第二部分. 进程管理

第六章: 进程同步
Process Synchronization

1. 背景
2. 临界区问题
3. Peterson`s 算法
4. 硬件同步
5. 信号量（Semaphores）
6. 经典同步问题
7. 管程（Monitors）

Note : These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts.*

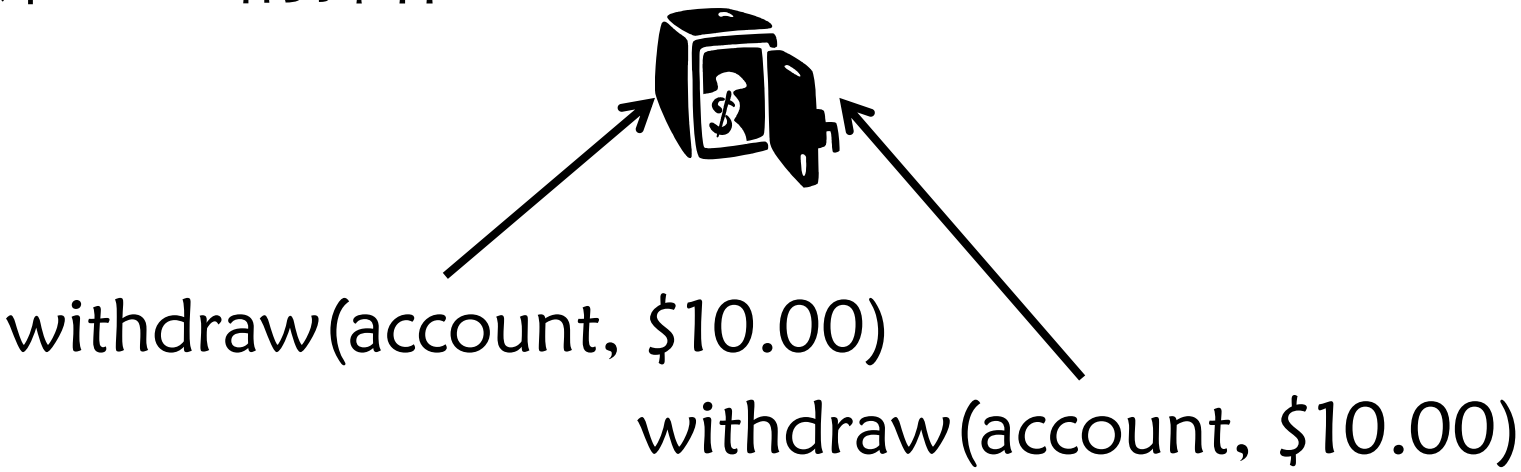
1. 背景

- 多个进程共享数据，并协同工作
- 存取这些共享数据，需要确保数据的一致性(data concurrency)
- 操作系统必须提供一个协同工作进程之间的共享数据的同步保障机制，以确保共享数据的一致性

- 问: 如果没有进程间的同步保障机制会发生什么样的问题?
- 答：发生竞争条件(race condition)问题

进程同步

- 假设1. 实现一个从银行账户上取款的函数 withdraw
- 假设2. A和B共享这个银行账号，银行账号的余额为 $100.00
- 假设3. A和B同时在不同的ATM取款机上同时各取款$10.00的操作



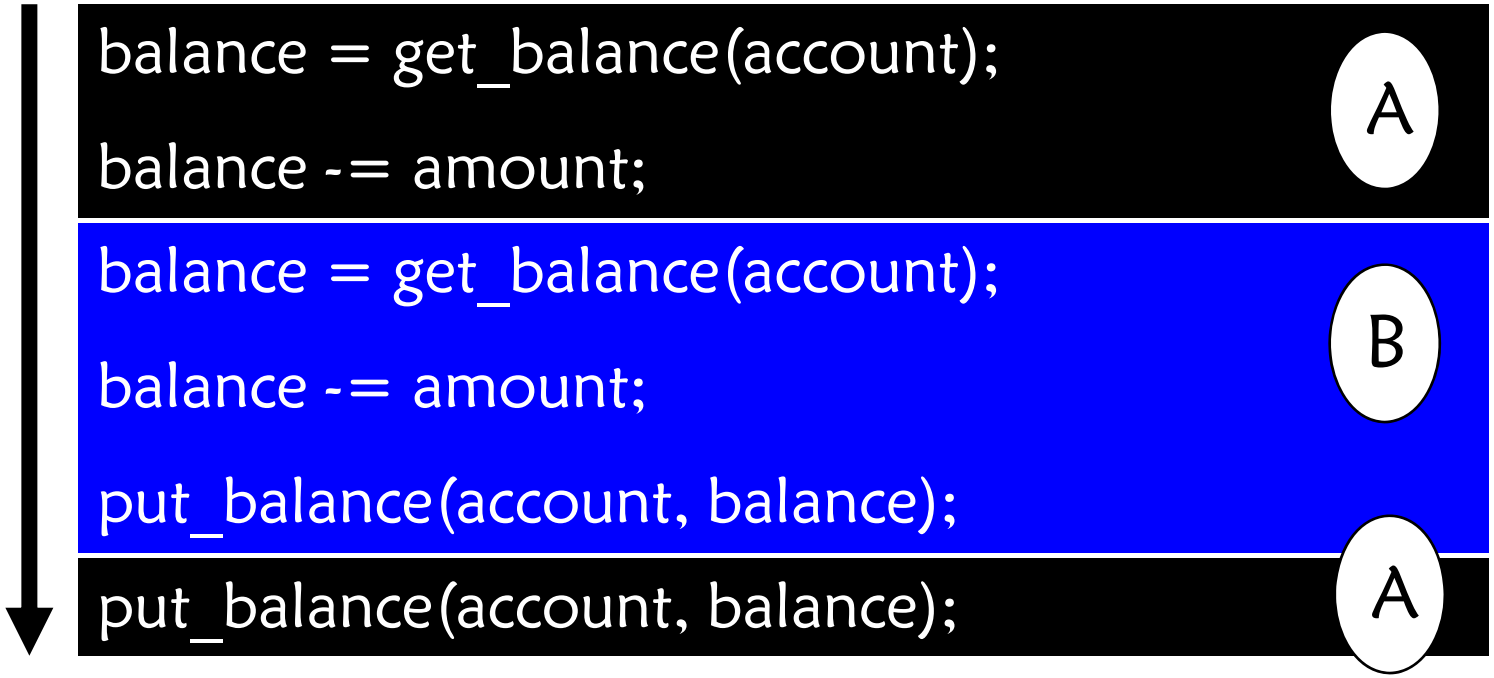
竞争条件（Race Condition）

```
// account：账号， amount：取款额
// get_balance：获取账号余额，
// put_balance：存入余额
int withdraw(account, amount) {
      balance = get_balance(account);
            balance = balance - amount;
            put_balance(account, balance);
            return balance;
}
```

银行系统的主机上可能有两个进程（进程A和进程B）同时在执行取款函数 withdraw( )

竞争条件（Race Condition）

- 假设3. 银行系统支持抢占调度，而且两个进程交叉执行(interleaved execution)如下

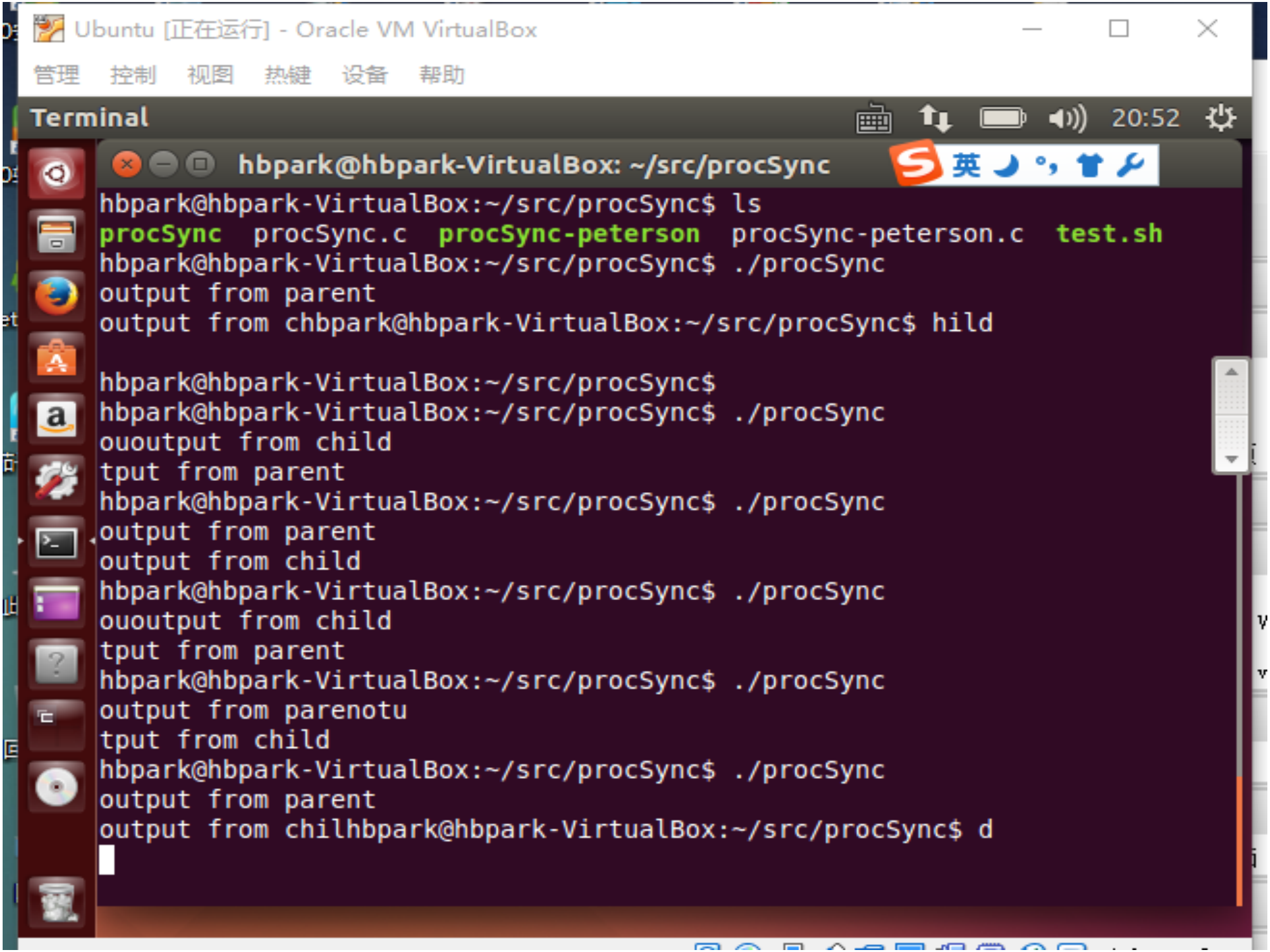


What is the result of the bank account?

```
#include <sys/types.h>

static void charatatime(char *str){
      char     *ptr; int c;
      setbuf(stdout, NULL); /* set unbuffered */
      for (ptr = str; c = *ptr++; )
                putc(c, stdout);
}
int main(void){
      pid_t pid;
      if ( (pid = fork()) < 0)
        perror("fork error");
      else if (pid == 0)
        charatatime( “output from child\n” );
      else
        charatatime( “output from parent\n” );
      exit(0);
}
```

竞争条件例子代码


Ubuntu [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Terminal
20:52
hbpark@hbpark-VirtualBox: ~/src/procSync
hbpark@hbpark-VirtualBox:~/src/procSync$ ls
procSync procSync.c procSync-peterson procSync-peterson.c test.sh
hbpark@hbpark-VirtualBox:~/src/procSync$ ./procSync
output from parent
output from chbpark@hbpark-VirtualBox:~/src/procSync$ hild
hbpark@hbpark-VirtualBox:~/src/procSync$
hbpark@hbpark-VirtualBox:~/src/procSync$ ./procSync
ououtput from child
tput from parent
hbpark@hbpark-VirtualBox:~/src/procSync$ ./procSync
output from parent
output from child
hbpark@hbpark-VirtualBox:~/src/procSync$ ./procSync
ououtput from child
tput from parent
hbpark@hbpark-VirtualBox:~/src/procSync$ ./procSync
output from parenotu
tput from child
hbpark@hbpark-VirtualBox:~/src/procSync$ ./procSync
output from parent
output from chilhbpark@hbpark-VirtualBox:~/src/procSync$ d

竞争条件

- 所以，多个进程对共享数据进行操作的时候，有可能会发生竞争条件
- 而且，竞争条件的结果是不可预测的（unpredictable）

问：那么，避免发生竞争条件的关键问题是什么？

答：确保操作共享数据的代码段的执行同步，不能让两个以上的进程同时运行操作共享数据的代码段

2. 临界区问题（Critical Problem）

- 多个进程同时操作共享数据时，每个进程拥有操作共享数据的代码段（程序段），而这代码段称为临界区(critical section)。

- 如共享变量、共享表、共享文件等

2. 临界区问题（Critical Section Problem）

- 解决竞争条件问题的关键是，

1. 在某一个时间，确保单个进程在临界区内执行
2. 同时，也得确保其他进程也可以进入临界区

临界区问题的通用结构

do {

进入区 → entry section

临界区 → critical section

退出区 → exit section

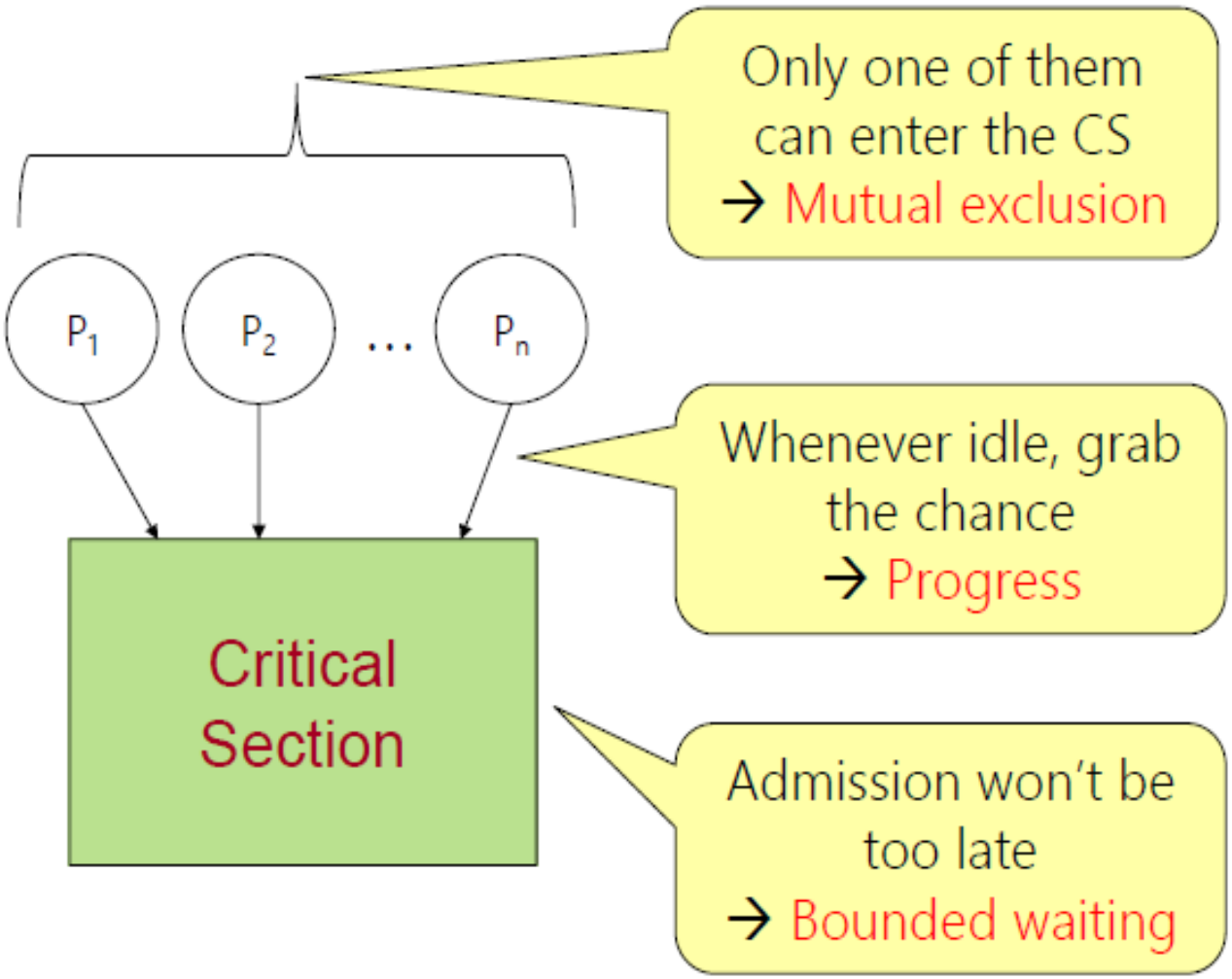
剩余区 → remainder section

}while(true)

解决临界区问题需要满足如下条件

1. 互斥（Mutual Exclusion ）
   - 某一个进程进入了临界区，其他进程就不能进入
2. 前进（Progress）
   - 如果没有进程在临界区执行，则必须确保一个进程进入临界区
3. 有限等待（Bounded Waiting）
   - 一个进程从请求进入临界区，直到该请求被允许，必须有限等待。

解决临界区问题的三个条件



3. Peterson 算法

A classic software-based solution to the critical-section problem, the solution is restrict to two processes.

- 设置两个变量
  1. int turn
  2. boolean flag[2]

- turn: 变量turn表示哪个进程可以进入其临界区
- flag : 数组flag表示哪个进程想要进入其临界区. flag[i] = true 表示进程 i 可以进入临界区。

前提条件是加载和存储指令是原子指令，即加载和存储指令是不可被中断的指令。

Peterson`s 算法

进程*i*和进程 *j*

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

Giving way to the other process

- Provable that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

```
Initialization
flag[2] = {false; false};
turn = 0;
```

```
PROCESS 0
do{
  flag[0] = true;
  turn = 1;
  while(flag[1] && turn == 1);
      //critical section
  flag[0] = false;
} while(true)
```

```
PROCESS 1
do{
  flag[1] = true;
  turn = 0;
  while(flag[0] && turn == 0);
      //critical section
  flag[1] = false;
} while(true)
```

如果没有turn 可以吗？

```
Initialization
flag[2] = {false; false};
```

```
PROCESS 0
do{
  flag[0] = true;
  while(flag[1]);
      //critical section
  flag[0] = false;
} while(true)
```

```
PROCESS 1
do{
  flag[1] = true;
  while(flag[0]);
      //critical section
  flag[1] = false;
} while(true)
```

满足解决临界区问题的三个条件

prove by contradiction

1. 互斥（假设两个进程都想进入了临界区）
   - 那么flag[0]=flag[1]=true, 但 turn 只能有一个值，要么0要么1

2. 前进（假设两个进程都在while循环不能前进）
   - Entry section: turn is key
   - Exit section: flag[] is key

3. 有限等待
   - 只要进入临界区的进程退出，就可以进入临界区

4. 硬件同步

- 问：我们可不可以用禁用中断的方式解决临界区问题？分别考虑单处理器系统和多处理器系统
- 答：单多皆可，但代价高

- 硬件方法解决：许多系统都拥有简单硬件指令，并描述如何用它们解决临界区问题
- 现代计算机系统提供特殊指令叫原子指令（atomic instructions）
  1. TestAndSet（）：检查和设置字的内容
  2. swap（）：交换两个字的内容

不可中断的指令

4.1 TestAndSet（）指令

假设定义如下原子指令:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

```
boolean TestAndSet(boolean * target) {
        boolean rv = * target;
        * target = TRUE;
        return rv;
}
```

```
PROCESS 0
do{
        while(TestAndSet(&lock));
        //critical section
        lock = false;
        //remainder section
}while (true);
```

声明全局变量 lock 初始化为lock = false

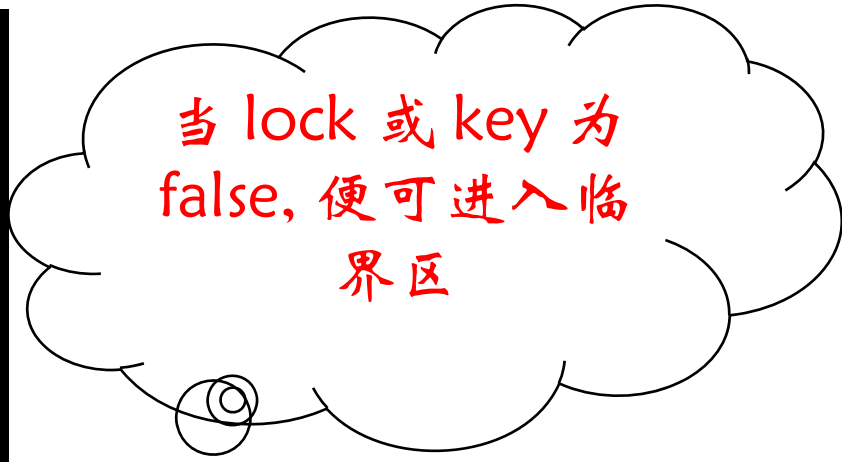
```
PROCESS 1
do{
        while(TestAndSet(&lock));
        //critical section
        lock = false;
        // remainder section
}while (true);
```

4.2 swap( ) 指令

声明一个布尔全局变量lock, 初始化为lock=false
另外，每个进程也有一个局部变量key, 初始化为true.

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

```
while (true)  {
     key = true;
     while ( key == true)
        swap (&lock, &key );
        // critical section
     lock = false;
        // remainder section
}
```



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

```
PROCESS 0
while (true)  {
    key = true;
    while ( key == true)
       swap (&lock, &key );
       // critical section
    lock = false;
    // remainder section
 }
```

```
PROCESS 1
while (true)  {
     key = true;
     while ( key == true)
       swap (&lock, &key );
       // critical section
     lock = false;
      // remainder section
 }
```

存在的问题

- Peterson`s 算法、TestAndSet( ) 和 swap( ) 原子指令(atomic instruction)操作会存在忙等待的问题 （busy waiting）

```
while(TestAndSet(&lock))
while(swap(&lock, &key))
```

Infinite loop

有限等待原子指令

声明一个布尔变量 waiting 确保有限的等待

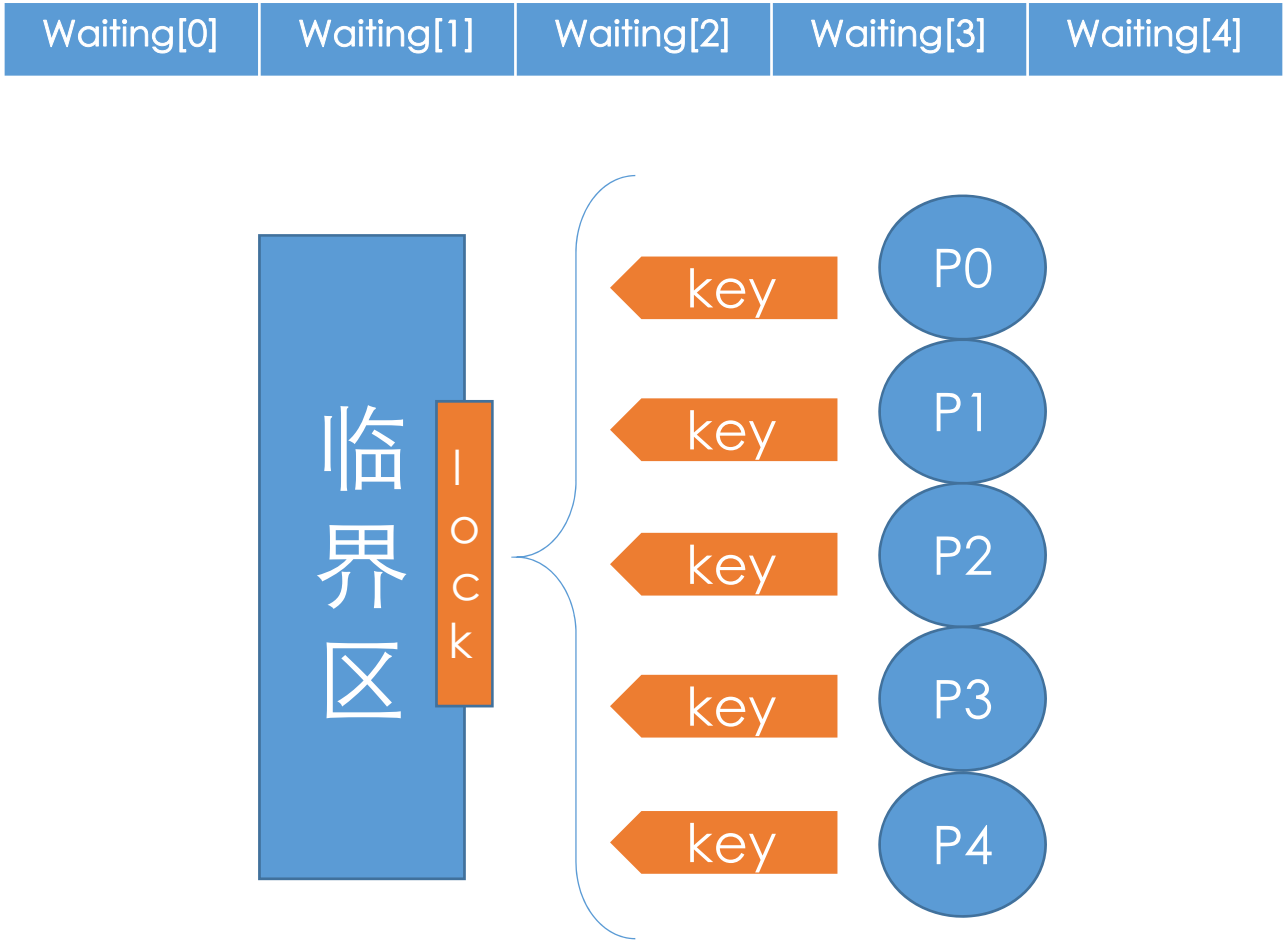
1. 局部变量 key，初始化为true; 每个进程
2. 全局变量 lock，初始化为false;
3. 用变量 waiting[i] 表示进程等待进入临界区，初始化为false;

可以进入临界区的条件是：

当 lock 或 waiting[i] 为 false时

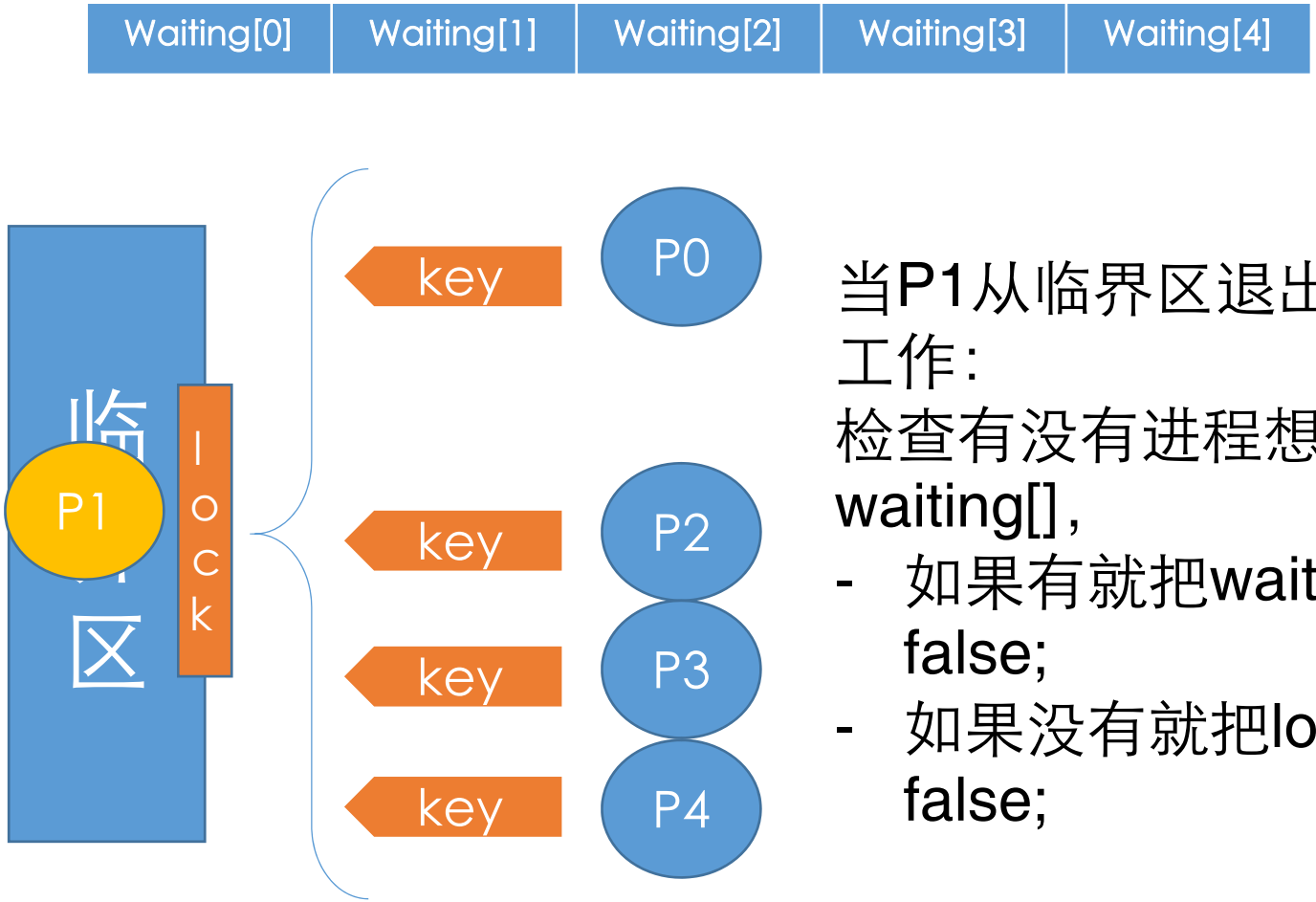
算法说明

可以进入临界区的条件：
当 lock 或 waiting[i] 为 false时



算法说明

可以进入临界区的条件：
当 lock 或 waiting[i] 为 false时



当P1从临界区退出时，做以下工作：
检查有没有进程想进入临界区waiting[]，

- 如果有就把waiting设置成false;
- 如果没有就把lock设置成false;

n ={ 1, 2, 3, 4, 5}

```
-----------------------------------
j = (i + 1) % n;
while( j != I && !waiting[j])
    j = (j + 1) % n;
-----------------------------------
```

依次检查每个进程是否想进入临界区

Let us consider the following cases:

- When i = 1
- When i = 2
- When i = 3
- When i = 4
- When i = 5

有限等待原子指令 TestAndSet( )

```
do {
    waiting[i] = true;
    key = true;
①   while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = false;
        // critical section
②   j = (i + 1) % n;
    while(( j != i) && !waiting[j])
        j = (j + 1) % n;
③   if (j  == i) // entered
        lock = false;
④   else  // waiting
        waiting[j] = false;
    // remainder section
} while (true);
```

1. i 表示当前运行的进程
2. n 表示进程的数量
3. j 表示 i 进程下一个进程

检查下个进程 j 是否想进入临界区

1. 要是waiting[j] == false，即不想进入临界区或已经进入临界区，那么跳过，检查在下一个进程； ②
2. 要是waiting[j] == true，即想进入临界区，就允许进入，并把 waiting[j] 设置成false; ④
3. 检查了一遍，没有进程想进入临界区，就把lock设置成false; ③

注意：waiting[j] 有可能是true，也有可能是 false;

When i = 0, n = 2, Process 0

```
do {
   waiting[0] = true;
   key = true;
   while(waiting[0] && key)
      key = TestAndSet(&lock);
   waiting[0] = false;
      // critical section
   j = (0 + 1) % 2;              // j = 1
   while(( j != 0) && !waiting[j])
      j = (j + 1) % 2;
   if (j  == 0) // entered
      lock = false;
   else  // waiting
      waiting[j] = false;
   // remainder section
} while (true);
```

When i = 1, n = 2, Process 1

```
do {
   waiting[1] = true;
   key = true;
   while(waiting[1] && key)
      key = TestAndSet(&lock);
   waiting[1] = false;
      // critical section
   j = (1 + 1) % 2;              // j = 0
   while(( j != 1) && !waiting[j])
      j = (j + 1) % 2;
   if (j  == 1)
      lock = false;
   else
      waiting[j] = false;
   // remainder section
} while (true);
```

有限等待原子指令

表明 waiting[0] 的值还是 false,
如果P1进入第二次循环，
waiting[0] 的值变为true

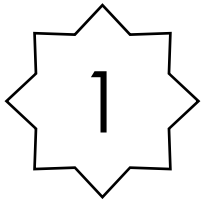
P1 waiting[0]	P2 …	P3 waiting[2]
false	false	false
true	true	true
false	true	true
false	false	true

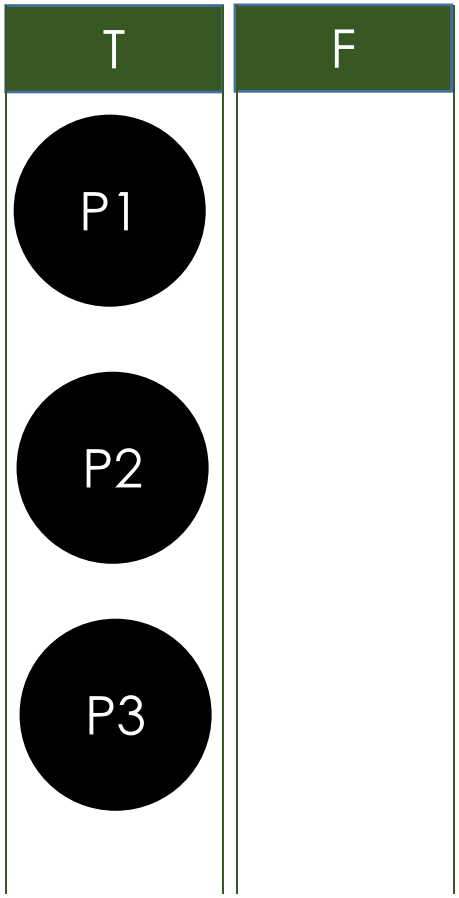
false	false	false

false	false	true

- 初始化值
- P1，P2，P3 都想进入临界区
- P1进入了临界区，lock=true, key1=false
- P1退出临界区，但还没有进入下一次循环
- 这时，因 waiting[1] 变为 false, P2进入临界区, lock=true, key2=true
- P2退出临界区，但还没有进入下一次循环
- 这时，因 waiting[2] 变为 false, P3进入临界区, lock=true, key3=true
- P3退出临界区，并进入第二次循环, lock=false 因 j == i

P1，P2，P3, 都想进入临界区

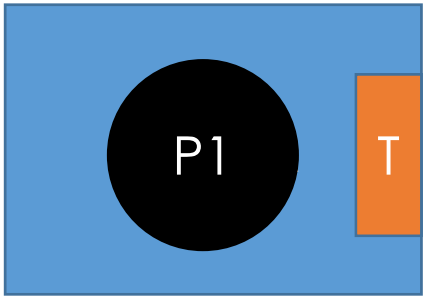


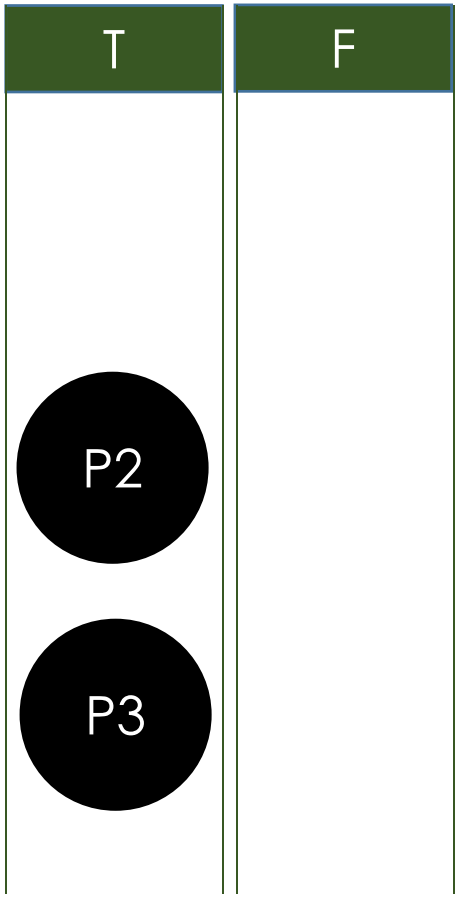


lock

waiting[]

P1 进入了临界区，lock 设置为 true



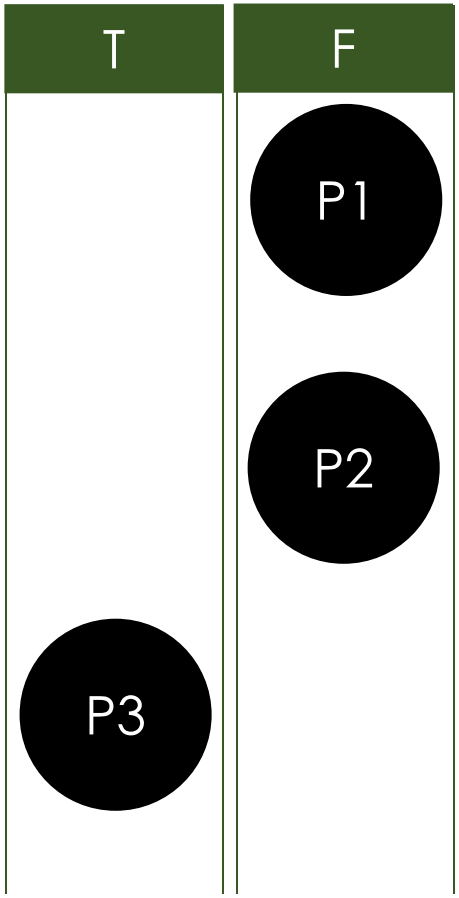


P1 退出临界区，P2的waiting 为false

3.1

当P1退出临界区时，有以下两种可能性

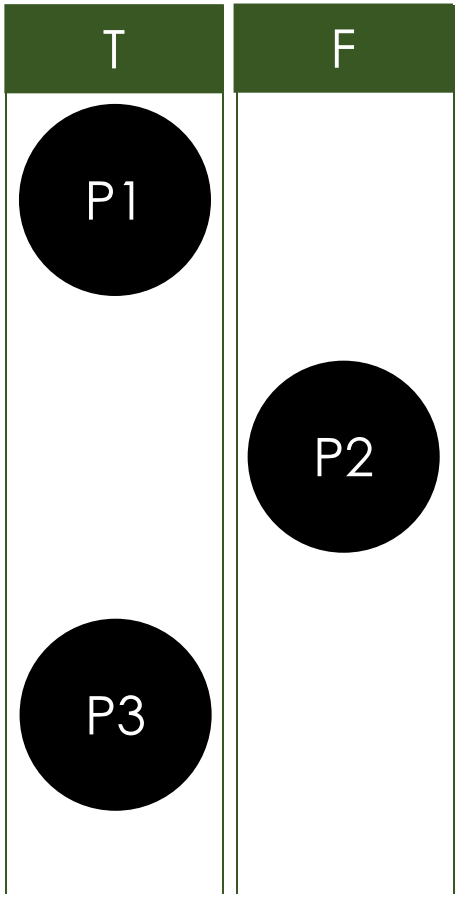
1. P1还没有进入下一次循环
2. P1进入了下一次循环



P1 退出临界区

当P1退出临界区时，有以下两种可能性

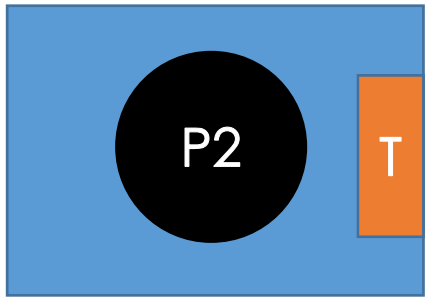
1. P1还没有进入下一次循环
2. P1进入了下一次循环

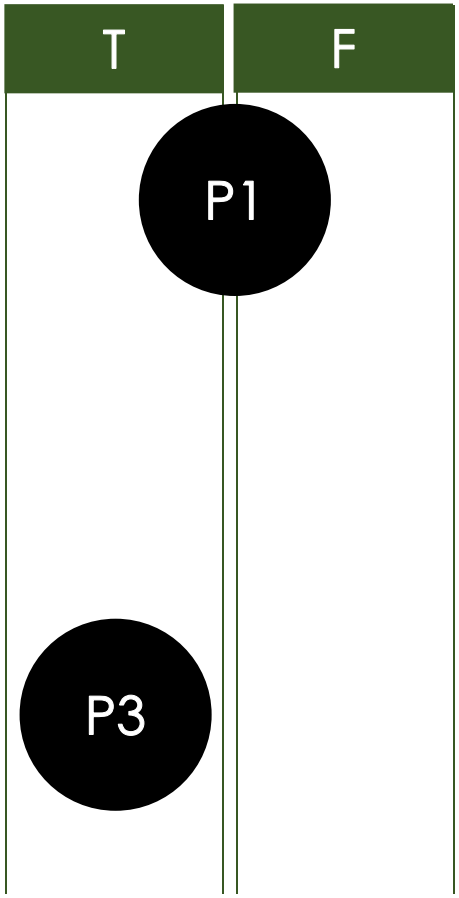


lock

waiting[]

P2进入临界区





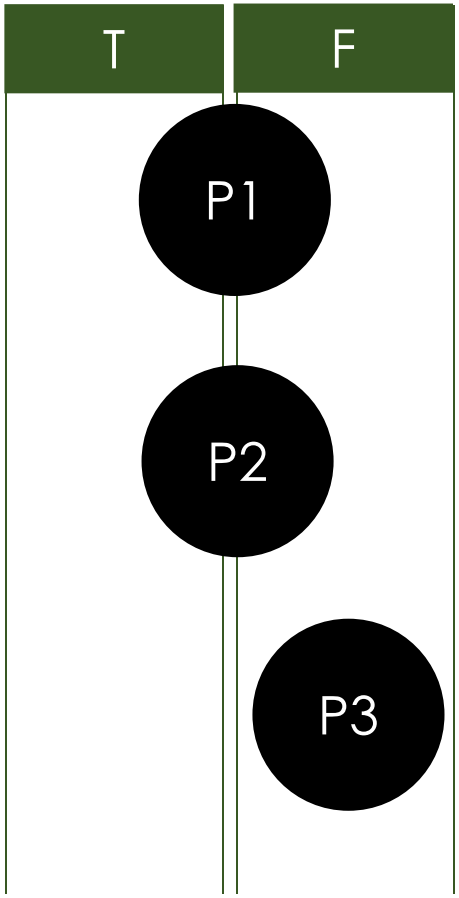
lock

waiting[]

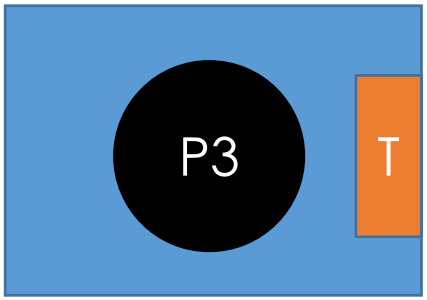
P2 退出临界区，P3的waiting 为false

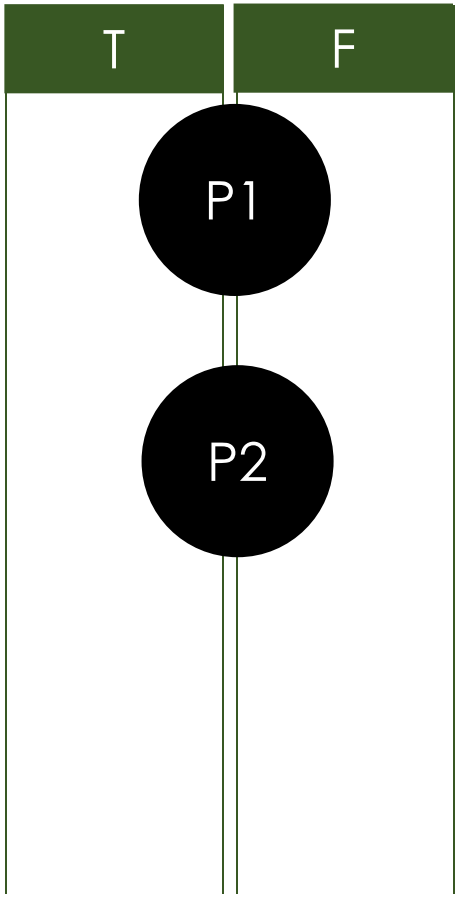
当P2退出临界区时，有以下两种可能性

1. P2还没有进入下一次循环
2. P2进入了下一次循环



P3进入临界区

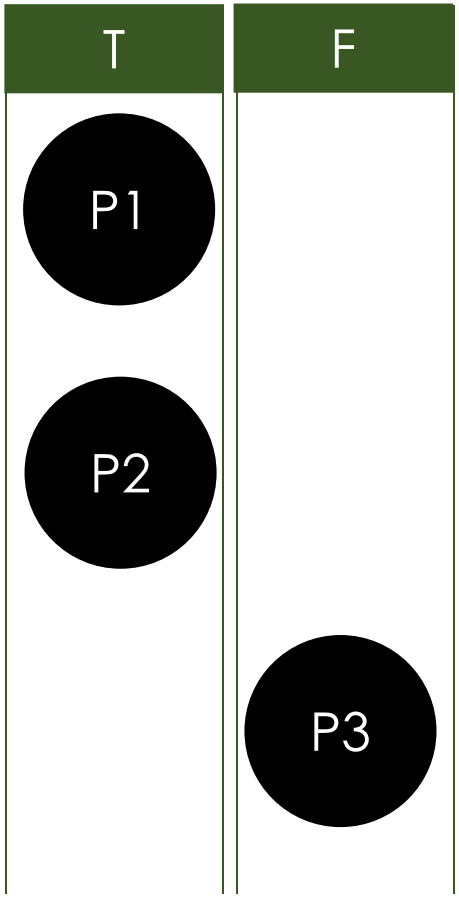




P3退出临界区

当P3退出临界区时，有以下两种可能性

1. 有等待进入临界区的进程
2. 没有等待进入临界区的进程



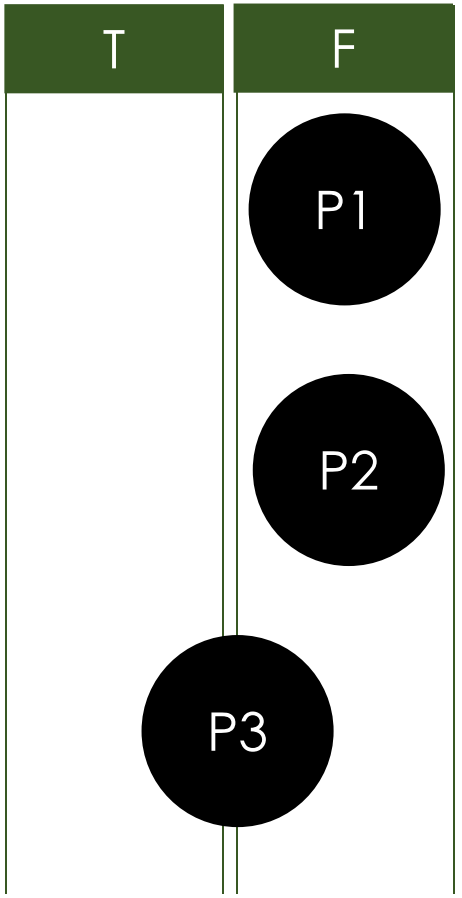
lock

waiting[]

P3退出临界区

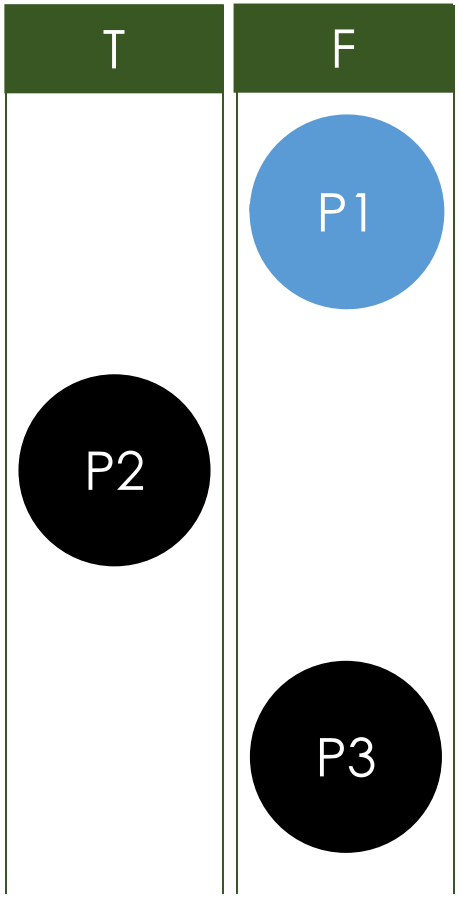
当P3退出临界区时，有以下两种可能性

1. 有等待进入临界区的进程
2. 没有等待进入临界区的进程



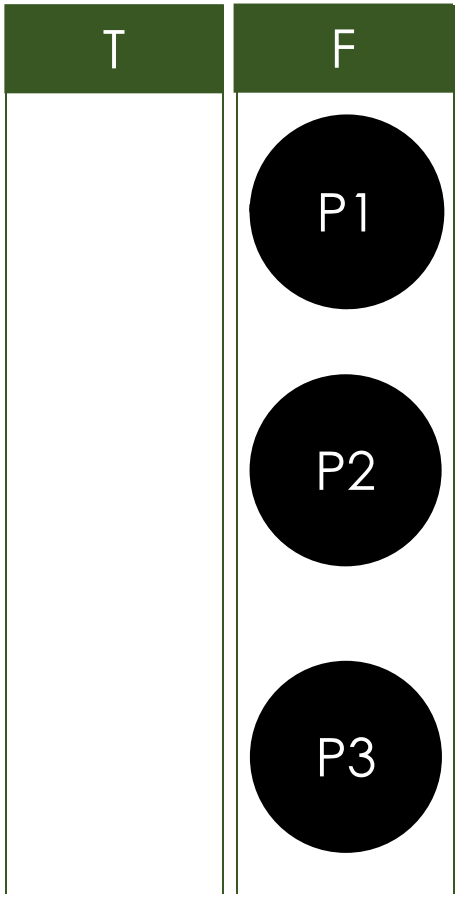
有等待进入临界区的进程

如果有等待进入临界区的进程的话，就把它的waiting 设置为 false, 从而使它可以进入临界区



有等待进入临界区的进程

如果没有等待进入临界区的进程的话，
就把 lock = false



5. 信号量（Semaphore）

- 无忙等待的同步工具，于1965年，由 Edsger Dijkstra提出
- 适用于单个或多个资源的同步操作
- 用S来表示信号量，S是整数变量
- 只能通过标准原子操作(atomic operation) 来访问信号量
  1. wait(S) operation : P(S) → S--
  2. signal(S) operation : V(S) → S++

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

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

信号量

1. 二进制信号量(binary semaphore)

   又称互斥锁（mutex lock）

   - 适用于单资源的共享，mutex值为资源数量，初始化为1
   - 信号量的值只能为 0 或 1

```
do{
    wait (mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (TRUE);
```

```
do{
    wait (mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (TRUE);
```

信号量

2. 计数信号量(counting semaphore)

- 适用于多资源共享，共享资源的数量为 n
- wait(n) 操作为减，signal(n) 操作为加，当n为0时表明所有资源都被占用

- 可以适用于优先约束（precedence constraint）

例子，假设要求P1的语句operation1完成之后，执行P2的语句operation2, 共享信号量 synch，并初始化为0

```
P1:
        operation 1;
        signal(synch);
P2:
        wait(synch);
        operation 2;
```

```
P0:
do{
    wait (mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (TRUE);
```

假设共享同类的两个资源（n = 2）

- 信号量 mutex 初始化为 2
- P0, P1, P2 竞争

```
P1:
do{
    wait (mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (TRUE);
```

```
P3:
do{
    wait (mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (TRUE);
```

信号量的实现

- 信号量的实现关键是保障wait( )和signal( )操作的原子执行，即必须保障没有两个进程能同时对同一信号量执行wait( )和signal( )操作。

- 保障方法

1. 单处理器环境下，禁止中断
2. 多处理器环境下，禁止每个处理器的中断。但这种方法即困难又危险

无忙等待的信号量实现

- 问题：忙等待（busy waiting），自旋锁（spinlock）
- 为了解决忙等待的问题，让忙等待的进程挂起（blocking operation），可以进入临界区时，让进程重新启动（wakeup operation）
- 挂起的含义是进程从运行状态转换成等待状态，
- 重启的含义是进程从等待状态转换成就绪状态
- 我们将信号量定义如下：

```
typedef struct {
        int value; //是整数值，是资源数量
        struct process *list;
} semaphore
```

无忙等待的信号量实现

- 把wait( ) 和 signal( )操作定义如下

S→list 是处于等待状态的进程队列

```
wait (semaphore *S) {
        S→value--;
        if (S→value < 0) {
                add this process to S->list;
                block();
        }
}
```

```
signal (semaphore *S) {
        S→value++;
        if (S→value <=0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

无忙等待的信号量实现

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

绝对值是被挂起的进程数量

This (<=) means there is [are] a process [processes] waiting to be awakened.

Now the calling process and P run concurrently. But there is no way to know which process will continue on a uniprocessor system

Which is better?

Comparison

busy-waiting vs. block()-wakeup()

Length of critical section vs. block-wakeup overhead

A：depend on the context switching or length of critical section

死锁（deadlock）

- 死锁：两个或多个进程无限地等待一个事件，而该事件只能由这些等待进程之一来产生

- Two or more processes are waiting infinitely for an event that can be caused by only one of the waiting processes

死锁（deadlock）

- 如以下P0和P1两个进程共享信号量（共享资源） *S* 和 *Q*，初值为1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
…	…
signal(S);	signal(Q);
signal(Q);	signal(S);

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

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

S, Q are initialized to 1, P0 and P1 interleaved execution

P0

```
wait(S)
wait(Q)
….
….
signal(S)
signal(Q)
```

P1

```
wait(Q)
wait(S)
….
….
signal(Q)
signal(S)
```

饥饿（starvation）

与死锁相关的另一个问题是饥饿问题

饥饿：无限期的等待，即进程在信号量内无限期的等待

Q: Then, what kind of case the indefinite blocking may occur ?

A: Add and remove processes from the list associated with a semaphore in LIFO(Last In First Out).

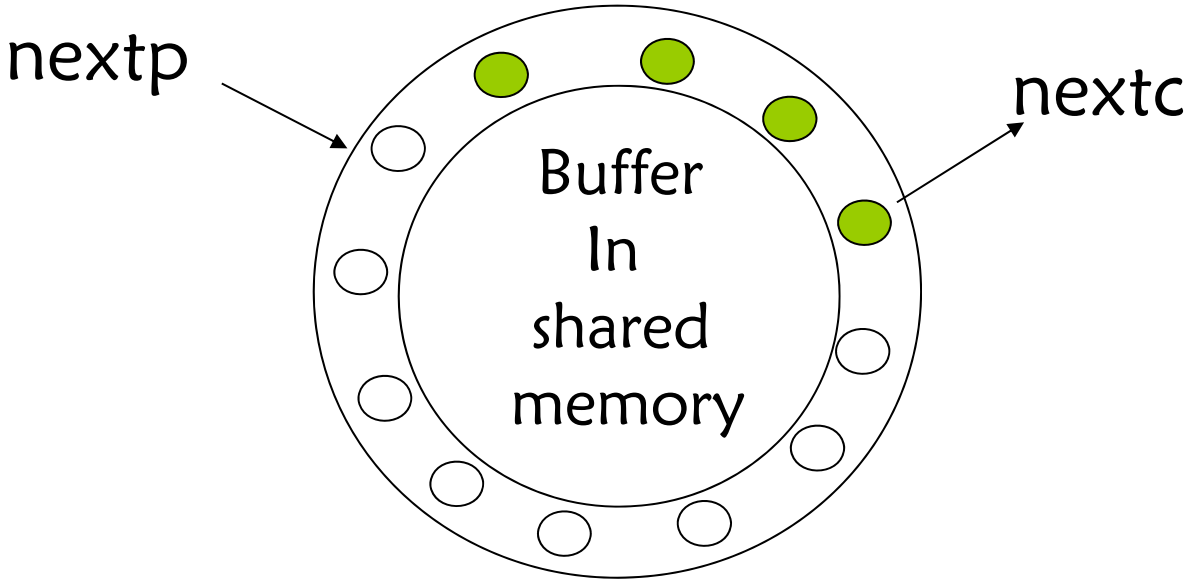
6. 经典同步问题

1. 有限缓冲问题(bounded buffer problem)
2. 读者和写者问题(reader and writer problem)
3. 哲学家进餐问题(dining philosophers problem)

6.1 有限缓冲问题

假定缓冲池中有 n 个缓冲项，每个缓冲项能存一个数据项

1. 当缓冲池满的时候，不能写（full）
2. 当缓冲池空的时候，不能读（empty）
3. 读的时候不能写，写的时候不能读（互斥）



6.1 有限缓冲问题

假定缓冲池中有 n 个缓冲项, 每个缓冲项能存一个数据项

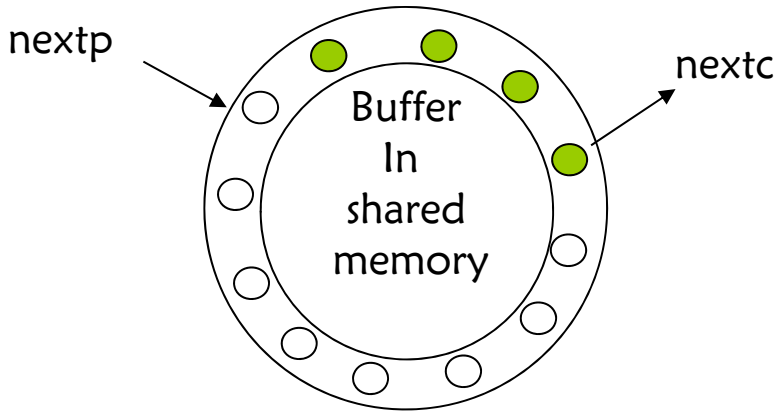
1. 用信号量 empty: 表示空缓冲项的个数
2. 用信号量 full： 表示满缓冲项的个数
3. 用信号量 mutex : 提供对缓冲池的读写互斥

- 信号量 mutex 初始化为1
- 信号量 full 初始化为 0
- 信号量 empty 初始化为n.

有限缓冲问题-生产者（写）

```
while (true) {
      //   produce an item
      wait (empty);//只要有空缓冲项，就写
      wait (mutex);
      // add the item to the  buffer
      signal (mutex);
      signal (full);
}
```

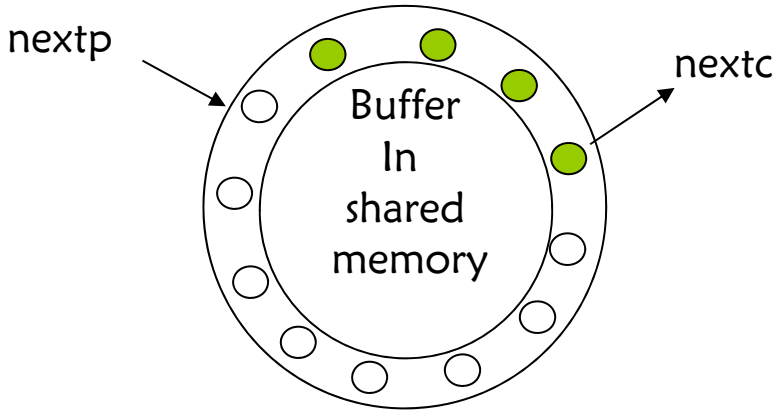
满缓冲项



有限缓冲问题-消费者（读）

```
while (true) {
    wait (full); //只要缓冲项里有数据，就读
    wait (mutex);
    //  remove an item from  buffer
    signal (mutex);
    signal (empty);
    //  consume the removed item
}
```

空缓冲项



有限缓冲问题

- The structure of the producer process

```
do {
        ...
       /* produce an item
        in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
       /* add next produced
          to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
       /* remove an item
          from buffer to
          next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
       /* consume the item
        in next_consumed */
        ...
} while (true);
```

6.2 读者-写者问题

读者与写者问题，如何确保同步

1. 读的时候不能写，写的时候不能读
2. 多位读者可以同时访问数据，需要知道读者数量
3. 只能由一个写者写数据，不能多个写者写数据

实现：

1. 读和写、写与写者之间互斥（信号量 wrt）
2. 跟踪读者（信号量 readcount）
3. readcount 的互斥 （信号量 mutex）

写者进程

1. 信号量 wrt 为读者和写者进程共享，初始化为 1，从而达到写操作互斥的目的
2. 变量 readcount 用来跟踪有多少进程正在读对象，初始化为 0
3. 信号量 mutex 用于确保在更新变量 readcount 的互斥，初始化为 1

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

读者进程

```
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) ;
}
```

第一位读者加锁

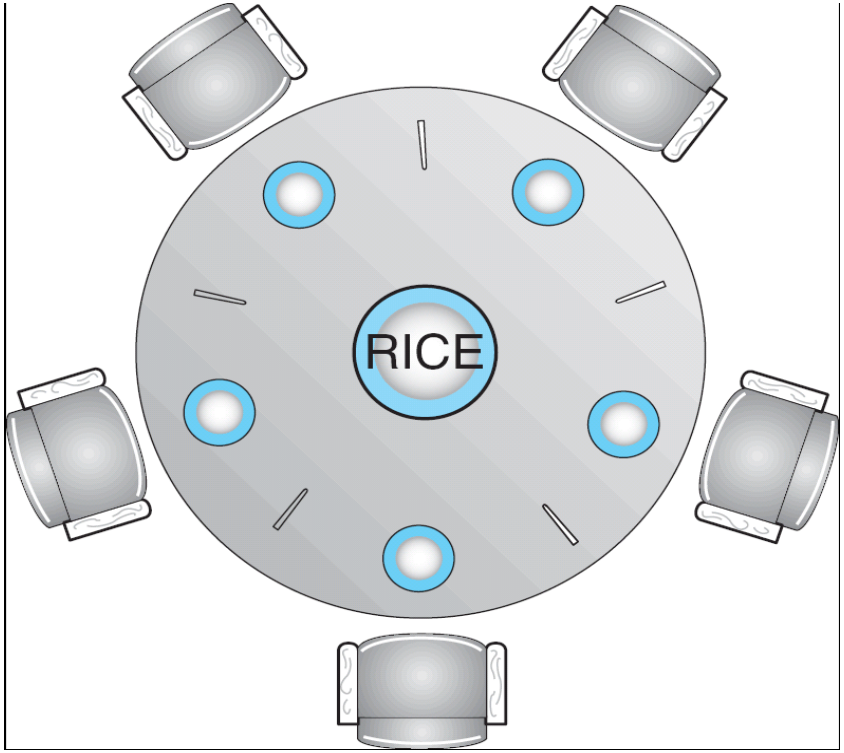
最后一位读者解锁

6.3 哲学家进餐问题

原来吃饭是哲学问题呀！

1. 哲学家们用一生来思考和吃饭
2. 一碗米饭
3. 5只筷子
4. 同时有两只筷子才能吃饭

是多个进程之间多个资源共享的问题



共享数据

semaphore chopstick[5];

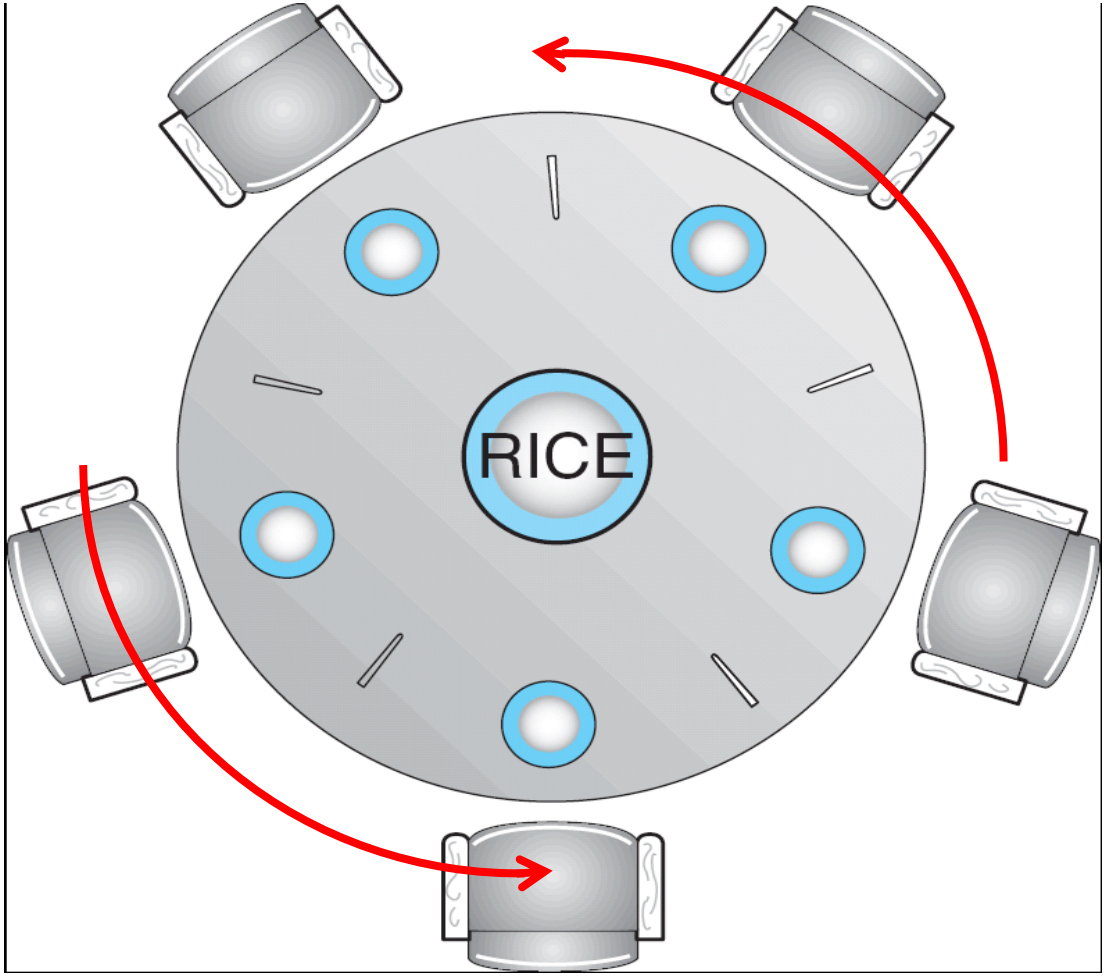
// Initially all values are 1

哲学家 i 进程

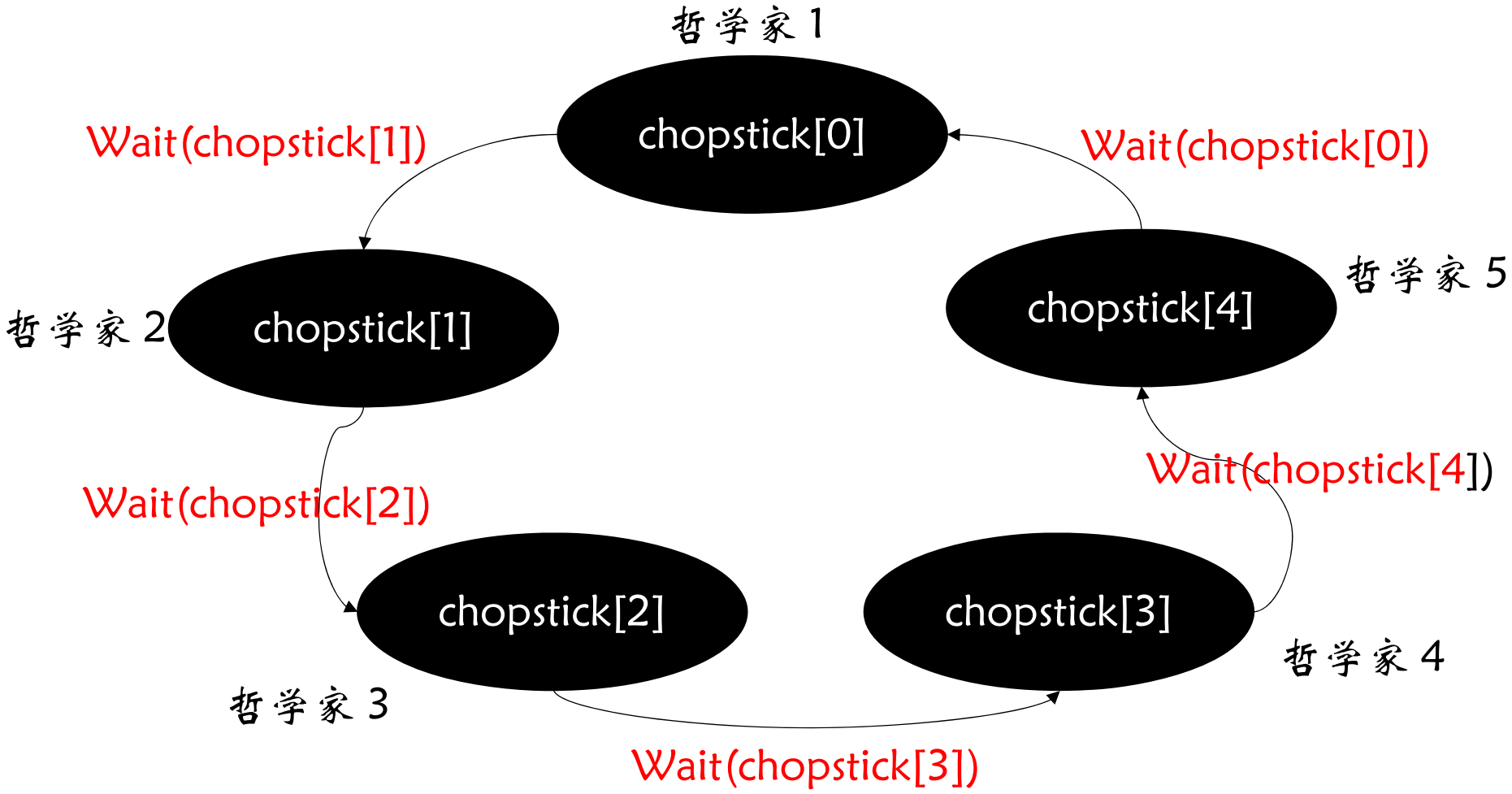
```
do {
    wait(chopstick[i])  //左手边筷子
    wait(chopstick[(i+1) % 5]) //右手边筷子
    …
   // eat
    …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    …
   // think
    …
} while (true);
```

What is the problem of this algorithm?

死锁（Deadlock）



死锁


哲学家 1
chopstick[0]
Wait(chopstick[1])
Wait(chopstick[0])
哲学家 2
chopstick[1]
chopstick[4]
哲学家 5
Wait(chopstick[4])
Wait(chopstick[2])
chopstick[2]
chopstick[3]
哲学家 4
哲学家 3
Wait(chopstick[3])

7. 管程（monitor）

- 由于发生如下操作错误可能会出现死锁、饥饿
  1. 交换wait() 和 signal() 操作顺序
  2. 用 wait() 替代了 signal() 操作
  3. 省略了 wait() 或 signal() 操作
  4. 。。。

- 管程(语言构造）的实现
  1. 利用抽象化(abstraction)概念
  2. 利用信号量

管程

1. 管程是一种用于多线程互斥访问共享资源的程序结构
   - 采用面向对象方法，简化了进程见得同步控制
   - 任一时刻最多只有一个县城执行管程代码
   - 正在管程中的线程可临时放弃管程的互斥访问，等待事件出现时恢复
2. 为什么要引入管程
   - 把分散在各进程中的临界区集中起来进行管理
   - 防止进程有意或无意的违法同步 操作
   - 便于用高级语言来书写程序，也便于程序正确性验证。

管程

1. 管程的使用
   - 在对象/模块中， 收集相关共享数据
   - 定义访问共享数据的方法
2. 管程的组成
   - 一个锁：控制管程代码的互斥访问
   - 0或多个条件变量：管理共享数据的并发访问
   - 一个条件变量对应于一个等待队列，每个条件变量有一个wait()和signal()操作

条件变量

- 当调用管程过程的进程无法运行时，用于阻塞进程的一种信号量

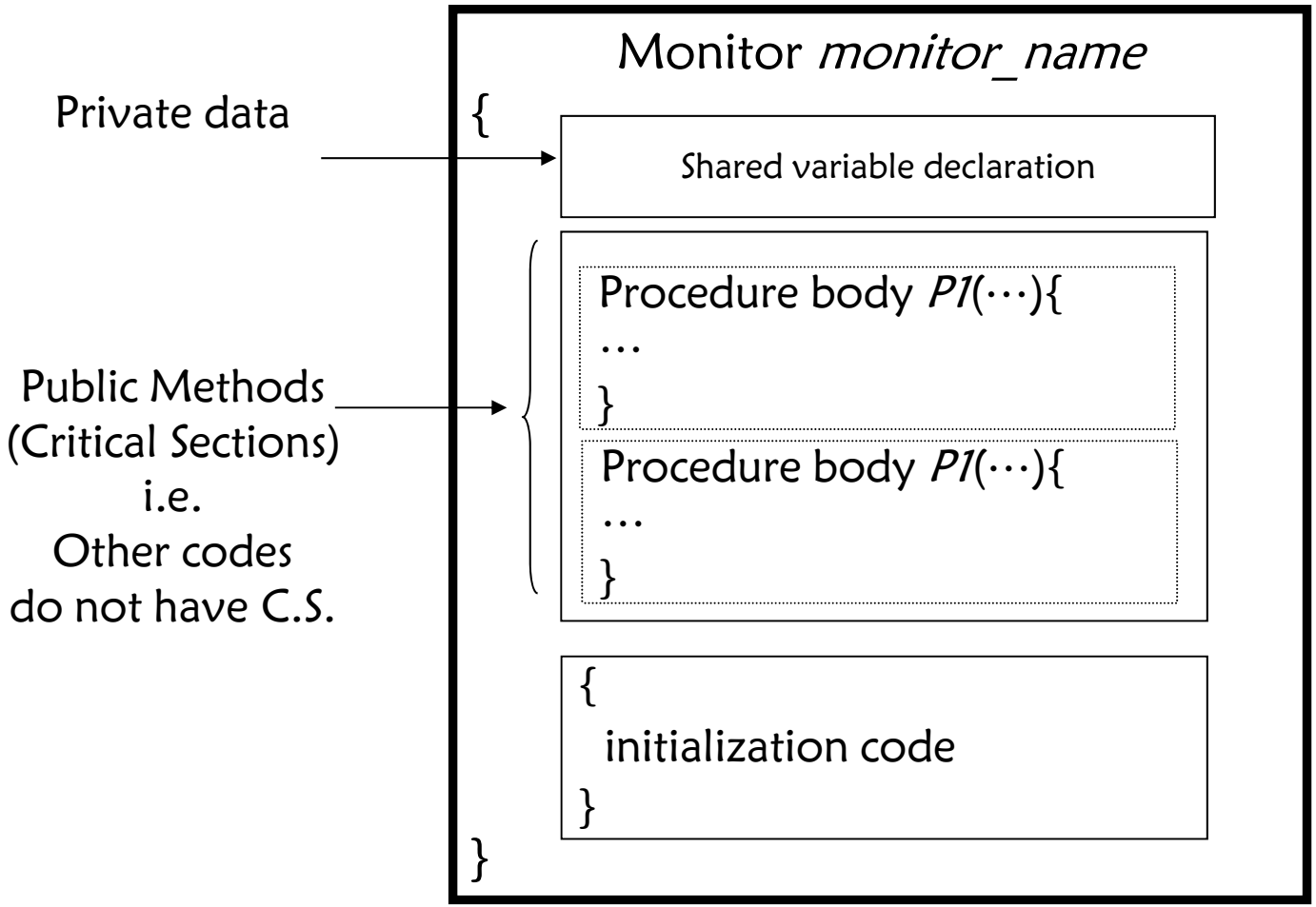
- 当一个管程过程发现无法继续时，它在某些条件变量 condition 上执行wait 操作，这个动作引起调用进程阻塞
- 另一个进程可以通过对其伙伴在等待的同一个条件变量condition 上执行同步原语signal操作来唤醒等待的进程

抽象化管程的语法

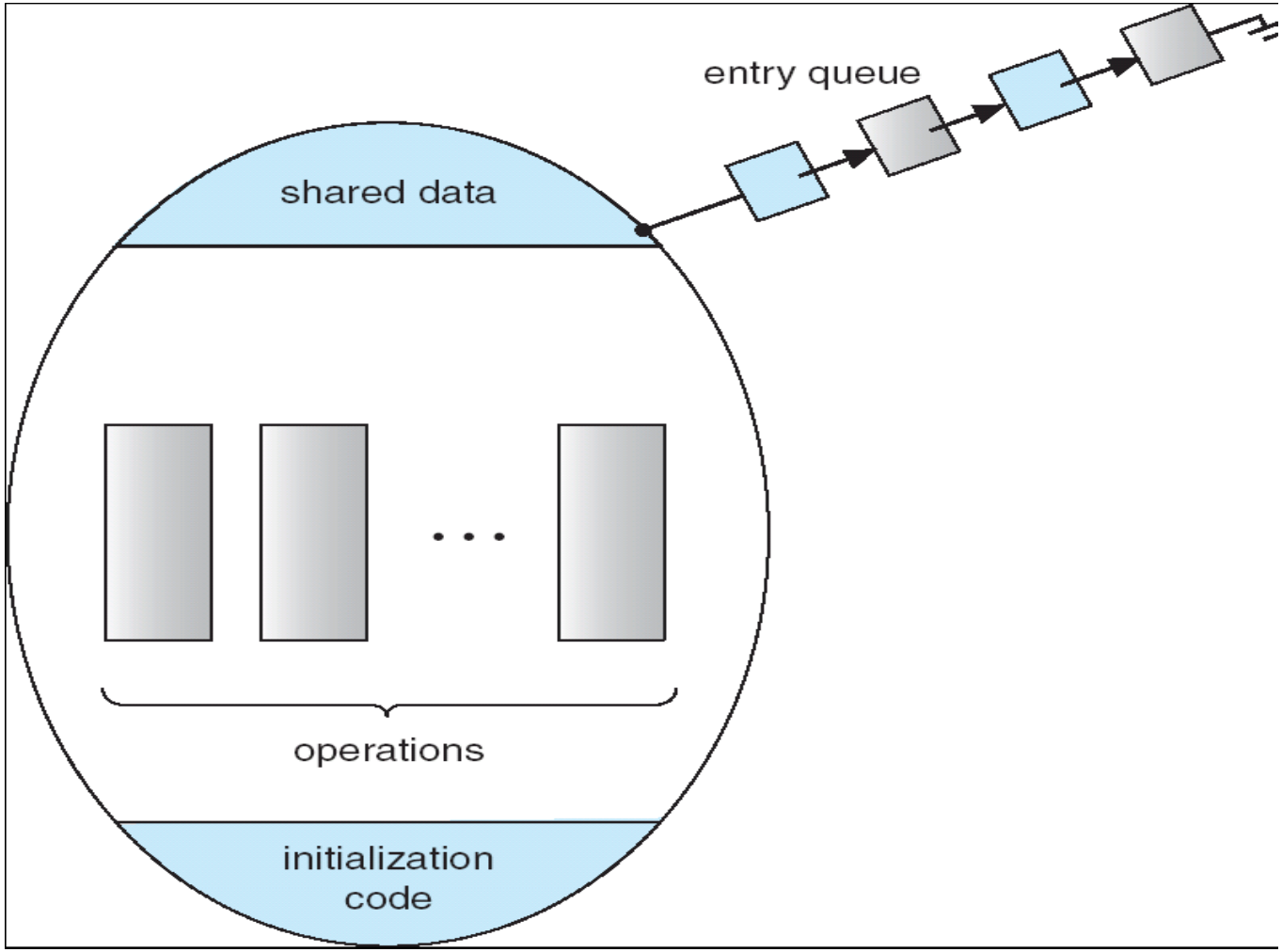
```
monitor monitor_name
{
    // shared variable declarations
    condition x;
    procedure P1 (…) { …. }
        …
    procedure Pn (…) {……}

   initialization code ( ….) { … }
        …
}
```

管程的语法

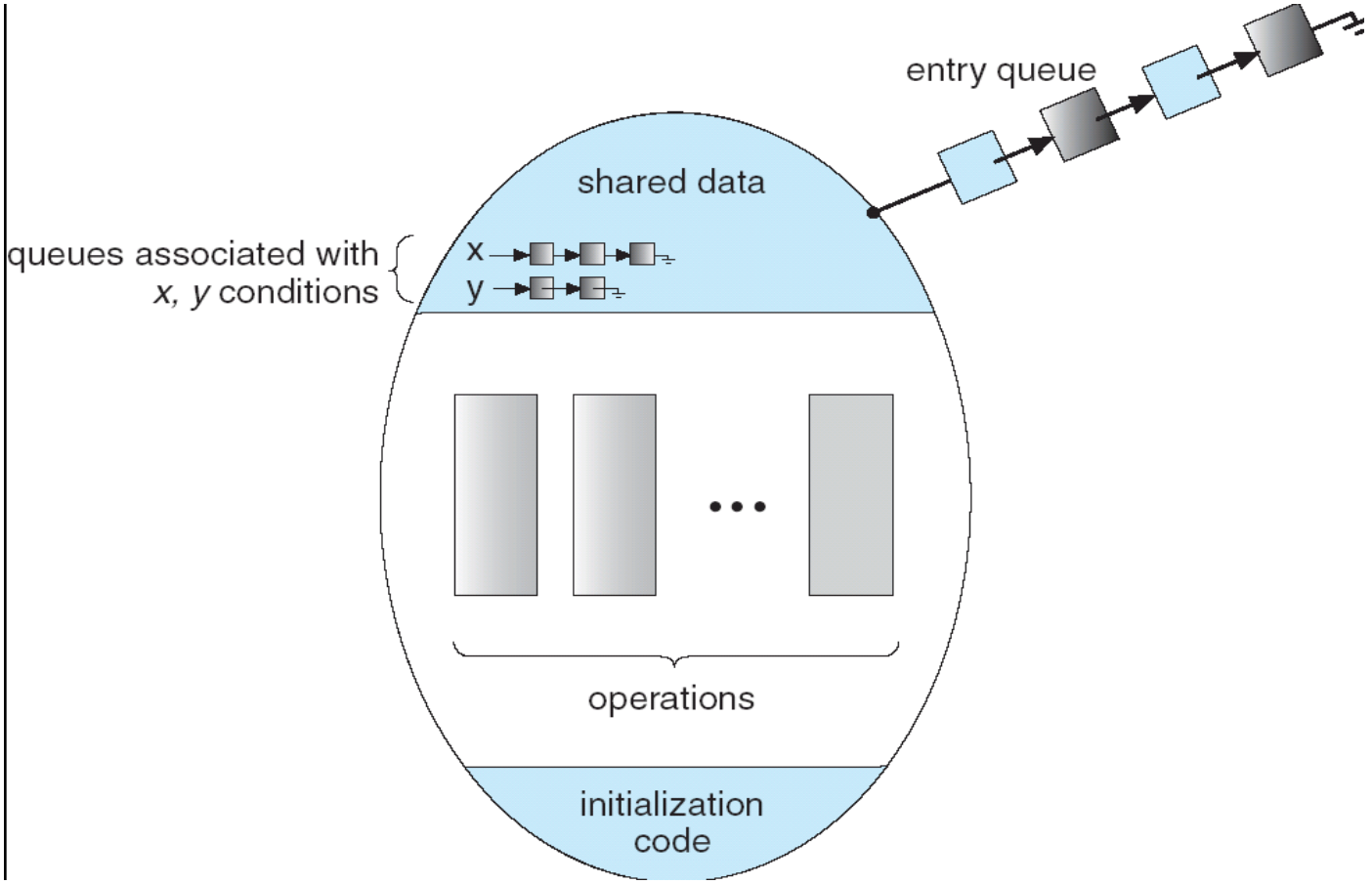

Monitor *monitor_name*
{
Private data
Shared variable declaration
Public Methods
(Critical Sections)
i.e.
Other codes
do not have C.S.
Procedure body *P1*(…){
…
}
Procedure body *P1*(…){
…
}
{
initialization code
}
}

管程的语法



条件变量

- 设置条件变量（单个，多个），对条件变量仅提供的wait ( ) 和signal ( )操作



哲学家进餐问题的管程解决方案

制定如下规定

1. 区分哲学家所处的三个状态
   - THINKING, HUNGRY, EATING
2. 哲学家 i 只有在其两个邻居不进餐时，才能拿起筷子进餐

(state[(i+4)%5] != EATING) and (state[(i+1)%5] != EATING)

3. 声明条件变量-哲学家
   - condition self[5]， 对条件变量仅有的操作是wait()和 signal()

哲学家进餐问题的管程解决方案

```
monitor DP {
        enum { THINKING; HUNGRY, EATING) state [5] ;
        condition self [5];

        void pickup (int i) {
                state[i] = HUNGRY;
                test(i);
                if (state[i] != EATING)
                        self[i].wait();
        }

        void putdown (int i) {
                state[i] = THINKING;
                // test left and right neighbors
                test((i + 4) % 5); // front
                test((i + 1) % 5); // back
}
```

哲学家进餐问题的管程解决方案

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

 initialization_code() {
      for (int i = 0; i < 5; i++)
          state[i] = THINKING;
 }
}
```

哲学家进餐问题的管程解决方案

- 每位哲学家 i 进行 pickup() 和 putdown() 操作

DP.pickup (i)

… eat …

DP.putdown (i)

… thinking…

利用信号量实现管程机制

- 对实现管程之间的互斥引入
  - 信号量 mutex
- 直到用餐的哲学家放下筷子，其他哲学家不能拿起筷子，即等待(ready and wait to active)用餐哲学家引入
  - 信号量 next
  - 等待中的哲学家数量 - 整数变量 next_count
- 为实现条件变量（blocking）哲学家自身，引入
  - 信号量 x_sem
  - 条件变量的数量 - x_count

基于信号量实现管程机制

信号量

semaphore mutex; // (初始化为 1)
semaphore next; // (初始化为 0)

计数变量 in the ready queue

int next_count = 0;

每个子程序实现为

```
wait(mutex);
    …
  body of F;
    …
if (next_count > 0)//如果有等待的进程就释放
  signal(next)
else
  signal(mutex);
```

基于信号量实现管程机制

- 条件变量 *x*, 设置如下:

  semaphore x_sem; // (initially = 0)

  int x_count = 0; //in the blocking queue

- 对条件变量 x.wait() 操作实现如下

```
Begin
    x_count++; // the number of blocking process
    if (next_count > 0)
        signal(next);
    else
        signal(mutex);
    wait(x_sem);
    x_count--;
End
```

基于信号量实现管程机制

- 对条件变量 x的signal() 操作实现如下

```
Begin
    if (x_count > 0) {
            next_count++;
            signal(x_sem);
            wait(next);
            next_count--;
    }
End
```

Q & A