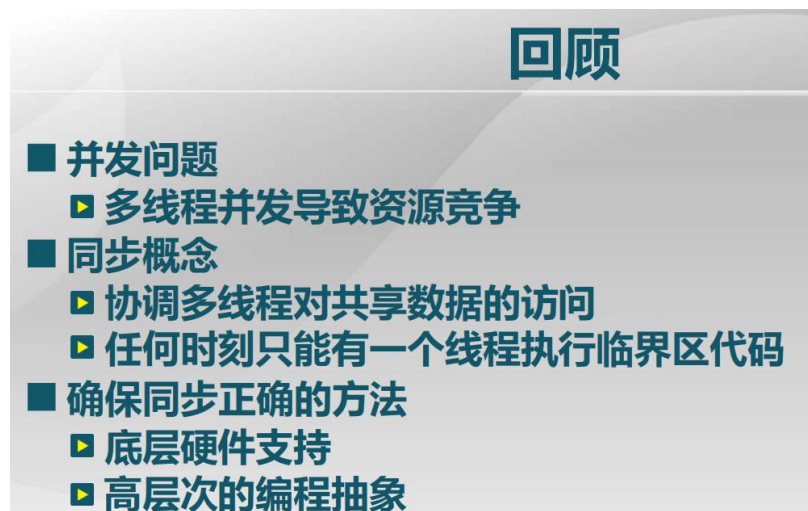


第一节 信号量

信号量和管程是操作系统提供的两种同步方法。



信号量和管程也属于高层次的编程抽象。



协调各个线程对临界资源的访问是最终要解决的问题。17 讲介绍的方法只有锁，如上图所示，本讲引入信号量，如上图所示。

信号量(semaphore)

- 信号量是操作系统提供的一种协调共享资源访问的方法
 - ▣ 软件同步是平等线程间的一种同步协商机制
 - ▣ OS是管理者，地位高于进程
 - ▣ 用信号量表示系统资源的数量
- 由Dijkstra在20世纪60年代提出
- 早期的操作系统的主要同步机制
 - ▣ 现在很少用（但还是非常重要在计算机科学研究）

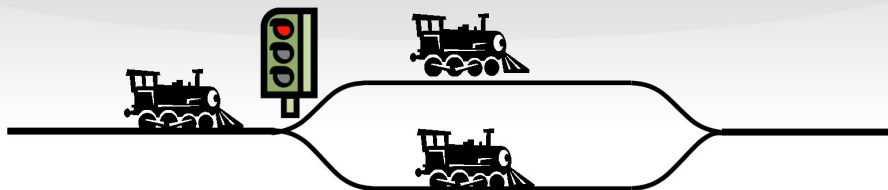
软件同步是平等线程间的一种同步协商机制(没有裁判); 而信号量机制中, OS 是管理者, 地位高于进程(有裁判)。

信号量(semaphore)

- 信号是一种抽象数据类型
 - ▣ 由一个整数 (**sem**) 变量和两个原子操作组成
 - ▣ **P()** (Prolaag (荷兰语尝试减少))
 - ▣ sem减1
 - ▣ 如sem<0, 进入等待, 否则继续
 - ▣ **V()** (Verhoog (荷兰语增加))
 - ▣ sem加1
 - ▣ 如sem≤0, 唤醒一个等待进程

■ 信号量与铁路的类比

- ▣ 2个站台的车站
- ▣ 2个资源的信号量



整型变量 **sem** 的取值可理解为要共享的资源的数目。**P()**, 申请资源时用。**V()**, 释放资源时用。**P()**操作的 $\text{sem} < 0$ 注意理解: 考虑边界条件, 上一个可占用资源的进程将 **sem** 从 1 减为 0, 而这个等待的进程先将 **sem** 减一, 因此发现 $\text{sem} < 0$, 进入等待。**V()**同理, 因为是 **sem** 先加一, 因此 $\text{sem} == 0$ 时, 说明 **P()**中 $\text{sem} == -1$, 有等待进程。

信号量的特性

- 信号量是**被保护的整数变量**
 - ▣ 初始化完成后，只能通过P()和V()操作修改
 - ▣ 由操作系统保证，PV操作是原子操作
- **P()可能阻塞，V()不会阻塞**
- 通常假定信号量是“公平的”
 - ▣ 线程不会被无限期阻塞在P()操作
 - ▣ 假定信号量等待按先进先出排队

自旋锁能否实现先进先出？

由于操作系统执行它的代码的时候，优先级高于进程的用户代码，所以能保证操作系统P,V 代码在执行过程中，不受应用进程代码执行的干扰，这样就能保证其原子性。P 操作可能由于没有资源而进入等待(阻塞)状态，V 操作只会释放资源，可以使处于等待状态的资源变为就绪状态。实际上，在实际系统中，信号量的公平有些偏差。P()操作实际上有一个等待最长时限的参数，超时后直接错误返回。

自旋锁做不到先进先出，因为自旋锁要占用 CPU，随时查询锁的状态，当临界区的使用者退出时，查询的进程不一定是哪一个。

下面看信号量的实现：

信号量的实现

```
class Semaphore {  
    int sem;  
    WaitQueue q;  
}
```

```
Semaphore::P() {  
    sem--;  
    if (sem < 0) {  
        Add this thread t to q;  
        block(p);  
    }  
}
```

```
Semaphore::V() {  
    sem++;  
    if (sem <= 0) {  
        Remove a thread t from q;  
        wakeup(t);  
    }  
}
```

sem 和 q 都是在操作系统内核中的。进入时 P()操作：若无资源，将当前进程 t 放入等待队列 q 中，并且阻塞 t(ppt 里我想打错了吧，应该是 block(t))。

退出时 V()操作：若有线程在等待，从对应的等待队列 q 中把相应的线程放到就绪队列中。

与前面讲的方法的区别：由于操作系统的保护，P()操作和 V()操作不会被打断。

第二节 信号量的使用

本节介绍如何利用信号量解决同步互斥问题。

信号量分类

- 可分为两种信号量
 - ▣ **二进制信号量**: 资源数目为0或1
 - ▣ **资源信号量**: 资源数目为任何非负值
 - ▣ 两者等价
 - ▣ 基于一个可以实现另一个
- 信号量的使用
 - ▣ 互斥访问
 - ▣ 临界区的互斥访问控制
 - ▣ 条件同步
 - ▣ 线程间的事件等待

首先看如何用信号量实现临界区的互斥访问:

用信号量实现临界区的互斥访问

每个临界区设置一个信号量, 其初值为1

```
mutex = new Semaphore(1);
```

```
mutex->P();  
Critical Section;  
mutex->V();
```

- 必须成对使用P()操作和V()操作
 - ▣ P()操作保证互斥访问临界资源
 - ▣ V()操作在使用后释放临界资源
 - ▣ PV操作**不能次序错误、重复或遗漏**

因为临界区同时只能有一个线程访问, 因此设置信号量初值为 1.

再看如何用信号量实现条件同步:

用信号量实现条件同步

每个条件同步设置一个信号量, 其初值为0

```
condition = new Semaphore(0);
```

线程A

```
... M ...  
condition->P();  
... N ...
```

线程B

```
... X ...  
condition->V();  
... Y ...
```

我们希望: 在 B 执行到 X 之后, A 才执行 N 模块, 例如 B 的 X 是准备数据, A 的 N 是使用数据。①若 B 先执行 V(), sem 由 0 变 1, 说明 X 已执行完, 这时 A 的 P 操作可以通过, 可以执行 N ②若 A 先执行 P 操作, 则 A 被阻塞, 进入等待状态, 只有等待 B 执行完 X, 进

行 V 操作后, A 才可以继续执行。上面就是条件同步的具体含义, 及如何用信号量实现条件同步。

下面看一个更实际的生产者消费者的例子:



举例: 消费者是打印进程, 生产者是应用程序, 若干个应用程序会产生打印作业放在缓冲区里, 打印服务器负责从缓冲区里读出打印数据然后打印。

缓冲区未满, 生产者才能往里写数据。缓冲区不空, 消费者才能从中读数据。

注意: 任何时刻只能有一个生产者或消费者可访问缓冲区, 指的是缓冲区是一个临界区, 同时只能有一个线程访问。



mutex 用于互斥访问, fullBuffers 对应有数据消费者才能往外读(full 对应缓冲区中有数据填充的含义), emptyBuffers(empty 对应缓冲区中无数据填充的含义)对应必须缓冲区有空地生产者才能写。FullBuffers+emptyBuffers 是缓冲区的总大小。

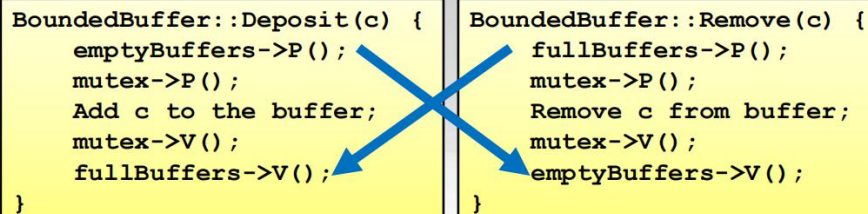
具体实现如下:

用信号量解决生产者-消费者问题

```
Class BoundedBuffer {  
    mutex = new Semaphore(1);  
    fullBuffers = new Semaphore(0);  
    emptyBuffers = new Semaphore(n);  
}
```

```
BoundedBuffer::Deposit(c) {  
    emptyBuffers->P();  
    mutex->P();  
    Add c to the buffer;  
    mutex->V();  
    fullBuffers->V();  
}
```

```
BoundedBuffer::Remove(c) {  
    fullBuffers->P();  
    mutex->P();  
    Remove c from buffer;  
    mutex->V();  
    emptyBuffers->V();  
}
```



■ P、V操作的顺序有影响吗？

首先初始化：二进制临界区信号量初值为 1，刚开始缓冲区无任何数据，因此满缓冲区 fullBuffers 初值为 0，空缓冲区 emptyBuffers 初值为 n

Deposit 是生产者，Remove 是消费者。

P、V 的操作顺序是有影响的，独立分析即可，略。

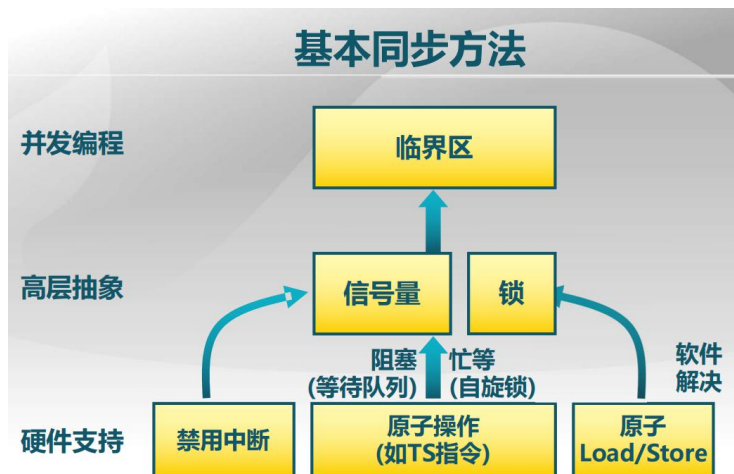
使用信号量的困难

- 读/开发代码比较困难
 - ▣ 程序员需要能运用信号量机制
- 容易出错
 - ▣ 使用的信号量已经被另一个线程占用
 - ▣ 忘记释放信号量
- 不能够处理死锁问题

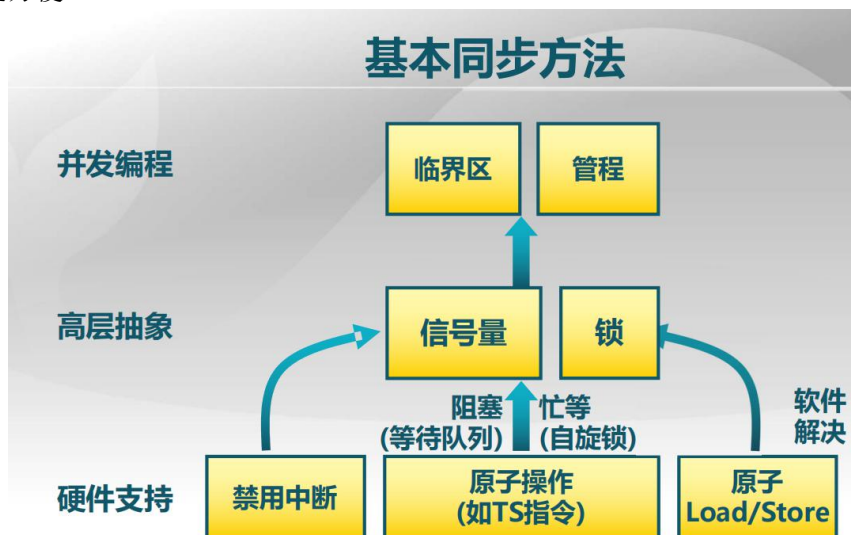
P、V 操作极易遗漏以及搞错顺序。

第三节 管程

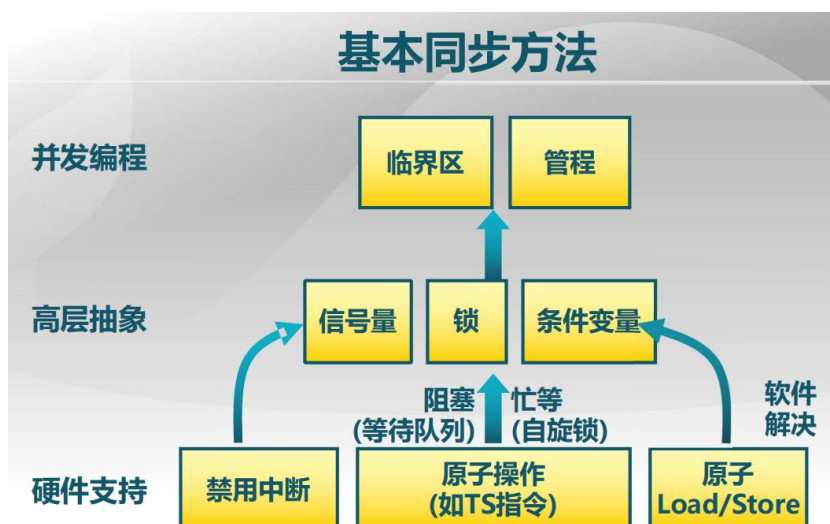
管程也是一种同步方法。



在刚才的同步方法介绍中，有了信号量，使得我们在操作系统的参与下处理临界区比原来更方便。



而管程是想改进信号量在处理临界区时的一些麻烦，比如在刚才的生产者消费者问题中，信号量的PV操作是分散在生产者和消费者两个不同的进程当中，这种情况下，PV操作的配对比较困难，我们试图把这些配对的PV操作集中到一起，这就是管程，它也是一种并发程序的编程方法，为了支持管程，还需要添加条件变量，这是在管程内部使用的一种同步机制：



下面看管程的具体做法。

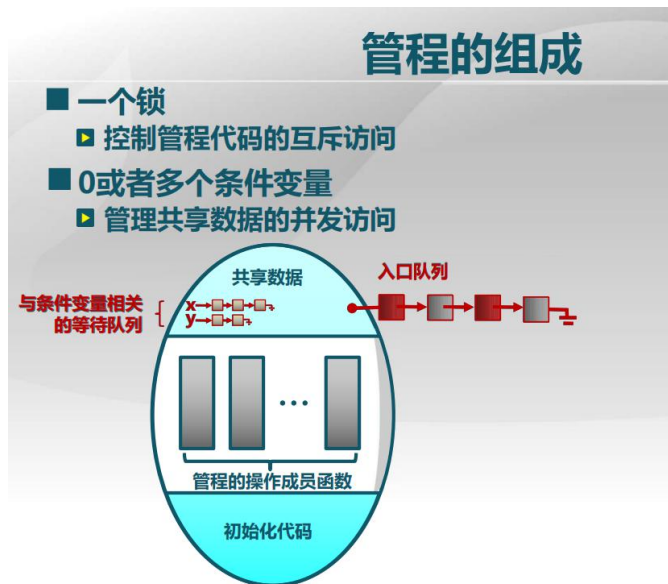
管程 (Monitor)

- **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，简化了线程间的同步控制
 - ▣ 任一时刻最多只有一个线程执行管程代码
 - ▣ 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
- **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

管程采用面向对象方法，即把共享资源相关的 PV 操作集中到一起，从而简化了线程间的同步控制。任一时刻最多只有一个线程执行管程代码，听起来和临界区一样，但区别体现在：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。即一个线程在临界区中执行，它必须执行到它退出临界区，它才可能放弃临界区的互斥访问；而管程允许在执行过程中临时放弃，临时放弃后，其它线程就可以进到管程的区域里。

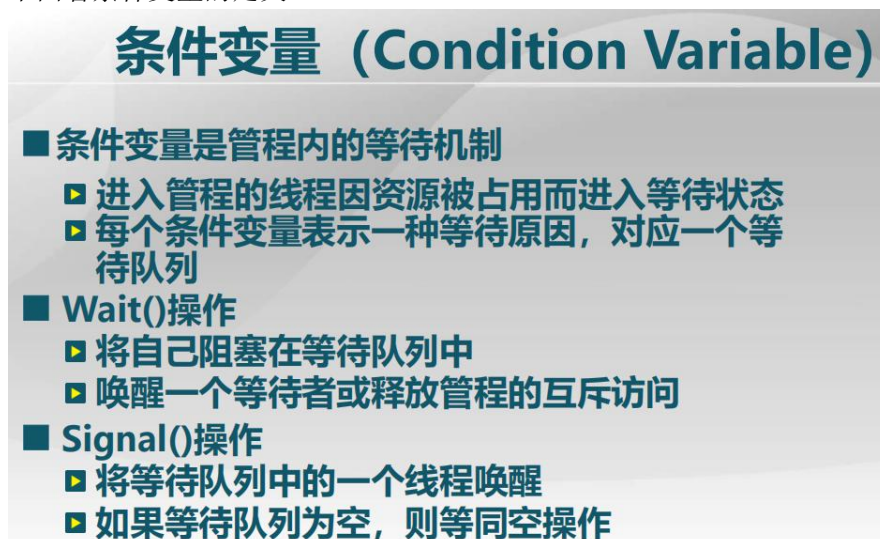
管程可在要同步的模块中，收集共享的数据，然后编写对这些共享数据的访问方法，有了这个，在各处用时，它就不用和其它地方再做同步互斥这些操作。

管程是如何做到这一点的？



首先，我们把这个区域里的代码理解为管程，如果将它视作一个临界区，只需在入口处加一个互斥访问的锁，管程也一样：在入口队列中，只允许一个线程在管程内部执行，如果在代码内部没有其它共享数据，这时就和临界区完全一样，即任何一个线程在进入时都在入口排队，允许只有一个线程进来，在进入线程退出时，其它线程可以进入。而管程在临界区基础上又加了一条，即零个或多个条件变量，如果是 0 个，就等同于一个临界区；如果是有一个以上的条件变量，这时就是管程所特有的了，它用来管理共享数据的并发访问，需要共享资源时，有相应的条件变量允许访问，才可以用上图中中间管程的(互斥的)操作成员函数。以上便是管程的基本思路。

下面看条件变量的定义：



Wait()即等待操作中，唤醒一个等待者或释放管程的互斥访问就是允许另外一个线程进入管程。

下面看条件变量的实现：

条件变量实现

```
Class Condition {  
    int numWaiting = 0;  
    WaitQueue q;  
}
```

```
Condition::Wait(lock) {  
    numWaiting++;  
    Add this thread t to q;  
    release(lock);  
    schedule(); //need mutex  
    require(lock);  
}
```

```
Condition::Signal() {  
    if (numWaiting > 0) {  
        Remove a thread t from q;  
        wakeup(t); //need mutex  
        numWaiting--;  
    }  
}
```

条件变量 Condition 中有一个整型变量和一个等待队列，与信号量不同的是，条件变量的初值为 0，而在信号量中，初值是和资源数相同的。

对于 Wait 等待操作，numWaiting 加一，表示等待的线程数加 1，然后将该线程 t 放到等待队列 q 中，然后释放管程的互斥访问权，执行调度，即调度后可以切换到另外的进程或者线程来执行，待调度完成，回来继续请求管程的访问权限。

对于释放 Signal 操作，如果 numWaiting 小于等于 0，就相当于空操作，否则，有另外的线程等在这个条件变量上，则这时将某线程 t 从等待队列中移出，放到就绪队列中让其执行，然后计数减一，也就是等的线程的数目少了一个。

现在看如何利用管程解决生产者消费者问题：

用管程解决生产者-消费者问题

```
class BoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition notFull, notEmpty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
    while (count == n)  
        notFull.Wait(&lock);  
    Add c to the buffer;  
    count++;  
    notEmpty.Signal();  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
    while (count == 0)  
        notEmpty.Wait(&lock);  
    Remove c from buffer;  
    count--;  
    notFull.Signal();  
    lock->Release();  
}
```

首先初始化定义两个条件变量：notFull 和 notEmpty，一个入口等待队列 lock，即入口的锁，这和我们前面用信号量解决生产者、消费者问题的设置很类似，count 用来表示写到缓冲区里数据的数目。

生产者和消费者分别对应着一个函数，注意，管程中利用了锁的概念，具体见 lec17 高级抽象方法部分：

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire(); //管程进入申请(申请锁)  
    while (count == n) //查看是否有缓冲区可以写数据，若没有则  
        notFull.Wait(&lock); //等待
```

```

Add c to the buffer; //向缓冲区写一个数据
count++; //计数加一
notEmpty.Signal();
lock->Release();
}
BoundedBuffer::Remove(c) {
lock->Acquire(); //管程进入申请(申请锁)
while (count == 0)
notEmpty.Wait(&lock);
Remove c from buffer; //从缓冲区里读一个数据
count--; //计数减一
notEmpty.Signal();
lock->Release();
}

```

比较难理解的是：

```

while (count == n)
notEmpty.Wait(&lock);

```

中为什么用 **while**，以及如何理解 **Wait()** 函数，现做一说明。

首先，当生产者发现缓冲区满了时，也就是 **count==n** 时，会进入 **wait** 函数，**wait** 函数如下：

```

Condition::Wait(lock){
numWaiting++; //首先将等待队列计数器加 1
Add this thread t to q; //将当前生产者(进程)放入等待队列 q
release(lock); //释放锁，以让其他进程可以进入管程
schedule(); //执行调度(因为锁已经释放，所以别的进程进入管程会成功)
require(lock); //继续请求锁
}

```

实际上，执行调度 **schedule** 后，会发生两种情况，第一种，调度的仍是一个生产者(记为生产者 2)，这时比较麻烦，这个生产者在 **BoundedBuffer::Deposit** 中成功获取锁之后，仍会因为缓冲区不够而进入 **wait()**，试想这个生产者 2 执行 **release(lock)** 后，生产者 1 立刻 **require(lock)** 成功，返回 **Deposit**。则如果是 **if**，不会重新判断缓冲区是否合规，这样就会出现错误。但如果是 **while** 情况就不同了，这时会继续判断一次缓冲区够不够，判断结果是依然不够，于是就继续等待。

此外注意，**require(lock)** 若获得锁，会将在等待队列中等待锁的进程移出，则 **numWaiting--**。

notEmpty 中存有生产者进程，**notEmpty** 中存有消费者进程。

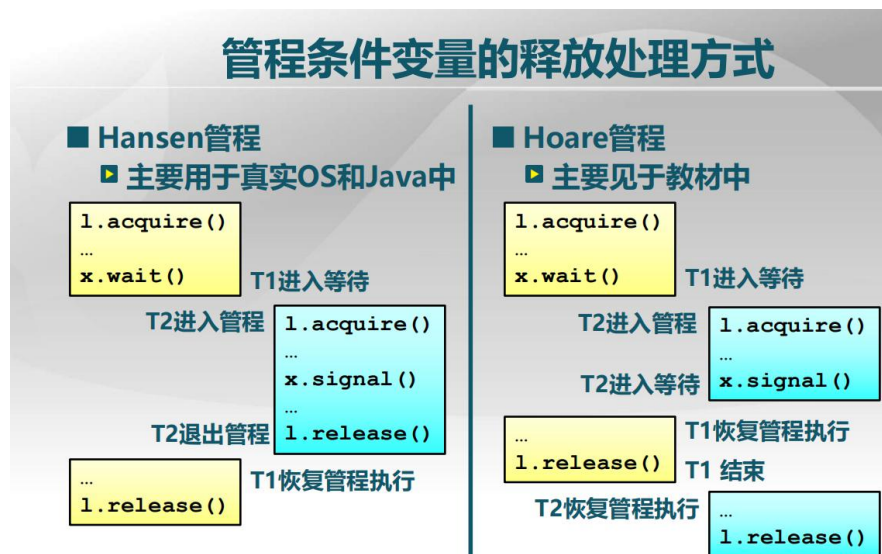
注意 **wait()** 中 **release(lock)** 后，该等待进程会自动被放入等待队列，因为此时属于该进程等待资源(锁)的情况，是应该被放入等待队列的。

Signal 起到将线程从等待队列中唤醒(被唤醒后，此时获得资源即锁的线程可以运行)的作用。

管程可以将 **PV** 操作都集中到一个模块中，从而简化和降低同步机制的实现难度。

下面看管程中条件变量到底是内部的线程优先执行还是正占用管程处于执行状态的线

程有更高权限来执行。两种不同的处理办法对应不同的管程：



注意本质：

上面两图，有一点是共同的，假设前提是，在 `x.wait()` 中释放锁之后，是一个消费者 T2 得到了调度，这个消费者来释放已满的缓冲区，就有了图中接下来的讨论。

关键是对蓝色框的消费者 `l.release()` 的考量：对于 Hansen 管程，在消费者释放锁之后，可能有另一个生产者在一直请求锁(有两个生产者了，假设图中生产者 A，可能的生产者 B)，①生产者 A 先获得锁，则生产者 A 此时回到 `while` 判断可以生产，然后 B 获得锁，但却发现不可生产 ②生产者 B 先获得锁，发现可以生产，然后生产者 A 从 `wait()` 回到 `while` 处发现不可生产(这也就是为什么一定要用 `while` 的原因)

而对于 Hoare 线程：由于 T2 一直未释放锁，T2 是直接将控制权转交给 T1(不需要释放锁)，所以不会有另一个生产者和 T1 抢占的问题，T1 直接从 `wait` 返回，得到未满的缓冲区进行生产即可，不需要再在 `while` 中进行一次条件判断，因此这里用 `if` 即可。

在 Hansen 中，一个线程 T1 等待条件变量，它进入等待状态，这时允许进程 T2 开始执行，T2 执行的过程当中给了 `signal` 释放信号，允许 x 条件变量对应的线程开始运行，在 Hansen 中，释放完后当前 T2 会继续执行，一直到它放弃管程的互斥访问权限，然后 T1 才恢复执行。这种做法是当前正在执行的线程更优先。

而 Hoare 的做法是第一个线程执行到等待条件变量的时候进入等待状态，然后 T2 开始执行，这与 Hansen 一样，区别在于 T2 认为 T1 等待的事件已经出现了，这时它唤醒 T1，唤醒完之后 T2 立即放弃管程的互斥访问权限，T1 马上开始执行，等 T1 执行结束，T2 才继续执行。从通常的优先角度讲，T1 更优先是合理的，这就是 Hoare 的做法，但真实系统中，用 Hansen 的做法：因为连续执行，效率更高，少了一次切换。

基于这两种不同做法，我们再看其在生产者消费者问题中的体现：

Hansen 管程与 Hoare 管程

```
Hansen-style :Deposit() {  
    lock->acquire();  
    while (count == n) {  
        notFull.wait(&lock);  
    }  
    Add thing;  
    count++;  
    notEmpty.signal();  
    lock->release();  
}
```

```
Hoare-style: Deposit() {  
    lock->acquire();  
    if (count == n) {  
        notFull.wait(&lock);  
    }  
    Add thing;  
    count++;  
    notEmpty.signal();  
    lock->release();  
}
```

■ Hansen管程

- ▶ 条件变量释放仅是一个提示
- ▶ 需要重新检查条件

■ 特点

- ▶ 高效

■ Hoare管程

- ▶ 条件变量释放同时表示放弃管程访问
- ▶ 释放后条件变量的状态可用

■ 特点

- ▶ 低效

signal 即表示管程的释放。

课后题：管程的主要特点有 ABD

- A.局部数据变量只能被管程的过程访问
- B.一个进程通过调用管程的一个过程进入管程
- C.不会出现死锁
- D.在任何时候，只能有一个进程在管程中执行

注意：管程可能出现死锁！

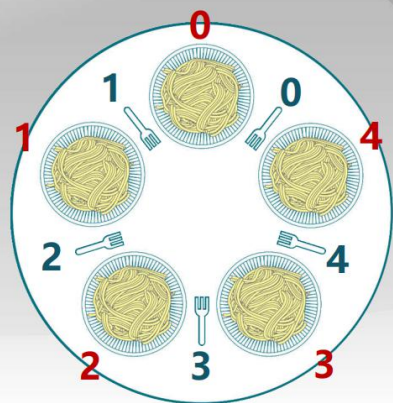
第四节 经典同步问题：哲学家就餐问题

下面将用信号量和管程的方法解决几个经典的同步问题：哲学家就餐问题和读者-写者问题。首先看哲学家就餐问题：

哲学家就餐问题

问题描述：

- 5个哲学家围绕一张圆桌而坐
 - ▶ 桌子上放着5支叉子
 - ▶ 每两个哲学家之间放一支
- 哲学家的动作包括思考和进餐
 - ▶ 进餐时需同时拿到左右两边的叉子
 - ▶ 思考时将两支叉子放回原处
- 如何保证哲学家们的动作有序进行？
如：不出现有人永远拿不到叉子



首先尝试信号量的办法：

方案1

```
#define N 5 // 哲学家个数
semaphore fork[5]; // 信号量初值为1
void philosopher(int i) // 哲学家编号: 0 - 4
{
    while(TRUE)
    {
        think( ); // 哲学家在思考
        P(fork[i]); // 去拿左边的叉子
        P(fork[(i + 1) % N]); // 去拿右边的叉子

        eat( ); // 吃面条中....
        V(fork[i]); // 放下左边的叉子
        V(fork[(i + 1) % N]); // 放下右边的叉子
    }
}
```

不正确, 可能导致死锁

假设每把叉子是一个资源, 对应一个信号量, 5 个信号量, 每个初值均为 1, 哲学家的编号是 i , i 的编号为 0, 1, 2, 3, 4, 利用 P 操作去表示拿叉子, 吃完后, 用 V 操作表示释放叉子。多数情况下该算法没问题, 但是考虑一种极端情况: 假定刚开始时所有叉子都放在那里, 如果 5 个哲学家进行连续的 5 个 P(fork[i]) 操作, 那么大家一起拿左边的叉子。这时就会发生每人都拿到左边的刀叉, 但是拿右边时, 每个哲学家都不成功, 则所有人都开始等待, 则没有任何一个人可以开始就餐, 所有人都无限期等待下去, 也就可能进入死锁。

方案二是一种极端的做法:

方案2

```
#define N 5 // 哲学家个数
semaphore fork[5]; // 信号量初值为1
semaphore mutex; // 互斥信号量, 初值1
void philosopher(int i) // 哲学家编号: 0 - 4
{
    while(TRUE) {
        think( ); // 哲学家在思考
        P(mutex); // 进入临界区
        P(fork[i]); // 去拿左边的叉子
        P(fork[(i + 1) % N]); // 去拿右边的叉子
        eat( ); // 吃面条中....
        V(fork[i]); // 放下左边的叉子
        V(fork[(i + 1) % N]); // 放下右边的叉子
        V(mutex); // 退出临界区
    }
}
```

互斥访问正确, 但每次只允许一人进餐

在系统中, 我们希望将资源得到充分的利用, 然而一旦没有办法充分利用, 可以把所有资源打成一个整包, 只有一个进程可以占用这一整包的资源, 不管它是不是都需要, 这种情况下, 系统也能正常运行下去, 只是效率低。这就是方案二的做法: 设置了一个互斥变量 `mutex`, 保证任何时刻只有一个哲学家可以就餐, 这时 `fork` 的申请一定不会遇到问题。这种方案虽然正确, 但效率低。这种做法我们认为可以接受, 但是性能不好。

于是我们给出解决方案三:

方案3

```
#define N 5 // 哲学家个数
semaphore fork[5]; // 信号量初值为1
void philosopher(int i) // 哲学家编号: 0 - 4
{
    while(TRUE)
    {
        think( ); // 哲学家在思考
        if (i%2 == 0) {
            P(fork[i]); // 去拿左边的叉子
            P(fork[(i + 1) % N]); // 去拿右边的叉子
        } else {
            P(fork[(i + 1) % N]); // 去拿右边的叉子
            P(fork[i]); // 去拿左边的叉子
        }
        eat( ); // 吃面条中....
        V(fork[i]); // 放下左边的叉子
        V(fork[(i + 1) % N]); // 放下右边的叉子
    }
}
```

没有死锁，可有多人同时就餐

方案三仍然用 5 把刀叉，但是不会设置一个全局的锁把就餐的事变成一个临界区的操作。针对刚才说到的 5 个哲学家一起拿起左边的刀叉之后所面临的死锁的情况，令他们每个人拿刀叉的时候有差异，不至于是所有的人全部都一起拿起左边的，这样就不会出现构成环路的情况，总会有先有后，最后保证大家都能拿到。通常情况下，可以两个人一起就餐。

偶数编号的先拿左边后拿右边，奇数的先拿右边后拿左边，这样就不会出现 5 个人都拿到一部分资源构成环路的情况。

是否有更好办法？留给自己思考。

第五节 读者-写者问题

下面我们讨论另一个经典的同步问题：读者-写者问题。读者-写者问题是我们在计算机系统中经常碰到的一个经典问题，它存在于数据库等很多共享资源的访问当中。

读者-写者问题描述

- 共享数据的两类使用者
 - ▣ 读者：只读取数据，不修改
 - ▣ 写者：读取和修改数据
- 读者-写者问题描述：对共享数据的读写
 - ▣ “读 - 读” 允许
 - ▣ 同一时刻，允许有多个读者同时读
 - ▣ “读 - 写” 互斥
 - ▣ 没有写者时读者才能读
 - ▣ 没有读者时写者才能写
 - ▣ “写 - 写” 互斥
 - ▣ 没有其他写者时写者才能写

首先尝试用信号量解决读者-写者问题：

用信号量解决读者-写者问题

■ 用信号量描述每个约束

- ▣ 信号量WriteMutex
 - ▣ 控制读写操作的互斥
 - ▣ 初始化为1
- ▣ 读者计数Rcount
 - ▣ 正在进行读操作的读者数目
 - ▣ 初始化为0
- ▣ 信号量CountMutex
 - ▣ 控制对读者计数的互斥修改
 - ▣ 初始化为1

CountMutex 用来保护读者计数，如果有多个读者进行读，它们都要对 Rcount 进行修改(即加 1)，那么 Rcount 可能出现同步问题，因此需要对各读者进行协调，这时对 Rcount 的访问应该是互斥的(可理解为写写互斥)，于是就有了 CountMutex 这个信号量。CountMutex 保证任何时刻只允许一个线程来修改计数 Rcount。

用信号量解决读者-写者问题

Writer

```
P (WriteMutex) ;  
  
write ;  
  
V (WriteMutex) ;
```

此实现中，读者优先

Reader

```
P (CountMutex) ;  
if (Rcount == 0)  
    P (WriteMutex) ;  
++Rcount ;  
V (CountMutex) ;  
  
read ;  
  
P (CountMutex) ;  
--Rcount ;  
if (Rcount == 0)  
    V (WriteMutex) ;  
V (CountMutex)
```

首先读和写的核心分别为 write 和 read, 然后进行互斥保护，即 P(WriteMutex),V(WriteMutex),写者完全理解为临界区问题；读者这头，如果与写者相对应，那它也是一个临界区，前面是 P 操作，后面是 V 操作，但它有些不同：如果第一个读者进来时申请 P(WriteMutex)操作，与写者进行互斥,第二个读者来时就不能这么申请了(为保证读允许)，所以 P(WriteMutex)只是对于第一个读者，因此加一个判断 if(Rcount==0),即如果当前读者计数为 0，即第一个进来时，需要申请 P(WriteMutex)，并计数加一，第二个读者进来时就不用了，第二个只需要计数加一。释放时也是一样，计数减一是每个都要减的，但只是对最后一个离开的读者，才需要释放读写互斥的信号量，这样才便于写者进行写操作。读者计数也要保护，这时在前面加上计数的信号量 CountMutex。

该方法特征：读者优先，但对于数据更新讲，我们希望写者优先。这就涉及读者写者问题的两种优先策略：

读者/写者问题：优先策略

■ 读者优先策略

- ▣ 只要有读者正在读状态，后来的读者都能直接进入
- ▣ 如读者持续不断进入，则写者就处于饥饿

■ 写者优先策略

- ▣ 只要有写者就绪，写者应尽快执行写操作
- ▣ 如写者持续不断就绪，则读者就处于饥饿

如何实现？

也有其它优先策略，如两边公平的方法，即有读者就绪了，那么写者就不能往里面写，有写者就绪了，读者不能往里写。这些做法如何实现？自己思考。

下面看用管程解决读者写者问题：

用管程解决读者-写者问题

■ 两个基本方法

```
Database::Read() {  
    Wait until no writers;  
    read database;  
    check out - wake up waiting writers;  
}
```

```
Database::Write() {  
    Wait until no readers/writers;  
    write database;  
    check out - wake up waiting readers/writers;  
}
```

■ 管程的状态变量

```
AR = 0;           // # of active readers  
AW = 0;           // # of active writers  
WR = 0;           // # of waiting readers  
WW = 0;           // # of waiting writers  
Lock lock;  
Condition okToRead;  
Condition okToWrite;
```

管程是把同步操作封装到一个类中，对读者写者问题，我们对外提供的基本方法有两个：

一是读即 `Read()`：如果在前面已经有写者，则一直等到写者结束才开始读，读完出来时看是不是有其他的写者，有则将它们唤醒。

另一个是写操作即 `Write()`：看前面是否有读者或写者，若没有，则开始写，写完后，看有没有另外的读/写者，有则将它们唤醒，让他们继续操作。

管程中维护了 4 个变量：`AR`:当前正在读的读者数目 `AW`:当前正在写的写者数目 `WR`:等待读的读者数目 `WW`:等待写的写者数目。 有一些关系：正在读和正在写只会有一个大于 0 的(读写互斥)。

管程中设置一个锁 `lock`,然后设置两个条件变量：一个是可以去读 `okToRead`，一个是可以去写 `okToWrite`,详细的代码如下：

解决方案详情：读者

```
AR = 0; // # of active readers
AW = 0; // # of active writers
WR = 0; // # of waiting readers
WW = 0; // # of waiting writers
Lock lock;
Condition okToRead;
Condition okToWrite;
```

```
Public Database::Read() {
    //Wait until no writers;
    StartRead();
    read database;
    //check out - wake up waiting writers;
    DoneRead();
}
```

```
Private Database::StartRead() {
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}
```

```
Private Database::DoneRead() {
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0) {
        okToWrite.signal();
    }
    lock.Release();
}
```

读者：

对外提供开始读:StartRead()和完成读: DoneRead()

StartRead 的内部实现：对于管程，我们都会申请管程的互斥访问，在内部开始读时，一定是当前正在读的计数要加 1，即 AR++，而何时会进行等待？如果有写者正在写，或者有写者在申请写，这时：等的计数加 1(WR++)，并且将其排到条件变量上(okToRead.wait(&lock))，等到返回时，等待读的计数减一；上面方法可见：写者是优先的，只要有写者，它都优先于这个读者。

DoneRead():首先申请互斥访问管程：lock.Acquire()，然后正在读的计数减一(AR--),如果没有读者且有写者等着，那就释放写者。AR==0 这一条件是为保证读写互斥，试想当前若有一进程正在读，这是因为之前判断时没有进程在写，若判断(AR==0&&WW>0)时恰好有一进程要写，那么 AR==0 可以排除正在有线程在读的情况，从而避免一边读一边写。

解决方案详情：写者

```
AR = 0; // # of active readers
AW = 0; // # of active writers
WR = 0; // # of waiting readers
WW = 0; // # of waiting writers
Lock lock;
Condition okToRead;
Condition okToWrite;
```

```
Public Database::Write() {
    //Wait until no readers/writers;
    StartWrite();
    write database;
    //check out-wake up waiting readers/writers;
    DoneWrite();
}
```

```
Private Database::StartWrite() {
    lock.Acquire();
    while ((AW+AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.Release();
}
```

```
Private Database::DoneWrite() {
    lock.Acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    }
    else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

写者：开始写 StartWrite()和完成写 DoneWrite()

StartWrite():有正在写的写者或正在读的读者时，便等待，那么如果有等待申请读的读

者，是不等待的。则这种方法是写者优先的。

DoneWrite():优先唤醒等待写的，如果没有等待写的，才去唤醒读者。

由以上实现可知，管程实现是写者优先的。

利用管程，可以更方便的把优先策略体现出来。从这一角度说，管程是简化了处理同步问题的实现方法。