第二讲

1、彼得森算法（Peterson’s Algorithm）

上次课介绍了（适用于两个线程）锁的一种实现 LockOne。LockOne 用两个共享变量 flag 实现了互斥，但不满足无死锁性质。为了避免死锁，需要引入新的机制。

彼得森算法在 LockOne 的基础上，多用一个共享变量 victim：

```
class Peterson implements Lock {
private boolean[] flag = new boolean[2];
private int victim;
public void lock() {
 flag[i] = true;                    //告诉另一进程我要进入临界区
 victim = i;                        // 让对方先进入临界区
 while (flag[j] && victim == i)) {} //等待对方离开临界区
}

public void unlock() {
 flag[i] = false;
}
```

锁应当满足的三个基本性质：互斥(安全性)，无死锁(活性)，无饥饿(活性)

1.1 Peterson 算法满足互斥

证明：（反证法）

假设 A、B 两个线程都在临界区，并且，不失一般性，假定 A 是最后一个写 victim 的线程：

$$\mathbf{write_B(victim=B) \rightarrow write_A(victim=A)} \quad (1)$$

考虑每个线程在进入临界区前在对 flag 的最后的读和写。从代码可得

$$\mathbf{write_B(flag[B]=true) \rightarrow write_B(victim=B)} \quad (2)$$

以及

$$\mathbf{write_A(victim=A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)} \quad (3)$$

结合（2）、（1）、（3）,得到

$$\mathbf{write_B(flag[B]=true) \rightarrow write_B(victim=B) \rightarrow}$$

$write_A$(victim=A) ➔ $read_A$(flag[B]) ➔ $read_A$(victim)

即 A 进入临界区前读 flag[B] == true 且 victim == A，因此 A 不可能进入临界区。这与 A、B 两个都在临界区的假设矛盾。因此 Peterson 算法满足互斥。

1.2 Peterson 算法满足无死锁

证明：根据代码，死锁只可能发生在循环中：

```
public void lock() {
  …
  while (flag[j] && victim == i) {};
```

死锁意味着：

flag[B]==true and victim==A 且 flag[A]==true and victim==B，

而 victim 不可能同时既为 A 又为 B。所以不可能出现死锁。

Peterson 算法使用一个共享变量 **victim** 成功地避免了 LockOne 存在的死锁问题。

1.3 Peterson 算法满足无饥饿

证明：反证法。假定 A、B 两个线程中有一个饥饿，不妨设是 A。即 A 永远陷在 lock() 方法的 while 循环中，这需要 flag[B] == true && victim == A。

在这种情况下，B 只有两种可能：
（1））B 也陷在 lock() 方法的 while 循环中，需要满足的条件是 flag[A] == true && victim == B。但 victim 不可能同时既为 A 又为 B。矛盾。
（注：这就是死锁的情况）

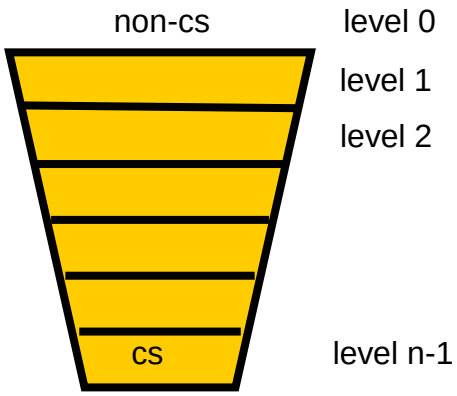
（2）B 反复进入、离开临界区。当 B 离开临界区时，它调用 unlock()，置 flag[B] == false。当它下次再进入临界区时，它调用 lock()，置 victim == B。这与 A 永远陷在 lock() 方法的 while 循环中的条件 flag[B] == true && victim == A 矛盾。
所以 Peterson 算法不可能导致线程被饿死，

注意：这里证明的是 Peterson 算法自身不会导致线程被饿死，这不能排除由于其它因素，如调度算法不公平而导致的饥饿。

Peterson 算法只适用于两个线程的系统。下面介绍两个适用于多个线程的互斥算法。第一个是过滤器算法。它是 Peterson 算法的直接推广。另一个是 Lamport 的 Bakery 算法。

2. 过滤器算法（The Filter Algorithm）

设系统中线程个数为 n。算法设置了 n-1 个“等待室”，称为“级”。线程必须逐级通过这些等待室才能进入临界区。一个在 i 级上的线程也在所有比 i 小的级上。

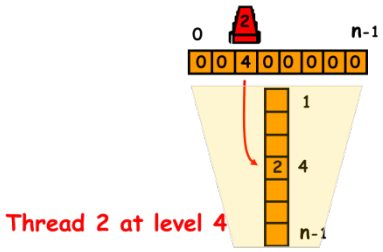


代码实现：

```
1 class Filter implements Lock {
2   int[] level;
3   int[] victim;
4   public Filter(int n) {
5     level = new int[n];
6     victim = new int[n]; // use 1..n-1
7     for (int i = 0; i < n; i++) {
8       level[i] = 0;
9     }
10 }

11 public void lock() {
12   int me = ThreadID.get();
13   for (int L = 1; L < n; L++) { // attempt level L
14     level[me] = L;
15     victim[L] = me;
16     // spin while conflicts exist
17     while ((∃k != me) (level[k] >= L && victim[L] == me)) {};
18   }
19 }

20 public void unlock() {
21   int me = ThreadID.get();
22   level[me] = 0;
23   }
24 }
```



初始时所有线程都在 0 级。一个试图进入临界区（n-1 级）的线程 A 需要从 1 级开始，逐一进入下一级（第 13 行的 for 循环）。每次为了进入下一级，A 都需要执行第 17 行的 while 循环。如果当线程 A 执行完一次 while 循环时 level[A] ≥

L，就说 A 处于 L 级。

level[i] = L 表示线程 i 试图进入 L 级。与 Peterson 算法类似，victim[L] = i 表示线程 i 让其它（也试图进入 L 级的）线程先进入。

等待室的设置具有两个重要的性质：

如果有进程试图进入 L 级，那么其中至少有一个成功进入；如果试图进入的线程不止一个，那么至少有一个被阻止在它所在的级上继续等待。

这就保证了至多 n 个线程可以同时进入 0 级，至多 n-1 个线程可以同时进入 1 级，…，至多 1 个线程可以进入 n-1 级（即临界区）。

引理：对所有 0 与 n-1 之间的 L，至多只有 n-L 个线程处于 L 级。
证明：施归纳于 L。
基始步骤 L=0。易见至多只有 n 个线程处于 0 级。
归纳步骤。假设引理对 L-1 成立。由归纳假设，至多只有 n-L+1 个线程处于 L-1 级。用反证法证明至多只有 n-L 个线程处于 L 级。

假定有 n-L+1 个线程处于 L 级。令 A 是 L 级上的线程中最后一个写 victim[L]的。那么对所有 L 级上的其它线程 B,

$write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

从代码可得

$write_B(level[B]=L) \rightarrow write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

从代码还可得

$write_A(victim[L]=A) \rightarrow read_A(level[B]) \rightarrow read_A(victim[L])$

综合以上可得

$write_B(level[B]=L) \rightarrow write_B(victim[L]=B) \rightarrow$
$write_A(victim[L]=A) \rightarrow read_A(level[B]) \rightarrow read_A(victim[L])$

因此每次 A 在第 17 行 读 level[B] ≥ L 且读 victim[L]=A，所以 A 不可能完成 while 循环进入 L 级。矛盾。

根据这个引理，至多只有 1 个线程能够进入 n-1 级，即临界区。因此有：

推论：Filter 算法满足互斥。

其它性质：

（1）无饥饿：在每一级都如同 Peterson 算法，不会有线程饥饿。

（2）无死锁。无饥饿的直接推论。

（3）公平性：Filter lock 只满足很弱的公平性，一个线程可能被其它线程超越任意多次。

2、面包房算法（Bakery Algorithm）（Lamport 1974）

“先到先服务”。基本思想是让每个要进入临界区的线程取个号，号小的比号大的先进入临界区。号是单调增加的。

代码如下：

```
1  class Bakery implements Lock {
2    boolean[] flag;
3    Label[] label;
4    public Bakery (int n) {
5     flag = new boolean[n];
6     label = new Label[n];
7     for (int i = 0; i < n; i++) {
8       flag[i] = false; label[i] = 0;
9     }
10 }

11   public void lock() {
12   int i = ThreadID.get();
13   flag[i] = true;
14   label[i] = max(label[0], ...,label[n-1]) + 1;
15   while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16   }

17   public void unlock() {
18    flag[ThreadID.get()] = false;
19   }
20 }
```

我们将 lock 方法中的 13、14 两行称为“门关”（doorway）。线程 A 的门关用 D_A 表示。为了达到“先到先服务”，需要知道一个线程比另一个线程“先到”。而当两个线程并发调用 lock()时，没有机制能告诉我们谁先调用，谁后调用。我们的解决办法是看谁先执行完 doorway：如果 $D_A \rightarrow D_B$ 就认为 A 先到。Flag 和 label 都是共享变量。一个线程调用 lock()后，就进入门关。这里它首先将自己的 flag 置为 **true**，让其它线程知道它要进临界区。然后计算自己的序号 label[i]。为此（以任意顺序）读取所有 n 个线程（包括该线程自己）的序号，算出这 n

个序号的最大值，再加 1，作为自己的新序号。因此新序号一定比原序号大。但是两个（或多个）并发调用 lock()的线程的序号可能相等。为了解决这个问题，比较两个线程先后时，按字典序比较 label[i] 和线程标识号 i 组成的对偶 (label[i], i))：

(a,i) < (b,j)： a < b 或者 a = b 且 I < j

由于线程在计算自己的序号时读取了所有线程，包括自己，的当前序号，然后取所有序号的最大值加 1；另外当线程释放锁时，没有重置 label。所以每个线程的序号是严格递增的。

2.1 Bakery 算法满足无死锁。

证明：在执行第 15 行 while 循环、等待进入临界区的线程中，一定有一个线程 A，它的(label[A],A)最小。这个 A 线程一定会进入临界区。

2.2. Bakery 算法满足先到先服务（First-Come-First-Served）。

证明：假设 D_A ➔ D_B。由代码可知，

$write_A(label[A])$ ➔ $read_B(label[A])$ ➔ $write_B(label[B])$ ➔ $read_B(flag[A])$

由此可知 label[A] < label[B]。所以 B 必须等到 A 释放锁，将 flag[A]置为 false 后 才有可能完成第 15 行的 while 循环进入临界区。

先到先服务蕴含了公平性。

由于无死锁和先到先服务蕴含了无饥饿，我们有

2.3. Bakery 算法满足无饥饿。

2.4. Bakery 算法满足互斥

证明：用反证法。假设两个线程 A 和 B 同时在临界区内。令 $Labeling_A$ 和 $Labeling_B$ 分别是线程 A 和 B 获得序号的时刻。假定 (label[A],A) << (label[B],B)。

当 B 进入临界区前，它看到的第 15 行 while 循环的条件必须为假。由于 (label[A],A) << (label[B],B)，只能是 flag[A] = false。由此推出

$Labeling_B$ ➔ $read_B(flag[A])$ ➔ $write_A(flag[A])$ ➔ $Labeling_A$

因此 label[A] < label[B]。这与(label[A],A) << (label[B],B)的假定矛盾。

Bakery 算法提出时（1974 年）主要是面向分布式并发系统的。在这种系统中没有共享存储，每个进程的 flag 和 label 变量都在它自己的内存中。任意进程都可

能出故障（失效）。可以假定当一个进程失效时，其它进程读它的 flag 和 label 是返回的值是 0（false）。Bakery 算法容许进程失效（假定进程失效后立即跳到非临界区并终止）。不过如果一个进程不断地失效、重启，那仍会导致整个系统死锁。

四、固有的代价

Bakery 算法简短、精炼、公平，并且不依赖于低层的互斥机制。但它需要读 n 个全局变量，而读全局变量是代价较高的操作。有没有可能降低这个代价呢？

共享存储多处理器系统中，线程之间通过读、写共享的内存单元实现协作。共享内存单元有几种类型：

多读单写（MRSW，Multi-Reader-Single-Writer）：本线程可以读写，别的线程只可以读。例如 flag[]。

多读多写（MRMW，Multi-Reader-Multi-Writer）：本线程和别的线程都可以读写。例如 Victim[]。

另外还有单读多写 SRMW 和 单读单写 SRSW，没有多大实际意义。

我们已经看到 Bakery 算法使用了 2n 个 MRSW 变量（flag[]和 label[]）实现了 n 个线程互斥。如果用 MRMW 变量是否可以数量级地降低变量的个数呢？下面将证明这是不可能做到的：实现 n 个线程互斥至少需要 n 个 MRMW 变量。

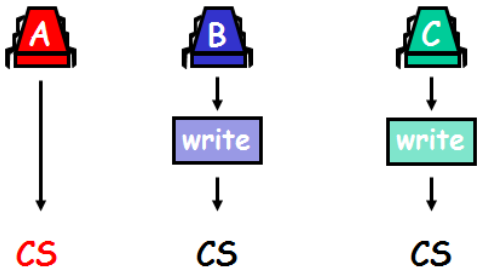
如果一个线程的下一步操作就是去写某个内存单元，我们就说这个单元被该线程所*覆盖*。如果一个系统的每一个共享内存单元都被一个线程所覆盖，我们就说这个系统处于*被覆盖状态*。

4.1 **定理** 任何通过读、写共享内存单元实现 n 个线程无死锁互斥的算法至少需要使用 n 个内存单元。

证明：用反证法。假设有一个算法使用 n-1 个内存单元实现了 n 个线程无死锁互斥。

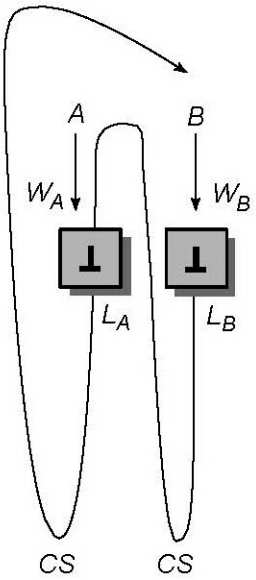
首先注意到，任何一个线程在进入临界区前都必须写至少一个共享内存单元。否则其它线程无法知道它在临界区，不可能实现互斥。如果只用多读单写的内存单元（如 Bakery 算法），那直接可以推出至少需要 n 个这样的内存单元才能实现无死锁互斥。

现在考虑使用多读多写内存单元。先假定我们能够使系统进入一个被覆盖状态，即系统中的 n-1 个内存单元各被一个线程覆盖，如下图所示（3 个线程，2 个内存单元）其中线程 B 和 C 分别覆盖一个内存单元：



现在让线程 A 运行。由于算法满足无死锁性，A 最终会进入临界区。A 进入临界区前，可能会在 2 个内存单元中留下信息。然后让 B 和 C 运行。他们会抹去两个内存单元中的信息。这样就不可能知道 A 已经进入临界区，从而无法保证互斥。

下面证明我们总可以让系统进入覆盖状态。以 2 个内存单元为例。



如果让线程 B 经过临界区 3 次，每次进入之前它都要写某个内存单元。由于只有 2 个内存单元，B 会写某个内存单元 2 次。将这个内存单元记为 L_B。再让 B 运行，直到它即将写 L_B。由于 B 还没有写，如果现在让 A 运行，它就会进入临界区。如果 A 在进入临界区之前只写 L_B，那么让 A 进入，让 B 运行，B 就会抹去 A 留在 L_B 中的信息。这样就不可能知道 A 已经进入临界区，从而无法保证互斥。因此 A 必须在进入临界区之前写 L_A。

现在让 A 运行，直到它即将写 L_A。这还不是覆盖状态，因为 A 可能也在 L_B 中留下了信息。再让 B 运行，抹去 A 留在 L_B 中的信息，进入、离开临界区至多 3 次，停在它即将再次写 L_B 的时刻。这时就达到了覆盖状态。

这个证明的思路可以通过归纳法推广到任意 n 的情形。

从上面的证明可以看到读、写操作存在的一个局限：一个线程写在一个内存单元中的信息在还没有被任何其它线程读取之前就可能被抹去。现代的多核架构一般都提供专门的指令以克服这个局限。