

进程并发控制： 信号量的应用

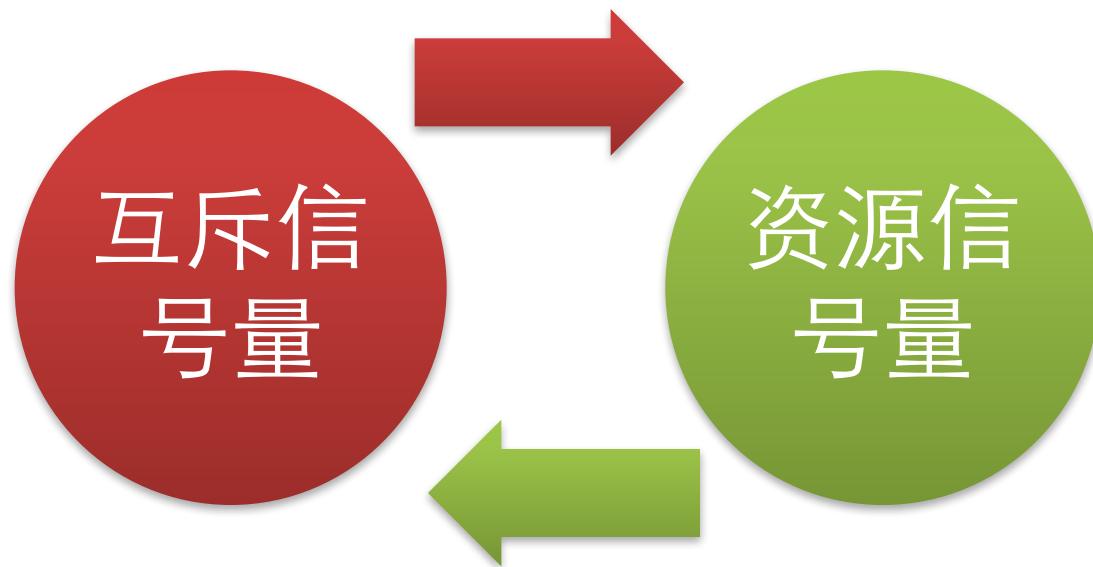
薛瑞尼

计算机科学与工程学院

16/3/22

信号量类型

Mutex



Binary semaphore

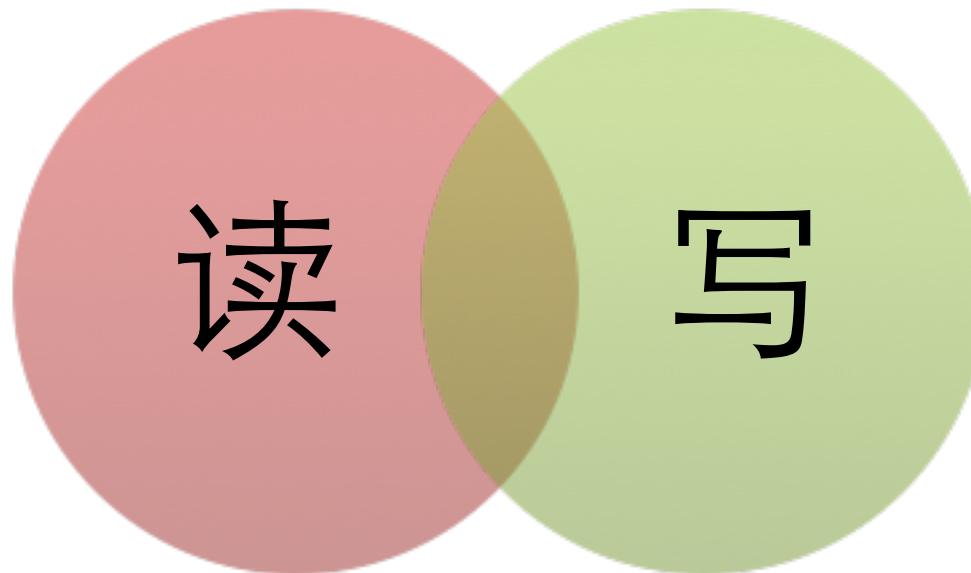
General semaphore

资源信号量

Counting semaphore

临界资源的独享访问

- Data race：数据竞争
- Race condition



对共享变量的访问

- 观察者和报告者是两个并发执行的进程。
- 观察者不断观察并对通过的卡车计数；
- 报告者不停地将观察者的计数打印，并归零。
- 请用P、V原语进行正确描述。

```
int count := 0;  
Semaphore mutex := 1;
```



类型、初值

Process observer

```
Begin  
do  
    观察车辆;  
    P(mutex);  
    count:=count+1;  
    V(mutex);  
    while (true);  
End;
```

Process reporter

```
Begin  
do  
    P(mutex);  
    print count;  
    count := 0;  
    V(mutex)  
    while (true);  
End;
```



P/V操作清晰
不必拘泥语法

图书馆问题

- 图书馆有N个座位，一张登记表，要求：
 - 读者进入时需先登记，取得座位号；
 - 出来时注销
- 用P、V原语描述读者的使用过程。

信号量mutex：登记表互斥控制，初值为1；

```
int SN = N  
  
semaphore mutex = 1  
  
Reader()  
    Enter();  
    阅读;  
    Exit()  
End
```

```
Enter()  
Begin  
repeat  
    local OK = False  
    P(mutex);  
    if (SN > 0) {  
        SN--;  
        OK = True;}  
    V(mutex);  
    until OK  
End
```

```
Exit()  
Begin  
P(mutex);  
SN++;  
V(mutex);  
End
```

Semaphore mutex = 1: 登记表互斥控制；

Semaphore SN = N: 可用座位数；

Reader()

BEGIN

Enter();

阅读；

Exit()

End

Enter()

Begin

P(SN);

P(mutex);

登记；

V(mutex);

End

调换？

Exit()

Begin

P(mutex);

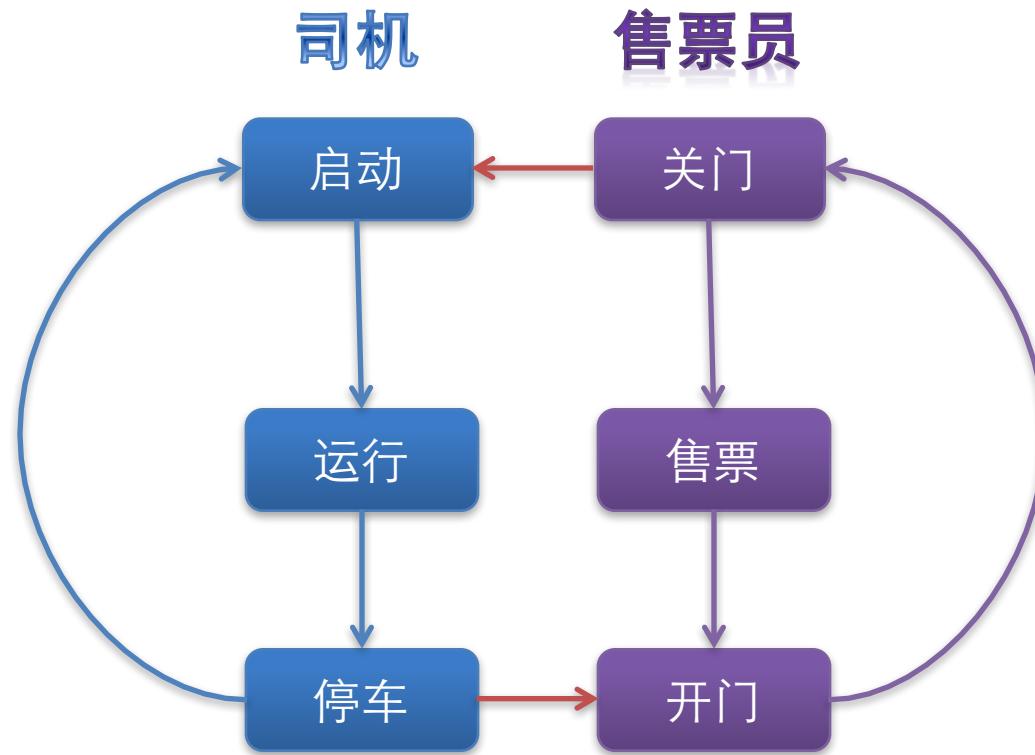
注销；

V(mutex);

V(SN);

End

公交车场景



- Semaphore S1=0: 是否允许司机启动汽车;
- Semaphore S2=0: 是否允许售票员开门;

Driver()

```
While (1){
    P(S1);
    启动汽车;
    正常行车;
    到站停车;
    V(S2);
}
```

Conductor()

```
While (1){
    关车门;
    V(S1);
    售票;
    P(S2);
    开车门;
}
```

sem S1=0;

sem S2=0;

Main{

Cobegin

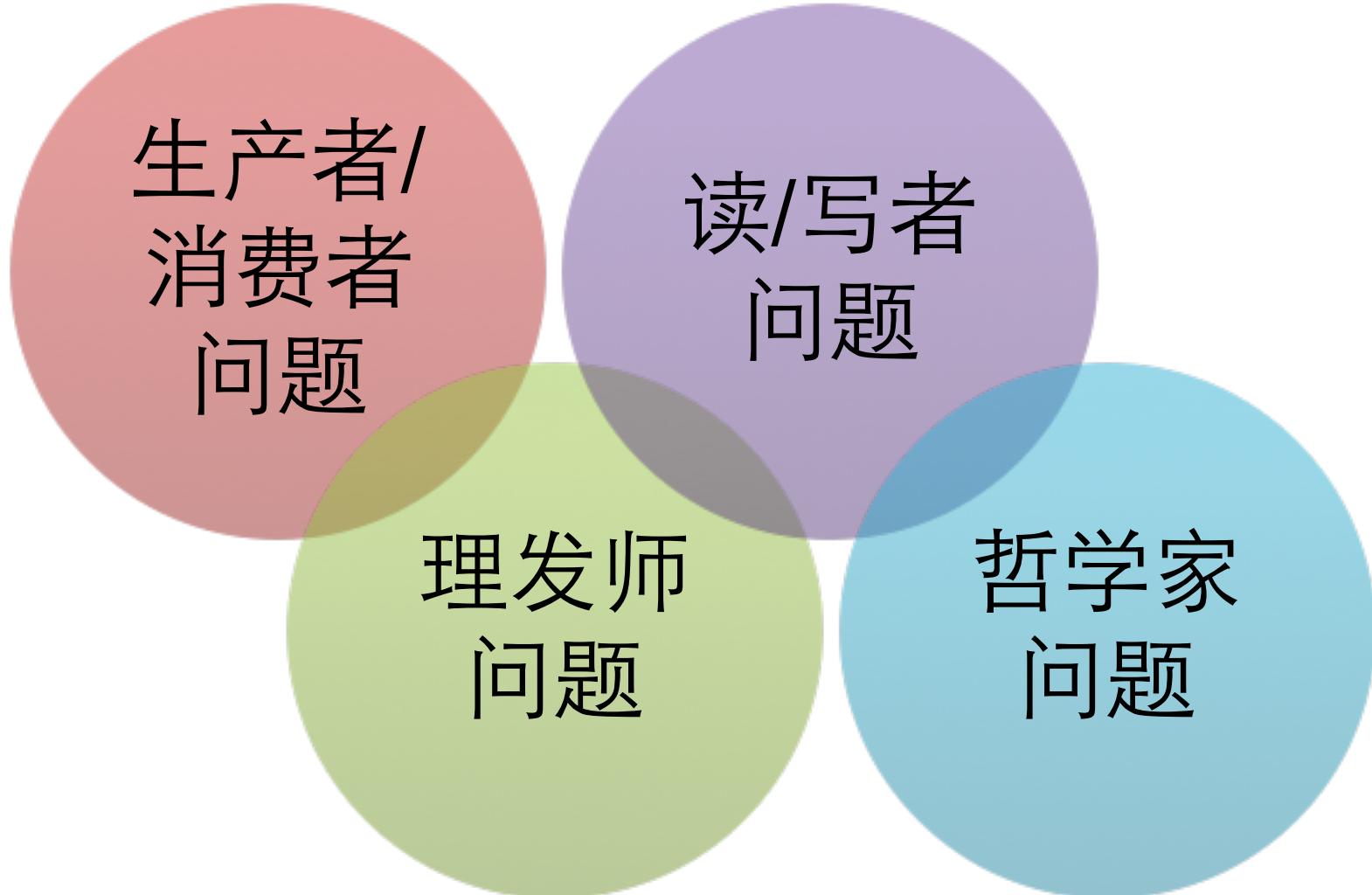
Driver();

Busman();

Coend

}

经典同步问题



生产者与消费者问题

- 荷兰计算机科学家E. W. Dijkstra把广义同步问题抽象成一种“生产者与消费者问题”
 - Producer Consumer Problem
 - Bounded-Buffer Problem

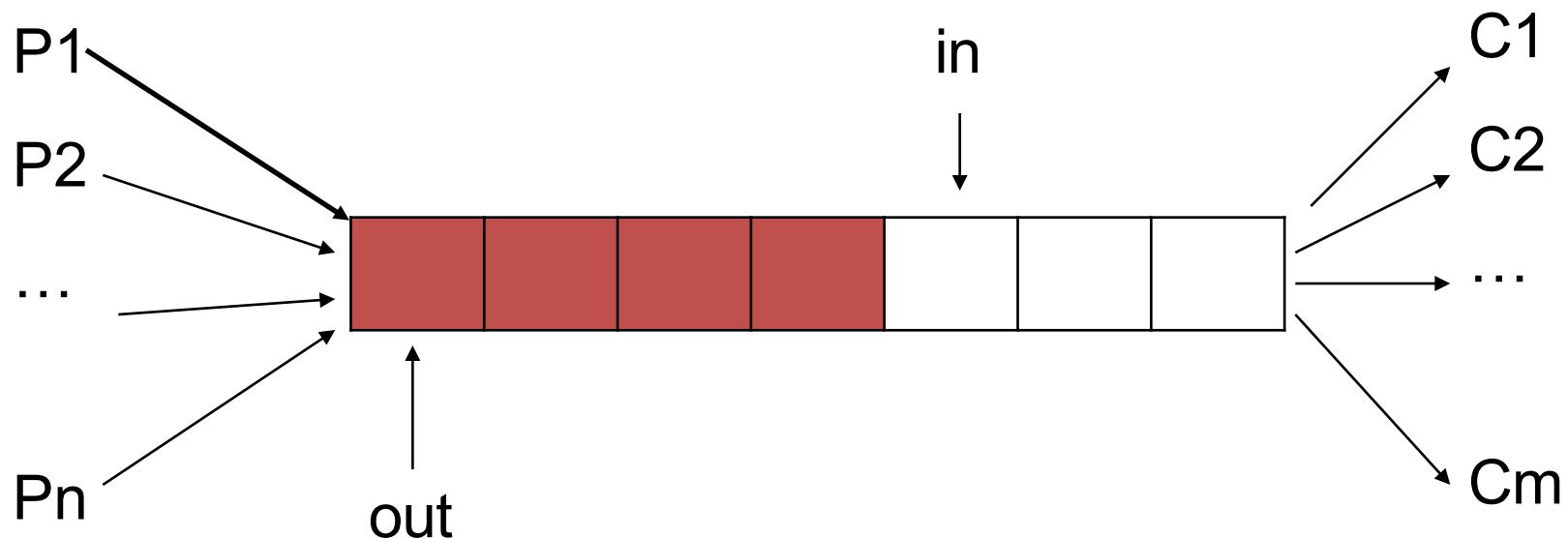
生产者/消费者模型

- 生产者：满则等待，空则填充
- 消费者：空则等待，有则获取
- 不允许同时进入缓冲区

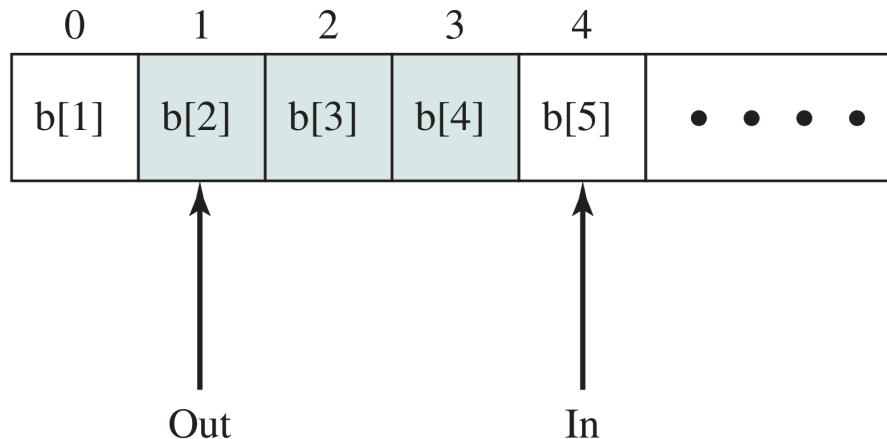


抽象模型

- 多个生产者，多个消费者
- 一个共享缓冲区



无限缓冲 (Infinite Buffer)



Note: Shaded area indicates portion of buffer that is occupied

```
while (true) {  
    produce item;  
    buffer[in] = item;  
    in = in + 1;  
}
```

生产者

```
while (true) {  
    while (in <= out)  
        /* do nothing */;  
    item = buffer[out];  
    out = out + 1;  
    consume item;  
}
```

消费者

方案

```
semaphore num = 0;  
semaphore mutex = 1;
```

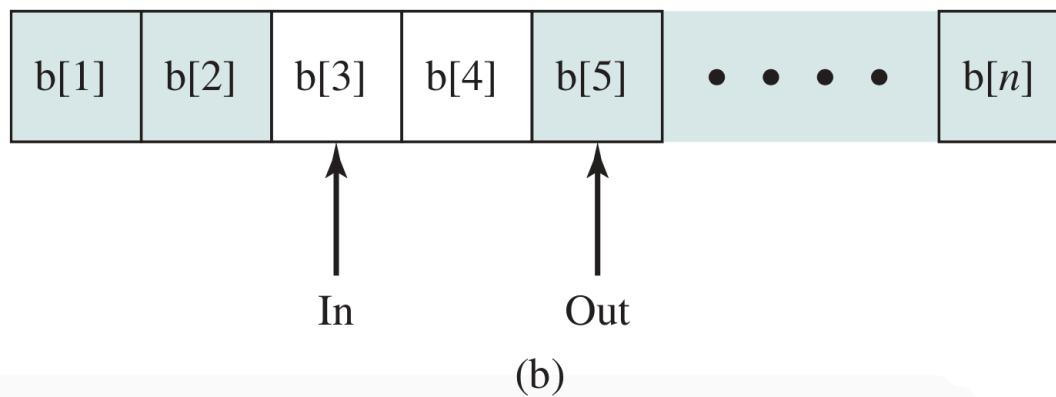
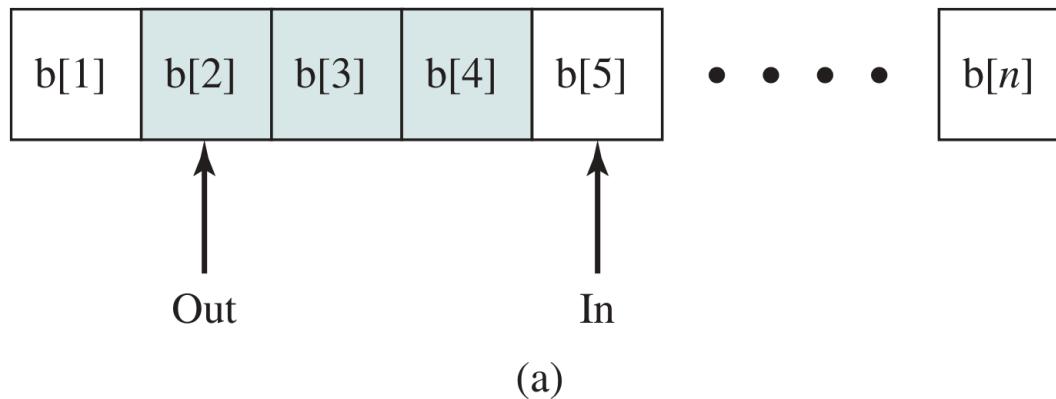
```
while (true) {  
    produce();  
    P(mutex);  
    填充buffer;  
    V(mutex);  
    V(num);  
}
```

生产者

```
while (true) {  
    P(num);  
    P(mutex);  
    消费buffer;  
    V(mutex);  
    consume();  
}
```

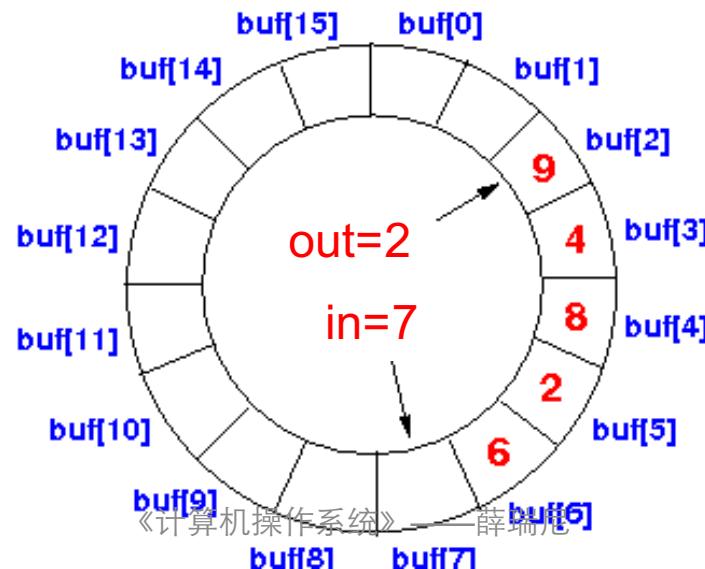
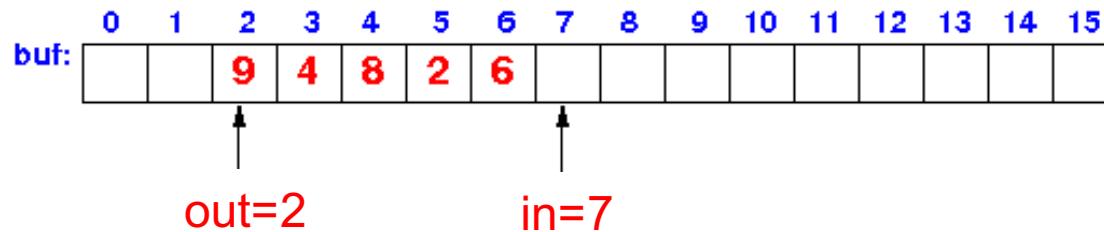
消费者

有限循环/环形缓冲区



有限循环/环形缓冲区

- 缓冲区被看作一个循环缓冲区
- 指针表达为按缓冲区的大小取模



有限循环/环形缓冲区

```
while (true) {  
    produce item;  
    while ((in + 1) % n == out)  
        /* do nothing */;  
    buffer[in] = item;  
    in = (in + 1) mod N;  
}
```

生产者

信号量避免忙等

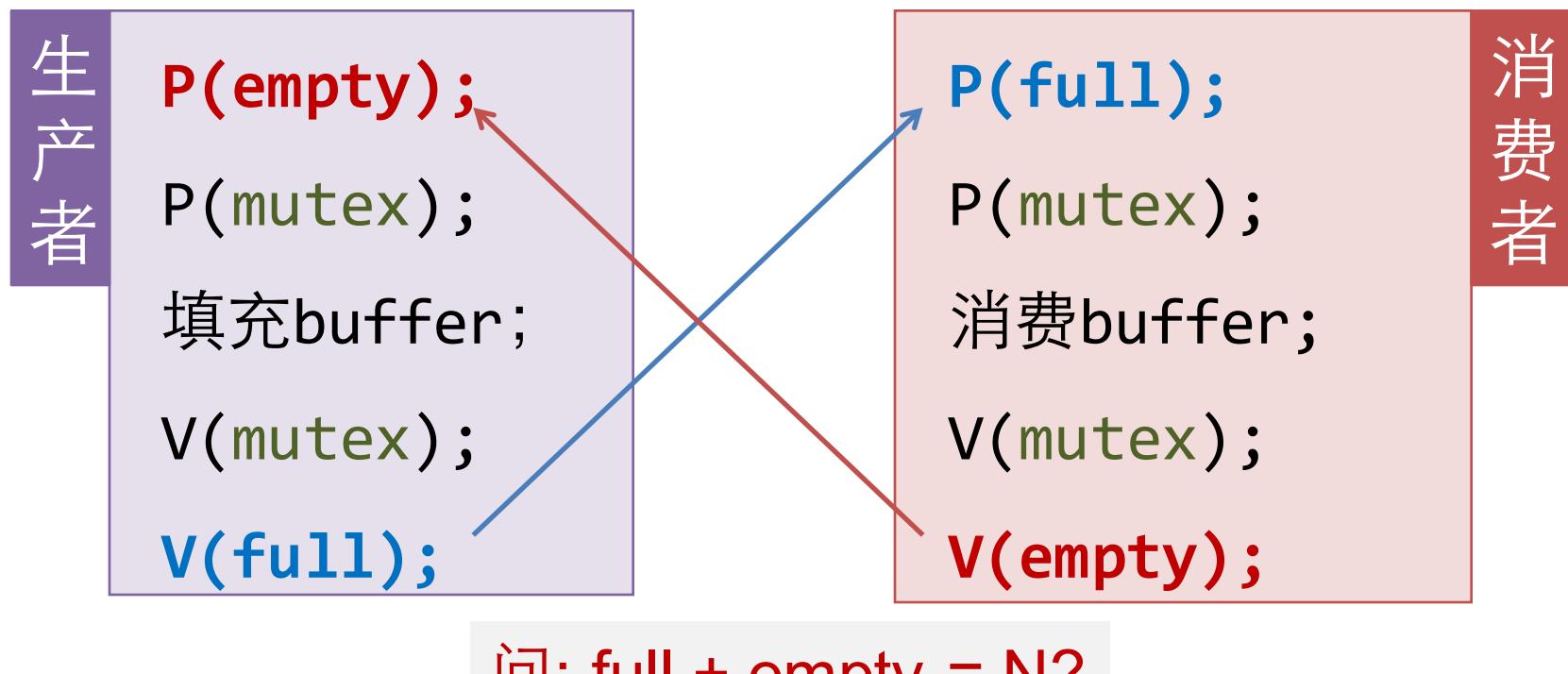
信号量实现同步

```
while (true) {  
    while (in == out)  
        /* do nothing */;  
    item = buffer(out);  
    out = (out + 1) mod N;  
    consume item;  
}
```

消费者

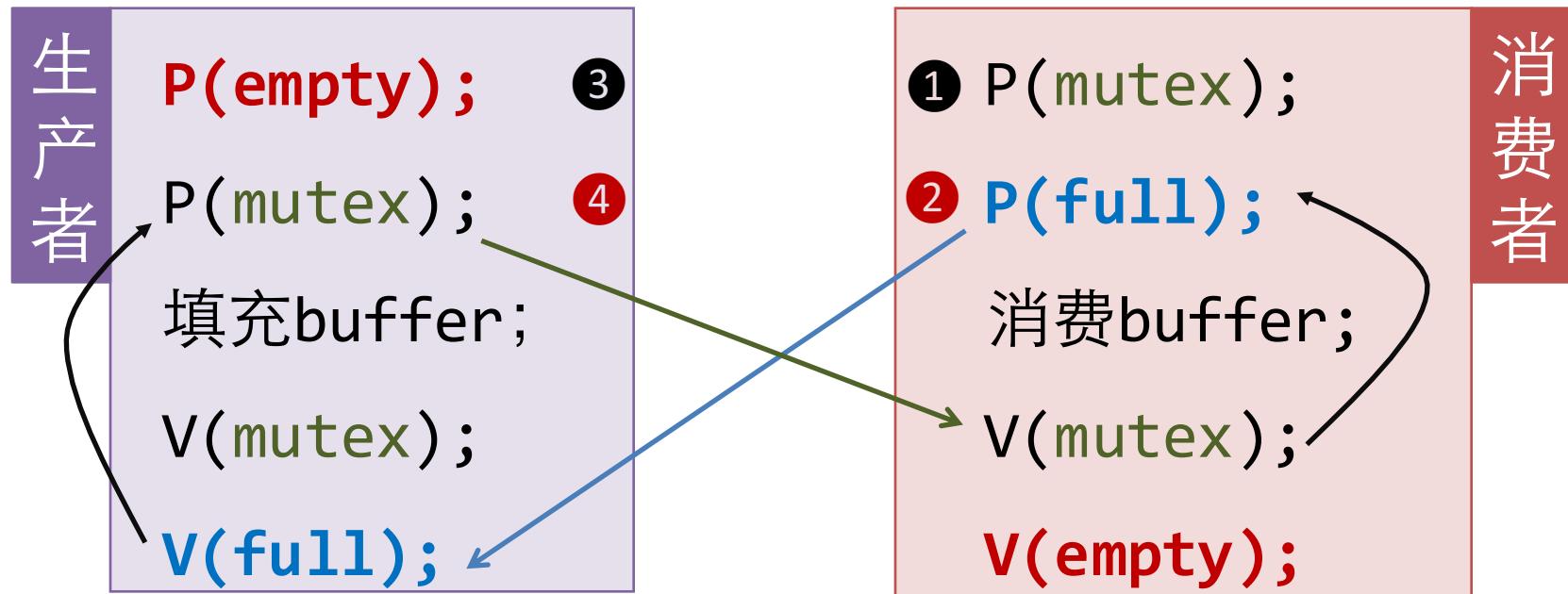
有限循环P/V操作

- semaphore full = 0 → “满” 缓冲区数量
- semaphore empty = N → “空” 缓冲区数量
- semaphore mutex = 1 → 互斥访问问缓冲区



P操作顺序与死锁

若缓冲区为空，按照以下顺序执行：



问: V操作顺序是否会导致死锁?

基于管程的Producer/Consumer

```
void main() {  
    parbegin  
        producer(); consumer();  
    paren; } }
```

```
void producer( ) {  
    while(1) {  
        produce an item;  
        PC.put(item);  
    } } }
```

```
void consumer( ) {  
    while(1){  
        item = PC.get();  
        consume the item;  
    } } }
```

生产者消费者问题之管程解决

```
monitor PC { //管程本身实现了互斥
```

```
    int in=0, out=0, count=0;  
    item buffer[n];  
    condition empty, full;
```

```
put(item) {  
    if (count >= n)  
        full.wait();  
  
    buffer[in]=item;  
    in =(in + 1) % n;  
    count++;  
  
    if (empty.queue)  
        empty.signal();  
}
```

```
get() {  
    if (count <= 0)  
        empty.wait();  
  
    item = buffer[out];  
    out =(out + 1) % n;  
    count--;  
  
    if (full.queue)  
        full.signal();  
  
    return item;  
}
```

思考&练习

- 生产者消费者模型的效率是不是最高的?
 - 如何控制生产者，消费者同时进入？

例：与进程的执行顺序有关的问题

- 有3个进程PA，PB和PC合作解决文件打印问题：
 - PA将文件记录从磁盘读入主存的缓冲区1，每执行一次读一个记录；
 - PB将缓冲区1的内容复制到缓冲区2，每执行一次复制一个记录；
 - PC将缓冲区2的内容打印出来，每执行一次打印一个记录。缓冲区的大小等于一个记录大小。
 - 请用P，V操作来保证文件的正确打印。



信号量

- `empty1, empty2`: 分别表示缓冲区1及缓冲区2是否为空，初值为1。
- `full1, full2`: 分别表示缓冲区1及缓冲区2是否有记录可供处理，初值为0。
- `mutex1, mutex2`: 分别表示缓冲区1及缓冲区2的访问控制，初值为1。

PA() {

while (1) {

 从磁盘读一个记录；

 P(empty1); P(mutex1);

 将记录存入缓冲区1；

 V(mutex1); V(full1); } }

PB() {

while (1) {

 P(full1); P(mutex1);

 P(empty2); P(mutex2);

 从缓冲区1中取出记录放去缓冲区2；

 V(mutex1); V(empty1);

 V(mutex2); V(full2); } }

PC() {

while (1) {

 P(full2); P(mutex2);

 从缓冲区2取一个记录；

 P(mutex2); V(empty2);

 打印记录；

} }

sem empty1=1; empty2=1;

full1=0; full2=0;

mutex1=1; mutex2=1;

Main() {

parbegin

 PA(); PB(); PC();

parend }

思考

- 生产者/消费者问题的启示
 - 资源数量：资源信号量
 - 资源访问：互斥信号量
 - 先申请资源，再申请访问权
 - 资源信号量P、V操作分布在不同进程
 - 互斥信号量P、V操作出现在同一进程

读者/写者问题

- Readers-Writers Problem
- 三个角色
 - 一个共享的数据区；
 - Reader: 只读取这个数据区的进程；
 - Write: 只往数据区中写数据的进程；
- 三个条件
 - 多个Reader可同时读数据区；
 - 一次只有一个Writer可以往数据区写；
 - 数据区不允许同时读写。

同步

- “读 - 写” 互斥
- “写 - 写” 互斥
- “读 - 读” 允许

思考

- 是不是普通的同步互斥问题?
 - 读者效率
- 是不是生产者消费者问题的扩展?
 - 生产者≠写者：没有empty
 - 消费者≠读者：没有full

读者优先

- 如有读者正在读数据，则允许多个读者同时进入读数据，只有当全部读者退出，才允许写者进入写数据。
- 读者插队
- 信号量
 - wsem：互斥信号量，用于Writers互斥Writers和 Readers，以及第一个Reader互斥Writers
 - readcount：统计同时读数据的Readers个数
 - mutex：对变量readcount互斥算术操作

正确性与进程顺序无关!

```
int readcount=0;
```

```
semaphore mutex=1, wsem=1;
```

```
void reader() {  
    while (1) {  
        P(mutex);  
        readcount++;  
        if (readcount == 1) P(wsem);  
        V(mutex);  
        READ;  
        P(mutex);  
        readcount--;  
        if (readcount == 0) V(wsem);  
        V(mutex);  
    }  
}
```

```
void writer() {  
    while (1) {  
        P(wsem);  
        WRITE;  
        V(wsem);  
    }  
}
```

序列:

1. R R R
2. W W W
3. R W
4. R R W
5. R W R R
6. W W R
7. W R R W

writer
饥饿

写者优先

- 只要有一个写者申请写数据，则不再允许新的读者进入读数据
- 写者会插队
- 信号量
 - rsem：信号量，当至少有一个写者申请写数据时互斥新的读者进入读数据
 - writecount：用于控制rsem信号量
 - mwc：信号量，控制对writecount的互斥加减操作

```
int readcount, writecount;  
semaphore mrc=1, mwc=1, wsem=1, rsem=1;
```

```
void reader( ) {  
    while (1) {  
        P(rsem);  
        P(mrc);  
        readcount++;  
        if (readcount == 1) P(wsem);  
        V(mrc);  
        V(rsem);  
        READ;  
        P(mrc);  
        readcount--;  
        if (readcount == 0) V(wsem);  
        V(mrc);  
    }  
}
```

```
void writer( ) {  
    while (1) {  
        P(mwc);  
        writecount++;  
        if (writecount == 1) P(rsem);  
        V(mwc);  
        P(wsem);  
        WRITE;  
        V(wsem);  
        P(mwc);  
        writecount--;  
        if (writecount == 0) V(rsem);  
        V(mwc);  
    }  
}
```

序列：

1. R R W R
2. W R R
3. W R R W
4. W R ... R W

```
int readcount, writecount;  
semaphore mrc=1, mwc=1, z=1, wsem=1, rsem=1;
```

```
void reader( ) {  
    while (1) {  
        P(z);  
        P(rsem);  
        P(mrc);  
        readcount++;  
        if (readcount == 1) P(wsem);  
        V(mrc);  
        V(rsem);  
        V(z);  
        READ;  
        P(mrc);  
        readcount--;  
        if (readcount == 0) V(wsem);  
        V(mrc);  
    }  
}
```

```
void writer( ) {  
    while (1) {  
        P(mwc);  
        writecount++;  
        if (writecount == 1) P(rsem);  
        V(mwc);  
        P(wsem);  
        WRITE;  
        V(wsem);  
        P(mwc);  
        writecount--;  
        if (writecount == 0) V(rsem);  
        V(mwc);  
    }  
}
```

4. WR...RW

思考

- 教材中描述 “ z 信号量的好处是：限制了 $rsem$ 阻塞队列的长度” , “无法唤醒全部阻塞读者”
- $P(z)$ 和 $P(rsem)$ 能否互换位置？
- Reader “饥饿” 的问题

进一步思考

- 如何实现公平的访问控制?
 - R, R, R, W, W, R, R, R, W, W...
 - 先来后到

方案1

```
int readcount = 0;  
semaphore mrc = 1, r = 1, w = 1;
```

```
READER {  
    P(r);  
    P(mrc);  
    readcount++;  
    if (readcount == 1) P(w);  
    V(mrc);  
    V(r);  
  
    Read();  
  
    P(mrc);  
    readcount--;  
    if (readcount == 0) V(w);  
    V(mrc);  
}
```

```
WRITER {  
    P(r);  
    P(w);  
  
    Write();  
  
    V(w);  
    V(r);  
}
```

方案2

```
semaphores: waiting=1, accessing=1, counter_mutex=1;  
shared variables: int nreaders = 0;  
local variables: int prev=0, current=0;
```

READER:

```
P(waiting);  
P(counter_mutex);  
    prev := nreaders;  
    nreaders := nreaders + 1;  
V(counter_mutex);  
if prev = 0 then P(accessing);  
V(waiting);  
... read ...  
P(counter_mutex);  
    nreaders := nreaders - 1;  
    current := nreaders;  
V(counter_mutex);  
if current = 0 then V(accessing);
```

WRITER:

```
P(waiting);  
P(accessing);  
V(waiting);  
    ... write ...  
V(accessing);
```

http://en.wikipedia.org/wiki/Readers-writers_problem

课外练习

- Binary Semaphore: 只能取[0, 1]
- 请用Binary Semaphore实现通用信号量

solution

```
Semaphore S1, S2, S3; // BINARY!!
int C = N; // N is # locks

down_c(sem) {
    downB(S3);
    downB(S1);
    C = C - 1;
    if (C<0) {
        upB(S1);
        downB(S2);
    }
    else {
        upB(S1);
    }
    upB(S3);
}

up_c(sem) {
    downB(S1);
    C = C + 1;
    if (C<=0) {
        upB(S2);
    }
    upB(S1);
}
```

理发师问题

- Sleeping Barber Problem / Barber Shop Problem
 - Edsger Dijkstra
- 角色和资源
 - 一个理发师
 - 一个理发椅
 - 一排座位
 - 随机到来的客户
- 场景
 - 理发师：有客干活，无客睡觉
 - 客户：唤醒理发师，有位等待，无位离开 **饥饿？死锁？**



信号量

```
/* # of customers waiting */  
semaphore customers = 0;  
  
/* barber status */  
semaphore barbers = 0;  
  
/* mutual exclusion to access seats */  
semaphore mutex = 1;  
  
/* # of available seats */  
int nas = N;
```

```
void barber(void) {  
    while (TRUE) {  
        P(customers);  
        P(mutex);  
        nas++;  
        V(barbers);  
        V(mutex);  
        cut_hair();  
    }  
}
```

```
void customer(void) {  
    P(mutex);  
    if (nas > 0) {  
        nas--;  
        V(customers);  
        V(mutex);  
        P(barbers);  
        get_haircut();  
    } else {  
        V(mutex);  
        leave_shop();  
    }  
}
```

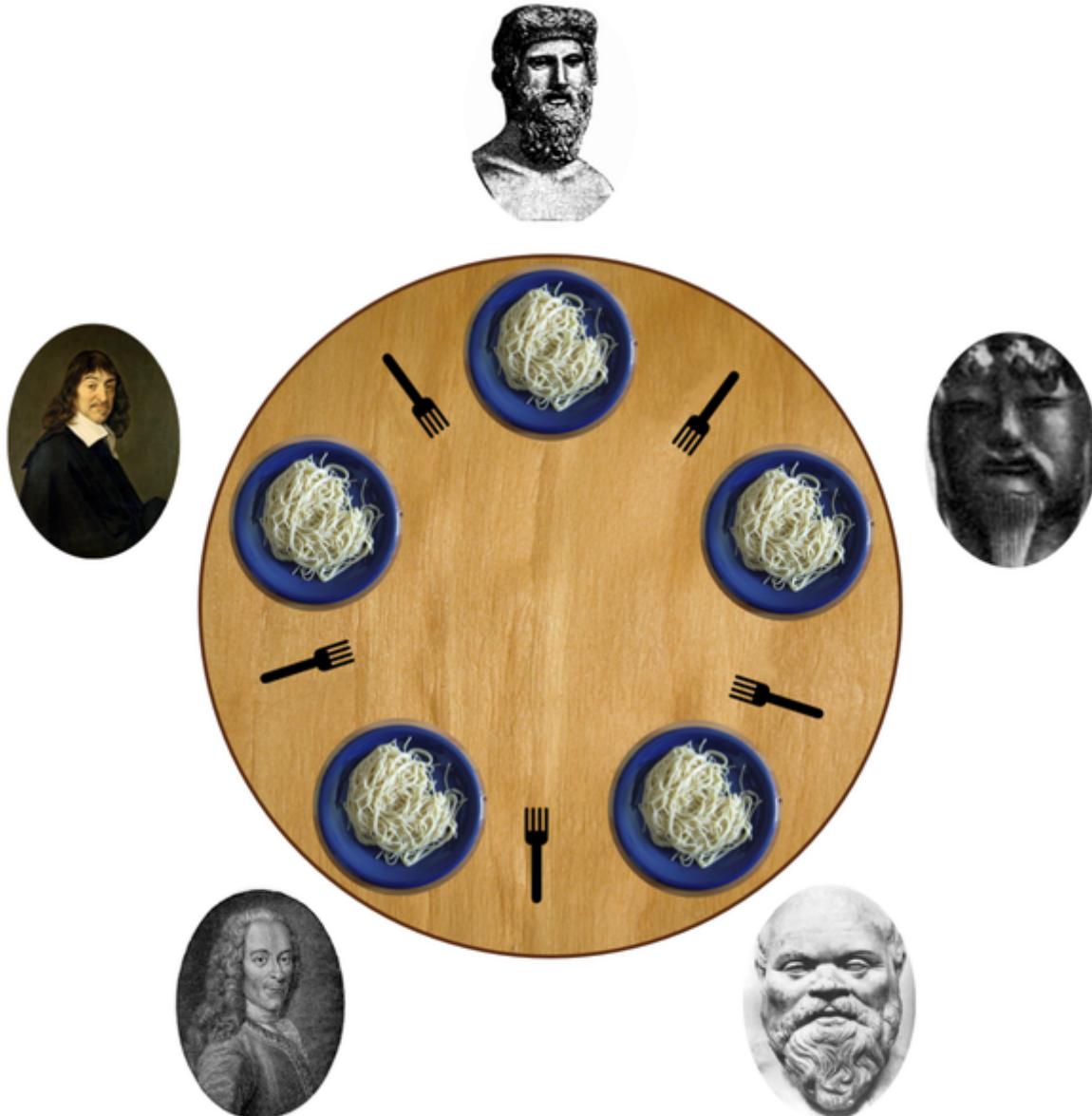
为什么不用nas做信号量，代替
customers?

思考

- 此算法有没有“饥饿”或“死锁”？
 - 无死锁，有饥饿
- 如何解决饥饿？
 - 用户排队
- 扩展理发师问题（包括收银员）
 - 见*Operating Systems Internals and Design Principles*

哲学家就餐问题

- Dinning Philosopher Problem
- 1965年，**Dijkstra**出了一道同步考试题：假设有五台计算机都试图访问五份共享的磁带驱动器。后来，这个问题被**Hoare**重新表述为哲学家就餐问题。这个问题可以用来解释死锁和资源耗尽。
- 描述
 - 5个哲学家围坐一张餐桌
 - 5只餐叉（筷子）间隔摆放
 - 思考或进餐
 - 进餐时必须同时拿到两边的餐叉
 - 思考时将餐叉放回原处



哲学家的生活1

repeat

思考；

取fork[i];

取fork[(i+1) mod 5];

进食；

放fork[i];

放fork[(i+1) mod 5];

forever;

哲学家的生活2

repeat

 思考；

 取fork[i];

 timeout(取fork[(i+1) mod 5], T);

 进食；

 放fork[i];

 放fork[(i+1) mod 5];

forever;

活锁

思考

- 大家有什么办法？

服务员方法

- Conductor/Waiter Solution
- 最多允许4个哲学家同时进食

```
program diningphilosophers;
    semaphore fork[5] = {1,1,1,1,1};
    semaphore room = 4;
    int i;
    Void philosopher (int i) {
        while (true) {
            thinking();
            P(room);
            P(fork[i]);
            P(fork[(i+1) mod 5])
            eating();
            V(fork[i]);
            V(fork[(i+1) mod 5])
            V(room);
        }
    }
}
```

资源分级方案1

- Resource hierarchy solution
- 为资源（餐叉）分配一个偏序（partial order）或者分级（hierarchy）的关系，所有资源都按此顺序获取，按相反顺序释放。
- 给哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之。

```
semaphore fork[5] = {1,1,1,1,1};  
int i;  
void philosopher(int i) {  
    while (1) {  
        think();  
        if(i%2 ==0) {  
            P(fork[i]);  
            P(fork [(i+1) % 5]);  
        } else {  
            P(fork[(i+1) % 5]);  
            P(fork[i]);  
        }  
        eat();  
        V(fork [(i+1) % 5]);  
        V(fork[i]);  
    }  
}
```

资源分级方案2

- 餐叉编号为1至5，每个哲学家总是先拿起左右两边编号较低的餐叉，再拿编号较高的。用完餐叉后，他总是先放下编号较高的餐叉，再放下编号较低的。
- 练习（5min）

管程解法

- 为了避免死锁，把哲学家分为三种状态：
 - 思考
 - 饥饿
 - 进食
- 如果取筷子，每次要么拿到两只筷子，要么一只不拿。

```
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
monitor philosopherMeal {
    int state[N]={0, 0, 0, 0, 0};
    condition self[N]={0, 0, 0, 0, 0};

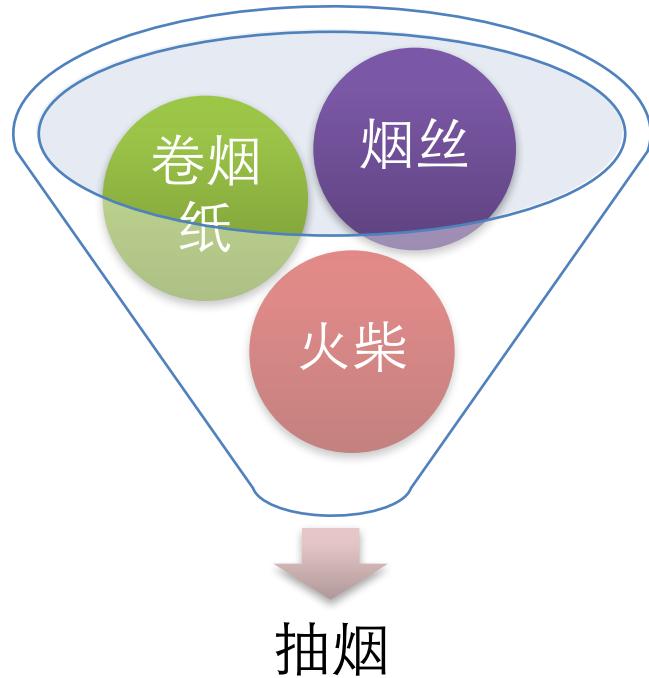
    void test(int i){
        if (state[i] == HUNGRY) &&
            (state[(i+4) % 5] != EATING) &&
            (state [(i+1) % 5] != EATING) {
            state[i] = EATING;
            self[i].signal;
        }
    }
}
```

```
void take_forks(int i){  
    state[i] = HUNGRY;  
    test(i);  
    if (state[i] != EATING)  
        self[i].wait;  
}  
  
void put_forks(int i) {  
    state[i] = THINKING;  
    test((i+4)%N);  
    test((i+1)%N);  
}  
} // end of monitor
```

```
void philosopher(int i) {  
    while (1) {  
        think();  
        philosopherMeal.take_forks(i);  
        eat();  
        philosopherMeal.put_forks(i);  
    }  
}
```

香烟问题

- S. S. Patil. @ 1971
- 三个要素：抽烟资源（烟草，卷纸，火柴）
- 四个人
 - 一个仲裁者
 - 三个烟民：手里各有一种资源
- 一张桌子
- 仲裁者（循环）
 - 随机选择2个烟民，将手头的资源放在桌上
 - 让第三个烟民收集桌上资源（+自己的）→ 开始抽烟
- 抽烟过程不能被打断
- 额外要求
 - 仲裁者的代码不能变化；不许使用条件判断和信号量数组
- http://en.wikipedia.org/wiki/Cigarette_smokers_problem



应用场景

- 生产者消费者
 - 打印机
 - 游戏中：金币管理、生命力（被攻击，医生）
- 读者写者
 - 数据库
 - 12306
- 理发师
 - 进程调度
- 哲学家

Take Away

- 5个术语
 - 同步: Synchronization
 - 互斥: Mutual Exclusion
 - 竞争: Race Condition
 - 临界区: Critical Section
 - 临界资源: Critical Resource
- 4种机制
 - 软件
 - 硬件指令互斥
 - 信号量
 - 管程
- 4个问题
 - 生产者消费者
 - 读者写者
 - 理发师
 - 哲学家就餐

参考阅读

- [http://en.wikipedia.org/wiki/Synchronization_\(computer_science\)](http://en.wikipedia.org/wiki/Synchronization_(computer_science))
- <http://en.wikipedia.org/wiki/Test-and-set>
- <http://en.wikipedia.org/wiki/Compare-and-swap>
- http://en.wikipedia.org/wiki/X86_instruction_listings
- [http://en.wikipedia.org/wiki/Semaphore_\(programming\)](http://en.wikipedia.org/wiki/Semaphore_(programming))
- [http://en.wikipedia.org/wiki/Monitor_\(synchronization\)](http://en.wikipedia.org/wiki/Monitor_(synchronization))

参考阅读

- http://en.wikipedia.org/wiki/Edsger_W._Dijkstra
- http://opera.ucsd.edu/pub_system_depend.html
- <https://computing.llnl.gov/tutorials/pthreads/>

谢谢！