3.2 临界区互斥软件实现算法

临界区问题可以通过精巧设计的软件算法来解决。在讨论具体算法之前，我们先考虑算法的通用结构。在任何一个进程或线程中，我们在临界区的前后各加入一小段代码，临界区之前的代码段称为进入区(entry section)，临界区之后可有退出区(exit section)。其他代码被称为剩余区(remainder section)。一个进程 Pi 的通用结构如图 3.1 所示，其中进入段和退出段被框起来以突出这些代码段的重要性。

```
do {
  进入区
  临界区
  退出区
  剩余区
} while (TRUE);
```

图 3.1 典型进程 Pi 的通用结构

临界区问题的解决方案必须满足如下三项要求：

·互斥(mutual exclusion)：如果某个进程正在其临界区内执行，那么其他进程都不能在其临界区内执行。

·前进（progress）：如果当前没有进程在其临界区内执行且有进程想进入临界区，那么应该选择这些想进入临界区的进程中的一个，允许其进入临界区，且这种选择不能无限推迟。

·有限等待(bounded waiting)：一个进程提出进入临界区的请求，在该请求被允许之前，其他进程允许进入其临界区的次数应该有一个上限。

讨论临界区问题的解之前，我们进一步假定每个进程的执行速度不为 0，即都在推进。然而，我们不对 n 个进程的相对速度（relative speed）做任何假设。

1．算法一

讨论 n 个进程之间的临界区问题的解之前，我们先讨论如何协调两个进程的算法。记这两个进程为 P0 和 P1。不失一般性，用 Pi 表示其中的一个进程，则另一个进程为 Pj，其中 j = 1-i。

我们先给出第一个算法。

我们引入共享变量 turn 来协调两个进程对临界区的访问。变量 turn 可以初始化为 0 或者 1。如果 turn == i，则表示进程 Pi 可以被允许进入它的临界区执行。进程 Pi 的程序结构如图 3.2 所示。

```
do {
  while (turn != i) ;
  临界区
  turn = j ;
  剩余区
} while (TRUE);
```

图 3.2 两进程 Pi 的通用结构

写成具体形式，图 3.2 也可以表达为图 3.3。

```
进程 P0:                                    进程 P1:

  do {                                        do {
    while (turn != 0) ;                         while (turn != 1) ;
    临界区                                      临界区
    turn = 1 ;                                  turn = 0 ;
    剩余区                                      剩余区
  } while (TRUE);                             } while (TRUE);
```

图 3.3 算法一两进程表达的具体形式

这个两进程的协调算法（算法一）满足互斥条件，即保证了在任何时候只有一个进程能在它的临界区中执行。但是，它不满足前面说的推进条件（progress requirement），因为它严格要求了两个进程必须以交替的方式进入临界区执行。换句话说，不管两个进程执行的相对速度如何，算法都要求两个进程必须你一次我一次地进入临界区，而不能让某个进程连续两次进入临界区。

2．算法二

针对算法一存在的问题，我们提出算法二。算法一的问题在于它没有保持两个进程的足够的状态信息，而仅仅记住了哪个进程被允许进入临界区。作为补救手段，我们把变量 turn 替换成一对标志变量数组：

```
boolean flag[2];
```

变量数组两个单元的初值都初始化为 false。如果 flag[i]的值为 true，那么就表示 Pi 准备进入临界区了。每个进程 Pi 的程序结构如图 3.4 所示。

```
do {
  flag[i] = TRUE;
  while (flag[j] == TRUE);
  临界区
  flag[i] = FALSE ;
  剩余区
} while (TRUE);
```

图 3.4 算法二两进程 Pi 的通用结构

在这个算法中，进程 Pi 首先把 flag[i]设置为 true，表示他已经准备进入他自己的临界区。然后，Pi 检查进程 Pj 并不准备也进入临界区。如果此时 Pj 也准备进入临界区，那么 Pi 会一直等待知道 Pj 指示它不再需要在临界区中执行为止；也就是说，一直等到 flag[j]成为 false。此时，Pi 才进入临界区。当 Pi 离开临界区时，Pi 把 flag[i]设置为 false，从而允许可能正在等待的另一个进程进入它的临界区。

我们再仔细考察一下这个算法。这里，互斥要求是能够被满足的。但是，推进要求却未必能被满足。仔细考虑以下情况：假设在进程 P0 执行了 flag[0] = true 语句之后，就发生了进程切换，然后 P1

执行了 flag[1] = true，那么，从此以后，两个进程都将在它们紧随其后的 while 语句上无限循环下去了。

所以，算法二的执行效果严重依赖于两个进程执行的时间关系。

你可能会想到，是否交换一下检测对方标志 flag[j]的 while 语句和设置自身标志 flag[i]为 true 的语句的顺序。遗憾的是，那也不能解决我们的问题。经过仔细思考，你会发现，那样会造成两个进程同时进入临界区的情况，从而根本不能保证互斥要求的满足。

所以，构造一个正确的软件算法来实现两进程之间的互斥，并不是一个容易的事情。

3. Peterson 算法

幸运的是，早有学者提出了正确的软件实现的互斥算法。下面讨论的 Peterson 算法，就是一个经典的基于软件的临界区问题的解决算法。

Peterson 算法融合了上述算法一和算法二的思想，构造出一个能正确解决临界区问题的算法，并能说明它满足了互斥、前进、有限等待这三大要求。

仍然记两个进程分别为 P0 和 P1。为了方便，当使用 Pi 表示其中一个进程时，用 Pj 来表示另一个进程，即 j=1-i。

Peterson 算法需要在两个进程之间共享两个数据项：

```
int turn;
boolean flag[2];
```

变量 tum 表示哪个进程可以进入其临界区。即如果 turn == i，那么进程 Pi 允许在其临界区内执行。数组 flag 表示哪个进程想要进入其临界区。例如，如果 flag[i]为 true，即进程 Pi 想要进入其临界区。在理解了这些数据结构后，可以按图 3.5 所示来描述这一算法：

```
do {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] && turn == j);
  临界区
  flag[i] = FALSE;
  剩余区
} while (TRUE);
```

图 3.5 Peterson 算法中的进程 Pi 的结构

为了进入临界区，进程 Pi 首先设置 flag[i]的值为 true，且设置 tum 的值为 j，从而表示如果另一个进程 Pj 希望进入临界区，那么 Pj 能进入。如果两个进程同时试图进入，那么 tum 会几乎在同时设置成 i 和 j，但只有后执行一个赋值语句的结果会最终有效；虽然另一个赋值语句也会被执行，但会立即被重写。最终 tum 值决定了哪个进程能允许先进入其临界区。

现在证明 Peterson 算法是正确的，这需要证明：一、互斥要求满足；二、前进要求满足；三、有限等待要求满足。

为了证明第一点，要注意到只有当 flag[j]==false 或者 tum==i 时，进程 Pi 才进入其临界区。而且，注意到如果两个进程同时在其临界区内执行，那么 flag[0]==flag[l]== true。这两点意味着 P0 和 P1 不可能成功地同时执行它们的 while 语句，因为 tum 的值只可能为 0 或 l，而不可能同时为两个值。因

此，只有一个进程（如 Pj）能成功地执行完 while 语句，而进程 Pi 至少必须执行一个附加的语句（“tum==j”）。而且，由于只要 Pj 在其临界区内，fiag[j]==true 和 turn==j 就同时成立。结果是：互斥成立。

为了证明第二点和第三点，应注意到，只要条件 flag[j]==true 和 turn==j 成立，进程 Pi 陷入 while 循环语句，那么 Pi 就能被阻止进入临界区。如果 Pj 不准备进入临界区，那么 flag[j]==false，Pi 能进入临界区。如果 Pj 已设置 flag[j]为 true 且也在其 while 语句中执行，那么 tum==j 或 tum==i。如果 turn==i，那么 Pi 进入临界区；如果 tum==j，那么 Pj 进入临界区。然而，当 Pj 退出临界区，它会设置 flag[j]为 false，以允许 Pi 进入其临界区。如果 Pj 重新设置 flag[j]为 true，那么它也必须设置 tum 为 i。因此由于进程 Pi 执行 while 语句时并不改变变量 turn 的值，所以 Pi 会进入临界区（前进要求满足），且 Pi 最多在 Pj 进入临界区一次后就能进入（有限等待要求满足）。

所以，Peterson 算法是解决两进程之间互斥问题的一个正确的软件算法。