3.3 硬件同步指令

在前四节中描述了基于软件的临界区问题的解答。为了用软件算法来解决临界区问题，可谓费尽周章。回顾前面几节，软件解法之所以困难是因为在一个进程进入临界区之前，它需要完成两件工作：一是检查是否有别的进程正在请求进入临界区或者正在临界区，二是设置自己（想）进入临界区的标志。由于检查他人的标志和设置自己的标志无法成为原子操作，即无法在一条机器指令内完成，所以，在进程切换可能在任何两条相邻的指令之间发生的情况下，设计无错的软件临界区问题解决算法这项工作变得非常困难而且精巧。

为了更方便地解决这个问题，在一些体系结构上，硬件厂商提供了检测一个标志和设置这个标志同时完成的机器指令，从而大大简化了设计临界区问题解法的困难。

一般来说，可以说任何临界区问题都需要一个简单工具——锁。锁是一个进程之间共享的内存中的变量。通过要求临界区用锁来防护，就可以避免竞争条件，即一个进程在进入临界区之前必须得到锁，而在其退出临界区时释放锁，如图 3.7 所示。

```
do {
  请求锁
  临界区
  释放锁
  剩余区
} while (TRUE);
```

图 3.7 采用锁的临界区问题的解答

硬件特性能简化编程任务且提高系统效率。有了硬件指令的支持，我们下面将讨论更多的临界区问题的解决方案。这些方案采用了从硬件到应用程序员可见的软件 API 等一系列技术。所有这些解决方案都是基于锁为前提的。不过，我们也将看到，这些锁的设计可能非常复杂。

指令 TestAndSet 可以按图 3.8 所示定义。其主要特点是该指令能原子地执行。

```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
 }
```

图 3.8 TestAndSet 指令的定义

如果机器支持指令 TestAndSet，那么可这样实现互斥：声明一个 Boolean 变量 lock，初始化为 false。进程 Pi 的结构如图 3.9 所示。

```
do{
  while (TestAndSet(&lock))
    ; // do nothing
  // critical section
  lock = FALSE;
  // remainder section
```

```
} while (TRUE);
```

图 3.9 使用 TestAndSet 指令的互斥实现

另一种可用于支持互斥的机器指令是 swap 指令。指令 swap 操作两个数据，其定义如图 3.10 所示。与指令 TestAndSet 一样，它也是原子执行的。如果机器支持指令 swap，那么互斥可按如下方式实现：声明一个全局布尔变量 lock，初始化为 false。另外，每个进程定义一个局部 Boolean 变量 key。进程 Pi 的结构如图 3.11 所示。

```
void Swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

图 3.10 Swap 指令的定义

```
do{
  key=TRUE;
  while (key == TRUE)
    Swap(&lock, &key);
  // critical section
  lock = FALSE;
  // remainder sectiaa
}while (TRUE);
```

图 3.11 使用 Swap 指令的互斥实现

图 3.9 和图 3.11 的两个算法解决了互斥，且满足推进条件，但是并没有从理论上满足有限等待要求。下面，介绍一个使用 TestAndSet 指令的算法，如图 3.12 所示。该算法满足临界区问题的全部三个要求。先定义共用数据结构如下：

```
boolean waiting[n];
boolean lock;
```

这些数据结构均初始化为 false。

为了证明满足互斥要求，注意，只有 waiting[i]==false 或 key ==flase 时，进程 Pi 才进入临界区。只有当 TestAndSet 执行时，key 的值才变成 false。执行 TestAndSet 的第一个进程会发现 key==false；所有其他进程必须等待。只有其他进程离开其临界区时，变量 waiting[i]的值才能变成 false；每次只有一个 waiting[i]被设置为 false，以满足互斥要求。

```
do {
  waiting[i] = TRUE;
  key = TRUE;
  while (waiting[i] && key)
    key = TestAndSet(&lock);
  waiting[i] = FALSE;

  // critical section

j = (i+1)%n;
```

```
while ((j!=i)&& !waiting[j])
   j = (j + l) % n;
if(j == i)
   lock = FALSE;
else
   waiting[j] = FALSE;

  // remainder section

} while (TRUE);
```

图 3.12 使用 TestAndSet 的有限等待互斥

为了证明满足前进要求，有关互斥的论证也适用。由于进程在退出其临界区时或将 lock 设为 false，或将 waiting[j]设为 false。这两种情况都允许等待进程进入临界区以执行。

为了证明满足有限等待，当一个进程退出其临界区时，它会循环地扫描数组 waiting[i]（i+1，i+2，…，n-l，0，…，i-1），并根据这一顺序而指派第一个等待进程（waiting[j] ==true）作为下一个进入临界区的进程。因此，任何等待进入临界区的进程最多只需要等待 n-l 次。