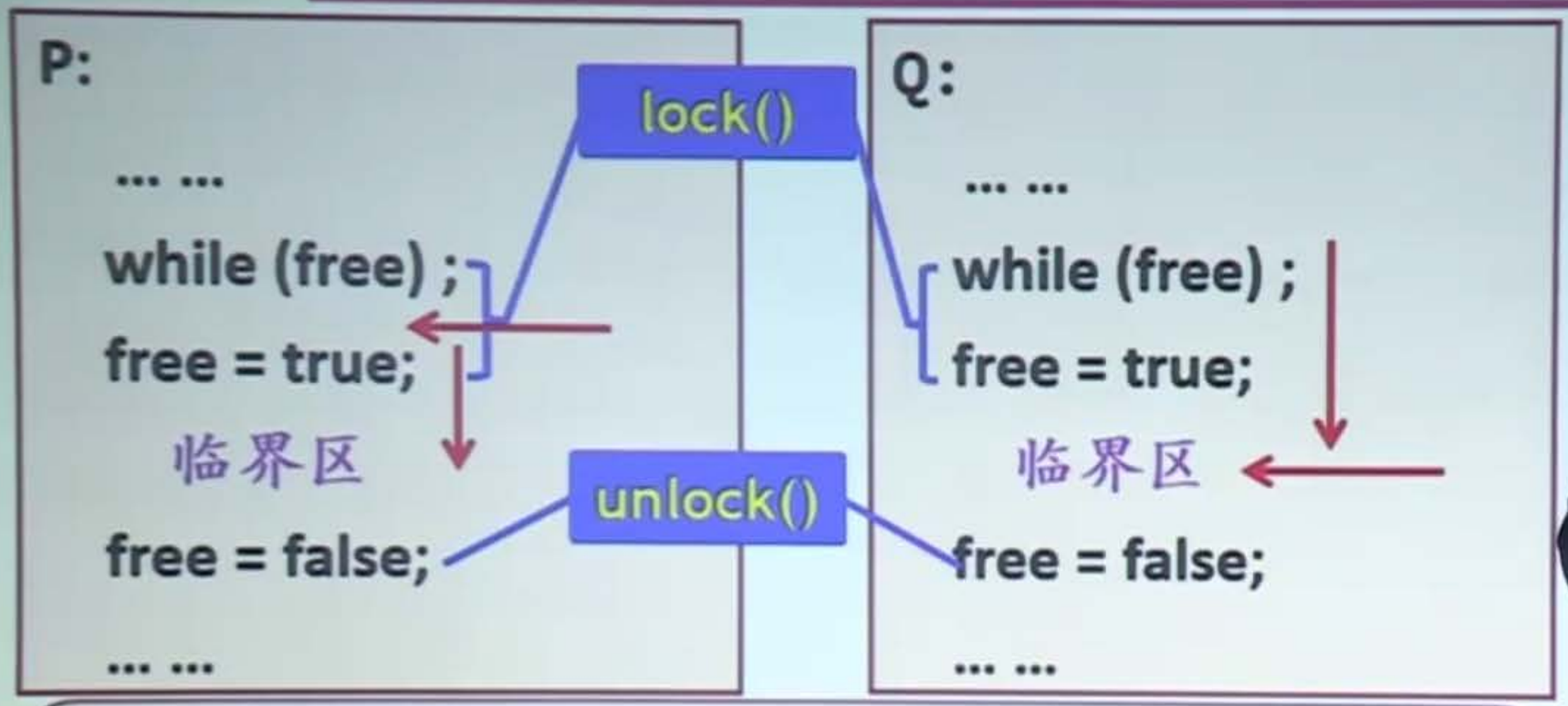
Dekker解法、Peterson解法

进程互斥的软件解决方案

下面我们介绍，进程互斥的软件解决方案 首先，我们引入第一个解决方案

软件解法1



free: 临界区空闲标志

true: 有进程在临界区

false:无进程在临界区

初值： free为false

软件解法2

```
P:
    … …
    while (not turn) ;
        临界区
    turn = false;
    … …
```

```
Q:
    … …
    while (turn) ;
        临界区
    turn = true;
    … …
```

turn: 谁进临界区的标志
    true: P进程进临界区
    false: Q进程进临界区
初值任意

我们再来看看，软件解法的第三种

软件解法3

*After you*问题

P:

```
... ...
pturn = true;
while (qturn) ;
    临界区
pturn = false;
... ...
```

Q:

```
... ...
qturn = true;
while (pturn) ;
    临界区
qturn = false;
... ...
```

pturn, qturn: 初值为false
P进入临界区的条件: pturn∧ not qturn
Q进入临界区的条件: not pturn∧ qturn

软件解法4——DEKKER算法

```
P:
    … …
  pturn = true;
  while (qturn) {
      if (turn == 2) {
          pturn = false;
          while (turn == 2);
          pturn = true;
      }}
    临界区
  turn = 2;
  pturn = false;
  … …
```

```
Q:
    … …
  qturn = true;
  while (pturn) {
      if (turn == 1) {
          qturn = false;
          while (turn == 1);
          qturn = true;
      }}
    临界区
  turn = 1;
  qturn = false;
  … …
```

在解法3基础上
引入turn变量

循环

第一个

不断地在测试，实际上是在浪费时间，这就是 DEKKER
算法，但是呢它确实解决了临界区的保护问题

软件解法5——PETERSON算法

```
#define FALSE 0
#define TRUE  1
#define N     2                    // 进程的个数
int turn;                          // 轮到谁?
    int interested[N];
  // 兴趣数组，初始值均为FALSE

void enter_region ( int process)
                     // process = 0 或 1
   {
       int other;
            // 另外一个进程的进程号
       other = 1 - process;
       interested[process] = TRUE;
            // 表明本进程感兴趣
       turn = process;
            // 设置标志位
       while( turn == process &&
   interested[other] == TRUE);
   }
```

循环

```
void leave_region ( int process)
   {
       interested[process] = FALSE;
            // 本进程已离开临界区
   }
```

```
进程i:
    …… ……
    enter_region ( i );
       临界区
    leave_region ( i );
    …… ……
```

Peterson算法解决了互斥访问的问题，而且克服了强制轮流法的缺点，可以完全正常地工作（1981）

0:00 下面我们介绍，进程互斥的软件解决方案 首先，我们引入第一个解决方案 这个解决方案的思想是，我们用一个 free 来代表临界区是否空闲这样一个标志 如果 free 等于 true ，就表示说 有进程在临界区，如果 free 等于 false ，就表示说临界区是空闲的 当然啦，初值， free 的初值呢是 false 好，我们来看一下，现在有两个进程 当它们都想进临界区的时候呢，首先要做的事情呢，是要判断一下 free 这个标志 如果 free 这个标志是 false 的话，那么这个循环就结束 然后呢，进到下面一条语句，也就是把这个标志设置为 true ，表示有进程在临界区 我们假设 P 先上的 CPU 它呢，因为初值啊，这个 free 的初值是 false ，所以呢 这个循环就结束了。 那么这个时候呢 如果，这个时候 进程 P 被切换下 CPU ，那么在这个时候下面上 CPU 的，如果正好又是进程 Q 那么进程 Q 呢，它也要判断 free 的这个标志是不是 false ，由于进程 P 还没有来得及改变它的值，因此 Q 检测的结果是 free 也等于 false ，所以 继续往下走啊，继续往下走。 所以我们看它继续往下走 那么把 free 变成 true ，然后进入了临界区 如果 Q 进入临界区之后又一次被切换下去了 也就是说它被切换下去了，在这个时候，而之后假设 正好又是 P 上 CPU 了，所以 P 进程上 CPU 之后呢，接着要做的事情就是把 free 变成 true ，然后进入临界区 那么，结果我们会看到，在临界区里头 两个进程都在里头，就没有 满足我们刚才提出的这几个，临界区的使用的原则 这就是一个错误的解法 但是呢，尽管它是一个错误的解法，我们可以中间得到一些启发，也就是说 这样两条语句，实际上呢是 相当于加锁这么一个概念，可以把这两条语句呢 给它写成一个 lock 函数 如果把 lock 函数的这个执行 给它设计成一个原语，也就是在执行过程中不容许被中断 那么，这个操作就是正确的了。 所以呢，我们假设这两条语句就 给它封装成一个 lock 函数，那么 lock 的实现的时候，我们把它实现成原语原子操作那么中间就不会被打断，就不会出现刚才我们所说的场景 同样，那么 free 等于 false ，我们也把它封装成一个 解锁的函数啊，也是一个原语操作 那么，因为这虽然是一条语句，但这条语句变成 了指令一级的话，会变成多条指令。 所以，同样我们需要把 unlock 这个函数啊，设计成一个原语啊，原语 所以，这是软件解法的第一种 我们再来看看，第二种软件解法 这里头，我们设置了一个 turn 这样一个标志 而 turn 这个标志呢，我们给它赋予一个 说明是， turn 如果等于 true 的话呢，表示 让 P 进程进临界区，如果 turn 等于 false 就让 Q 进程进临界区，初值呢是任意 那这里头，我们可以看到，那么可能会出现这样一个场景，就是说 P 进程想进临界区，由于 turn 等于 false ，所以呢，不该它进 所以 P 进程在

那里头，老在那啊等待进入临界区，它在那循环啊，循环 看看能不能进临界区。 可是，如果 Q 进程始终 没有进过临界区，也不想进临界区，那么 P 进程就进不了临界区，尽管临界区里头没有进程 也就是说，在临界区外的进程 Q ，阻止了 P 进程进临界区 啊，这是不容许的，这也是违反我们前面所介绍的使用临界区的原则的 我们再来看看，软件解法的第三种 那么这种情况是这样的，我们设置了两个标志 pturn 呢，是表示 P 进程要进临界区的意向 qturn 呢，表示 Q 进程要进临界区的想法 那么，进临界区它的条件判断是这样的 P 进程要想进临界区，那么首先 pturn 等于 true 然后，qturn 呢不等于 true 所以，这样的话呢，那么 P 进程可以进临界区 同样， Q 进程也是要判断自己想 进临界区并且对方不想进临界区的时候，要做这样一个判断 那我们来看一下啊，如果 P 进程上 CPU 了，那么 pturn呢 等于 true 了 那么这个时候呢，假设 P 进程又被切换下去了 所以， Q 进程上 CPU ，那么 Q 进程呢 它呢执行了说，它也想进临界区，所以 qturn 也等于 true ，接着 那么 Q 进程会去判断能不能进？也就是说判断 P 进程想不想进，可是刚才由于 P 进程已经设置 pturn 等于 true 了，所以这个地方，Q 进程就会在这不断的循环 因为， P 进程想进。好，那么时间片 到了，那么 Q 进程被切换下 CPU 于是呢，可能假设 P 进程又上 CPU 了，所以 P 进程呢也在 做下面的事情，要判断 Q 进程想不想进临界区 那么 Q 进程是想进临界区的，所以呢， P 进程也不进临界区 在那里头不断的去测试，看看 P 进程想不想进临界区 那么这样会出现什么情况呢？临界区是没有进程在里头的，而 P 进程 和 Q 进程也都不进临界区，这就是一个我们所说的，通过这样 两个这个不断的在这循环，谁都不去进临界区。 就得到这么一个场景 而这个场景呢，我们就称为 after you 问题，就大家都在谦让 谁也不进临界区，这也不满足我们前面所谈的临界区的使用原则 那么下面呢，我们来介绍一下 dekker 算法 这是第四种解法， dekker 算法呢，实际上是在 1965 年第一个用软件的方法解决了这个临界区的，这个保护问题 那么它的这个算法的思想啊，实际上就是在刚才我们所介绍的 算法 3 的基础之上，又引入了一个 turn 变量，这个 turn 变量是相当于是一个枚举类型由这个 turn 变量来决定，在两个进程都想 进又都谦让的这种情况下，由 turn 来决定谁进 啊基本思想是这样的。 我们来看一下，假设 还是 P 和 Q 都想进临界区，如果 P 进程在这个地方被中断了 那么 Q 进程上 CPU Q 进程上 CPU 之后呢，它也把这个 qturn 设置成了 true ，好，这就是我们 刚才碰到的这种

after you 的一个场景 而不管是 P 进程还是 Q 进程，它接着要执行的呢，我们来看一下 假如以 Q 进程为例，那么它判断 P 进程想不想进临界区，如果它判断 P 进程想进临界区，也就是说它们都想进临界区的情况下 就由 turn 来决定让谁进临界区 好，那我们看看，如果 turn 等于 1，假设这是让 P 进程进临界区，那么 Q 进程就做了一个决定 它把自己变成 false，qturn 变成 false，就是说我不进了，让出来了 那么让出来之后，它做的事情就不断的在那去循环，看看是不是该轮到自己进 啊，就是说如果 turn 等于 1，就是 P 进程要进，所以这个时候这个循环就一直执行下去 那么直到 turn 等于 2，这个循环才结束啊，才结束 所以，我们来看看，如果对 P 进程 它根据，如果 Q 进程也想进，所以 P 进程呢判断出 Q 进程想进的情况下 一定去判断 turn 这个值是不是等于 2，如果是等于 2，就让出 CPU 那么，它也实际上判断 turn 是不是等于 1，如果是就让出 CPU 所以，它们两个进程，总有一个根据 turn 的值会让出 CPU，而另外一个就可以直接进入临界区了 好，那么这两条语句就是说在 在让出 CPU 之后还要保持不断的循环 去判断是不是该轮到自己了，因此呢这里有一个循环 反复在那循环，直到它把它的时间片用完然后被切换下 CPU，始终在 CPU 上在循环 这里头呢这个就是一个我们称之为叫忙等待 busy waiting，也就是它在 CPU 上等，当然大家会去想说 假定只有一个 CPU 的情况下，你在 CPU 上等，那谁上 CPU？没有人进程上 CPU，所以你必须在 CPU 用完了 时间片才能被切换下去，所以浪费了这些时间，就是说你 不断地在测试，实际上是在浪费时间，这就是 DEKKER 算法，但是呢它确实解决了临界区的保护问题 那么下面我们来看一个 更好的软件的解法，叫 PETERSON 算法 PETERSON 算法呢它实际上是 1981年 被开发出来这么一个算法，它是解决了这个互斥访问的这样 一个问题，同时呢，也克服了这种强制轮流的这样一个缺点 像刚才 DEKKER 算法它有一个强制轮流这样一个缺点 可以完全非常正常地工作，而且它提出的这个 方案呢还对于开发的用户呢，还是非常容易使用的 我们来看一下。任何一个进程当它想进临界区的时候，它只需要调用一个 enter_region 这样一个函数 来看一看能不能安全地进入临界区 如果能安全地进入临界区，那么就是相当于这个函数执行结束，它就可以进临界区了 如果不能够安全地进入临界区，那么它就会去再等待 在这个函数当中去等待进入临界区。 那么 调用这个函数的时候呢是用进程号，当这个进程使用完 临界区的相关资源之后，那么它在出临界区的时候呢，那么就调用一个

leave_region，那么也就是可以让其他的进程进入临界区。基本的思想就是非常简单了 它去调用这样一个函数，那么我们来看看 enter_enter 这个函数是怎么实现的 如果只有一个进程想进临界区，所以 也非常简单，就进临界区就可以了，所以这个函数就使用比较简单 但是呢，我们碰到的往往是说，如果两个进程，当然我们这是一个最简单的情况，两个进程 都同时想进临界区，我们来看一下 那么这个代码当中，那么 interested 是一个数组，它是表示出 哪个进程想进临界区，要把这个意愿表达写在这个数组的某个单元里头 它是这样一个，第一个，这是一个标志，标志想不想进临界区 那么调用的时候呢是用进程号来调用的，那么 那么假设我们有两个进程，进程号是 0 和 1，所以另外一个进程它的 编号呢，就 other 呢就是 1 减去当前这个进程的进程号 然后做的事情呢就是把这个意愿表达出来，就是在这个数组里头表达出 某个进程想进临界区，把这个单元设置成 true 在此之后呢，有一个非常重要的 变量，叫 turn，turn 这个变量呢就是用 当前的这个进程进程号赋给这个 turn 好，那么谁要想进临界区，就用自己的进程号赋给这个 turn 那么我们看到，如果两个进程同时想进临界区的话 那么就对 turn，因为 turn 是一个，大家都可以往里头赋值的这么一个共享的变量 所以两个进程，比如说 0 和 1 都想进临界区，那就看谁 先赋值给 turn，谁后赋值给 turn 我们假设 0 号进程先赋值给 turn 那么后，后面的是 1 号进程赋值给 turn，因此 先赋值给 turn 的那个值呢实际上就被后面赋值的这个覆盖掉了 所以 turn 里头始终保持是后面那个 来的要想进临界区的进程的进程号 那么下面我们来看一下，就要做判断了，到底是两个进程都想进，让谁先进呢？ 那我们当然希望说谁先来的谁先进，也就是谁先给 turn 赋值的，谁先进临界区 那我们来看一下这里头 这有一个判断，这个判断呢 是这样的，turn 等不等于 process，那么好了，我们来看 process 呢是当前这个进程的进程号 可是如果你是要先进临界区，先赋值给 turn 的，那么你的值就 turn 的值后面已经被后面那个进程给覆盖掉了，所以现在 turn 不等于 0 了，所以这个条件就不成立，这个条件不成立，这个循环就结束了 也就是换句话说，0 号进程实际上这个 循环呢一次都没有做就出来了，出来之后呢就进临界区了 那我们来看 1 号进程，1 号进程是后把这个 进程号赋给这个 turn 的，所以呢 turn 的值呢就等于 1 因此呢，对于第一个条件，那么肯定是成立的 然后我们再看第二个，因为第二个是表达出你想不想进 临界区，那么大家都是想进临界区的，所以呢第二个条件也成立，所以

两个条件都成立，- 所以对于 进程 1 来讲，实际上它调用了 region 这个 enter_region，实际上是在这个地方做循环 所以它在不断地去循环。 那么这个呢是离开这个临界区 之后要调用的这么一个函数，实际上就是把 interested 这个单元呢设置成 false 表达出已经出临界区了就可以了，所以呢，这是一个 PETERSON 算法 那么以上我们介绍了五种解法，当然了 DEKKER 算法呢和 PETERSON 算法呢都是正确的算法 同时呢，PETERSON 算法由于它使用这个 就在临界区使用之前是非常简单地 调用了这个 enter_region 和 leave_region，所以对于编程人员来讲是非常简单的一件事情 这也给我们后面的一些解决方案呢给出了一些启示