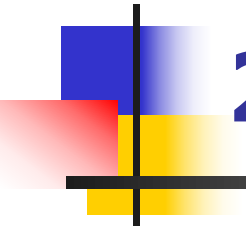


2.4 进程同步

2.4 进程同步

- 进程异步性
- 资源无序使用
- 必须协调多个相关进程的执行顺序
- 共享资源，相互合作
- 使程序的执行具有可再现性

2.4.1 进程同步的基本概念

- 间接相互制约关系
  - 共享资源导致
- 直接相互制约关系
  - 进程间合作
- 临界资源
  - 系统中某些资源一次只允许一个进程使用
  - 互斥方式使用

GET:

Begin local g

g←stack[top] （1）

top=top-1 （2）

End

程序段PA：取栈顶数据

IN:

Begin

top=top＋1 （1）

stack[top] ←ad （2）

End

程序段PB：数据入栈

如果执行顺序：

PB(1)→ PA(1) → PA(2) → PB(2)

则？

生产者—消费者问题

- 错误？

```
buffer[] //大小为n的缓冲区
int counter;
int in =1 ; int out=1;
```

```
Product:
While(true){
   while (counter==n );
   buffer[in] = item;
   in = in +1 mod n;
    counter++
}
```

```
Consume:
While(true){
     while (counter==0);
     get = buffer[out];
     out =  out + 1 mod n;
     counter --;
}
```

- 生产者—消费者问题
  - 分别执行正确，并发执行发生错误
  - 原因在于对共享变量 **counter**的操作
  - 必须对共享资源加以管理

```
Counter ++;
        mov R1 , counter
        inc  R1
        mov  counter, R1
```

```
Counter --;
        mov R2 , counter
        dec  R2
        mov  counter, R2
```

```
      (正确结果：Counter = 5)
mov R1,counter  ( R1= 5 )
inc   R1              ( R1= 6 )
mov R2 , counter (R2 =5 )
dec  R2               (R2 = 4)
mov counter,R1    (counter = 6)
mov counter,R2    (counter = 4)
```

2.4.1 进程同步的基本概念

- 临界区
  - 进程中访问临界资源的那段代码
  - 进程互斥进入临界区，实现对临界资源的互斥访问
  - 代码进入临界区之前，检查临界资源是否被访问
  - 退出临界区时，把临界资源的访问标志修改为未访问

2.4.1 进程同步的基本概念

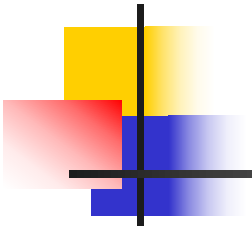
- 同步机制应遵循的规则
  - 空闲让进
    - 各进程具有平等独立竞争公有资源的权利；
    - 当无进程在临界区时，任何有权使用临界区的进程可以进入；
    - 并发进程的某个进程不在临界区内时，它不阻止其它进程进入临界区。
  - 忙则等待
    - 不允许两个以上的进程同时进入临界区,每次至多有一个进程处于临界区
  - 有限等待
    - 任何进入临界区的要求应在有限的时间内得到满足；进程在临界区内仅逗留有限的时间
  - 让权等待
    - 进程不能进入自己的临界区时，释放处理机，不允许“忙等”

2.4.2 硬件同步机制

1 基于关闭中断的互斥实现

当一个进程进入临界区后，关闭所有的中断；当它退出临界区时，再打开中断。

- 进程的切换是由中断引发的，关闭中断后，CPU不会被分配给其他进程，其他进程无法执行；
- 操作系统内核经常使用这种方法来更新内部的数据结构（变量、链表等）。



问题：

- 如果进程在临界区中执行大量的计算，结果会如何？
- 这种方法能否用于用户进程？
- 这种方法能否用在多CPU的系统中？

2 基于繁忙等待的互斥实现

方法1. 加锁标志位法

```
while ( lock );
lock = 1;
临界区
lock = 0;
```

lock的初始值为0，当一个进程想进入临界区时，先查看lock的值，若为1，说明已有进程在临界区内，只好循环等待。等它变成了0，才可进入。每个进程的操作类似。

3 TSL指令

```
while ( lock );
lock = 1;
临界区
lock = 0;
```

加锁标志位法的缺点在于可能出现针对共享变量 lock 的竞争
状态。例如，当进程 0 执行完循环判断语句后，被时钟中断打断，从而可能使多个进程同时进入临界区。

能不能把查询lock变量与修改lock变量这两个操作捆绑在一起，使它们不会被打断？这就是硬件上的
TSL（Test and Set Lock）指令。

TSL指令（续）

该指令读入内存变量LOCK的值，保存在寄存器RX
中，然后存放一个非零值到LOCK当中。TSL是一个原子操作，即不可分隔的操作。

Intel x86系列：BTS (Bit Test and Set)指令。

问题：如何使用TSL指令来实现进程间互斥？

TSL指令（续）

- 使用LOCK来作为加锁标志位，协调各个进程对共享资源的访问；
- 当LOCK为0时，任何进程均可使用TSL指令把它设置为非0，进而访问共享资源；
- 当LOCK为非0时，循环等待；
- 在退出临界区时，把LOCK置为0。

```
enter_region:
    TSL  REGISTER, LOCK
    CMP  REGISTER, #0
    JNE  enter_region
    RET
```

```
leave_region:
    MOVE  LOCK, #0
    RET
```

4 SWAP 指令

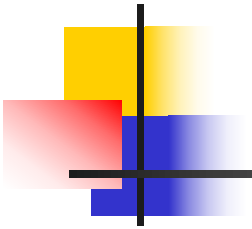
- 使用**LOCK**来作为加锁标志位，协调各个进程对共享资源的访问；
- 当**mutex**为**0**时，任何进程均使用**swap**指令把它设置为非**0**，进而访问共享资源；
- 当**mutex**为非**0**时，等待；
- 在退出临界区时，把**mutex**置为**0**。

lock:

```
MOV AL, 1H
XCHG mutex,AL
CMP AL, 0H
JNE lock
RET
```

unlock:

```
MOV mutex, 0H
RET
```



小结

以上的各种方法，都是基于繁忙等待(busy waiting)的策略，都可归纳为一种形式：当一个进程想要进入它的临界区时，首先检查一下是否允许它进入，若允许，就直接进入了；若不允许，就在那里循环地等待，一直等到允许它进入。

缺点：1）浪费CPU时间；2）可能导致预料之外的结果（如：一个低优先级进程位于临界区中，这时有一个高优先级的进程也试图进入临界区）

- 一个低优先级的进程正在临界区中；
- 另一个高优先级的进程就绪了；
- 调度器把CPU分配给高优先级的进程；
- 该进程也想进入临界区；
- 高优先级进程将会循环等待，等待低优先级进程退出临界区；
- 低优先级进程无法获得CPU，无法离开临界区。

2.4.3 信号量机制，

- 1965年Dijkstra 提出
- 1 整型信号量
  - 信号量定义为一个整型量
  - 除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和signal(S)来访问。这两个操作一直被分别称为P、V操作。

```
wait(S){   while S≤0 do no-op；
           S：=S-1;
        }
signal(S){ S：=S+1; }
```

  - wait操作，只要是信号量S≤0， 就会不断地测试。
  - 该机制并未遵循“让权等待”的准则， 而是使进程处于“忙等”的状态。

2.4.3 信号量机制，

- **2 记录型信号量**
  - 基本思想
    - 用一个整型变量代表资源数目
    - 进程链表链接上述的所有等待进程
    - 采用了记录型的数据结构

- 记录型信号量
  - 整型值表示资源数目
  - **Wait**操作请求一个单位的该类资源，成功则－**1**；若**<0**，说明无资源可用，则自我阻塞，插入信号量链表
  - **Signal**释放一个资源，＋**1**；如有等待该资源的进程则唤醒

```
typedef  struct {
        value:integer;囍
        struct PCB *list;囍
        }semaphore
```

```
wait( semaphore S){
      S.value--;囍
      if (S.value＜0)  block(S->list)囍
 }囍
```

```
signal(semaphore S){囍
        S.value++;囍
        if (S.value≤0)wakeup(S.->list);
      囍
      }囍
```

2.4.3 信号量机制，

- 需要两个以上共享资源的情况
  - 进程中**A**和**B**都要包含对**Dmutex**和**Emutex**的操作
  - **process A:**
    wait(Dmutex);
    wait(Emutex);
    。 。 。 。 。 。

    process B:囍
    wait(Emutex);囍
    wait(Dmutex);囍
    。 。 。 。 。 。 。 。

若进程**A**和**B**按下述次序交替执行**wait**操作？囍

process A: wait(Dmutex); 于是**Dmutex=0**囍
process B: wait(Emutex); 于是**Emutex=0**囍
process A: wait(Emutex); 于是**Emutex=-1 A**阻塞囍
process B: wait(Dmutex); 于是**Dmutex=-1 B**阻塞

发生了两个进程都无法继续运行的僵持状态！

2.4.3 信号量机制，

- **AND型信号量**
  - 一次性分配原则
    - 将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程
    - 待进程使用完后再一起释放
    - 只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。
  - 亦即，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。
  - 在**wait**操作中，增加了一个“**AND**”条件，故称为**AND**同步，或称为同时**wait**操作， 即**Swait(Simultaneous wait)**定义如下

```
Swait(S1, S2, ..., Sn){
 While(TRUE){囍
   if (Si≥1 && ... && Sn≥1){囍
      for (i =1;i<=n;i++) si--;
      break;
   } else{囍
      place the process in the waiting
      queue associated with the first
      Si found   with Si＜1, and set
      the program counter of this
      process to the beginning of
      Swait operation囍
   }
 }囍
}
```

```
Ssignal(S1, S2, ..., Sn){
 While(TRUE)囍{
     for (i =1;i<=n;i++){
        Si ++; 囍
        Remove all the process waiting
        in the queue associated with Si
        into the ready queue.囍
    }
  }
}
```

??????

```
Swait(S1, S2, ..., Sn){囍
   if (S1≥1 and ... and Sn≥1){囍
      for (i =1;i<=n;i++) si--;
   } else{囍
      place the process in the waiting
      queue associated with the first
      Si found   with Si＜1, and set
      the program counter of this
      process to the beginning of
      Swait operation囍
   }囍
}
```

```
Ssignal(S1, S2, ..., Sn){囍
      for (i =1;i<=n;i++){
         Si ++; 囍
         Remove all the process waiting
         in the queue associated with Si
         into the ready queue.囍
      }
}
```

```
Swait(S1, S2, ..., Sn){囍
  while(TRUE){
    if (S1≥1 and ... and Sn≥1){囍
        for (i =1;i<=n;i++) si--;
        break;
    } else{囍
       block(Si→list);
       // where Si is the first
       semaphore found    with value
       less than 1 囍
    }
 }囍
}
```

```
Ssignal(S1, S2, ..., Sn){囍
      for (i =1;i<=n;i++){
          Si ++; 囍
         Remove all the process waiting
         in the queue associated with Si
         into the ready queue.囍
     }
}
```

2.4.3 信号量机制，

- **4 信号量集**
  - 一次分配多个某类资源
  - 每次分配前测试该资源的数量是否大于下限值
  - **S为信号量；d为需求值；t为下限值(假定d<=t,否则,应在Swait中检测si>=d是否满足)**

```
Swait(S1, t1, d1, …, Sn, tn, dn){囍
   if (S1≥t1 and … and Sn≥tn){囍
       for (i =1;i<=n;i++){ 囍
       Si：=Si-di;囍
   }  else囍{
      Place the executing process in
      the waiting queue of the first Si
      with Si＜ti and set its program
      counter to the beginning of the
      Swait Operation. 囍
      }
}
```

```
signal(S1, d1, …, Sn, dn){囍
 for (i =1;i<=n;i++){ 囍
    Si ： =琼Si+di;囍
     Remove all the process
     waiting in the queue
     associated with Si into the
     ready queue囍
  }
}
```

2.4.3 信号量机制。

- 一般“信号量集”的几种特殊情况：囍
  - **(1) Swait(S, d, d)**。 此时在信号量集中只有一个信号量**S**， 但允许它每次申请**d**个资源，当现有资源数少于**d**时，不予分配。囍
  - **(2) Swait(S, 1, 1)**。 此时的信号量集已蜕化为一般的记录型信号量**(S＞1**时**)**或互斥信号量**(S=1**时**)**。囍
  - **(3) Swait(S, 1, 0)**。这是一种很特殊且很有用的信号量操作。当**S≥1**时，允许多个进程进入某特定区；当**S**变为**0**后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

2.4.4 信号量的应用，

- **1 实现互斥**

```
semaphore mutex =琈1;囍
```

```
process1(){
囍
  while(1){囍
     wait(mutex);囍
     critical section囍
     signal(mutex);囍
     remainder seetion囍
   }
}
```

```
process 2(){
  while(1){囍
     wait(mutex);囍
     critical section囍
     signal(mutex);囍
     remainder seetion囍
  }
}
```

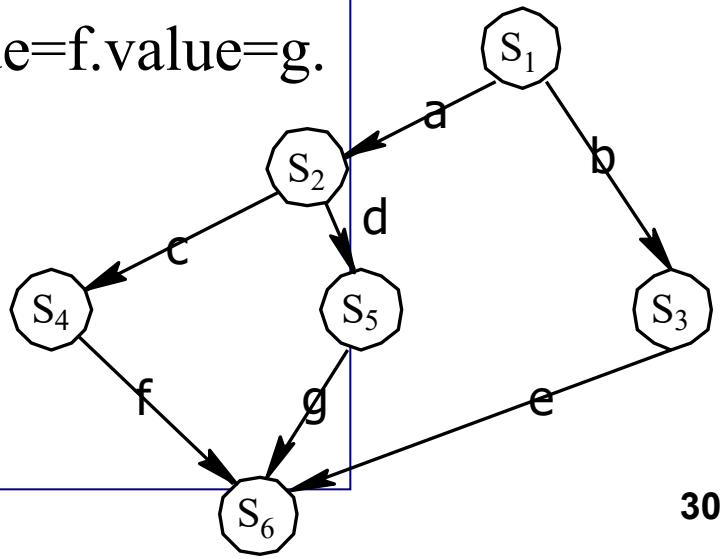
2.4.4 信号量的应用，

- **2 实现前驱关系**
  - 语句**S1**和**S2**分属不同进程
  - 若希望执行语句**S1**后再执行语句**S2**
    - 使二者共享公用信号量**s**，赋初值为**0**，
    - **Signal**（**s**）放在语句**S1**之后
    - **Wait**（**s**）放在**S2**语句之前
    - 即
      - 进程**P1**：　**S1;**　**signal(s);**
      - 进程**P2**：　**wait(s);**　　**S2;**

2.4.4 信号量的应用。

- **2 实现前驱关系**

```
semaphore a,b,c,d,e,f,g;
P1(){ S1; signal(a); signal(b); }囍
P2(){ wait(a); S2; signal(c); signal(d); }囍
P3(){ wait(b); S3; signal(e); }囍
P4(){ wait(c); S4; signal(f); }囍
P5(){ wait(d); S5; signal(g); }囍
P6(){ wait(e); wait(f); wait(g); S6; }囍
main(){
a.value=b.value=c.value=d.value=e.value=f.value=g.
value=0;
cobegin
    p2();p2();p3();p4();p5();p6();
coend
}
```



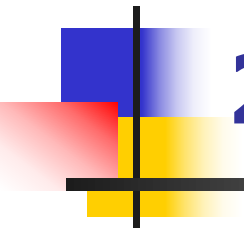
2.4.5 管程机制

- 进程自己控制同步操作**Wait，Signal**

- 定义
  - 代表共享资源的数据结构；
  - 对该数据结构实施操作的一组过程；
  - 共同构成操作系统的资源管理模块
- 请求和释放资源通过管程完成
- 组成
  - 名称；内部数据结构；过程；初始化语句
- 条件变量
  - **cwait，csignal**原语同步
  - 显式说明条件变量：形如 **condition x, y;**
  - 只能由管程访问
  - 抽象数据类型，由链表记录因该条件变量而阻塞的进程
  - **cwait(x):** 调用管程的进程因**x**而阻塞，则调用该过程讲其插入到**x**条件的等待队列。释放管程。
  - **csignal(x):** 调用管程的进程发现**x**条件变化，调用该过程重新启动阻塞进程。

管程机制

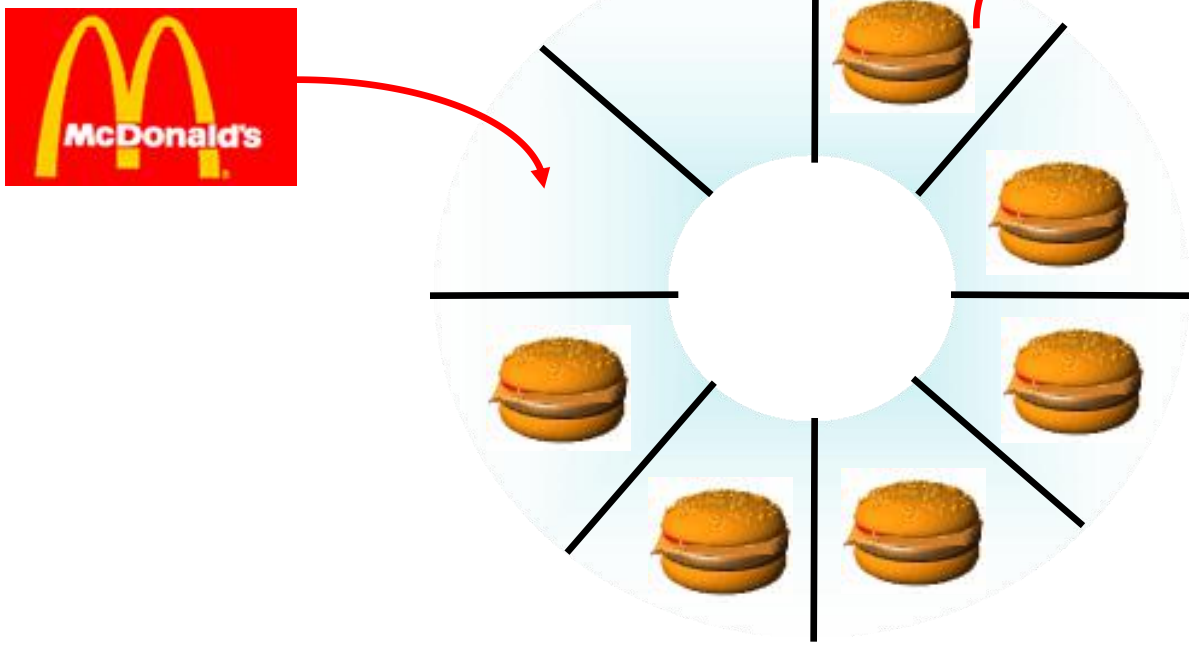
```
Monitor boundedbuffer;
Char buffer[N];
int nextin, nextout;
cond notfull,notempty;
void append (char x)
{ if (count ==N) cwait (notfull);
   buffer[nextin]=x;
   nextin = (nextin +1)%N;
   count ++;
   csignal(nonempty);
}

void take (char x)
{ if(count ==0) cwait (notempty);
  x= buffer[nextout];
  nextout = (nextout+1)%N;
 count--;
 csingal (notfull);
}
```



2.5 经典进程同步问题

2.5.1 生产者—消费者问题，

- 关于并发进程互斥和同步的一般模型
- 消费者
  - 使用某一类资源的进程，称为该资源的消费者。
- 生产者
  - 释放某一类的资源。
- 生产者和消费者之间满足以下条件
  - 通过有**n**个缓冲区的公共缓冲池交换信息
  - 消费者想接收数据，缓冲区中至少有一个单元是满的。
  - 生产者想发送数据，缓冲区中至少有一个单元是空的。



生产—消费者问题

2.5.1 生产者—消费者问题，

- **1 记录型信号量**

```
semaphore mutex=1, empty=n, full=0;
item buffer[n];
int     in=0, out=0;
```

```
producer(){
  do{
        ...
        producer an item nextp;
        ...
        wait(empty);
        wait(mutex);
        buffer(in)=nextp;
        in =(in+1) mod n;
        signal(mutex);
        signal(full);
     } while(true);
}
```

```
consumer(){
    do{
        wait(full);
        wait(mutex);
        nextc=buffer(out);
        out =(out+1) mod n;
        signal(mutex);
        signal(empty);
        consumer the item in nextc;
   }while(true);
}
```

```
main(){
   cobegin
      producer(); consumer();
   coend
}
```

- **注意：**
  - *成对出现；次序不能颠倒*

2 利用AND信号量解决生产者消费者问题

- 伪码

```
semaphore mutex=1, empty=n, full=0;
item buffer[n];囍
int     in=0, out=0;
```

```
producer(){
  do{囍
      produce an item in nextp;囍
       …囍
      Swait(empty, mutex);囍
      buffer(in)  =琇nextp;囍
      in  =琇(in+1)mod n;囍
      Ssignal(mutex, full);囍
 } while(true);囍
}
```

```
consumer(){囍
    do{ 囍
        Swait(full, mutex);囍
        nextc =琇buffer(out);囍
         out  =琇(out+1) mod n;囍
         Ssignal(mutex, empty);囍
         consumer the item in nextc;
 } while(true);囍
}
囍
```

利用管程解决生产者-消费者问题，

- **1) put(item)**过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量**count**来表示在缓冲池中已有的产品数目，当**count≥n**时， 表示缓冲池已满， 生产者须等待。
- **(2) get(item)**过程。消费者利用该过程从缓冲池中取出一个产品，当**count≤0**时，表示缓冲池中已无可取用的产品， 消费者应等待。

3 利用管程解决生产者-消费者问题，

- 伪码

```
monitor producer-consumer{囍
 int  in=0,out=0,count=0;囍
 item buffer[N];囍
 condition notfull, notempty;囍
```

```
void put(item x){囍
    囍
        if count≥N then cwait(notfull);
囍      buffer[in] = x;囍
         in坳=琊(in+1) % N;囍
         count++;囍
         csignal(notempty);囍
 }囍
```

```
 void get(item x){囍
        if (count≤0)cwait(notempty);囍
        x =琊buffer[out];囍
        out =琊(out+1) % N;囍
        count--;囍
        csignal(notfull);
 }囍
}
```

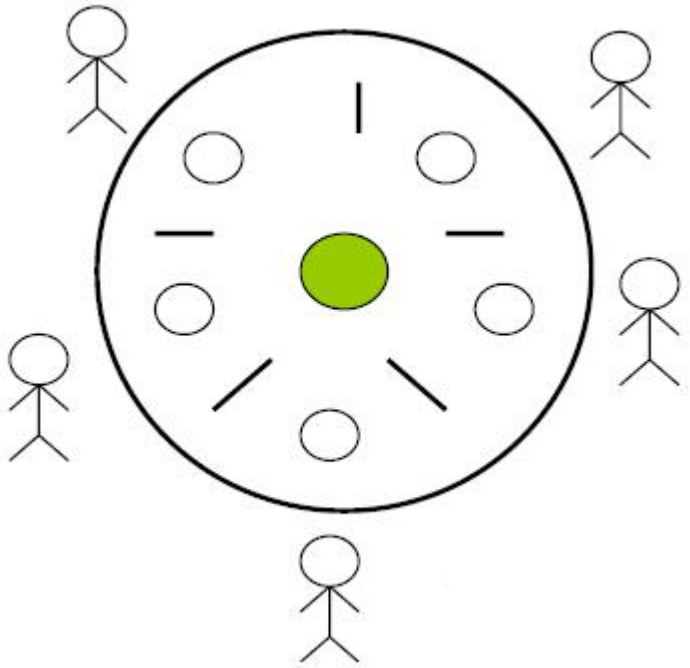
利用管程解决生产者-消费者问题。

- 生产者和消费者可描述为：

```
Producer(){
  item x;
  while(1){囍
          produce an item in x;囍
          PC.put(x);囍
   }
}囍
Consumer(){囍
      item x
      while(1)囍
          PC.get(x);囍
          consume the item in x;囍
      };囍
}
```

2.5.2 哲学家进餐问题

- （Dijkstra，1965）
- 有五个哲学家围坐在一圆桌旁，桌子中央有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子。每个哲学家思考、饥饿、吃通心面。
- 欲吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。



1 记录型信号量解决哲学家进餐问题

- 解
  - 每一把叉子都是必须互斥使用的，因此，应为每把叉子设置一个互斥信号量，初值均为**1**。
  - 当一个哲学家吃通心面之前必须获得自己左边和右边的两把叉子，即执行两个**P** 操作，吃完通心面后必须放下叉子，即执行两个**V** 操作。
- 隐患
  - 五位同时拿到左边筷子？
  - 只允许**4**个人同时取左边筷子
  - 仅当左右叉子都可用时才进餐
  - 规定奇数编号哲学家先左后右；偶数编号相反。
    - **1**、**2**号哲学家竞争**1**号筷子；**3**、**4**号哲学家竞争**3**号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

```
semaphore chopstick[5] ={1,1,1,1,1};
do{囍
   wait(chopstick［i］);囍
   wait(chopstick［(i+1) % 5］);囍
   eat;囍
   signal(chopstick［i］);囍
   signal(chopstick［(i+1) % 5］);囍
   think;囍
}while(true);
```

2 使用AND信号量机制解决哲学家就餐

- 要求每个哲学家先获得两个临界资源(筷子)后方能进餐，

```
semaphore chopstick[5] ={1,1,1,1,1};
do{
        think;
        Sswait(chopstick［(i+1) % 5］, chopstick ［i］);
        eat;
        Ssignal(chopstick ［(i+1) % 5］, chopstick ［i］);
    }while(true);
```

2.5.3 读者—写者问题

- 要求
  - 多个进程共享文件或记录
  - 允许多个进程同时读共享数据
  - 不允许写进程和其它读写进程同时访问数据
  - 无读进程时可写
- 解
  - 读、写进程需要互斥（信号量）
  - 记录读进程的数目，无读进程才可写（信号量）

记录型信号量解决读者—写者问题

- 伪码

```
semaphore rmutex=1, wmutex=1;
int readcount =0;
```

```
Void reader(){
  do{
      wait(rmutex);
       if (readcount=0) wait(wmutex);
       readcount++;
       signal(rmutex);

       perform read operation;

      wait(rmutex);
      readcount--;
      if (readcount=0) signal(wmutex);
       signal(rmutex);}
  while(true);
}
```

```
void writer(){
  do{
      wait(wmutex);
      perform write operation;
      signal(wmutex);}
  while(true);
}
```

利用信号量集机制解决读者写者问题

- 最多允许N个读者同时读

```
#define  N  88
semaphore L=N, mx=1;
void reader()
  do{囍
        swait(L,1,1);//控制读者数目囍
        swait(mx,1,0);//无写进程可读囍
          鲋 perform read operation;
        ssignal(L,1);
     }while(true);囍
  }
```

```
囍
    void writer(){囍
        do{囍
            swait(mx,1,1; L,N,0);囍
            //无写进程，
            //无读进程，即L＝N；
             perform write operation;囍
             ssignal(mx,1);囍
        }while(true)
囍}
```