操作系统2023

第5讲 互斥和同步(1/2)

清华大学电子工程系

马洪兵

大纲

01 互斥和同步问题

02 临界区互斥问题的算法

03 信号量

04 经典IPC问题

05 管程

进程并发执行结果不确定的示例

- 订票系统，两个终端，运行T1、T2程序

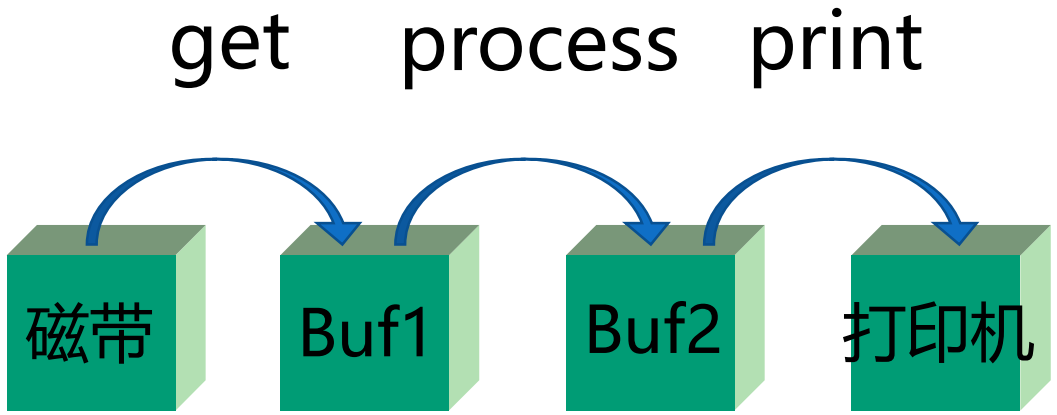
```
T1:
      ...
①  read(x);
②  if x>=1 then
③      x:=x-1;
④  write(x);
      ...
```

```
T2:
      ...
①  read(x);
②  if x>=1 then
③      x:=x-1;
④  write(x);
      ...
```

- 假设执行顺序为T1①②③T2①②③④T1④，卖出2张票，写回数据库的票款余额只减了1

进程并发执行结果不确定的示例

- 操作顺序冲突
  - 3个进程get、process和print协作完成事务处理
  - 有6种可能的操作顺序，只有一种结果是正确的



进程间的制约关系

- 并发执行的多个进程，都是独立地和异步地向前推进，彼此间似乎互不相关，但实际上每个进程在其运行过程中总是存在着某种间接或直接的制约关系
- 直接相互制约关系（同步）
- 间接相互制约关系（互斥）
- 互斥与同步问题对于线程同样适用

进程的同步

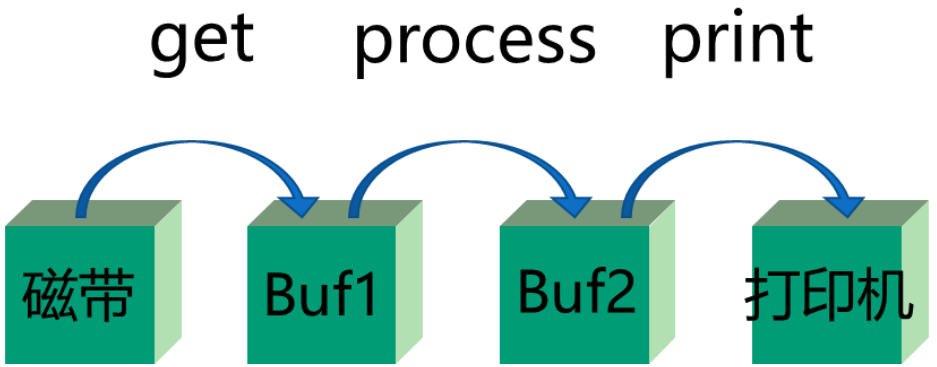
- 指系统中一些进程需要相互合作，共同完成一项任务，这种协作进程之间相互等待对方消息或信号的协调关系称为进程同步
- 具体说，并发进程在一些关键点上可能需要互相等待与互通消息，进程间的相互联系是有意识的安排的
- 产生的原因：进程合作

进程的同步

- 同步问题有两类
  - 保证一组合作进程按逻辑需要的执行次序执行
  - 保证共享缓冲区（共享数据）的合作进程的同步

司机的活动：
- 启动开车
- 行车
- 到站停车

列车员的活动：
- 关车门
- 服务
- 开车门



进程的互斥

- 是指若干个进程同时竞争一个需要互斥使用的资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到该资源被释放。进程间要通过某种中介发生联系，是无意识安排的
- 产生的原因：资源共享
- 互斥是一种特殊的同步
  - 逐次使用互斥资源，也是对进程使用资源次序上的一种协调

临界资源

- 系统中某些共享资源一次只允许一个进程使用，称这样的资源为临界资源(critical resource)
- 硬件临界资源：打印机、光盘刻录机
- 软件临界资源：只能互斥访问的变量、表格、队列
- 并非所有共享资源都是临界资源，如只读数据

临界区

■临界区(critical section)：进程中访问临界资源的代码片断

```
entry section          /*进入区*/
    critical section   /*临界区*/
exit section           /*退出区*/
    remainder section  /*剩余区*/
```

临界区

- 进入区(entry section)：在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应“正在访问临界区”标志
- 退出区(exit section)：将“正在访问临界区”标志清除
- 剩余区(remainder section)：代码中的其余部分

```
entry section          /*进入区*/
    critical section   /*临界区*/
exit section           /*退出区*/
    remainder section  /*剩余区*/
```

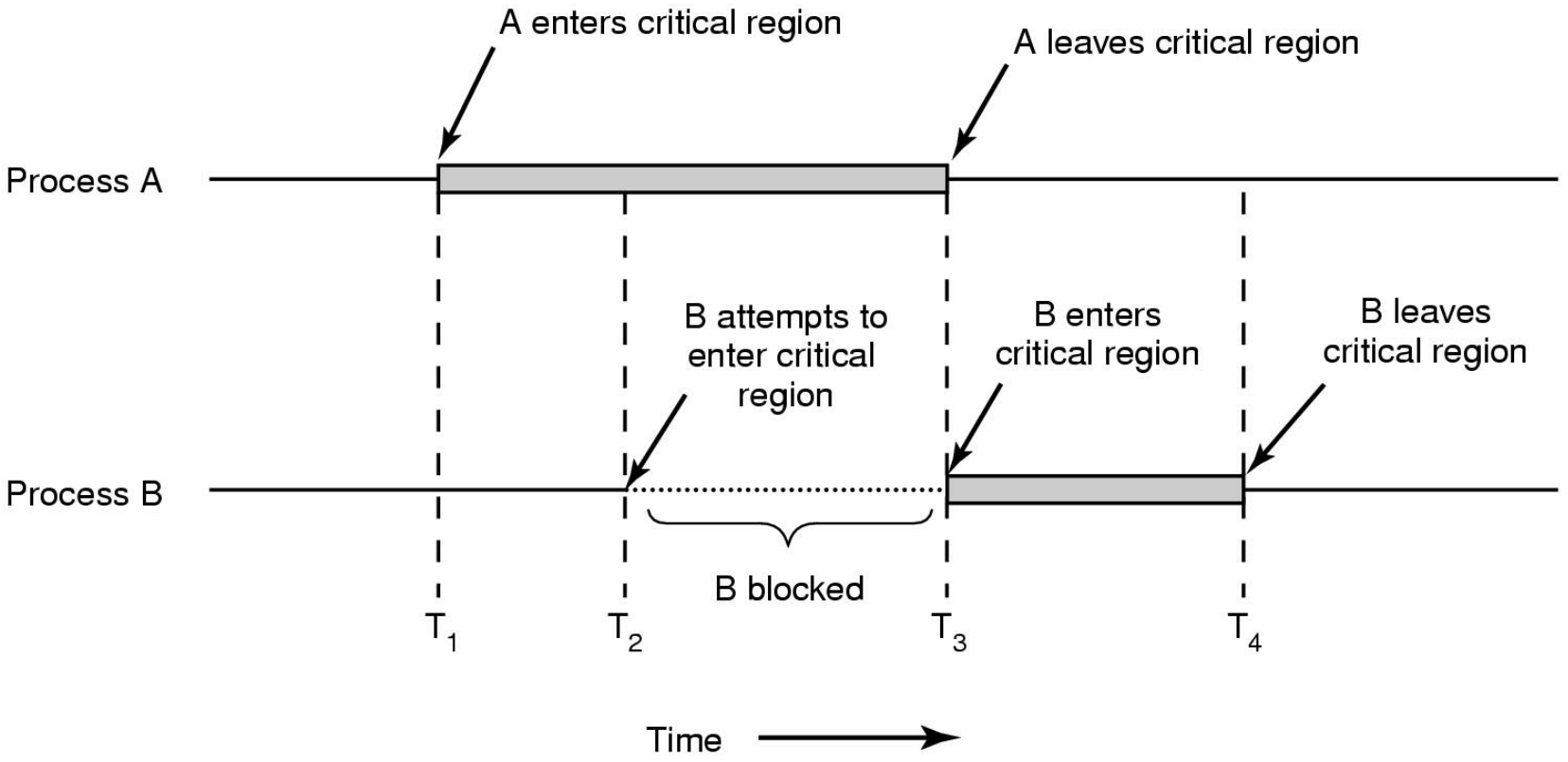
访问临界区应遵循的原则

1. 空闲让进—— 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
2. 忙则等待——不允许两个以上的进程同时进入临界区
3. 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
4. 让权等待——不能进入临界区的进程应放弃占用CPU

- 目的：保证使用临界资源的进程能够正确和高效地进行协作，避免竞争条件
- 竞争条件(race condition)：两个或多个进程访问某些共享资源，最终结果取决于进程运行的精确时序

访问临界区应遵循的原则

- 使用临界区



大纲

01 互斥和同步问题

02 临界区互斥问题的算法

03 信号量

04 经典IPC问题

05 管程

锁变量

- 共享锁变量lock:
  - 0：无进程在临界区(初值)
  - 1：有进程在临界区

- 存在的问题：
  - 存在违反条件2的可能
  - 忙等待(违反条件4)

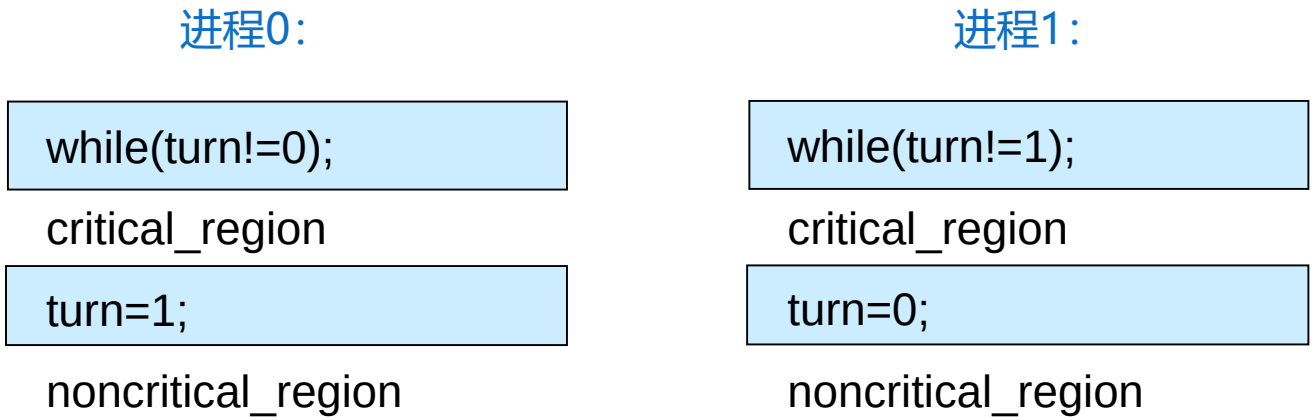
```
while(lock==1);      <a>
lock=1;              <b>
critical_region
lock=0;
noncritical_region
```

忙等待

- 忙等待(busy waiting)：连续测试一个变量直到出现某个值为止
- 忙等待浪费CPU时间，所以应该避免。只有等待时间非常短的情况下才使用忙等待
- 用于忙等待的锁，称为自旋锁(spin lock)

轮转法

- 设置整型变量turn：用于记录轮到哪个进程进入临界区
  - 0：轮到进程0(初值)
  - 1：轮到进程1



轮转法

- 轮转法存在的问题
  - 该算法要求两个进程严格轮流进入临界区，因而可能违反条件1
  - 程序代码不对称
  - 忙等待

进程0：

```
while(turn!=0);
critical_region
turn=1;
noncritical_region
```

进程1：

```
while(turn!=1);
critical_region
turn=0;
noncritical_region
```

Peterson算法

- Peterson算法需要设置整形变量turn和数组interested，turn表示轮到哪个进程进入临界区，数组interested表示进程是否想要进入临界区

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Peterson算法

- 当一个进程想进入临界区时，先调用enter_region函数，判断是否能安全进入，不能的话等待
- 当进程从临界区退出后，需调用leave_region函数，允许其它进程进入临界区

进程0：

```
enter_region(0);
critical_region
leave_region(0);
noncritical_region
```

进程1：

```
enter_region(1);
critical_region
leave_region(1);
noncritical_region
```

Peterson算法

- 只有当interested[other]==FALSE或者turn==other时，process才能进入临界区。如果other已经在临界区中，那么interested[other]==TRUE并且turn==process，因此process只能在while循环上忙等，而进不了临界区

```
void enter_region(int process);
{
        int other;

        other = 1 – process;
        interested[process] = TRUE;
        turn = process;
        while (turn == process && interested[other] == TRUE);
}
```

Peterson算法

- 考虑极端情况，两个进程几乎同时希望进入临界区，此时while循环中的interested[other]==TRUE成立，但是turn==process只可能对一个进程成立，turn的取值为后一个修改turn变量的进程号（前一个的进程号被覆盖），因此后一个进程在while循环上忙等，前一个进程进入临界区——这是合理的，先要进入临界区的进程进入，后要进入临界区的进程等待

```
void enter_region(int process);
{
        int other;

        other = 1 – process;
        interested[process] = TRUE;
        turn = process;
        while (turn == process && interested[other] == TRUE);
}
```

Peterson算法

- 优点：
  - 解决了互斥访问的问题，而且克服了轮转法的缺点，可以正常地工作
- 缺点：
  - 忙等待

硬件方法——禁止中断

- 进入临界区前执行“关中断”指令
- 离开临界区后执行“开中断”指令
- 只有在发生时钟中断或其他中断时才会进行进程切换
- 优点：简单
- 缺点：
  - 把禁止中断的权利交给用户进程导致系统可靠性较差
  - 不适用于多处理器

硬件指令方法

- 利用处理机提供的专门的硬件指令，对一个字的内容进行检测和修改
- 硬件方法的目的是解决共享变量的完整性和正确性——其读写操作由一条指令完成，因而保证读操作与写操作不被打断

硬件指令方法

- 利用测试并设置(Test and Set)指令，例如X86的BTS/BTR指令
- BTS OP1, OP2
  - 将OP1中第OP2位的值保存到标志寄存器FLAGS的进位标志CF中，并将被测试的位置1
- BTR OP1, OP2
  - 将OP1中第OP2位的值保存到进位标志CF中，并将被测试的位清0

硬件指令方法

- X86的BTS/BTR指令

```
enter_region:
        LOCK BTS lock, 0
        JC enter_region
        RET
```

```
leave_region:
        LOCK BTR lock, 0
        RET
```

- 指令前缀LOCK用于锁内存总线，以防其他处理器访问内存

硬件指令方法

- 利用SWAP(对换)指令，例如X86的XCHG指令
- XCHG OP1, OP2
  - 将OP1和OP2的内容对换

```
enter_region:
        MOV reg, 1
        LOCK XCHG lock, reg
        CMP reg, 0
        JNZ enter_region
        RET
```

```
leave_region:
        LOCK MOV lock, 0
        RET
```

硬件指令方法

- 当一个进程想进入临界区时，先调用enter_region，判断是否能安全进入，不能的话等待；当它从临界区退出后，需调用leave_region，允许其它进程进入临界区

```
enter_region();
critical_region
leave_region();
noncritical_region
```

硬件指令方法

- 硬件指令方法的优点
  - 适用于任意数目的进程
  - 简单，容易验证其正确性
  - 可以支持进程中存在多个临界区，只需为每个临界区设立一个布尔变量
- 硬件方法的缺点
  - 忙等待，耗费CPU时间

优先级反转问题

- 忙等待不但浪费CPU时间，而且可能导致优先级反转问题(priority inversion problem)
- 考虑两个进程H、L，H的优先级高于L，假设调度规则规定，只要H处于就绪态它就可以运行。若某一时刻L处于临界区中，此时H变成就绪态并被调度，从而开始忙等待，但是由于H的优先级高于L，使得L不会被调度，也就无法离开临界区

大纲

01 互斥和同步问题

02 临界区互斥问题的算法

03 信号量

04 经典IPC问题

05 管程

信号量的引入

- Peterson算法和硬件指令方法是正确的算法，但是都具有忙等待的缺点
- 它们是平等进程间的一种协商机制
- 为此，需要一个地位高于进程的管理者来解决公有资源的使用问题。OS可从进程管理者的角度来处理互斥和同步的问题
- 1965年荷兰学者Dijkstra提出了信号量概念，这是一种卓有成效的进程互斥和同步机制，成为OS提供的管理临界资源的有效手段

信号量

- 每个信号量s包括一个整数值s.count(计数），以及一个进程等待队列s.queue，队列中是阻塞在该信号量上的各个进程的标识
- 信号量只能通过初始化和两个标准的原语来访问——作为OS核心代码执行，不受进程调度的打断

信号量计数值的含义

- s.count>0表示有count个资源可用
- s.count=0表示无资源可用
- s.count<0则|count|表示s等待队列中的进程个数
- 以上意义的信号量也称为资源信号量
- 如果不需要信号量的计数能力，只用于对临界资源的互斥访问，这样的信号量也称为互斥信号量

P、V原语

- P、V来自荷兰语的proberen(降低)和verhogen(升起)
- P操作表示申请一个资源，它使信号量之值减1，如果信号量之值小于0，则进程被阻塞而不能进入临界区。P操作也称为wait操作、down操作
- V操作表示释放一个资源，它使信号量之值增1，如果信号量之值不大于0，则唤醒一个被阻塞的进程。V操作也称为signal操作、up操作

P、V原语

■P原语

```
P(s)
{
        --s.count;                  //表示申请一个资源;
        if (s.count <0)             //表示没有空闲资源;
        {
                调用进程进入等待队列s.queue;
                阻塞调用进程;
        }
}
```

P、V原语

■V原语

```
V(s)
{
        ++s.count;              //表示释放一个资源;
        if (s.count <= 0)       //表示有进程处于阻塞状态;
        {
                从等待队列s.queue中取出一个进程P;
                进程P进入就绪队列;
        }
}
```

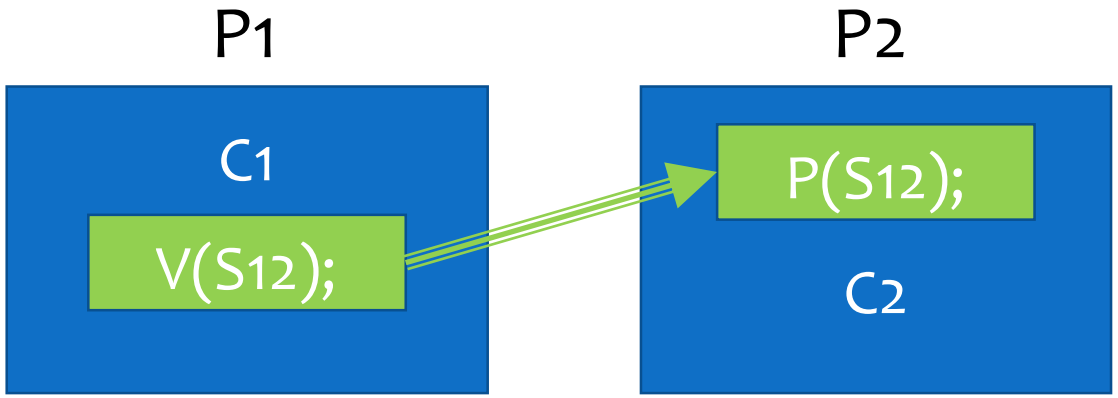
利用信号量实现互斥

- 为临界资源设置一个互斥信号量mutex(MUTual Exclusion)，初值为1
- 在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间

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

利用信号量实现同步

- 并发执行的进程P1和P2中，分别有代码C1和C2，要求C1在C2开始前完成——前趋关系
- 为每个前趋关系设置一个互斥信号量S12，其初值为0



P、V原语的使用要点

- 必须成对使用P和V原语，不能重复或遗漏
  - 遗漏P原语则不能保证互斥访问
  - 遗漏V原语则不能在使用临界资源之后将其释放(给其他等待的进程)
- 当为互斥操作时， P和V原语同处于同一进程
- 当为同步操作时， P和V原语不在同一进程中出现

P、V原语的使用要点

- P、V原语不能次序错误
- 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要，一个同步P操作与一个互斥P操作在一起时同步P操作必须在互斥P操作前（为什么？）
- 两个V操作顺序无关紧要

信号量的应用

- 订票系统——引入互斥信号量mutex实现对临界资源x的互斥访问，初值为mutex=1

```
T1:                T2:                          Tn:
...                ...                          ...
P(mutex);          P(mutex);                    P(mutex);
read(x);           read(x);                     read(x);
if x>=1 then       if >=1 then        ...       if x>=1 then
    x:=x-1;            x:=x-1;                      x:=x-1;
write(x);          write(x);                    write(x);
V(mutex);          V(mutex);                    V(mutex);
...                ...                          ...
```

信号量的应用

- 列车行车问题——设两个信号量S和C，初值为S = 0；C = 0

```
司机
while(true) {
        行车
        到站停车
        V(S)
        P(C)
        启动开车
 }
```

```
列车员
while(true) {
        服务
        P(S)
        开车门
        关车门
        V(C)
}
```

信号量的应用

- 3个进程get、process和print协作完成事务处理
- 设置4个信号量empty1、 empty2、full1、full2， empty1、 empty2表示Buf1、Buf2是否为空，初值为1， full1、full2表示Buf1、Buf2是否有记录，初值为0

get进程
```
while(true) {
        从磁带读取一条记录
        P(empty1)
        将记录存入Buf1
        V(full1)
 }
```

process进程
```
while(true) {
        P(full1)
        从Buf1读取一条记录
        V(empty1)
        对记录进行处理
        P(empty2)
        将处理结果记录存入Buf2
        V(full2)
 }
```

print进程
```
while(true) {
        P(full2)
        从Buf2读取一条记录
        V(empty2)
        打印记录
}
```

大纲

01 互斥和同步问题

02 临界区互斥问题的算法

03 信号量

04 经典IPC问题

05 管程

经典IPC问题

- IPC（Inter-Process Communication，进程间通信），解决进程间竞争共享资源引起的同步-互斥问题
- 经典IPC问题：
  - 生产者-消费者问题
  - 读者-写者问题
  - 哲学家进餐问题
  - 睡眠理发师问题

生产者-消费者问题

- 生产者-消费者问题(the producer-consumer problem)
- 若干进程通过有限的共享缓冲区交换数据。其中，生产者进程不断写入，而消费者进程不断读出
- 共享缓冲区共有n个数据单元
- 任何时刻只能有一个进程可对共享缓冲区进行操作

生产者-消费者问题

- 信号量：full是“满”数目，初值为0，empty是“空”数目，初值为n，mutex用于访问缓冲区时的互斥，初值是1
- full和empty存在关系：full + empty = n

Producer

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

Consumer

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

生产者-消费者问题

- 每个进程中各个P操作的次序是重要的：先检查资源数目，再检查是否互斥—否则可能死锁

Producer

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

Consumer

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

读者-写者问题

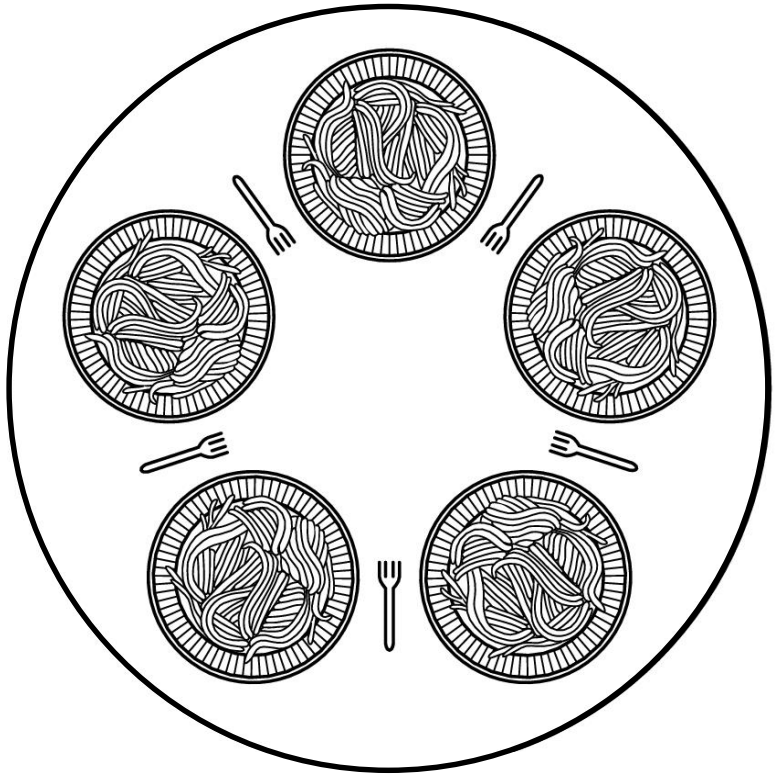
- 读者-写者问题(the readers-writers problem)
- 有两组并发进程：读者和写者，共享一组数据区，要求：
  - 允许多个读者同时执行读操作
  - 不允许读者、写者同时操作
  - 不允许多个写者同时操作

读者-写者问题

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                    /* repeat forever */
            down(&mutex);               /* get exclusive access to 'rc' */
            rc = rc + 1;                /* one reader more now */
            if (rc == 1) down(&db);     /* if this is the first reader ... */
            up(&mutex);                 /* release exclusive access to 'rc' */
            read_data_base( );          /* access the data */
            down(&mutex);               /* get exclusive access to 'rc' */
            rc = rc - 1;                /* one reader fewer now */
            if (rc == 0) up(&db);       /* if this is the last reader ... */
            up(&mutex);                 /* release exclusive access to 'rc' */
            use_data_read( );           /* noncritical region */
      }
}


void writer(void)
{
      while (TRUE) {                    /* repeat forever */
            think_up_data( );           /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base( );         /* update the data */
            up(&db);                    /* release exclusive access */
      }
}
```

哲学家进餐问题

- 哲学家进餐问题(the dining philosophers problem) 由Dijkstra首先提出并解决
- 5个哲学家围绕一张圆桌而坐，桌子上放着5把叉子，每两个哲学家之间放一支；哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两把叉子，思考时则同时将两把叉子放回原处。如何保证哲学家们的动作有序进行?



哲学家进餐问题

- 一种错误算法

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

哲学家进餐问题

- 正确算法

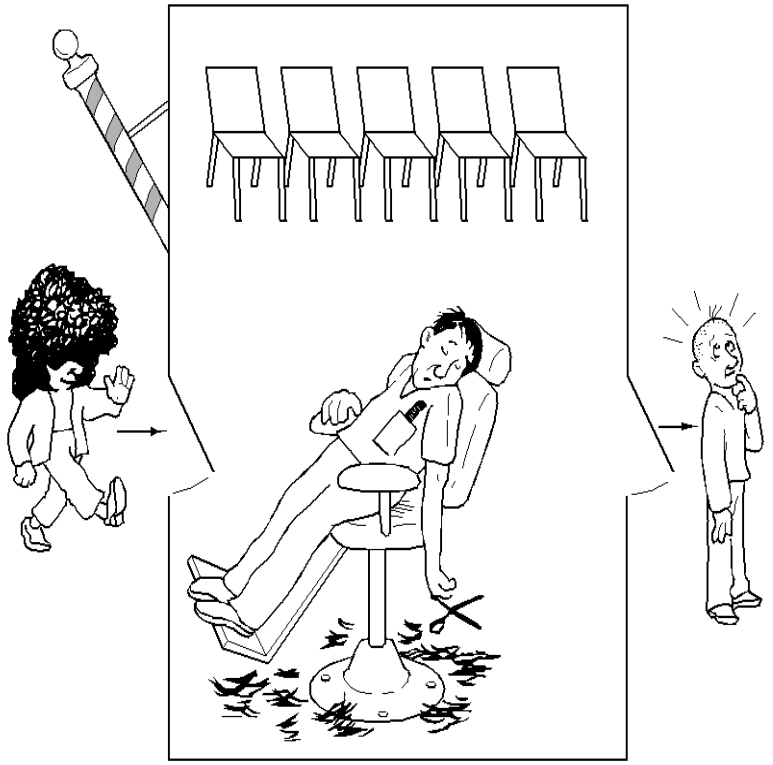
```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N–1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}
```

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

睡眠理发师问题

■睡眠理发师问题(The Sleeping Barber Problem)

- ■理发店里有一位理发师，一把理发椅和N把供等候理发的顾客坐的椅子
- ■如果没有顾客，则理发师便在理发椅上睡觉。当一个顾客到来时，他必须先唤醒理发师
- ■如果顾客到来时理发师正在理发，则如果有空椅子，可坐下来等；否则离开



睡眠理发师问题

```
#define CHAIRS 5                        /* # chairs for waiting customers */

typedef int semaphore;                  /* use your imagination */

semaphore customers = 0;                /* # of customers waiting for service */
semaphore barbers = 0;                  /* # of barbers waiting for customers */
semaphore mutex = 1;                    /* for mutual exclusion */
int waiting = 0;                        /* customers are waiting (not being cut) */

void barber(void)
{
      while (TRUE) {
            down(&customers);           /* go to sleep if # of customers is 0 */
            down(&mutex);               /* acquire access to 'waiting' */
            waiting = waiting - 1;      /* decrement count of waiting customers */
            up(&barbers);               /* one barber is now ready to cut hair */
            up(&mutex);                 /* release 'waiting' */
            cut_hair( );                /* cut hair (outside critical region) */
      }
}


void customer(void)
{
      down(&mutex);                     /* enter critical region */
      if (waiting < CHAIRS) {           /* if there are no free chairs, leave */
            waiting = waiting + 1;      /* increment count of waiting customers */
            up(&customers);             /* wake up barber if necessary */
            up(&mutex);                 /* release access to 'waiting' */
            down(&barbers);             /* go to sleep if # of free barbers is 0 */
            get_haircut( );             /* be seated and be serviced */
      } else {
            up(&mutex);                 /* shop is full; do not wait */
      }
}
```

大纲

01 互斥和同步问题

02 临界区互斥问题的算法

03 信号量

04 经典IPC问题

05 管程

信号量的缺点

- 用信号量可实现进程间的同步和互斥，但由于信号量的控制分布在整个程序中，其正确性分析很困难
  - 同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁(如P、V操作的次序错误、重复或遗漏)
  - 易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统中并发执行的各个程序
  - 不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局
  - 正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误

管程的引入

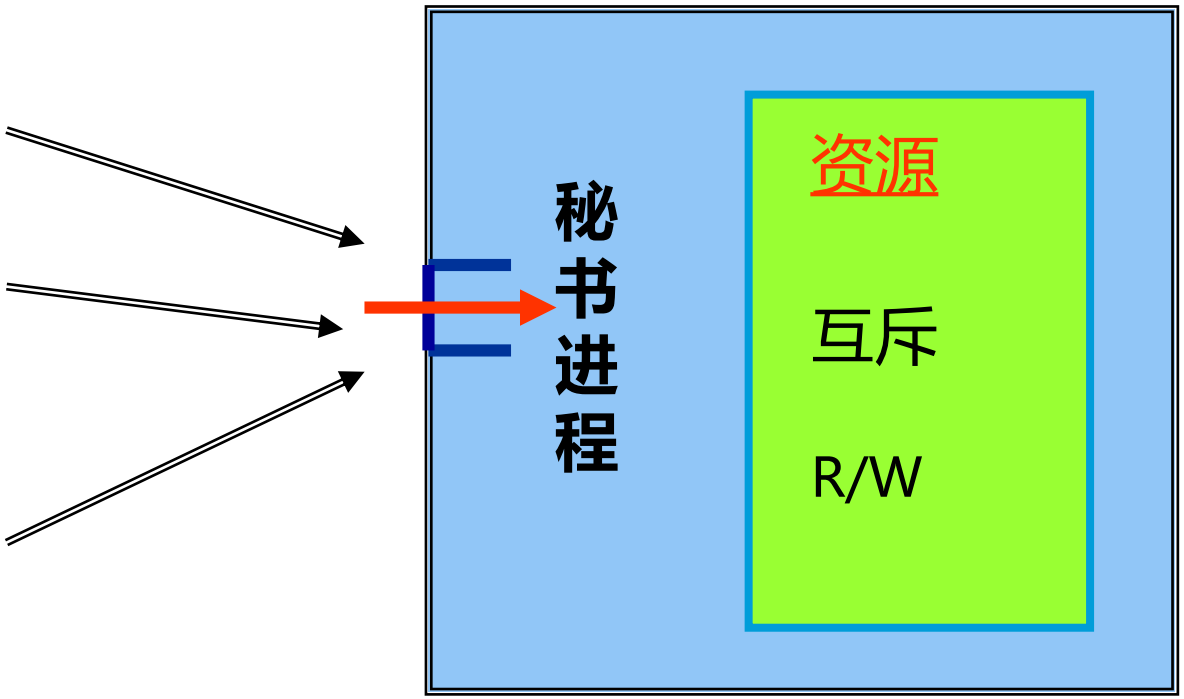
- 1971年，Dijkstra提出“秘书”进程的思想
- “秘书”每次仅让一个进程来访，这样既便于对共享资源的管理，又实现了互斥访问

进程1请求使用共享资源

进程2请求使用共享资源

……

进程n请求使用共享资源



管程的引入

- 1973年，Hansen和Hoare把Dijkstra的“秘书”进程的思想推广为“管程”
- 管程的基本思想是把信号量及其操作原语封装在一个对象内部。即：将共享变量以及对共享变量能够进行的所有操作集中在一个模块中
- 管程的定义：管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块
- 引入管程可提高代码的可读性，便于修改和维护，正确性易于保证

管程的主要特性

- 模块化：一个管程是一个基本程序单位，可以单独编译
- 抽象数据类型：管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码
- 信息封装：进程可调用管程中实现的某些过程(函数)，至于这些过程是怎样实现的，在其外部则是不可见的
- 管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的过程(函数)来间接地访问管程中的共享变量

管程的格式

- 类Pascal语言描述的管程

管程名：example
共享变量：i
条件变量：c
管程过程：producer()、comsumer()

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

管程互斥进入

- 管程是编程语言的组成部分，编译器采用与其他过程调用不同的方法处理对管程的调用
- 典型的处理方法：当一个进程调用管程过程时，该过程将检查在管程中是否有其他活跃进程。如果有，调用进程将被阻塞，直到另一个进程离开管程而将其唤醒；如果没有，则该调用进程可以进入
- 因为管程是互斥进入的，所以当一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待，因而在管程的入口处应当有一个进程等待队列，称作入口等待队列

资源等待队列

- 管程是用于管理资源的，当进入管程的进程因资源被占用等原因不能继续运行时应当释放管程的互斥权，即将等待资源的进程加入资源等待队列
- 资源等待队列可以由多个，每种资源一个队列
- 资源等待队列由条件变量维护
- 条件变量(condition variables)是管程内的一种数据结构，且只有在管程中才能被访问，它对管程内的所有过程是全局的，只能通过wait和signal两个原语操作来控制

条件变量

- wait(c)：调用进程阻塞并移入与条件变量c相关的等待队列中，并释放管程，直到另一个进程在该条件变量c上执行signal( )唤醒等待进程并将其移出条件变量c队列
- signal(c)：如果c链为空，则相当于空操作，执行此操作的进程继续；否则唤醒c链中的第一个等待者

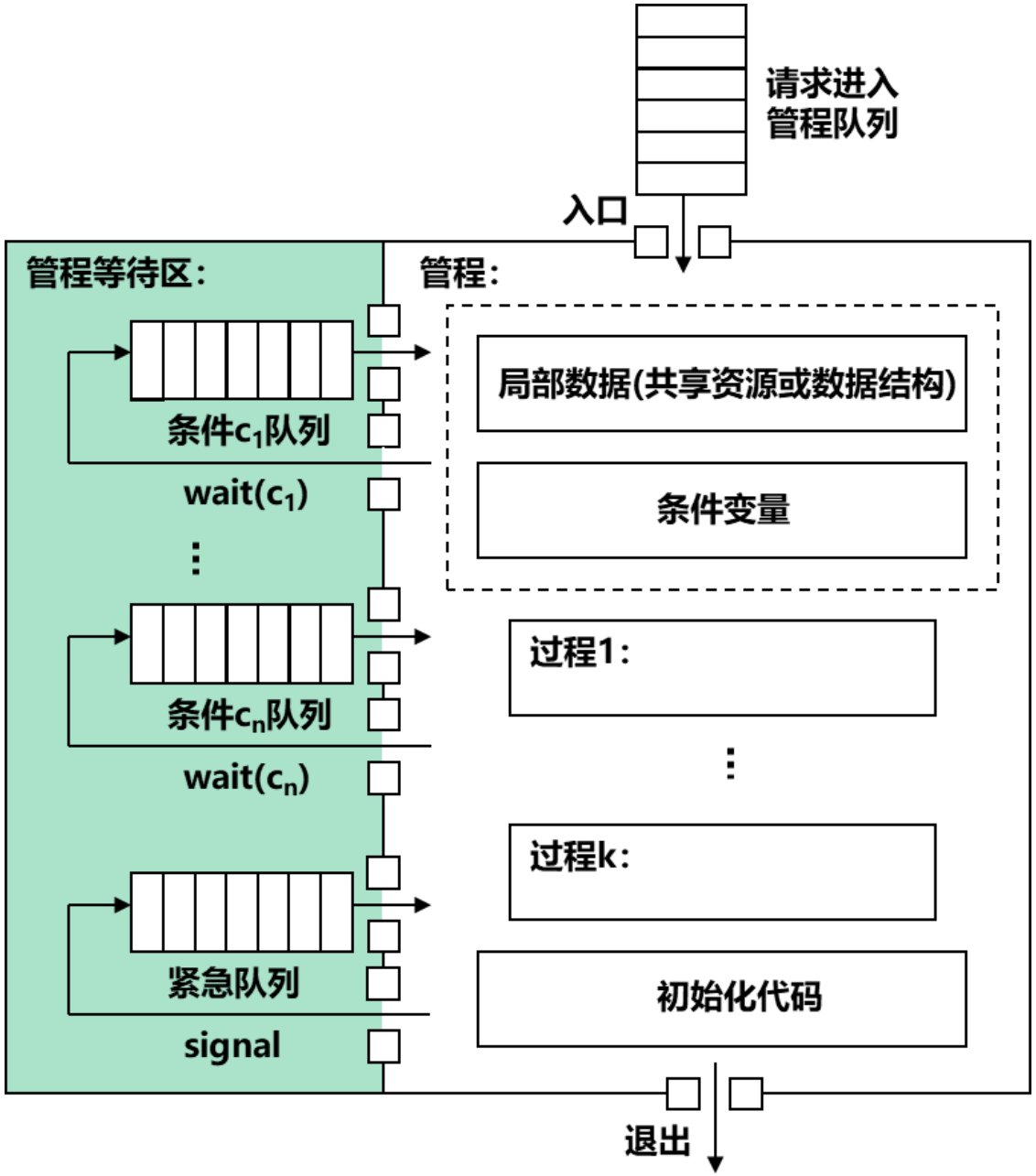
紧急等待队列

- 当一个进入管程的进程执行唤醒操作时(如P唤醒Q)，管程中便存在两个同时处于活动状态的进程
- 处理方法有两种：
  - P等待，Q继续，直到Q退出或等待(Hoare)
  - 规定唤醒为管程中最后一个可执行的操作(Hansen)

紧急等待队列

- 假设进程P唤醒进程Q，则P等待Q继续。如果进程Q在执行时又唤醒进程R，则Q等待R继续，……，于是，在管程内部，由于执行唤醒操作，可能会出现多个等待进程，因而还需要有一个进程等待队列，这个等待队列被称为紧急等待队列

管程的结构


请求进入
管程队列
入口
管程等待区:
管程:
局部数据(共享资源或数据结构)
条件c_1队列
wait(c_1)
条件变量
条件c_n队列
wait(c_n)
过程1:
过程k:
紧急队列
signal
初始化代码
退出

管程的缺点

- 管程是一个编程语言概念，编译器必须要识别管程并用某种方式对其互斥做出安排
- Java、Pascal 、 Modula-2等语言支持管程
- C、C++及多数程序设计语言都不支持管程
- C、C++及多数程序设计语言也没有信号量，但是支持信号量十分容易——编译器甚至不需知道它们的存在，信号量是OS机制，系统只需提供相应系统调用即可

用管程实现生产者-消费者问题

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

谢谢