操作系统实验指导书

2013 年 3 月 21 日

实验一　线程同步

1.1　实验简介

本实验讨论临界区问题及其解决方案。实验首先创建两个共享数据资源的并发线程。在没有同步控制机制的情况下，我们将看到某些异常现象。针对观察到的现象，本实验采用两套解决方案：

- 利用 Windows 的 mutex 机制
- 采用软件方案

然后比较这两种方案的性能优劣。

1.2　制造混乱

Windows 操作系统支持抢先式调度，这意味着一线程运行一段时间后，操作系统会暂停其运行并启动另一线程。也就是说，进程内的所有线程会以不可预知的步调并发执行。

为了制造混乱，我们首先创建两个线程 t1 和 t2。父线程（主线程）定义两个全局变量，比如 *accnt1* 和 *accnt2*。每个变量表示一个银行账户，其值表示该账户的存款余额，初始值为 0。线程模拟在两个账户之间进行转账的交易。也即，每个线程首先读取两个账户的余额，然后产生一个随机数 r，在其中一个账户上减去该数，在另一个账户上加上该数。线程操作的代码框架如下：

```
1  counter=0;
2  do {
3    tmp1 = accnt1;
4    tmp2 = accnt2;
```

```
5     r = rand();
6     accnt1 = tmp1 + r;
7     accnt2 = tmp2 - r;
8     counter++;
9   } while ( accnt1 + accnt2 == 0 );
10  print(counter);
```

两个线程执行相同的代码。只要它们的执行过程不相互交叉，那么两个账户的余额之和将永远是 0。但如果发生了交叉，那么某线程就有可能读到新的 *accnt1* 值和老的 *accnt2* 值，从而导致账户余额数据发生混乱。线程一旦检测到混乱的发生，便终止循环并打印交易的次数 (*counter*)。

请编写出完整的程序代码并运行，然后观察产生混乱需要的时间长短。因为这是我们编写的第一个程序，因此这里我给出了完整的代码，请参考。有能力的同学在参考下面的代码之前，请先自己尝试一下。

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <windows.h>
4
5   int  accnt1 = 0;
6   int  accnt2 = 0;
7
8   DWORD WINAPI run( LPVOID p) {
9     int  counter=0;
10    int  tmp1, tmp2, r;
11
12    do {
13      tmp1 = accnt1;
14      tmp2 = accnt2;
15      r = rand();
16      accnt1 = tmp1 + r;
17      accnt2 = tmp2 - r;
18      counter++;
19    } while ( accnt1 + accnt2 == 0 );
20    printf("%d\n", counter);
21  }
22
23  int  main(int argc, char *argv[])
24  {
```

```
25    CreateThread(NULL,
26          0,
27          run,
28          NULL,
29          0,
30          NULL);
31
32    CreateThread(NULL,
33          0,
34          run,
35          NULL,
36          0,
37          NULL);
38
39    system("PAUSE");
40    return 0;
41  }
```

反复运行该程序。请问，观察到了什么？你能解释这些现象吗?

1.3 临界区问题之解决方案

上面例子中，线程执行的代码叫做临界区，因为两个线程在这里访问了同样的数据，在没有保护的情况下，有可能发生混乱。解决该问题有两套方案。其一，如果操作系统提供了同步原语，例如 mutex，那么就可直接利用该原语对临界区进行排它性的存取保护。其二，如果操作系统不提供这样的原语，那么可用软件方案加以解决。本实验中，我们将实现并比较这两种方案。

1.3.1 mutex 方案

Windows 操作系统提供了 mutex 对象。mutex 状态可以是 signaled (unlocked) 或者是 nonsignaled (locked)。

利用 mutex 对象，可以方便地实现临界区保护。进入临界区时（在第一个读操作之前），锁住 mutex 对象；离开临界区时（在第二个写操作之后），打开 mutex 对象。线程的阻塞与唤醒由系统管理，程序员无需干预。

以下给出的是在 Windows 操作系统下有关 mutex 对象操作的提示。

创建一个未上锁 mutex 对象的代码如下：

```
HANDLE hMutex = CreateMutex(NULL,
    FALSE,
    NULL);
```

给 mutex 对象上锁的代码如下：

```
WaitForSingleObject( hMutex, INFINITE );
```

打开 mutex 对象的代码如下：

```
ReleaseMutex( hMutex );
```

根据以上提示，编写出用 mutex 对象保护临界区的解决方案。完成后，请思考以下问题：假设把加锁和开锁操作分别放置在第一个写操作之前和第二个写操作之后，能否实现临界区的保护，为什么?

1.3.2 软件方案

现在假设操作系统没有提供同步原语。这时，我们只能通过编程语言对变量的操作实现临界区保护。下面给出的是一个概念性的解决方案框架：

```
/* CS Algorithm: Peterson Solution */
int c1 = 0, c2 = 0, will_wait;
cobegin
p1: while (1) {
        c1 = 1;
        will_wait = 1;
        while( c2 && (will_wait==1) ); /*wait loop*/
        CS1;
        c1 = 0;
        program1;
    }
p2: while (1) {
        c2 = 1;
        will_wait = 2;
        while( c1 && (will_wait==2) ); /*wait loop*/
        CS2;
        c2 = 0;
        program2;
    }
```

上面的方案使用了三个变量 $c1, c2$ 和 $will\_wait$。线程 i 试图进入临界区时首先把变量 c_i 置为 1，接着把变量 $will\_wait$ 的值设置为 i（为什么?）阻塞通过临界区之前的循环实现。当线程退出临界区时，又把变量 c_i 的值设置为 0。

在我们的例子中，临界区始于第一个读操作，结束于第二个写操作。请把上面的概念框架转换为可运行的 C 代码，实现临界区的保护。为了加快程序的执行速度，可在阻塞循环中增加一个 Sleep(0) 语句。这可以让不能进入临界区的线程马上放弃处理器，而不用无谓消耗处理器资源。

最后，请比较 mutex 方案和软件方案的效率（执行相同的循环次数，计算消耗的时间。为了让结果更科学，请多次试验，然后计算平均值）。

提示：时间度量可以用

```
1 DWORD GetTickCount(VOID)
```

在操作开始之前调用一次，操作结束之后再调用一次，两次调用所得到的返回值的差便是该操作所消耗的大致时间（单位毫秒）。

1.4 参考文献

1. Peterson's Algorithm

2. The Little Book of Semaphores

3. Shared Memory Consistency Models: A Tutorial

4. MSDN Library