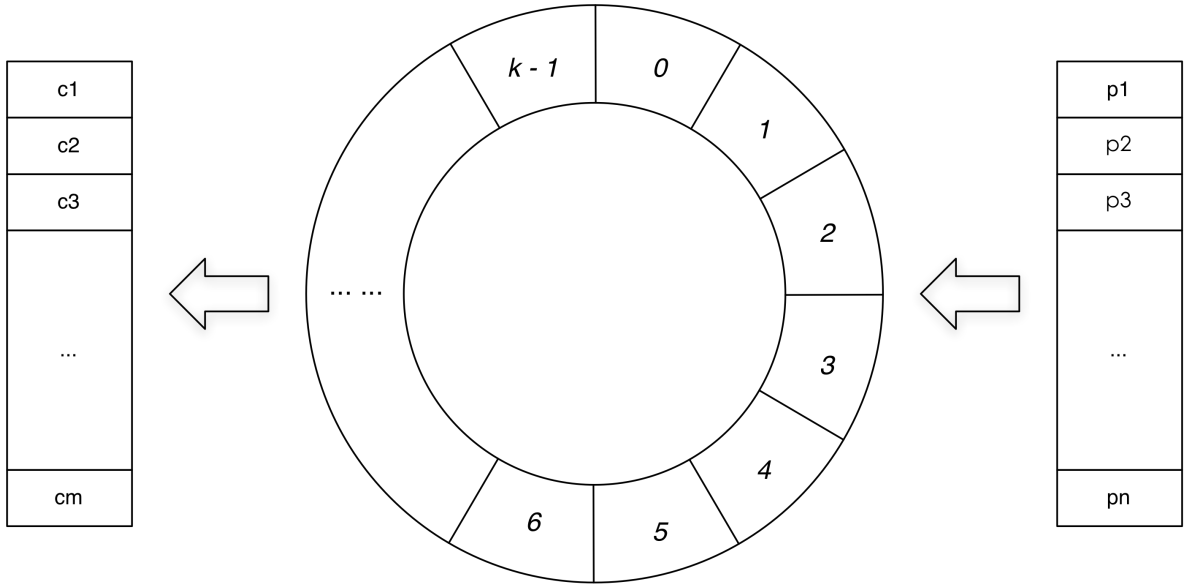
3.3 信号量与PV操作

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

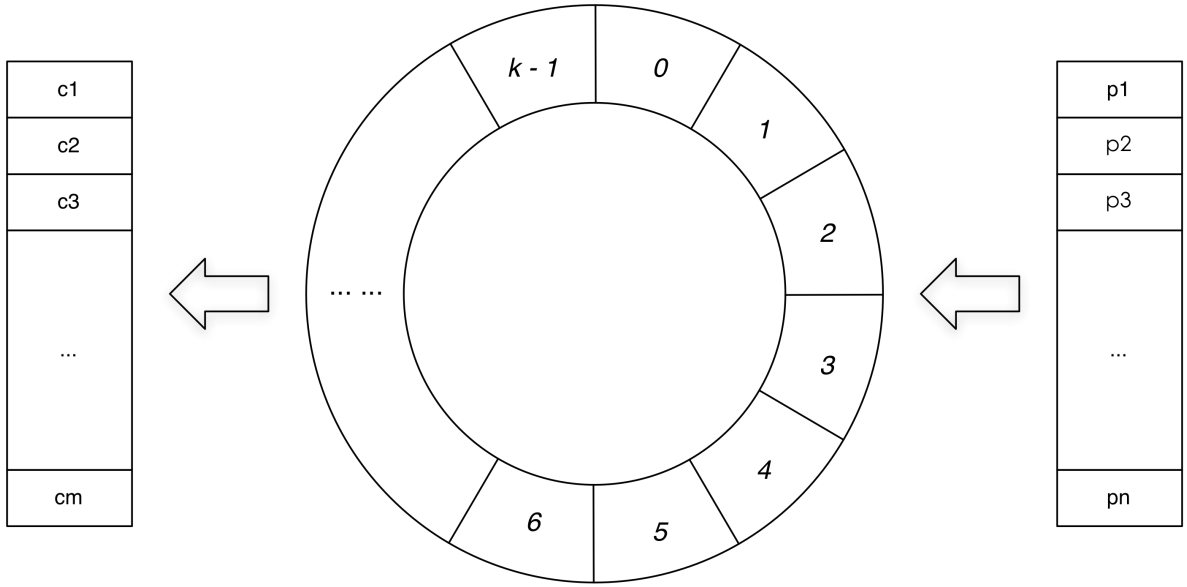
3.3.1 同步和同步机制

- 生产者
  - 计算进程
  - 发送进程
- 消费者
  - 打印进程
  - 接收进程



3.3.1 同步和同步机制

```
1.  int k;
2.  typedef anyitem item;
3.  item buffer[k]; //shared by producers and consumers
4.  int in;         //shared by producers
5.  int out;        //shared by consumers
6.  int counter;    //shared by producers and consumers
```



3.3.1（例）

process producer（）

```
1.  while (true) {
2.     {produce an item in nextp}
3.     if (counter == k) sleep(producer);
4.     buffer[in] = nextp;
5.     in = (in + 1)%k;
6.     counter++;
7.     if(counter==1) wakeup(consumer);
8.  }
```

3.3.1（例）

process consumer（）

```
1.  while (true) {
2.     if (counter == 0) sleep(consumer);
3.     nextc = buffer[out];
4.     out = (out+1)%k;
5.     counter--;
6.     if(counter == (k-1)) wakeup(producer);
7.     {consume the item in nextc}
8.  }
```

3.3.1（例）：竞争错误

P1	P2
1. while (true) {	while (true) {
2. {*produce an "A" in nextp*}	
3.	{*produce an "B" in nextp*}
4. if (counter == k) sleep(producer);	
5. buffer[in] = *nextp*; //buffer[0] = "A"	
6.	if (counter == k) sleep(producer);
7.	buffer[in] = *nextp*; //buffer[0] = "B"
8. in = (in + 1)%k; //in = 1	
9.	in = (in + 1)%k; //in = 2
10. counter++; //counter = 1	
11.	counter++; //counter = 2
12. if(counter==1) wakeup(consumer);	
13.	if(counter==1) wakeup(consumer);
14. }	}

3.3.1（例）：竞争错误

P1	P2
1. while (true) {	while (true) {
2. {*produce an "A" in nextp*}	
3.	{*produce an "B" in nextp*}
4. if (counter == k) sleep(producer);	
5. buffer[in] = *ne*	
6.	er == k) sleep(producer);
7.	] = *nextp*; //buffer[0] = "B"
8. in = (in + 1)%k	
9.	+ 1)%k; //in = 2
10. counter++; //counter = 1	
11.	counter++; //counter = 2
12. if(counter==1) wakeup(consumer);	
13.	if(counter==1) wakeup(consumer);
14. }	}

buffer, in, out 应该互斥访问

3.3.1（例）：同步错误

- 语句：counter++ 可被编译为机器指令（Intel x86）

++

```
1.  MOV AX, counter
2.  INC AX
3.  MOV counter, AX
```

- 语句：counter-- 可被编译为机器指令（Intel x86）

--

```
1.  MOV BX, counter
2.  DEC BX
3.  MOV counter, BX
```

3.3.1（例）：协作错误(续)

1.	{*produce an "A" in nextp*}			
2.	if (counter == k) sleep(producer);			
3.	buffer[in] = *nextp*;			
4.	in = (in + 1)%k;			
5.	MOV AX, counter			
6.	INC AX			
7.	MOV counter, AX			
8.			nextc = buffer[out];	//nextc = "A"
9.		//buffer[0] = "A"	out = (out + 1)%k;	//count = 1
10.		//counter = 1	MOV BX, counter;	//out = 1
11.		//in = 1	DEC BX	
12.	{*produce an "B" in nextp*}			
13.	buffer[in] = *nextp*;			
14.	in = (in + 1)%k;	//in = 2		
15.	MOV AX, counter	//buffer[1] = "B"		
16.	INC AX	**//counter = ?**		
17.	MOV counter, AX			
18.			MOV counter, BX	**//counter = ?**
19.				
20.				

3.3.1（例）：协作错误(续)

```
1.   {produce an "A" in nextp}
2.   if (counter == k) sleep(producer);
3.   buffer[in] = nextp;
4.   in = (in + 1)%k;
5.   MOV AX, counter
6.   INC AX
7.   MOV counter, AX
8.                                                    ;
9.
10.
11.
12.  {produce an "B" in nex
13.  buffer[in] = nextp;
14.  in = (in + 1)%k;
15.  MOV AX, counter
16.  INC AX
17.  MOV counter, AX
18.                                                   MOV counter, BX
19.
20.
```

//nextc = "A"
//count = 1
//out = 1

//in = 2
//buffer[1] = "B"
//counter = ?

//counter = ?

应保证同步变量，如counter，的操作原子性

3.3.1 进程同步机制

出现上述竞争与协作错误结果的原因在于各个进程访问缓冲区的速率不同

要得到正确结果，需要调整并发进程的速度。

需要通过在进程间交换信号或消息来调整相互速率，达到进程协调运行的目的。

3.3.1 进程同步机制（续）

- 操作系统实现进程同步的机制称为同步机制，它通常由同步原语组成
- 常用的同步机制有：
  - 信号量与PV操作
  - 管程
  - 进程通信

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

3.3.2 信号量与PV操作

- Section 3.2中采用软、硬件方法来解决临界区调度问题

1. 对不能进入临界区的进程，采用忙等测试法，浪费CPU时间
2. 将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重了用户编程负担
3. 方法只能解决进程竞争，而不能解决进程协作问题

3.3.2 信号量同步

- 1965年荷兰计算机科学家E.W.Dijkstra提出了新的同步工具--信号量和P、V操作。他将交通管制中多种颜色的信号灯管理交通的方法引入操作系统，让两个或多个进程通过特殊变量展开交互。
- 一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(Semaphore)。进程可以使用P、V两个特殊操作来发送和接收信号

3.3.2（续）

- 起信号灯作用的变量称为信号量
- 操作系统中，信号量是用来表示物理资源的实体，信号量是一种软资源
- 除赋初值外，信号量仅能由同步原语对其进行操作，没有任何其他方法可以检查和操作信号量

3.3.2（续）

- Dijkstra引进了两个信号量操作原语：
- P操作原语：测试“（Proberen）”/ wait()
- V操作原语：增量“（Verhogen）”/ signal()

- 利用信号量和P、V操作既可以解决并发进程的竞争问题，又可以解决并发进程的协作问题

3.3.2：信号量分类（用途）

- 公用信号量：初值常常为1，用来实现进程间的互斥。相关进程均可对其执行P、V操作
- 私有信号量：初值常常为可用资源数，多用来实现进程同步。拥有该信号量的一类进程可以对其执行P操作，而另一类进程可以对其执行V操作

3.3.2：信号量分类（取值）

- 二元信号量：仅取0和1 ，主要用于解决进程互斥问题
- 一般信号量：允许取值为非负整数，主要用于解决进程同步问题

随着信号量的发展，先后出现了以下几种形式的信号量：

1. 整型信号量
2. 结构型信号量
3. AND型信号量
4. 信号量集机制

3.3.2：整型信号量

- 数据类型：整型量S
- 操作：
  - 初始化（初值一般取可用资源的数目）
  - 原子操作P(S)
  - 原子操作V(S)
- 除初始化外，仅能通过P、V操作访问

3.3.2：整型信号量（续）

P(S)

```
1.  while ( S <= 0) {};
2.  S--;
```

V(S)

```
1.  S++;
```

未遵循“让权等待”原则

- 若S=0，则进程便无限循环进行等待，白白浪费CPU时间

为实现让权等待，引进了记录型信号量机制

3.3.2：结构型信号量

```
typedef struct semaphore{
  int value;          //可用资源数
  struct pcb *list; //阻塞队列
}
```

P(semaphore &s)

```
1.  s.value--;
2.  if(s.value < 0) {
3.     {add current process to s.list}
4.    block();
5.  }
```

V(semaphore &s)

```
1.  s.value++;
2.  if(s.value <= 0) {
3.     {remove a process P from s.list}
4.    wakeup(P);
5.  }
```

block() 和 wakeup(P) 是操作系统提供的基础系统调用

3.3.2：结构型信号量 推论

- 推论1：s .value>=0时， s. value值表示还可以执行P而不会被阻塞的进程数，每执行一次P就意味着一次分配一个单位的资源
- 推论2：当 s. value<0，表示没有可用资源，请求该资源的进程被阻塞。 s. value的绝对值表示被阻塞的进程数，执行一次V就意味着释放一个资源。若s. value<0表示还有被阻塞的进程，需要唤醒一个被阻塞的进程，使他到就绪队列中排队
- 推论3：通常，P操作意味着请求一个资源，V操作意味着释放一个资源；特定条件下P操作代表挂起进程的操作，V操作代表唤醒被挂起进程的操作

3.3.2：结构型信号量 缺点

- 整型及记录型信号量一次仅能获得一个单位的一种临界资源，从而容易导致死锁（违背有限等待原则）
- 例：现有A、B两个进程，它们都要求访问共享数据D和E，D、E应作为临界资源，为D、E分别设置互斥信号量Dmutex和Emutex，其初值均为1。下面是导致死锁的进程A、B的并发操作序列：

3.3.2 死锁样例

A	B
1. P(Dmutex);	
2.	P(Emutex);
3. P(Emutex);	
4.	P(Dmutex);

进程A阻塞

进程B阻塞

3.3.2：AND型信号量

- 基本思想：对于进程在整个运行过程中需要的所有资源，要么一次性地全部分配给该进程，要么一个也不分
- 数据结构：与记录型信号量结构类同

3.3.2：AND型信号量

Swait(S_1, S_2, …, S_n)

```
1.  if (S1≥1 && S2≥1 && … && Sn≥1){
2.    for(i=1; i<=n;i++){
3.      Si--;
4.    }
5.  } else {
6.    {place P to the first Si with Si < 1};
7.    {set P's program count to the beginning of Swait}
8.  }
```

Ssignal(S_1, S_2, …, S_n)

```
1.  for(i=1; i<=n;i++){
2.    Si++;
3.    if (Si <=0){wakeup all its associated processes}
4.  }
```

3.3.2：信号量集

- 解决问题：
  - 某进程一次需要N个某类临界资源时，避免执行N次P操作
  - 当资源量 < 下限值时不分配
- 数据结构：信号量S，需求值d，下限值t

3.3.2：信号量集

Swait(S_1,t_1,d_1, S_2 ,t_2,d_2, …, S_n,t_n,d_n,)

```
1.  if (S1≥t1 && S2≥t2 && … && Sn≥tn){
2.    for(i=1; i<=n;i++){
3.      Si -= di;
4.    }
5.  } else {
6.    {place P to the first Si with Si < 1};
7.    {set P's program count to the beginning of Swait}
8.  }
```

d：需求值
t：下限值

Ssignal(S_1, d_1, S_2 ,d_2, …, S_n ,d_n)

```
1.  for(i=1; i<=n;i++){
2.    Si += di;
3.    if (Si <=0){wakeup all its associated processes}
4.  }
```

3.3.2：信号量集（特例）

- 特例1：Swait（S,d,d）:此时在信号量集中只有一个信号量S，但允许每次申请d个资源，当现有资源少于d时不分配
- 特例2：Swait（S,1,1）:此时的信号量蜕化为一般的结构型信号量（S>1时）或互斥信号量（S=1时）
- 特例3：Swait（S,1,0）: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

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

3.3.3 信号量实现互斥

- 为临界资源设一互斥信号量mutex，初值为1，将临界区置于P(mutex)和V(mutex)之间。用于互斥的信号量mutex，其P 和V 操作一定要成对出现在同一个进程中

```
semaphore mutex = 1;
Pi() // i = 1, 2, ..., n
1.  P(mutex);
2.  {critical section}
3.  V(mutex);
```

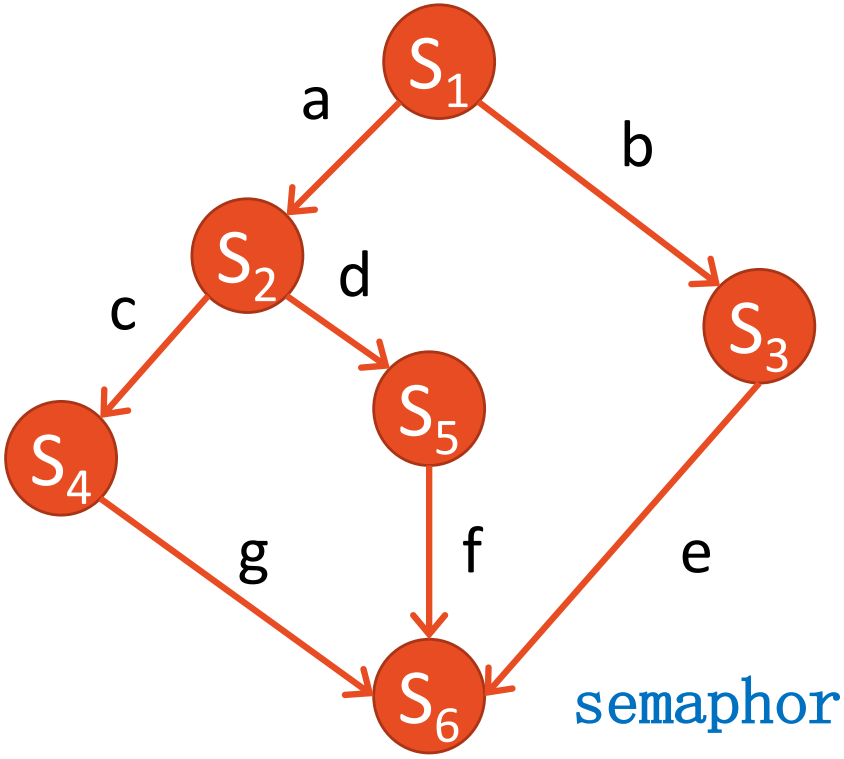
- 只对信号量测试一次，避免忙等

3.3.3（补充）前趋关系实现

- 设置一信号量m，初值为0，若希望进程P1中的语句S1先于进程P2中的语句S2，则：
  - 在P1中用：S1；V（m）；
  - 在P2中用： P（m）； S2；
- 这样， S2不会先于S1执行。

3.3.3（补充）前趋关系实现

- 例：语句S1~S6之间的前趋关系图如下，写出一个可并发执行的程序



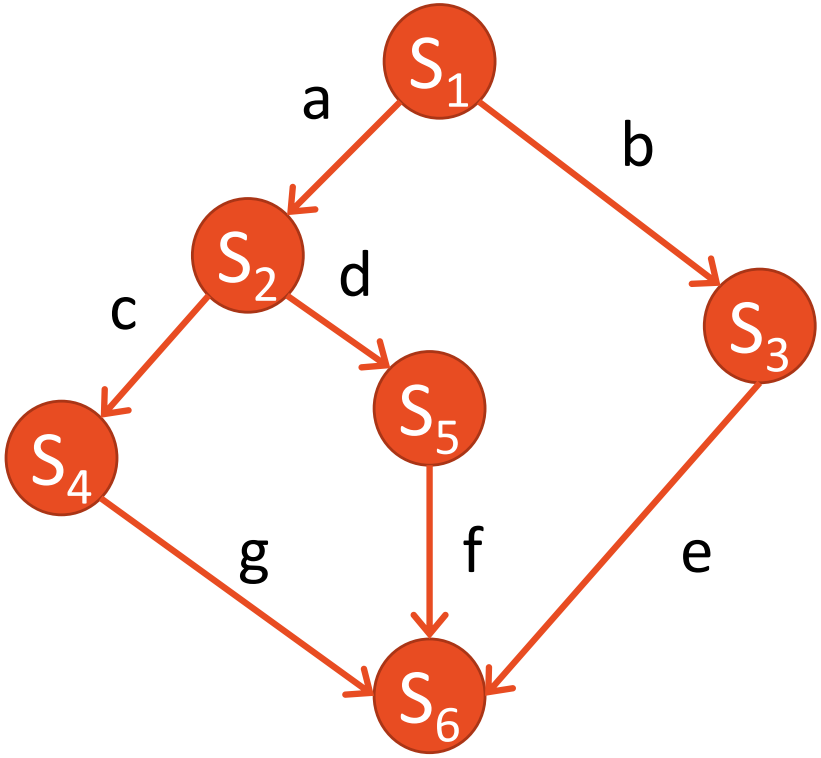
Tip：图中有多少条有向边就需要多少个信号量用来实现同步

图中共有7条有向边，所以需要定义7个信号量

```
semaphore a, b, c, d, e, f, g = 0;
```

3.3.3（补充）前趋关系实现

```
semaphore a, b, c, d, e, f, g = 0;
parbegin
p1: {S1; V(a); V(b);}
p2: {P(a); S2; V(c); V(d);}
p3: {P(b); S3; V(e);}
p4: {P(c); S4; V(g);}
p5: {P(d); S5; V(f);}
p6: {P(f); P(g); P(e); S6;}
parend
```



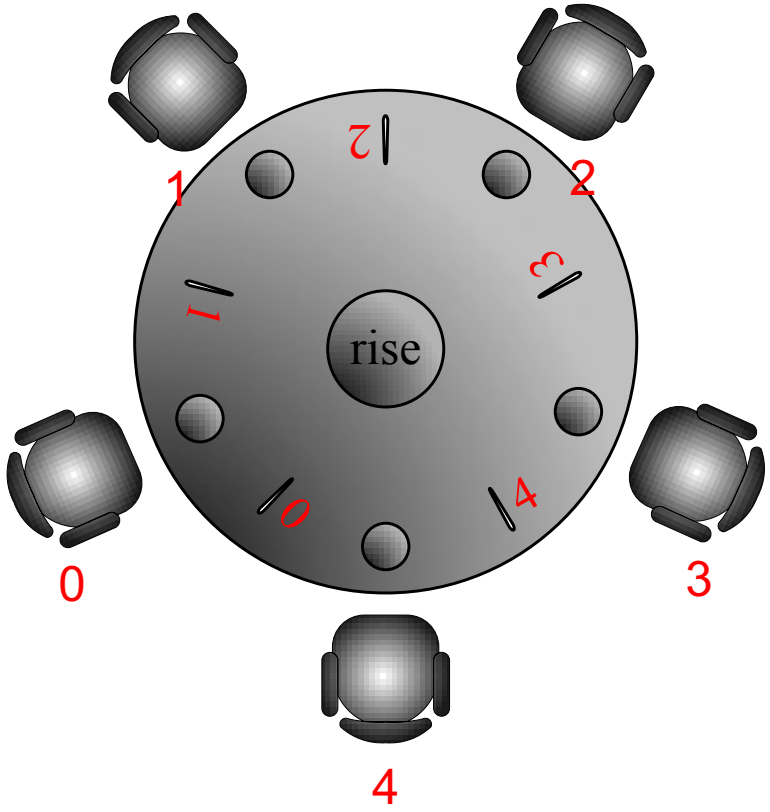
信号量的经典应用

- 哲学家进餐问题
  - Dining Philosophers problem
- 生产者 — 消费者问题
  - Bounded-Buffer problem
- 读 — 写者问题
  - Readers-Writers problem
- 理发师问题
  - Sleeping Barber problem

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

3.3.4 哲学家就餐问题

- 五位哲学家围坐一张圆桌，桌上放五根筷子，每位哲学家只能拿起与他相邻的两根筷子吃饭
- 哲学家的生活方式是交替地进行思考和进餐



3.3.4 采用结构型信号量

- 数据结构：每根筷子都是一个临界资源，都应定义一个信号量，为五根筷子定义一个信号量数组，每个信号量的初值为1

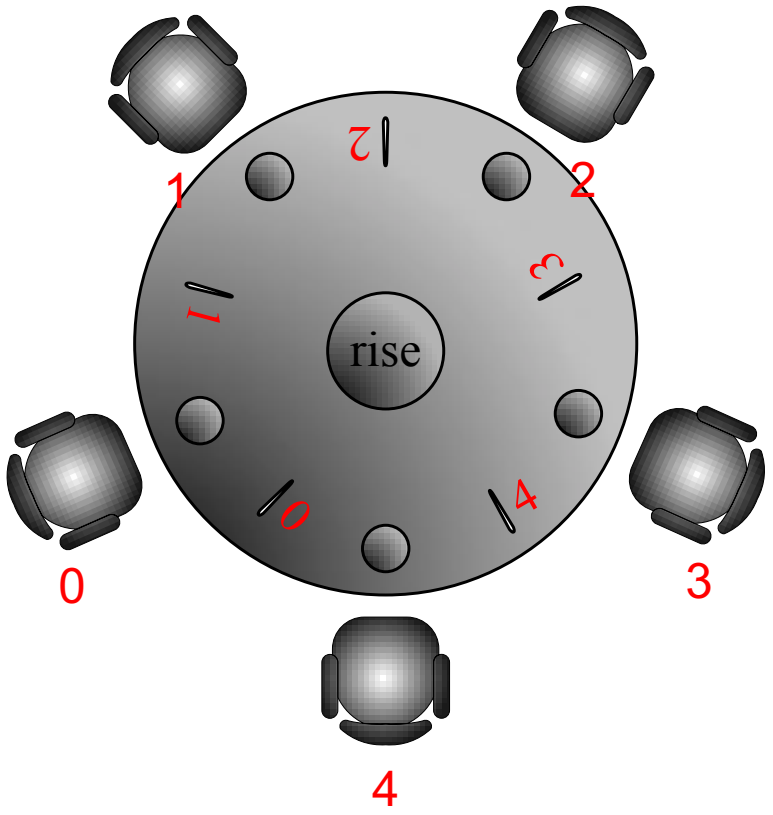
3.3.4 采用结构型信号量

```
semaphore fork[5]= {1,1,1,1,1};
```

Philosopher_i() // i = 1, 2, 3, 4, 5

```
1.  while (true) {
2.     {think()};
3.     P(fork[i]);

4.     P(fork[(i+1)%5];
5.     {eat()};
6.     V(fork[i]);
7.     V(fork[(i+1)%5]);
8.  }
```



可能造成死锁！

3.3.4 采用结构型信号量

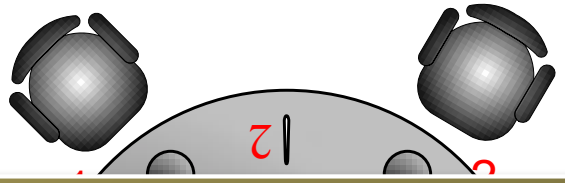
```
semaphore fork[5]= {1,1,1,1,1};
```

Philosopher_i() // i = 1, 2, 3, 4, 5

```
1.  while (true) {
2.     {think()};
3.     P(fork[i]);

4.     P(fork[(i+1)%5];
5.     {eat()};
6.     V(fork[i]);
7.     V(fork[(i+1)%5]);
8.  }
```

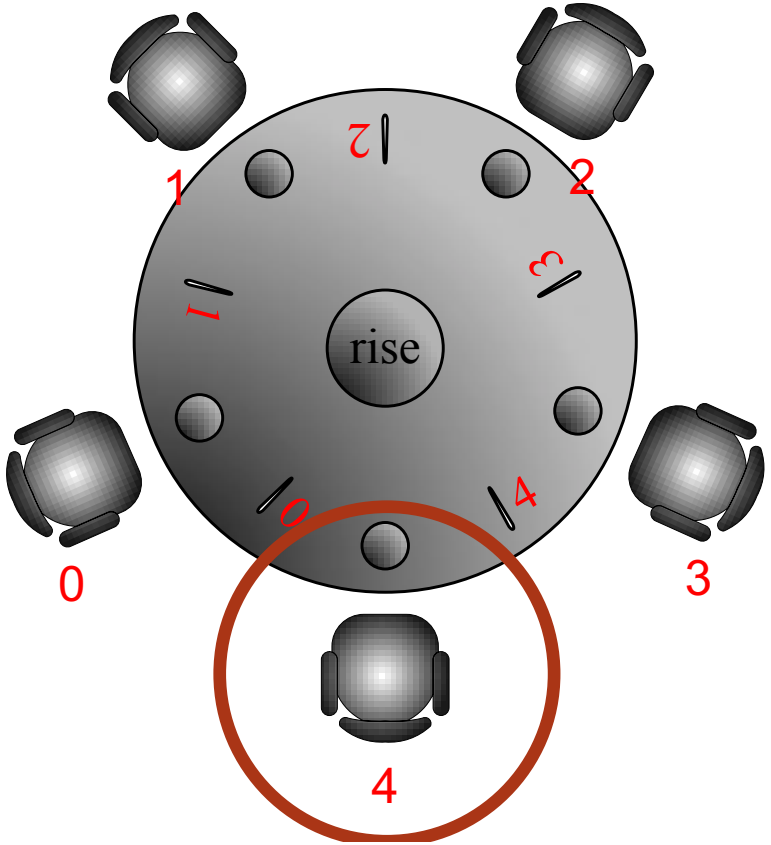
可能造成死锁！



若每个哲学家都各自拿起他左边的一根筷子，然后再去拿他右边的筷子时，将都拿不到右边的筷子，大家又都不会放下手中的筷子，大家在相互等待别人释放筷子，系统于是进入死锁状态

3.3.4：死锁解决方法

1：至多允许有四位哲学家同时去拿左边的筷子，最终保证至少有一位哲学家能够进餐，（至多可以有两位哲学家能够进餐）



3.3.4：死锁解决方法

2，仅当哲学家的左右两根筷子均可用时，才允许他拿起筷子进餐。这种方案可以采用AND信号量机制来实现

3，规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；偶数号哲学家先拿他右边的筷子，然后再去拿左边的筷子

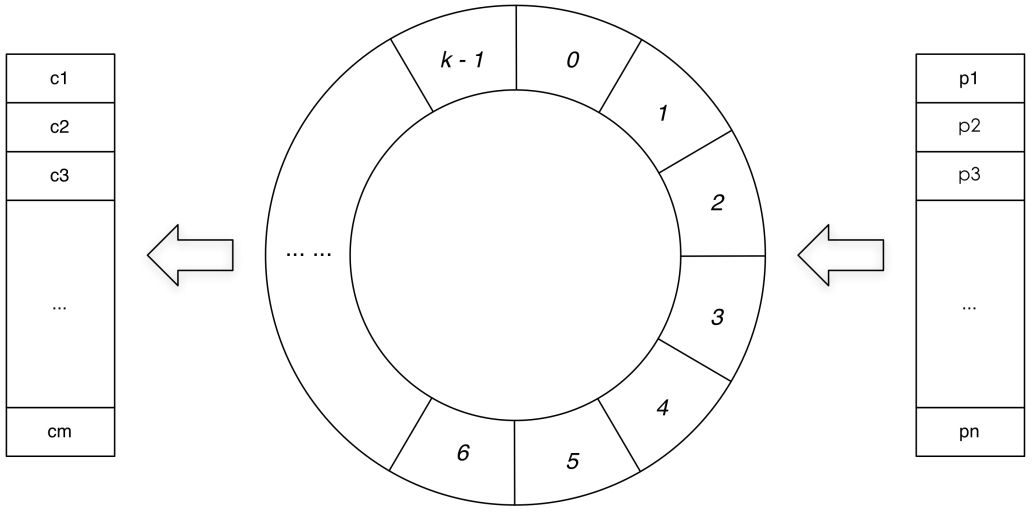
- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

3.3.5 生产者—消费者问题

数据结构：

1. 含有n个缓冲区的公用缓冲池
2. 互斥信号量mutex：实现诸进程对缓冲池的互斥使用，一次仅允许一个进程读或写公用缓冲池，初值为1
3. 资源信号量empty：记录空缓冲区个数，初值为n
4. 资源信号量full：记录满缓冲区个数，初值为0

```
item B[n]
semaphore mutex = 1;
semaphore full = 0;
semaphore empty = n;
```



3.3.5 生产者—消费者问题

process producer（）

```
1.  while (true) {
2.     {produce an item in nextp}
3.     P(empty);
4.     P(mutex);
5.     buffer[in] = nextp;
6.     in = (in + 1)%k;
7.     counter++;
8.     V(mutex);
9.     V(full);
10. }
```

3.3.5 生产者—消费者问题

process consumer（）

```
1.  while (true) {
2.    P(full);
3.    P(mutex);
4.    nextc = buffer[out];
5.    out = (out+1)%k;
6.    counter--;
7.    V(mutex);
8.    V(empty);
9.    {consume the item in nextc}
10. }
```

3.3.5（续）

- 在每个程序中用于互斥的P（mutex）和V （mutex） 必须成对出现

- 对资源信号量empty和full的操作也同样必须成对出现，但它们在不同的程序中

- 在每个程序中的多个P操作顺序不能颠倒，否则容易引起死锁

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题

3.3.6 读—写者问题

- 该问题为多个进程访问一个共享数据区，如数据库、文件、内存区、寄存器等数据问题建立了一个通用模型，其中读进程只能读数据，写进程只能写数据

- 例如一个联网售票系统，数据的查询和更新非常频繁，不可避免会出现多个进程访问一个数据的情况，多个进程同时读一个数据是允许的，但如果一个进程正在更新一个数据则不允许其他进程访问或修改该数据。否则就会将一个座位销售多次

3.3.6（续）

- 写者—写者冲突（wrt）
- 写者—读者同步（ReadCount）
- 读者间互斥ReadCount（mutex）

```
semaphore wrt = 1;
semaphore mutex = 1;
int ReadCount = 0;
```

3.3.6（续）

process reader()

```
1.  while (true) {
2.     P(mutex);
3.     readCount++;
4.     if (readCount == 1) P(wrt);
5.     V(mutex);

6.    {reading data … }
7.
8.     P(mutex);
9.     readCount--;
10.    if (readCount == 0) V(wrt);
11.    V(mutex);
12. }
```

process writer()

```
1.  while (true) {
2.  P(wrt);

3.  {writing data … }
4.
5.  V(wrt);
6.  }
```

1. 多读者，待写
2. 一写者，待读
3. 多读者并发

3.3.6：写者优先

- 写者—写者冲突（wrt）
- 写者—读者同步（ReadCount）
- 读者间互斥ReadCount（rmutex）
- 写者计数（WriteCount）
- 写者间互斥WriteCount(wmutex)
- 读—写者互斥（rdr）

```
semaphore wrt = 1;
semaphore rmutex = 1;
int ReadCount = 0;
semaphore wmutex = 1;
int WriteCount = 0;
semaphore rdr = 1;
```

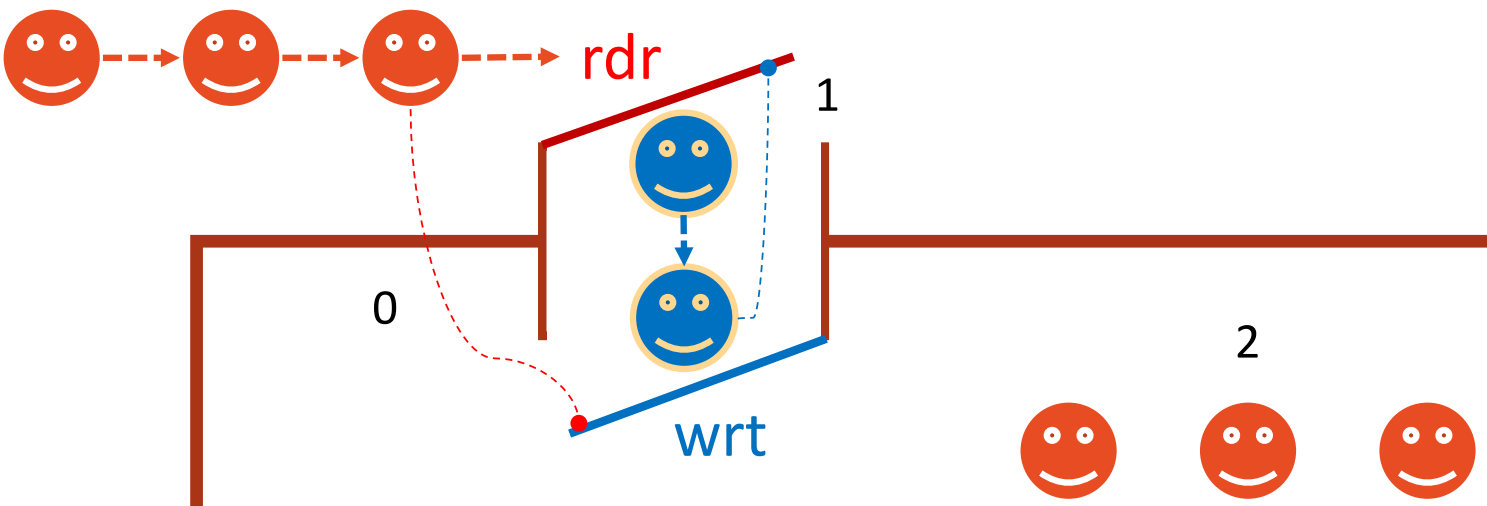
3.3.6 ：写者优先（续）

process reader()

```
1.  while (true) {
2.     P(rdr);
3.     P(rmutex);
4.     ReadCount++;
5.     if(ReadCount == 1) P(wrt);
6.     V(rmutex);
7.     V(rdr);
8.     {reading …}
9.     P(rmutex);
10.    ReadCount--;
11.    if (ReadCount == 0) V(wrt);
12.    V(rmutex);
13. }
```

process writer()

```
1.  while (true) {
2.     P(wmutex);
3.     WriteCount++;
4.     if(WriteCount==1) P(rdr);
5.     V(wmutex);
6.     P(wrt);
7.     {writing …}
8.     V(wrt);
9.     P(wmutex);
10.    WriteCount--;
11.    if(WriteCount==0) V(rdr);
12.    V(wmutex);
13. }
```



- 读者必须等待写者全部完成才可继续
- 一旦有写者，待所有现有读者完成后关门（rdr）、排除后继读者
- 写者关闭

3.3.6：信号量集的解决方案

- 特例3：Swait（S,t:1,d:0）: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

- 读者数目信号量Rcount ，初值为RN；
  - 规定最多只允许RN个读者同时读
- 信号量mutex，表示有无写者在写，初值为1

```
semaphore Rcount = RN;
semaphore mutex = 1;
```

3.3.6（续）

process reader()

```
1.  while (true) {
2.     Swait(Rcount, 1, 1;
3.        mutex, 1, 0);

4.    {reading data … }

5.     Ssignal (Rcount,1);
6.  }
```

process writer()

```
1.  while (true) {
2.  Swait(mutex,1,1;
3.     Rcount, RN, 0);

4.  {writing data … }
5.
6.  Ssignal(mutex, 1);
7.  }
```

1. 多读者，待写
2. 一写者，待读
3. 多读者并发

- 同步与同步机制
- 信号量与PV操作
- 信号量实现互斥
- 哲学家就餐问题
- 生产者—消费者问题
- 读—写者问题
- 理发师问题