上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L6. 进程同步

宋卓然

上海交通大学计算机系

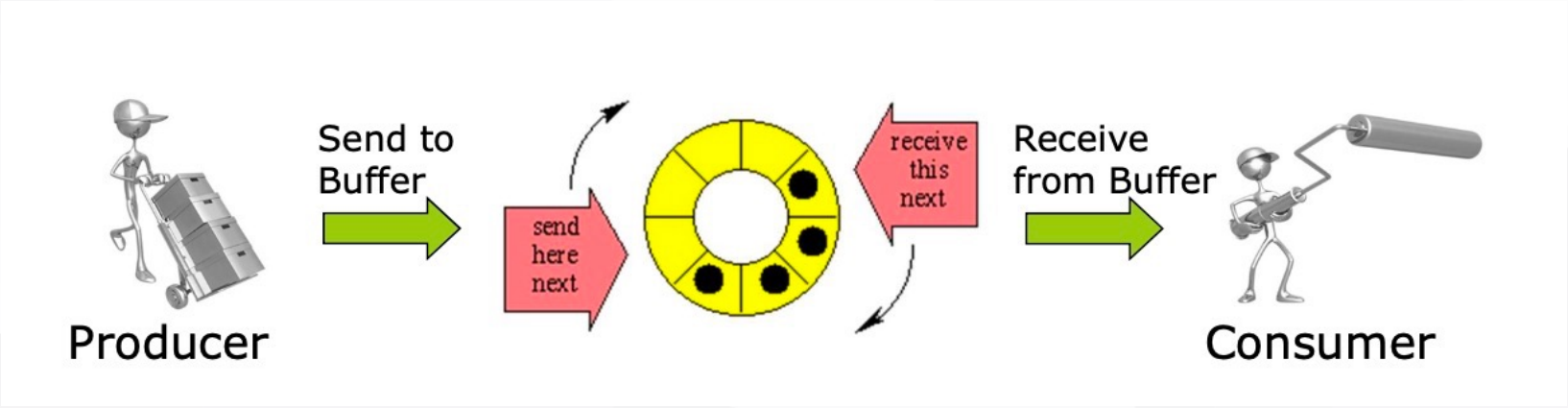
songzhuoran@sjtu.edu.cn

背景

- 进程可以并发访问
- 但并发访问共享数据可能导致数据不一致
- 如何保证数据的一致性、完整性

生产者-消费者问题

- 进程协作的范式，生产者产生由消费者消费的信息
- 一个进程产生的数据，可能被另一个进程所使用



生产者-消费者问题 共享内存

- 共享内存，大小为10

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer

Producer

```
while (true) {
  /* Produce an item */
 while (((in + 1) % BUFFER_SIZE)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
  while (in == out)
        ; // do nothing
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
      return item;
}
```

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer
- 缺陷
  - 最多只有BUFFER_SIZE-1 项
  - 一直处于等待
- 希望允许BUFFER_SIZE

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer

Producer

```
while (true) {
    /* produce an item */
    while (in % BUFFER_SIZE == out && in != out)
            ;           /* waiting */
    buffer[in % BUFFER_SIZE] = item;
    if ((in + 1) % BUFFER_SIZE == out)
            in = out + BUFFER_SIZE;
    else
            in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
    while (in == out)
            ; // do nothing

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
return item;
}
```

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer。限定BUFFER_SIZE项
- 假想为填充所有缓冲区的消费者-生产者问题提供一个更好的解决方案

Producer

```
while (true) {
    /*  produce an item and put in nextProduced  */
    while (counter == BUFFER_SIZE)  ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true)  {
    while (counter == 0) ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /*  consume the item */
}
```

生产者-消费者问题 共享内存

- 允许两个进程并发操作变量counter，导致错误
- 多个进程并发访问和操作同一数据并且执行结果与特定访问顺序有关，称为**竞争条件**
- 为了防止竞争条件，需要确保一次只有一个进程可以操作counter
- 对于抢占式内核，需要确保内核数据结构不会导致竞争条件

T_0:	生产者	执行	$register_1 = counter$	$\{register_1 = 5\}$
T_1:	生产者	执行	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	消费者	执行	$register_2 = counter$	$\{register_2 = 5\}$
T_3:	消费者	执行	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	生产者	执行	$counter = register_1$	$\{counter = 6\}$
T_5:	消费者	执行	$counter = register_2$	$\{counter = 4\}$

临界区问题

- 假设某个系统有n个进程
- 每个进程有一段代码，称为临界区
  - 进程在执行该区时可能修改公共变量、更新一个表、写一个文件
  - 当一个进程在临界区内执行时，其他进程不允许在它们的临界区内执行
- 临界区问题是设计一个协议以便协作进程
- 在进入临界区前， 每个进程应请求许可。实现这一请求的代码区段称为进入区。临界区之后可以有退出区。其他代码为剩余区

```
do {
        entry section          临界区
            critical section
        exit section           剩余区
            remainder section
} while (TRUE);
```

临界区问题

- 临界区问题的解决方案应满足如下三条要求
  - 互斥：如果进程P，在其临界区内执行，那么其他进程都不能在其临界区内执行
  - 进步：如果没有进程在其临界区内执行，并且有进程需要进入临界区，那么只有那些不在剩余区内执行的进程可以参加选择，以便确定谁能进入临界区
  - 有限等待：从一个进程做出进入临界区的请求直到这个请求允许为止，其他进程允许进入其临界区的次数具有上限

临界区问题的解决方案

- Peterson解决方案
- 硬件同步
- 互斥锁
- 信号量

Peterson解决方案

- 适用于两个进程交错执行临界区与剩余区
- 进程共享两个数据项
- 变量turn表示哪个进程可以进入临界区。即如果turn==i，那么进程Pi允许在临界区内执行 。数组flag表示哪个进程准备进入临界区。例如，flag[i]为true，那么进程Pi准备进入临界区
- turn的最终值决定了哪个进程允许先进入临界区

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

Peterson解决方案

- 证明
  - 互斥成立
    - turn不可能同时等于i和j
  - 进步要求成立
    - 当Pj退出临界区时，会设置flag[j]=false，允许Pi进入临界区
  - 有限等待成立
    - Pi进程在Pj进程进入临界区后最多一次就能进入临界区

硬件同步

- 所有方案都是基于加锁为前提的，通过锁来保护临界区
- 对于单处理器环境，临界区问题可简单地加以解决：在修改共享变量时禁止中断出现
- 但对于多处理器，中断禁止过于耗时，需要将消息传递到所有处理器
- 多处理器提供特殊硬件指令
  - 原子地交换两个字（不可中断）
  - TestAndSet()
  - Swap()

硬件同步

- TestAndSet()
  - 定义

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

在执行3条指令中间不允许产生中断

  - 从内存读值
  - 测试该值是否为1（然后返回真假）
  - 内存值设置为1

硬件同步

- 多个进程（可以大于2个）同时进入循环，希望获得锁
  - TestAndSet互斥实现：设置共享布尔变量lock，初始化为false

```
do {
        while ( TestAndSet ( &lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //    remainder section

} while (TRUE);
```

硬件同步

- TestAndSet()
  - 有限等待：设置布尔变量waiting[n]和lock，初始化为false；一个进程退出临界区时，会循环扫描数组（i+1,i+2,…,n-1,0,…,i-1），并根据这一顺序而指派第一个等待进程（waiting[j]==true）作为下次进入临界区的进程。

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

硬件同步

- Swap()
  - 定义：两个变量

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

  - 交换内存中的两个值

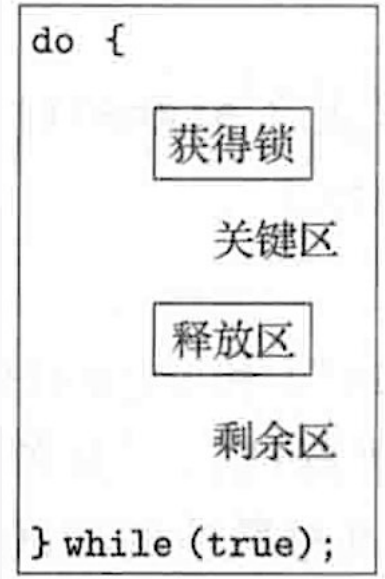
硬件同步

- Swap()
  - 互斥实现：将变量key初始化为true

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section

         lock = FALSE;

                //    remainder section

 } while (TRUE);
```

互斥锁

- 采用硬件同步较复杂，不能让程序员直接使用
- 软件工具---互斥锁（mutex lock）
  - acquire()获得锁
  - release()释放锁



互斥锁

- 互斥锁（mutex lock）有一个布尔变量available，表示锁是否可用
- available初始化为true

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

- 缺点：忙等待
  - 当一个进程在临界区，其他进程在进入临界区时需要连续循环调用acquire，也称为“自旋锁”，浪费CPU周期

信号量

- 互斥锁，最简单
- 希望设计一种更鲁棒、提供更高级功能的方法：信号量 S，整型变量
- 两个标准原子操作，用于修改S
  - wait()，最初被称为P()，把信号量减1
  - signal()，最初被称为V()，把信号量加1
- 信号量的修改应不可分割地执行

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

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

信号量

- 信号量
  - 二进制信号量：0、1，类似于互斥锁
  - 计数信号量：不受限制，用于控制访问具有多个实例的资源，初值为资源数量
    - 使用资源，对信号量执行wait()，减少信号量计数
    - 释放资源，对信号量执行signal()，增加信号量计数
    - 当信号量等于0，表示所有资源都在使用中，之后的进程进入阻塞
- 但现在依然具有忙等待的问题！
  - 选择释放哪个进程？使用FIFO

信号量

- 核心：当执行操作wait()并发现信号量不为正时，不是忙等待，而是阻塞自己
- 定义进程链表

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

- wait()

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

- signal()

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

信号量

- 当一个进程必须等待信号量时，被添加到进程链表，被阻塞
- signal()从等待进程链表取走一个进程，并加以唤醒
- 引入两个新操作
  - sleep()挂起/阻塞调用它的进程
  - wakeup()重新启动阻塞进程的执行
- 信号量可以为负数，为负数时，它的绝对值是等待它的进程数

信号量的使用 实现互斥

- 临界区

wait(mutex);　　P()操作

…

临界区

…

signal(mutex);　V()操作

死锁

- 两个或多个进程无限等待一个事件，而该事件只能由这些等待进程之一来产生，就出现了死锁

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

- 假设P0执行wait(S)，P1执行wait(Q)。当P0执行wait(Q)时，它必须等待，直到P1执行signal(Q)；类似的，当P1执行wait(S)时，它必须等待，直到P0执行signal(S)。由于两个进程都不执行signal()，产生死锁

优先级反转

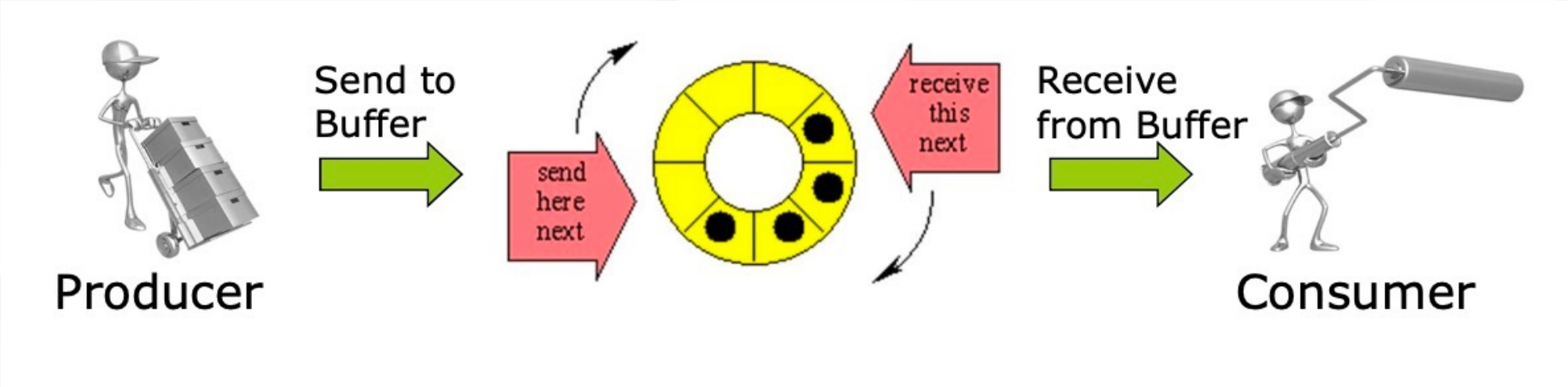
- 若一个较高优先级进程需要读取、修改内核数据，而该数据正在被较低优先级的进程访问，会出现调度挑战
- 较高优先级进程需要等待较低优先级完成，但若较低优先级被另一较高优先级进程抢占，则发生优先级反转
- 三个进程L、M、H，优先级为L<M<H，当H需要资源，而L正在使用，则H需要等待L；但此时若M进入可运行态，并抢占L，则H还会被M影响，称为“优先级反转”
- 解决方案：优先级继承协议，正在访问资源的进程获得需要访问它的更高优先级进程的优先级，直到它使用完成并恢复优先级

经典同步问题

- 有界缓冲问题
- 读者-作者问题
- 哲学家就餐问题

有界缓冲问题

- 有n个缓冲区，不能超过缓冲区访问数据



有界缓冲问题

- 共享数据结构
  - mutex=1
  - empty=n
  - full=0

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

判断是否写满n个缓冲区

获得锁

生产者

```
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);
```

消费者

读者-作者问题

- 读数据库的进程：读者
  - 不需要修改数据，允许多个读者同时
- 写数据库的进程：作者
  - 读取和修改数据，只允许一个作者
- 当作者在写入数据库同时读者希望访问数据库，产生读者-作者问题
- 要求作者在写入数据库时具有共享数据库独占访问权
- “第一”读者-作者问题
  - 要求读者不应保持等待，除非作者已获得权限使用共享对象
- “第二”读者-作者问题
  - 一旦作者就绪，就会尽快执行

“第一”读者-作者问题的解答方法（读者优先）

- 共享数据结构
  - wrt=1, readcount=0（有多少个读者）, mutex=1
- 一个读者在wrt上等待，n-1个读者在mutex上等待

```
do {
        wait (wrt) ;

        //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

作者

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
            wait (wrt) ;
        signal (mutex)

        // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

确保对readcount互斥保护

确保没有作者在写入

确保对readcount互斥保护

所有读者完成读操作释放锁

读者

“第二”读者-作者问题的解答方法（作者优先）

- 共享数据结构
  - int readcount = 0, writecount = 0;
  - mutexrc = 1, mutexwc = 1, wrt = 1, rd = 1;

作者

```
do {
    wait (mutexwc);
    writecount ++;
    if (writecount == 1)
        wait (rd);
    signal (mutexwc);

    wait (wrt);
    // writing is performed
    signal(wrt);

    wait (mutexwc);
    writecount - -;
    if (writecount == 0)
        signal (rd);
    signal (mutexwc);
} while (TRUE);
```

确保没有读者在读

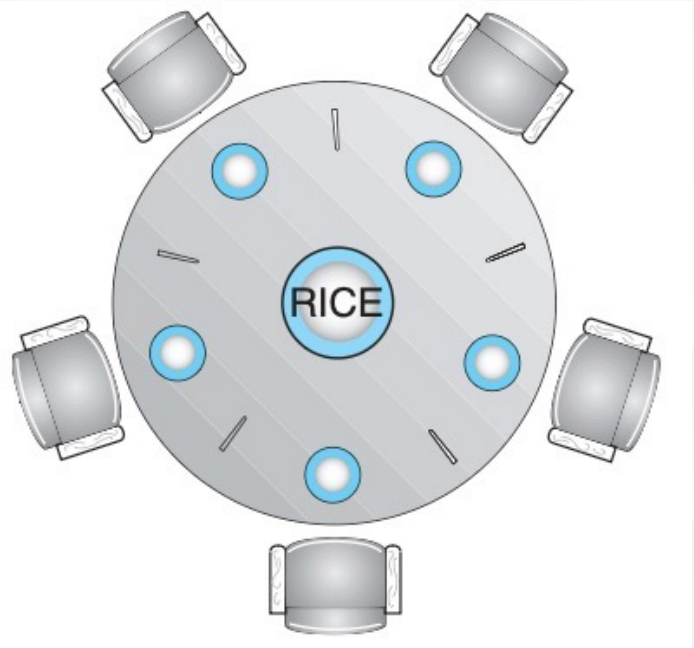
```
do {
    wait (rd);
    wait (mutexrc);
    readcount ++;
    if (readcount == 1)
        wait (wrt);
    signal (mutexrc);
    signal (rd);

    //reading is performed

    wait (mutexrc);
    readcount - -;
    if (readcount == 0)
        signal (wrt);
    signal (mutexrc);
} while (TRUE);
```

增加了rd变量，只有当作者都被处理完，才会进入下面的wait(mutexrc)

读者

哲学家就餐问题

- 桌上有5根筷子，当一个哲学家拿起两根筷子就可以吃饭，但互相不交流
- 经典的同步问题



哲学家就餐问题

- 共享数据结构
  - chopstick[5]，均初始化为1
- 可以确保两个邻居不同时进食，但可能导致死锁
  - 假设5个哲学家同时拿起左边的筷子，此时所有筷子的信号量为0，但当他们试图拿起右边的筷子，都会被推迟

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

死锁！

哲学家就餐问题

- 改进一下
  - 5个哲学家同时拿起左边的筷子，同时放下，同时等待

```
do{
        wait(chopstick[i]);
        if(chopstick[(i+1)%5]){
                wait(chopstick[[(i+1)%5]);
                break;
        }
        else{
                signal(chopstick[i]);
                wait some time();
        }
}while(true);
```

死锁！

哲学家就餐问题

- 改进一下
  - 5个哲学家同时拿起左边的筷子，同时放下，随机等待一段时间

```
do{
        wait(chopstick[i]);
        if(chopstick[(i+1)%5]){
                wait(chopstick[[(i+1)%5]);
                break;
        }
        else{
                signal(chopstick[i]);
                wait random time();
        }
} while(true);
```

可行，但不够完美

哲学家就餐问题

- 改进一下
  - 将拿筷子的部分保护起来，利用信号量mutex

```
do{
        wait(mutex);
        wait(chopstick[i]);
        if(chopstick[(i+1)%5]){
                wait(chopstick[[(i+1)%5]);
                break;
        }
        else{
                signal(chopstick[i]);
                wait random time();
        }
        signal(mutex);
} while(true);
```

可行，但只允许一位哲学家进食

哲学家就餐问题

- 改进一下
  - 要么不拿，要么一次性拿两个筷子

S1 哲学家进入饥饿状态

S2 如果左邻居或右邻居正在进餐，则等待；

否则转S3

S3 拿起两个筷子

S4 吃饭

S5 放下左右两个筷子

S6 重新循环

哲学家就餐问题

- 改进一下
  - 要么不拿，要么一次性拿两个筷子

S1 哲学家进入饥饿状态

S2 如果左邻居或右邻居正在进餐，则等待；否则转S3

S3 拿起两个筷子

S4 吃饭

S5 放下左右两个筷子

S6 重新循环

哲学家自己如何解决这个问题

计算机程序如何解决这个问题

S1 哲学家进入饥饿状态

S2 如果左邻居或右邻居正在进餐，则进程进入阻塞态；否则转S3

S3 拿起两个筷子

S4 吃饭

S5 放下左边的筷子，看看左边的邻居是否能进餐（饥饿状态、两个筷子都在），若能则唤醒它

S6 放下右边的筷子，看看右边的邻居是否能进餐，若能则唤醒它

S7 重新循环

哲学家就餐问题

- 共享数据结构
  - int **state**[5]（临界资源）
  - mutex = 1, s[N]=0;

```
void philosopher(int i){
        do{
                think();
                wait_chop(i);
                eat();
                put_chop(i);
        } while(true);
}

void test_take_chopstick(int i){
        if(state[i]== HUNGRY && state[left]!=EATING && state[right]!=EATING{
        state[i]=EATING;   两个叉子到手了
        signal(s[i]);          通知第i人可以吃饭了，s[i]=1
}
```

```
void wait_chop(int i){
        wait(mutex);
        state[i]=HUNGRY;
        test_take_chopstick(i);
        signal(mutex);
        wait(s[i]);              没有拿到叉子便阻塞自己
}

void put_chop(int i){
        wait(mutex);
        state[i]=THINKING;   把叉子放回去
        test_take_chopstick(left);
        test_take_chopstick(right);
        signal(mutex);
}
```

状态是一个需要被互斥保护的

试图唤醒其他人

哲学家就餐问题

- 可以确保两个邻居不同时进食，但可能导致死锁
  - 假设5个哲学家同时拿起左边的筷子，此时所有筷子的信号量为0，但当他们试图拿起右边的筷子，都会被推迟
- 死锁的补救措施
  - 允许最多4个哲学家同时坐在餐桌
  - 只有一个哲学家两边的筷子均可用时，他才拿起他们
  - 使用非对称解决方案。即单号哲学家先拿左边的筷子，接着右边的筷子；而双号哲学家先拿右边的筷子，接着左边的筷子

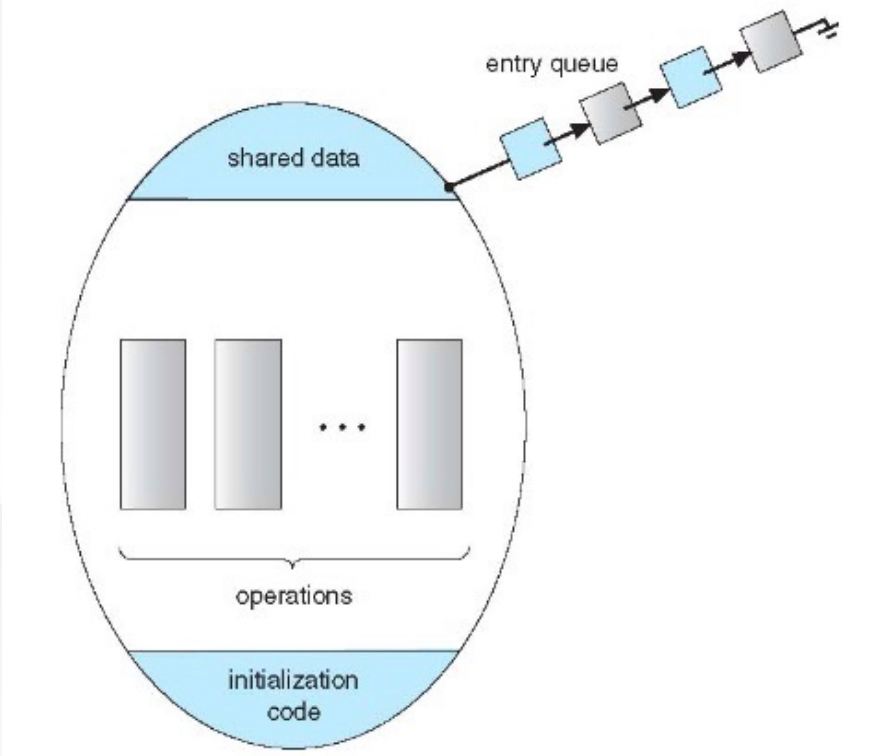
管程

- 高级语言工具，封装了数据及对其操作的一组函数
  - 管程提供了一组由程序员定义的、在管程内互斥的操作
  - 管程类型也包含一组变量，用于定义这一类型的实例状态，也包括操作这些变量的函数实现
- 只有管程内定义的函数才能访问管程内局部声明的变量和形式参数

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (...) { .... }

   procedure Pn (...) {......}

   Initialization code (...) { ... }
}
```

管程

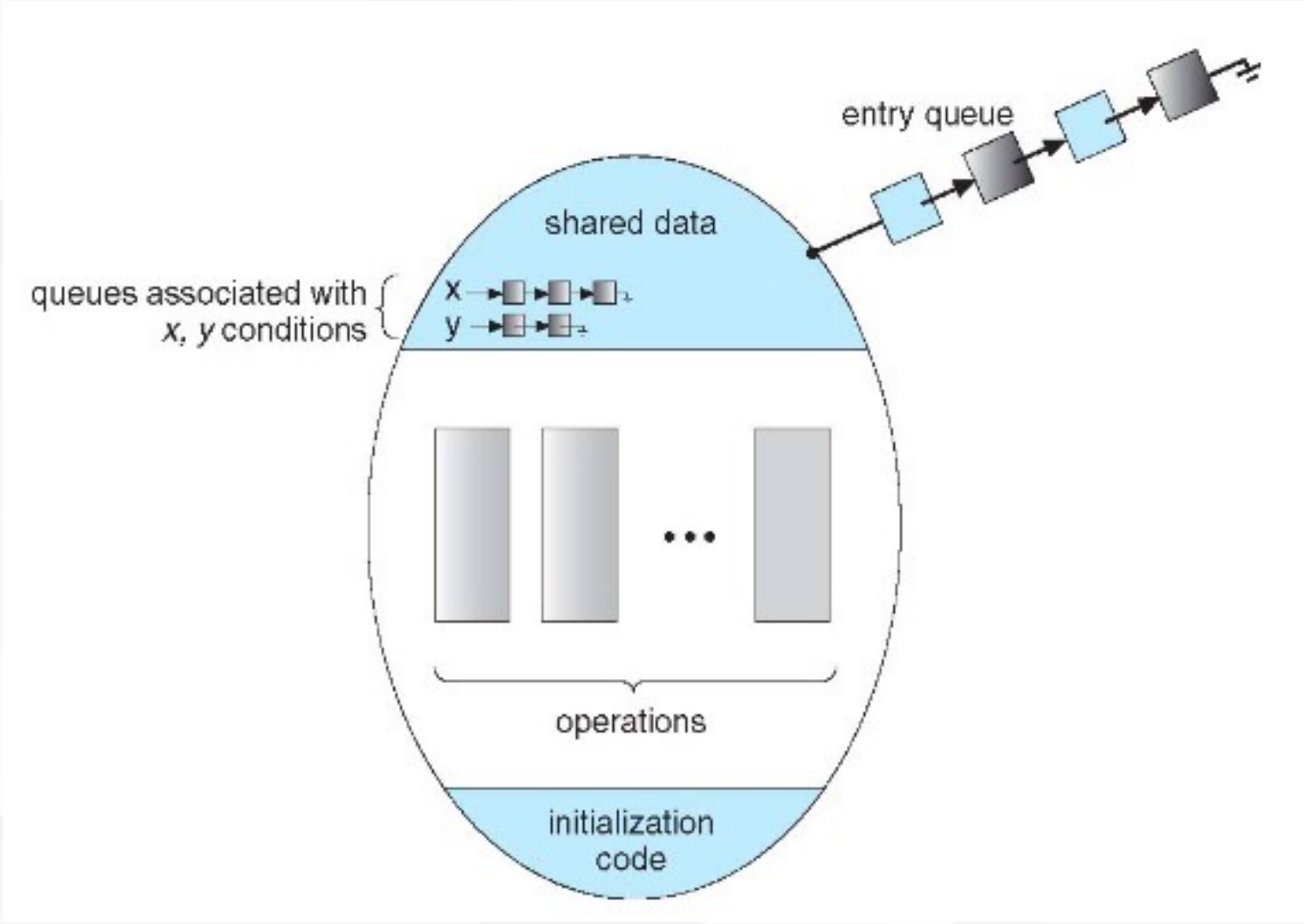
- 管程结构确保每次只有一个进程在管程内处于活动状态，程序员不需要明确编写同步约束
- 需要用shared data的进程都要排队（entry queue）



管程

- 定义附加的同步机制
  - 通过条件（condition）结构来提供：condition x;
- 针对条件，可以进行两种操作
  - x.wait () 调用这一操作的进程会被挂起，直到另一进程调用x.signal ()
  - x.signal () 重新恢复一个挂起的进程。如果没有挂起进程，那么x.signal () 就没有作用，即x的状态如同没有执行任何操作，这与信号量的操作signal()不同，后者始终影响信号量的状态

具有同步机制的管程


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

具有同步机制的管程

- 假设当操作x.signal () 被进程P调用时，在变量x上有一个挂起进程Q，会发生什么？
- 两种可能性
  - 唤醒并等待：进程P等待直到Q离开管程，或者等待另一个条件
  - 唤醒并继续：进程Q等待直到P离开管程或者等待另一个条件
- 两种各有优劣，由编程语言决定

哲学家就餐问题的管程解决方案

- 强加以下限制：只有一个哲学家两边的筷子均可用时，他才拿起他们
- 为区分哲学家的三种状态，引入数据结构
  - enum {THINKING, HUNGRY, EATING} state[5];
- 哲学家i只有在其两个邻居不在就餐时（(state[(i+4) % 5] != EATING)和(state[(i+1) % 5] != EATING) ），才能设置变量state[i] = EATING
- 另外，还需要声明：condition self[5];
  - 让哲学家i在饥饿且不能拿到筷子时，延迟自己

哲学家就餐问题的管程解决方案

```
monitor DiningPhilosophers
  {
    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);
           test((i + 1) % 5);
     }
```

```
          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;
          }
    }
```

哲学家就餐问题的管程解决方案

- 哲学家i在用餐前，应调用操作pickup()，这可能挂起该哲学家进程。在pickup()成功之后，他就可以进餐了。然后他调用putdown()。

```
DiningPhilosophers.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

课堂习题

假设两个进程P0和P1，共享变量：boolean flag[2]; int turn;

进程结构如图所示，请证明这个算法满足临界区问题的三个要求

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```