Chapter28

> ***28-1. 首先用标志 flag.s运行 x86.py。该代码通过一个内存标志“实现”锁。你能理解汇编代码试图做什么吗?***

查看 `flag.s` 汇编代码如下:

ostep-homework > threads-locks > flag.s

```
.var flag
.var count

.main
.top

.acquire
mov  flag, %ax      #
test $0, %ax        # flag = 0,  代表锁是空闲的
jne  .acquire       # 当前锁被其他线程占用，循环等待
mov  $1, flag       # 获得锁，flag置为1

# critical section
mov  count, %ax     #
add  $1, %ax        #
mov  %ax, count     # count++
# release lock
mov  $0, flag       # 释放锁

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top            # 如果bx的值大于0则循环回到.top处执行

halt
```

这段汇编代码实现了一个简单的***线程同步机制***，主要是为了***保护对共享变量 `count` 的访问***。它使用了 `flag` 作为一个简单的锁（自旋锁）：

1. **初始化**: 假设 `.var flag` 和 `.var count` 分别初始化为0和某个初始值。
2. **获取锁**:
   - `mov flag, %ax`: 将 `flag` 的值加载到寄存器 `%ax` 中。
   - `test $0, %ax`: 检查 `%ax` （即 `flag` 的值）是否为0。
   - `jne .acquire`: 如果 `flag` 不为0（意味着锁被其他线程持有），则跳转到 `.acquire` 标签处，继续检查直到 `flag` 变为0。
   - `mov $1, flag`: 当成功进入此部分，说明 `flag` 为0，将 `flag` 设置为1，表示获取到了锁。
3. **临界区**:
   - `mov count, %ax`: 将 `count` 的值加载到寄存器 `%ax` 中。
   - `add $1, %ax`: 在 `%ax` （即当前的 `count` 值）上加1。
   - `mov %ax, count`: 将增加后的值存回 `count`，完成对 `count` 的原子性递增操作。
4. **释放锁**:
   - `mov $0, flag`: 将 `flag` 设置回0，表示锁已被释放，其他等待的线程可以尝试获取锁。
5. **循环控制**:
   - 假设在开始执行这段代码之前，`%bx` 已经被初始化为一个循环次数。
   - `sub $1, %bx`: 每次循环结束时，`%bx` 减1，表示循环计数递减。
   - `test $0, %bx`: 检查 `%bx` 是否为0。
   - `jgt .top`: 如果 `%bx` 大于0，跳转回 `.top` 标签，重新开始整个过程（包括尝试获取锁、执行临界区操作等）。否则，执行到 `halt` 指令，程序停止。

但是这个锁有问题，并不是互斥的。用下图解释:

```
       Thread 0                Thread 1
 mov  flag, %ax # flag 此时为0
 test $0, %ax   # %ax 此时为0，只要thread 0 继续执行，它就必能获得锁
----- Halt;Switch -----  ----- Halt;Switch -----
                           mov  flag, %ax
                           test $0, %ax    # %ax 也是0，线程拥有独立的寄存器
                           jne  .acquire
                           mov  $1, flag   # thread 1 获得了锁

                           mov  count, %ax
                           add  $1, %ax
                           mov  %ax, count
                           ...             # thread 1 没有释放锁
----- Halt;Switch -----  ----- Halt;Switch -----
 jne  .acquire
 mov  $1, flag  # thread 0 获得了锁，此时两个线程都获得了锁
```

> ***28-2. 使用默认值运行时, flag.s是否按预期工作?它会产生正确的结果吗?使用-M 和-R 标志跟踪变量和寄存器(并使用-c 查看它们的值)。你能预测代码运行时 flag 最终会变成什么值吗?***

```
dinghaitong@dinghaitong-vm:~/OperatingSystem/HwAbout/ostep-homework/threads-intro$ ./x86.py -p flag.s -R ax,bx -M flag,count -c
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace flag,count
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False

 flag count      ax    bx          Thread 0                Thread 1
    0     0       0     0
    0     0       0     0   1000 mov  flag, %ax
    0     0       0     0   1001 test $0, %ax
    0     0       0     0   1002 jne  .acquire
    1     0       0     0   1003 mov  $1, flag
    1     0       0     0   1004 mov  count, %ax
    1     0       1     0   1005 add  $1, %ax
    1     1       1     0   1006 mov  %ax, count
    0     1       1     0   1007 mov  $0, flag
    0     1       1    -1   1008 sub  $1, %bx
    0     1       1    -1   1009 test $0, %bx
    0     1       1    -1   1010 jgt .top
    0     1       1    -1   1011 halt
    0     1       0     0   ----- Halt;Switch -----  ----- Halt;Switch -----
    0     1       0     0                           1000 mov  flag, %ax
    0     1       0     0                           1001 test $0, %ax
    0     1       0     0                           1002 jne  .acquire
    1     1       0     0                           1003 mov  $1, flag
    1     1       1     0                           1004 mov  count, %ax
    1     1       2     0                           1005 add  $1, %ax
    1     2       2     0                           1006 mov  %ax, count
    0     2       2     0                           1007 mov  $0, flag
    0     2       2    -1                           1008 sub  $1, %bx
    0     2       2    -1                           1009 test $0, %bx
    0     2       2    -1                           1010 jgt .top
    0     2       2    -1                           1011 halt
```

根据运行结果我们可知： `flags.s` 能按照预期工作，并能取得正确的结果。
这是因为默认中断间隔是50条指令，所以期间线程并没有切换，而是按顺序执行，所以不会产生锁冲突。
最后 `flag` 会被置为 0