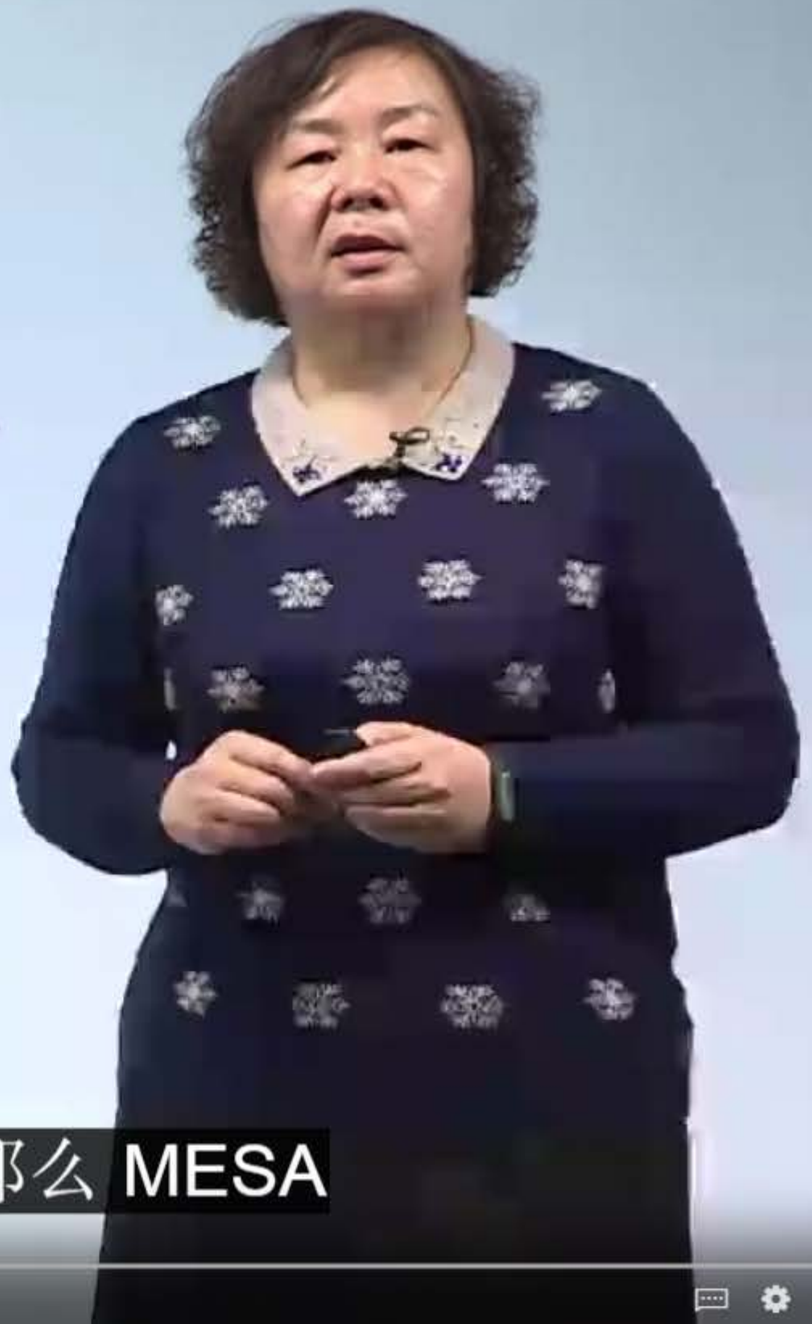
MESA管程与Hoare管程比较

MESA管程

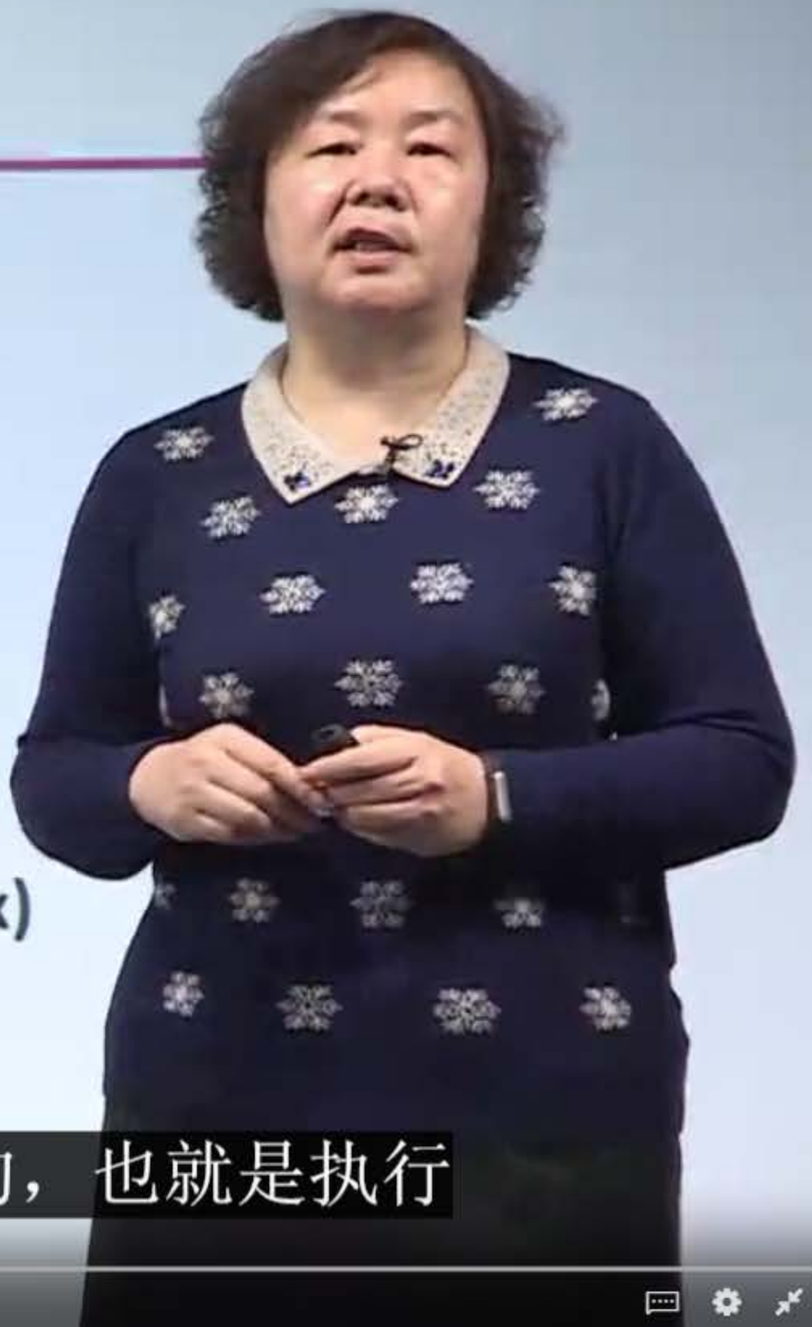


下面呢我们介绍一下 MESA 管程，那么 MESA

0:00 / 13:09

MESA管程

- Lampson和Redell，Mesa语言（1980）
- Hoare管程的一个缺点
  - ✓ 两次额外的进程切换
- 解决：
  - ✓ signal → notify
  - ✓ notify：当一个正在管程中的进程执行notify(x)时，它使得x条件队列得到通知，发信号的进程继续执行



那么发信号的这个进程呢 它还是继续执行的，也就是执行notify 的这个进程呢继续执行

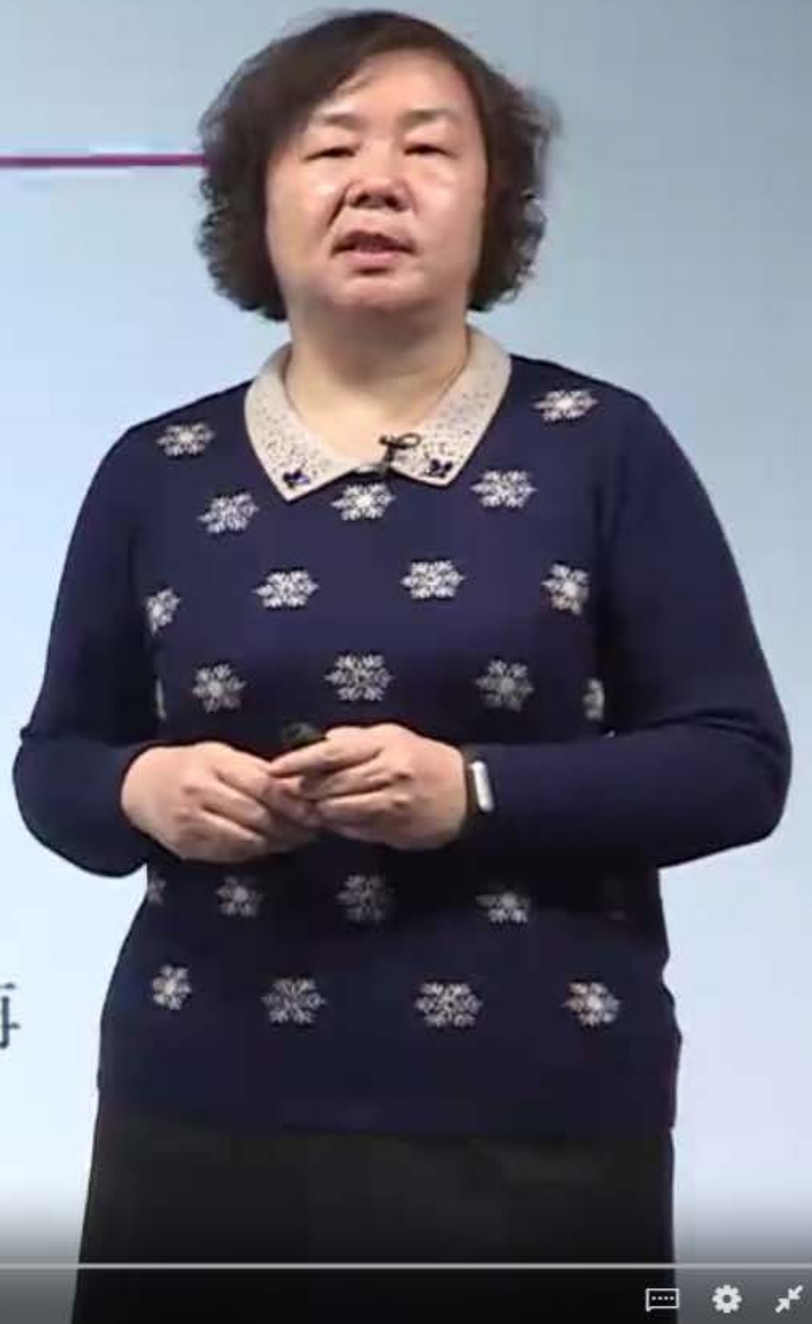
使用NOTIFY要注意的问题

- notify的结果：位于条件队列头的进程在将来合适的时候且当处理器可用时恢复执行

- 由于不能保证在它之前没有其他进程进入管程，因而这个进程必须重新检查条件

  → 用while循环取代if语句

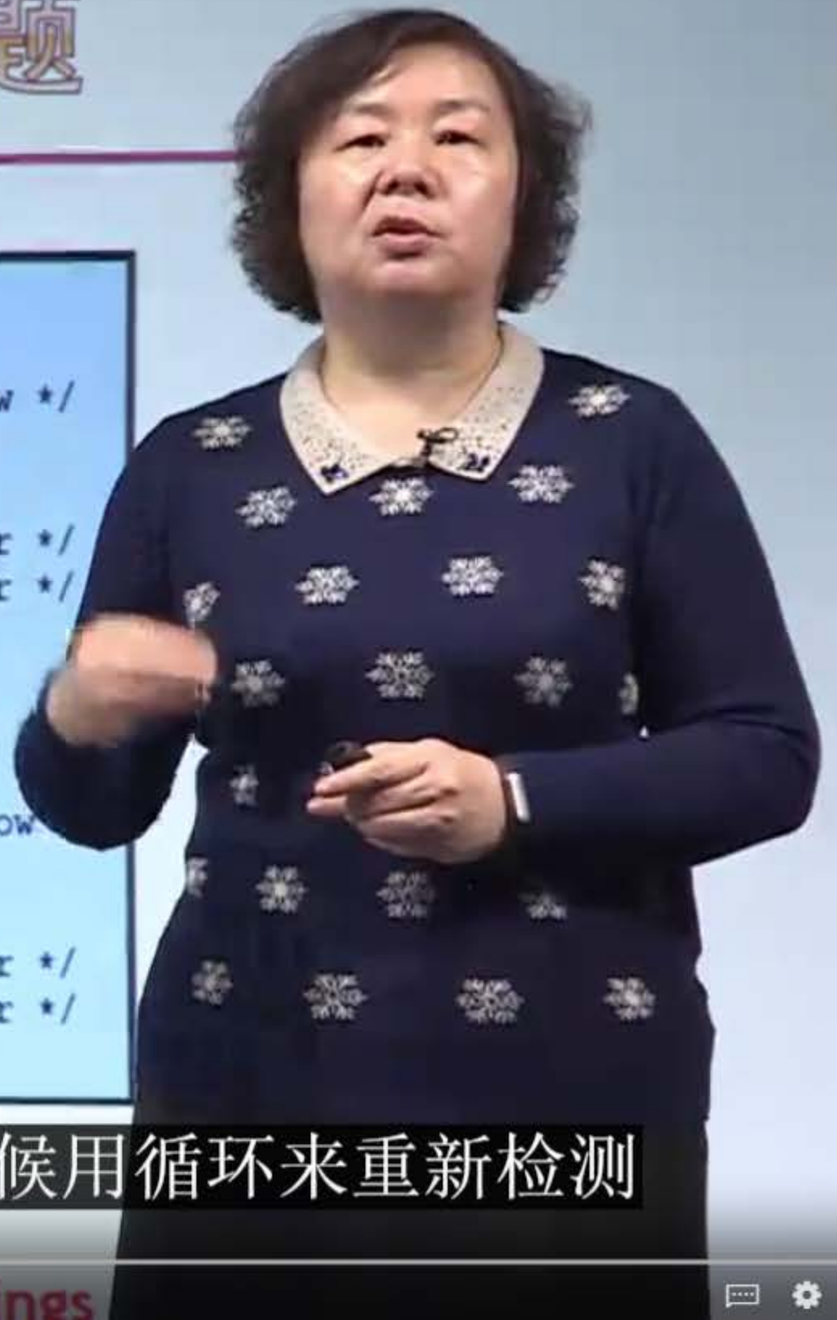
- 导致对条件变量至少多一次额外的检测（但不再有额外的进程切换），并且对等待进程在notify之后何时运行没有任何限制



MESA管程：生产者-消费者问题

```
void append (char x)
{
    while(count == N) cwait(notfull);   /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                                    /* one more item in buffer */
    cnotify(notempty);                         /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                    /* one fewer item in buffer */
    cnotify(notfull);                          /* notify any waiting producer */
}
```



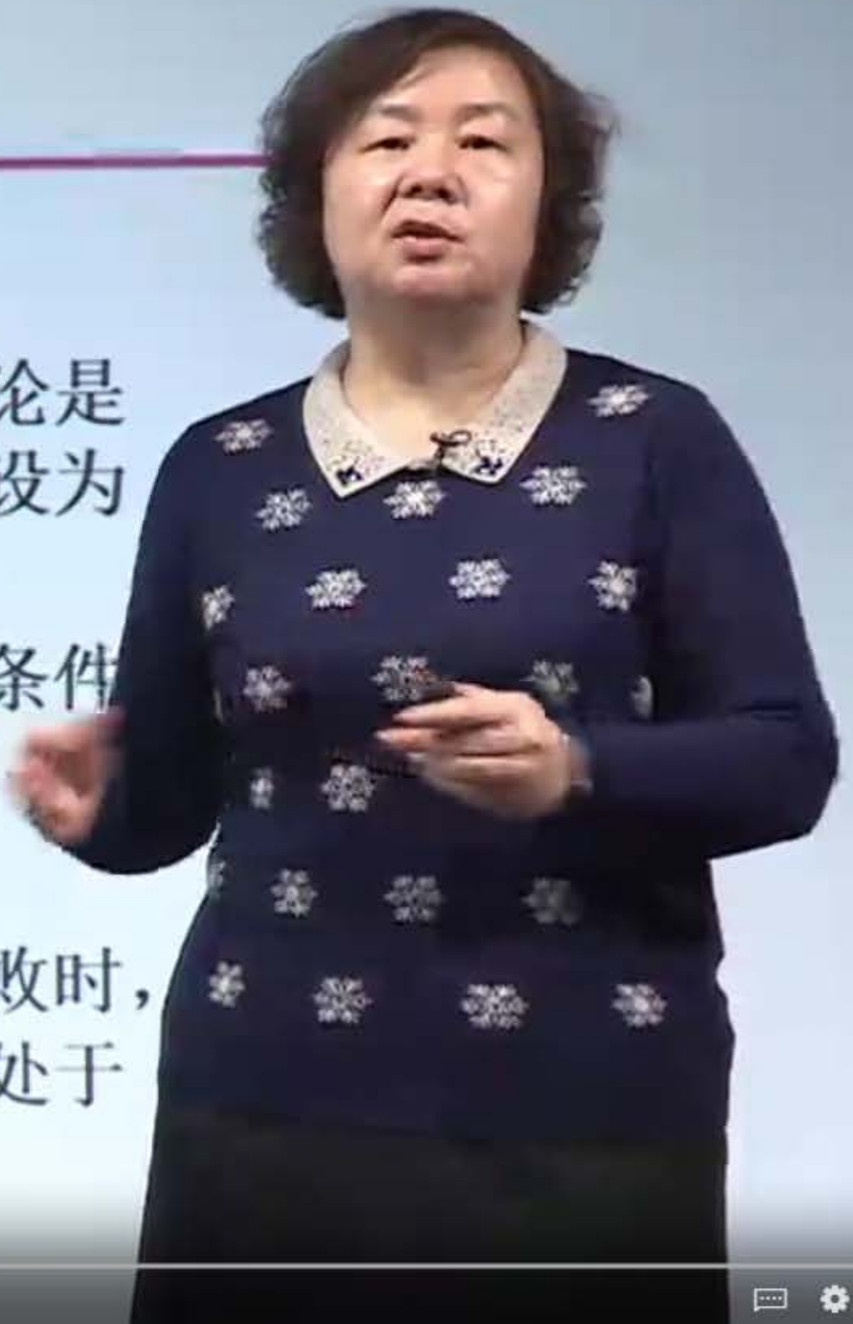
管程的使用都是一样的 只是在判断条件的时候用循环来重新检测

William Stallings

改进NOTIFY

- 对notify的一个很有用的改进
  - 给每个条件原语关联一个监视计时器，不论是否被通知，一个等待时间超时的进程将被设为就绪态
  - 当该进程被调度执行时，会再次检查相关条件，如果条件满足则继续执行
- 超时可以防止如下情况的发生：

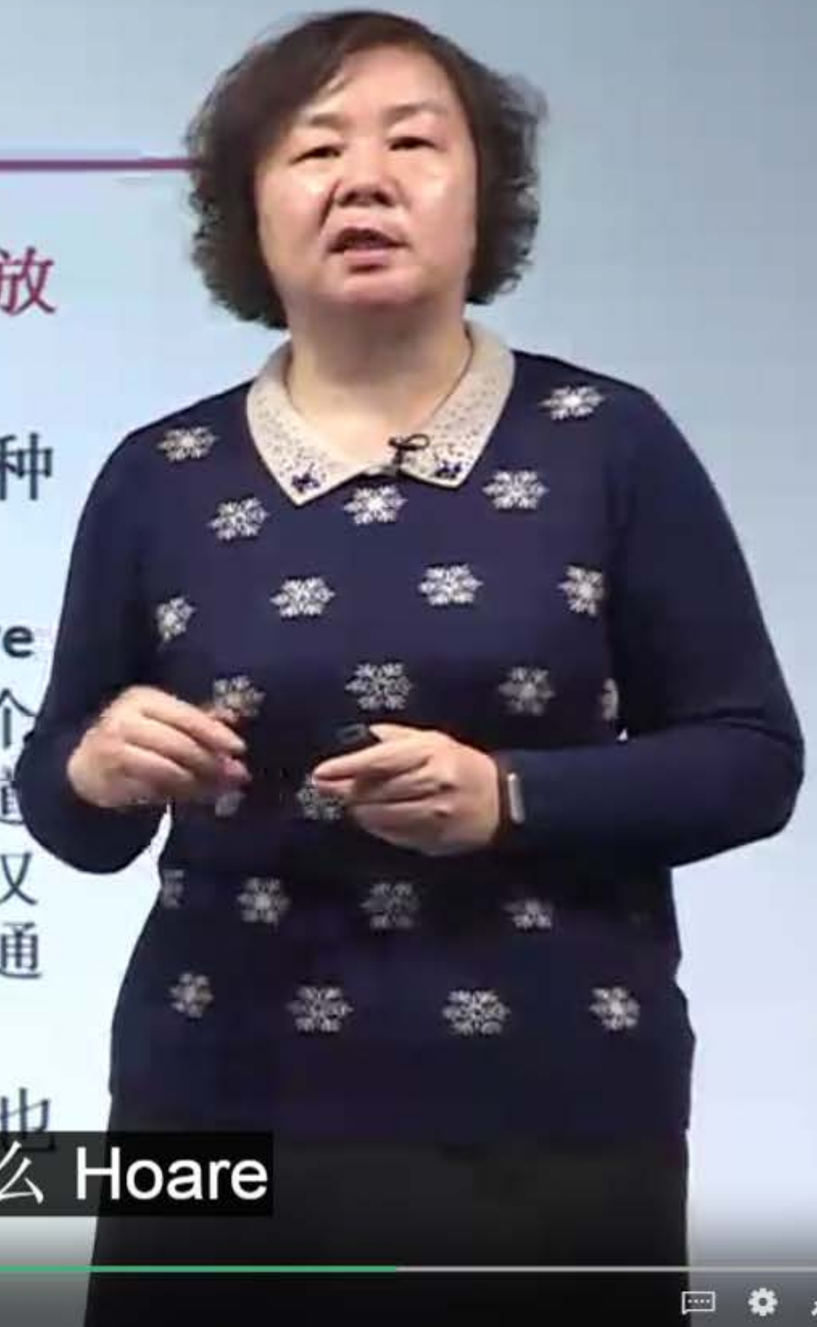
  当某些进程在产生相关条件的信号之前失败时，等待该条件的进程就会被无限制地推迟执行而处于饥饿状态



引入BROADCAST

broadcast：使所有在该条件上等待的进程都被释放并进入就绪队列

- 当一个进程不知道有多少进程将被激活时，这种方式是非常方便的
  - 例子：生产者/消费者问题中，假设**insert**和**remove**函数都适用于可变长度的字符块，此时，如果一个生产者往缓冲区中添加了一批字符，它不需要知道每个正在等待的消费者准备消耗多少字符，而仅仅执行一个**broadcast**，所有正在等待的进程都得到通知并再次尝试运行
- 当一个进程难以准确判定将激活哪个进程时，也可使用广播



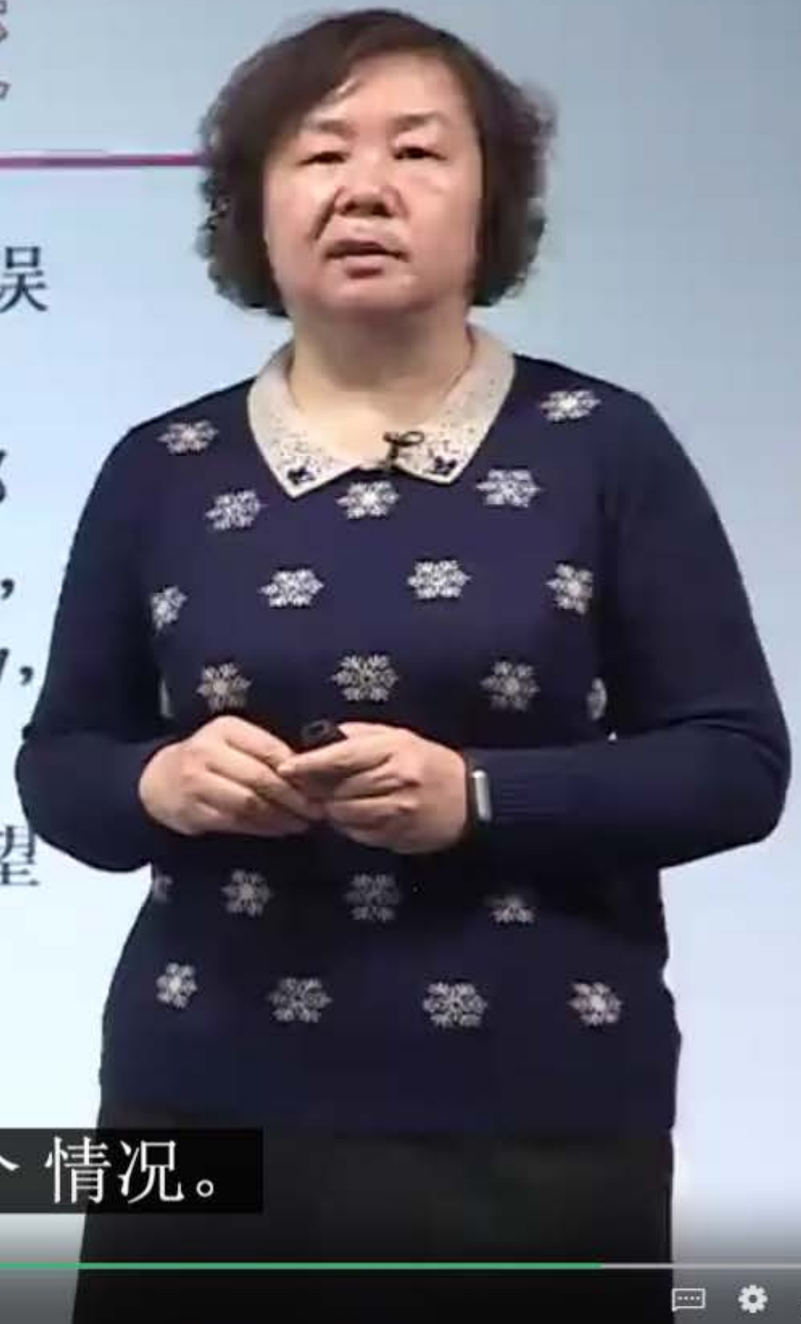
条件变量上操作的一些这个变化 那么 Hoare

10:48 / 13:09

HOARE管程与MESA管程的比较

- Mesa管程优于Hoare管程之处在于Mesa管程错误比较少
- 在Mesa管程中，由于每个过程 在收到信号后都重新检查管程变量，并且由于使用了while结构，一个进程不正确的broadcast广播或发信号notify，不会导致收到信号的程序出错

收到信号的程序将检查相关的变量，如果期望的条件没有满足，它会重新继续等待



Mesa 管程优于 Hoare 管程这样一个 情况。

管程小结

管程：抽象数据类型

有一个明确定义的操作集合，通过它且只有通过它才能操纵该数据类型的实例

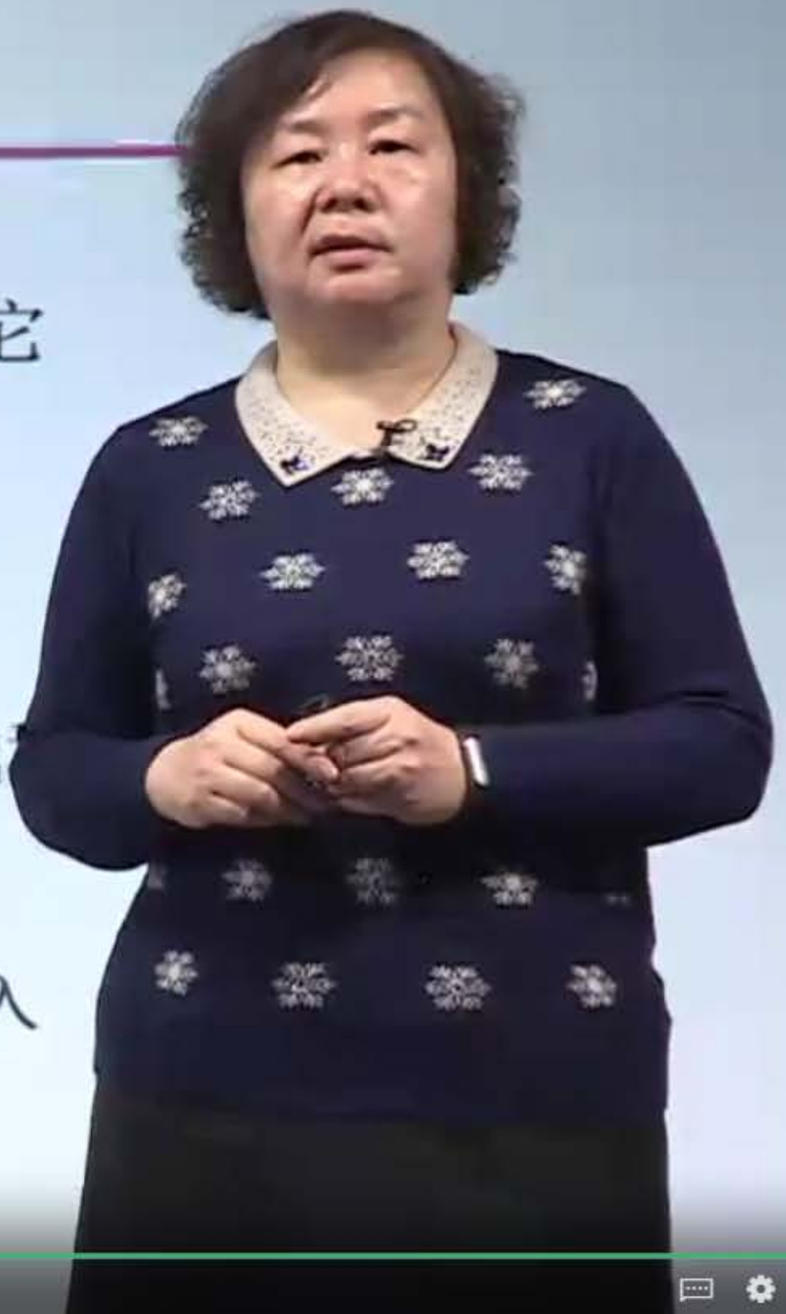
实现管程结构必须保证下面几点：

（1）只能通过管程的某个过程才能访问资源；

（2）管程是互斥的，某个时刻只能有一个进程或线程调用管程中的过程

条件变量：为提供进程与其他进程通信或同步而引入

- wait/signal 或 wait/notify 或 wait/broadcast



0:00 下面呢我们介绍一下 MESA 管程，那么 MESA 管程呢 大家可以看到，它是 1980 年它才提出的，呃 那么它就是由两个这个设计人呢设计了 Mesa 语言，那么这个 Mesa 语言中支持了这个管程 那么为什么又提出了这么新的一个解决方案呢？ 当然了，主要是针对了 Hoare 管程的一个缺点 那么 Hoare 管程是 p 进程唤醒了 q ，那么 p 进程去等待，让 q 进程执行 这样的话呢，就会引起一次进程的切换 那因为 p 进程要等待让 q 进程上 CPU 执行 当 q 进程执行完了，然后呢再 把 p 进程调度上 CPU，那么 p 进程再来一次切换，所以 Hoare 管程会导致两次额外的进程切换 因此，MESA 管程实际上它是 这样一种解决思路，就是 p 进程唤醒了 q 那么 p 进程继续执行，那么 q 进程呢 也重新去等待。 当然了， q 进程等待等在不同的地方 这就是 MESA 管程的语义，所以呢这里头从解决上我们来看一下 原来的 Hoare 管程用的是 signal 这样一个函数 到了 MESA 管程里头呢就改成了 notify， notify notify 的含义是什么呢？ 刚才我们也看到了这样一个词。 notify 是说 当一个正在管程当中，这个某个进程执行了 notify 的时候 那么执行 notify 比如说 x， x 是条件变量，那就 这个操作就会使得等在条件变量上的某个进程得到了一个通知 相当于把它唤醒，得到个通知 让它可以继续执行了。 那么发信号的这个进程呢 它还是继续执行的，也就是执行 notify 的这个进程呢继续执行 那我们看到 notify 的结果是使得 位于条件这个队列的第一个进程得到了通知 那么使得它在将来某个合适的时候 当处理机可用的时候，那么可以让它上 CPU 去执行 那它只是得到了通知，但是呢它没有马上去执行 因此，由于它不是马上执行 就不能保证在它下一次上 CPU 的时候 那个条件还依然成立 也就是换句话说，不能保证在它之前 上 CPU 之前没有其他进程进入管程 你不能保证这一点，因此呢，在这个进程再次上 CPU 执行的时候它要重新检查这个条件是不是成立 我们来看一下就是，当一个进程 检查条件不成立的时候呢，进入了条件队列里等待 当另外一个进程给它 发了一个通知之后，那么这个进程就可以啊 得到通知，然后重新等待不同的地方那么等待上 CPU，可以执行了，等待上 CPU 但是当它真正被调度上 CPU 的时候 可能会有其他的进程在它之前对条件又重新 进行了改变，因此呢这里头必须重新检查条件 那么重新检查条件 就靠的是用 while 循环来取代 if 语句。 我们原来在看代码的时候我们可以看到 判断条件是不是成立用的是 if 语句 但是如果是 MESA 管程里头，你要使用 这个 MESA 语义的话，那么你就必须用 while 循环来代替这个 if 语句。当一个 现进程那么它被唤醒，就是得到通知 它再次上 CPU 运行的时候呢，它需要重新检查条件，因此就是用 while 循环就可以做

到这一点 那么， MESA 管程它的缺点呢就是会导致对条件变量至少 多一次的一个额外的测试，但是因为它没有进程的 额外的切换，它是多了一次测试条件，啊，测试条件 而且呢，对等待进程在 notify 之后什么时候运行呢，也没有任何限制，啊，没有任何限制 所以呢，相对来讲，这个呢是比 Hoare 管程要简单一些 而且要效率高一些 下面我们看一个例子， MESA 管程来解决生产者-消费者问题 那么这个判断呢用 while 来判断 其他的和我们前面所讲的 Hoare 管程的使用都是一样的 只是在判断条件的时候用循环来重新检测 那么 有了这样一个 notify 之后，我们还可以做下面的一些改进 比如说，我们对 notify 做一个非常有实用价值的改进 就是给每个条件，这个原语呢关联一个 监视计时器，啊，有个计时器 就是说它进入了条件这个变量队列以后 不管它是不是得到了通知，只要 它等待的时间超过了一个时间量的时候，就是超时了 那就把它都进入了一个就绪态，原来在条件变量等待，就让它进入就绪态 所以这 notify 可以做这样一个改进 因此，因为什么呢？因为当 这个进程再度被调度运行的时候，等于它会重新 检查条件是不是满足，不满足就继续等待，满足就可以 上 CPU 执行。 那这种情况下 那么因为有再次检测条件的这么一个 把关，所以呢我们允许一些进程，我们可以对 notify 做这样的一个改进 让一些进程在条件这里等到一定的时间就自动地 出来，重新可以被调度。 这就是 notify 的一种改进 那么这种改进呢其实也就是说 通过对超时的一个控制，可以预防一些情况发生，比如说，某些进程 它在产生了这个相关的条件的这个信号之前，它就 失败了，比如说就退出了，它没有来得及去发这个信号 没有来得及发信号，那么等着信号的这个进程就会无限期地 被拖延，对吧，会拖延，就产生了这样一个饥饿现象 所以因此呢，你有这样一个小小的改进呢就可能对这 种问题呢得到一个合适的一个解决方案 还可以呢把 notify 再改，我们说 signal 改到 notify 通知 我们再把 notify 再去改，就引入了一个叫 broadcast notify 是一次通知一个进程，而 broadcast 呢，是所有 等在这个条件变量上的这些进程都被释放让它进入就绪队列 那么 broadcast 的好处呢 我们来看一下，当一个进程它不知道有多少个进程 将被激活的时候，那么用这种 broadcast 就比较好，反正全给你 释放，你们重新进入就绪，然后你们上 CPU 之后重新检查条件，所以也不会引起其他的问题 举一个小例子啊，还是以生产者/消费者为例 以前我们对生产者、消费者的界定是它的缓冲区的大小都是 等长的。 那么假如说有这么一种场景，就是说 insert 和 remove 它是一个可变

长度的这么一个插入和删除 呃，就取出这样一个操作，就是取出的这个数据长度是可变的 那么如果一个生产者，比如说往缓冲区里添加了一批字符 啊，添加了这批数据呢，因为长度是不等的，所以 它不知道到底通知有多少个消费者，因为长度不等，所以 呢它可能这次送入的这些数据可能够三个消费者这个使用 那么也可能是够两个消费者使用，所以呢，它不知道到底供 几个消费者可以使用，所以它干脆就用了什么呀？ broadcast 通过 broadcast 就把所有的等待的这个消费者都给它释放 然后由这些消费者上 CPU 以后重新再去检查条件，重新再去取数据 所以对于生产者而言呢，就比较简单。 好，那么 broadcast 呢还可能是这样，就是说当一个进程不知道 不能准确地去判断说激活某一个进程的时候呢，它也可以使用这种广播机制 所以呢，我们看到就是对于一个，啊，这个 条件变量上的这个操作，从 signal 我们演化到了 notify 那么再从 notify 还可以演化到这个一个改进的 notify 以及 broadcast，可以做不同的设计 那么这就是 Mesa 管程以及还有一些对这个 条件变量上操作的一些这个变化 那么 Hoare 管程与 Mesa 管程 我们看一下它的这个的比较啊 那么 Mesa 管程呢我们通常认为会优于 Hoare 管程 主要是由于 Mesa 管程的这个出错的几率比较小 因为在 Mesa 管程当中，由于每个进程在收到信号之后 它都会去重新检查这个相应的这个变量，所以呢 又通过使用了这个 while 结构，所以如果一个进程没有正确的 broadcast 这个广播或者是发信号的话 那么也不会导致收到这个信号的进程呢出错 所以这样的话呢，就是它的出错的几率会小 那么因为收到信号的进程呢，去检查，依然重新检查相关的变量 如果所期望的条件没有满足的话，它还会继续等待，所以这是 这个 Mesa 管程优于 Hoare 管程这样一个 情况。 下面我们对管程进行一个小结 管程是一个抽象的数据类型 它有一个明确定义的一个操作的集合 并且只能通过这个集合里的操作来操作 数据类型当中的实例。 那么实现管程这个 结构呢，要保证以下两点。 第一点是只能通过管程 提供的某个过程才能访问管程当中的资源。 第二点呢是管程是 互斥使用的，某一时候只能有一个进程或者线程去调用管程当中的过程 那么如何解决互斥问题呢？这是管程本身机制所决定，也就是它是语言 成分，因此呢是由编译器来保证的 那么如何解决同步问题呢？实际上是通过了条件变量以及在这个条件变量上 提供的若干操作，那么哪些操作呢？有 wait/signal 或者是 wait/notify ，或者是 wait/broadcast 这就是管程的主要的一个小结