2.3.6 互斥量

如果不需要信号量的计数能力，有时可以使用信号量的一个简化版本，称为互斥量（mutex）。互斥量仅仅适用于管理共享资源或一小段代码。由于互斥量在实现时既容易又有效，这使得互斥量在实现用户空间线程包时非常有用。

互斥量是一个可以处于两态之一的变量：解锁和加锁。这样，只需要一个二进制位表示它，不过实际上，常常使用一个整型量，0表示解锁，而其他所有的值则表示加锁。互斥量使用两个过程。当一个线程（或进程）需要访问临界区时，它调用mutex_lock。如果该互斥量当前是解锁的（即临界区可用），此调用成功，调用线程可以自由进入该临界区。

另一方面，如果该互斥量已经加锁，调用线程被阻塞，直到在临界区中的线程完成并调用mutex_unlock。如果多个线程被阻塞在该互斥量上，将随机选择一个线程并允许它获得锁。

由于互斥量非常简单，所以如果有可用的TSL或XCHG指令，就可以很容易地在用户空间中实现它们。用于用户级线程包的mutex_lock和mutex_unlock代码如图2-29所示。XCHG解法本质上是相同的。

```
mutex_lock:
    TSL REGISTER,MUTEX      | 将互斥信号量复制到寄存器，并且将互斥信号量置为1
    CMP REGISTER,#0         | 互斥信号量是0吗？
    JZE ok                  | 如果互斥信号量为0，它被解锁，所以返回
    CALL thread_yield       | 互斥信号量忙，调度另一个线程
    JMP mutex_lock          | 稍后再试
ok: RET                     | 返回调用者，进入临界区

mutex_unlock:
    MOVE MUTEX,#0           | 将mutex置为0
    RET                     | 返回调用者
```

图2-29 mutex_lock和mutex_unlock的实现

mutex_lock 的代码与图2-25中enter_region的代码很相似，但有一个关键的区别。当enter_region进入临界区失败时，它始终重复测试锁（忙等待）。实际上，由于时钟超时的作用，会调度其他进程运行。这样迟早拥有锁的进程会进入运行并释放锁。

在（用户）线程中，情形有所不同，因为没有时钟停止运行时间过长的线程。结果是通过忙等待的方式来试图获得锁的线程将永远循环下去，决不会得到锁，因为这个运行的线程不会让其他线程运行从而释放锁。

以上就是enter_region和mutex_lock 的差别所在。在后者取锁失败时，它调用thread_yield将CPU放弃给另一个线程。这样，就没有忙等待。在该线程下次运行时，它再一次对锁进行测试。

由于thread_yield只是在用户空间中对线程调度程序的一个调用，所以它的运行非常快捷。这样，mutex_lock和mutex_unlock都不需要任何内核调用。通过使用这些过程，用户线程完全可以实现在用户空间中的同步，这些过程仅仅需要少量的指令。

上面所叙述的互斥量系统是一套调用框架。对于软件来说，总是需要更多的特性，而同步原语也不例外。例如，有时线程包提供一个调用mutex_trylock，这个调用或者获得锁或者返回失败码，但并不阻塞线程。这就给了调用线程一个灵活性，用以决定下一步做什么，是使用替代办法还只是等待下去。

到目前为止，我们掩盖了一个问题，不过现在还是有必要把这个问题提出来。在用户级线程包中，多个线程访问同一个互斥量是没有问题的，因为所有的线程都在一个公共地址空间中操作。但是，对于大多数早期解决方案，诸如Peterson算法和信号量等，都有一个未说明的前提，即这些多个进程至少应该访问一些共享内存，也许仅仅是一个字。如果进程有不连续的地址空间，如我们始终提到的，那么在Peterson算法、信号量或公共缓冲区中，它们如何共享turn变量呢？

有两种方案。第一种，有些共享数据结构，如信号量，可以存放在内核中，并且只能通过系统调用来访问。这种处理方式化解了上述问题。第二种，多数现代操作系统（包括UNIX和Windows）提供一种方法，让进程与其他进程共享其部分地址空间。在这种方法中，缓冲区和其他数据结构可以共享。在最坏的情形下，如果没有可共享的途径，则可以使用共享文件。