

HW8

姓名: 陈锐林, 学号:21307130148

2023 年 10 月 29 日

Chapter 28

Question1

(1) 我理解了这段汇编。(2) 这里的锁 (acquire 部分), 是通过循环不断对一个变量判 0, 之后再下一步。

Question2

(1) 按默认设置运行 flag.s 是没问题的; 可以看到默认设置下, 一共 2 个线程, 并且线程中断达 50, 是远大于代码数的。(2) 鉴于前面的信息, 能看出两个线程会正常运行, 最后 flag 会正常被清 0; 在利用-M,-R,-c 后可验证属实。

Question3

(1) 如果在前面的命令加上"-a bx=2,bx=2" 对寄存器进行设置, flag 仍然是 0; 因为设置 bx=2, 相当于让整个大循环运行两次。(2) 如果想让 flag 最后不为 1, 应该是利用中断。比如下面这张图中, 右侧线程 2 本来 flag 为 1(-a bx=5,bx=4), 但是中断后就变为 0 了。所以如果要在线程 1 达到这个目的, 应该是在线程 1 最后一个循环对 flag 清零后又跳到线程 2, 置 1 后再回来; 但这后续是无法达到的, 因为默认设置中 thread interrupt 有 50, 线程 1 结束时 flag 为 1, 但是很快就会切换到线程 2, 继续操作, 最后 flag 总会被清零的。

1	4	1	4	1001 test \$0, %ax
1	4	1	4	1002 jne .acquire
1	4	1	4	1000 mov flag, %ax
1	4	1	4	1001 test \$0, %ax
1	4	5	1	----- Interrupt ----- Interrupt ----
1	5	5	1	1006 mov %ax, count
0	5	5	1	1007 mov \$0, flag
0	5	5	0	1008 sub \$1, %bx
0	5	5	0	1009 test \$0, %bx
0	5	5	0	1010 jgt .top
0	5	5	0	1011 halt
0	5	1	4	----- Halt;Switch ----- Halt;Switch ---
0	5	1	4	1002 jne .acquire
0	5	0	4	1000 mov flag, %ax
0	5	0	4	1001 test \$0, %ax
0	5	0	4	1002 jne .acquire
1	5	0	4	1003 mov \$1, flag

Question4

(1) 这题所谓的坏结果和好结果应该是针对 count 和 ax 的，因为第三题中我们已经知道 flag 固定为 0 结束。如这里让命令为 `-a bx=10,bx=10`；最后的 count 和 ax 应该都是 20；但是事实上根据中断频率的不同会得到不同结果。尝试了 1-15 的中断频率后，我发现只有 11 和 15 能得到好结果，其他都是坏结果。(2) 尝试解释下原因：原因在 `mov count,%ax / add $1,%ax / mov %ax,count`；这三句中间可能发生中断，最后 count 得到的 ax 值可能和之前相比没有增长。而中断频率等于 11，对应着完成一次大循环的代码数，所以不会有这个问题。而 15 也是正好避开了，如果再扩大 bx 可能会出问题。

Question5

(1) 上锁是用的 `xchg %ax, mutex`；这是一个原子操作，将 ax 和 mutex 中的值进行对换，并返回 mutex 的值。之后如果 ax 是 0(mutex 原本是 0)，就进行下一步；否则继续循环。(2) 开锁就是将变量 0 给到 mutex。

Question6

(1) 仍然取 `bx=10,bx=10`；考虑-i 从 1-15；可以看到最后得到的结果 mutex 都是 0，是正确的。(2) 而关于 CPU 的浪费问题，发现中断频率较小时总的指令数很大，比如取 4 时是 306；而取 11 时就只有 222。(该数据通过添加-S 标志得到)。应该是因为锁的问题导致的循环步数增加，在中断频率低的时候就比较容易遇到。

Question7

采用命令如：`python3 ./x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -a bx=10,bx=10 -P 0011`，就可以模拟题目中的情况。这段命令即 thread0 和 thread1 每两条交替运行；能观察到在 0 中的锁在 1 中被 acquire；仅从 mutex/ax/count 的值来看是正确的。但是还应该还要 CPU 的效率问题。

Question8

变量 turn 的存在保证了两个线程不会陷入竞争获得一个锁的死循环。这种情况下即使两个线程 flag 都是 1 了，也不会产生矛盾。

Question9

首先采取不同的-i 并不会导致最后结果的不同 (count flag cx ax 保持一致)。唯一不同的可能是 CPU 的效率，发现从 1-15，用指令最多的反而是-5 和-6，达到 50 多条。

Question10

通过改变不同的执行序列，我们能观察结果是否正确。而在上面已经采用了不同的-i，所以这里选择用一些数量不均衡或者顺序不对称的-P 即可。比如-P 010；-P 00001；-P 11011；-P 1001。在试验过程中，观察在线程切换前后是否有不正常行为发生；最后可以看到，只有用的指令数的差别，而没有逻辑上的错误。

Question11

(1)ticket.s 是一个 ticket 锁的定义和使用，和前文中的内容是对应的。(2) 这里使用命令“`python3 ./x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000,bx=1000 -c`”，能看到最后得到的结果是正确的，count,ticket,turn=2000。如下图：

Chapter 30

Question1

查看后知道，这是通过单一条件变量来解决生产者/消费者问题。

Question2

这里我用了下面 5 条指令，这些是题目中参数的不同组合：

```
./main-two-cvs-while -p 1 -c 1 -m 1 -v;  
./main-two-cvs-while -p 1 -c 1 -m 10 -v;  
./main-two-cvs-while -p 1 -c 1 -m 10 -l 100 -v;  
./main-two-cvs-while -p 1 -c 1 -m 1 -C 0,0,0,0,0,1 -v;  
./main-two-cvs-while -p 1 -c 1 -m 10 -l 10 -C 0,0,0,0,0,1 -v;
```

(1) 题目问代码的行为有没有随着缓存区变大而改变，有点没 get 到在问什么，应该没有变化。(2) 关于 NF 的变化，从第一条指令到第二条指令就能知道：只提升缓冲区大小是没法改变的，NF 都是 0 或者 1；而从第二条指令到第三条指令又能看出，只改变生产者数量也不能改变 NF；但是如果对比第四条和第五条指令就能知道，只要增加了睡眠时间，那么改变生产者和消费者数量就能改变 NF 分布；这里第五条指令下 NF 会从 0 变化到 10。

Question3

`./main-two-cvs-while -p 1 -c 1 -m 10 -l 10 -C 0,0,0,0,0,1 -v` 在 windows 和 linux 上，得到结果是不一样的，如下：

```
Consumer consumption:
```

```
C0 -> 10  
C1 -> 0  
C2 -> 0
```

```
Total time: 12.38 seconds
```

```
PS C:\Users\10848\Desktop\NEWHW\ostep-homework\threads-cv>
```

```
Consumer consumption:
```

```
C0 -> 0  
C1 -> 10  
C2 -> 0
```

```
Total time: 12.02 seconds
```

```
chenr1959@SK-20210701MSSI:~/NEWHW/ostep-homework/threads-cv$
```

Question4

根据题设，每个消费者会在获得锁之后睡眠 1s，这时其他线程没法进行，所以在忽略其他时间计算下，总的时间就是 c3 的个数，即 12s。从具体的行为来看是这样的：c1 消费第一个，无需等待；c0 醒了之后，没法消费，等了 1s 后陷入沉睡；c2 和 c0 类似；最

后 c0 和 c1 共等 9s，而最后三者各等 1s。

Question5

改变-m 的值为 3，会导致总的 c3 步数为 11，最后用时 11s。可以看到，随着容量的增大，生产者不必等待 EOS，这种情况发生了 1 次，节省了 1s。

Question6

这题的命令和 5 相比只有睡眠时间点的差距。最后用时 5s。是可以看到 c6 数量也是 12，但是最后的时间不应该是 12s，因为在解锁后睡眠，生产者可以继续进行，这样的情况占了 7 次。

Question7

还是用时 5s，但是 c6 数量是 13，但是仍能利用睡眠时间减去 8s。

Question8

不可能，因为这里只有 1 个生产者和 1 个消费者。

Question9

可以的，采用以下命令：`./main-one-cv-while -c 2 -v -P 0,0,0,0,0,1`。如下，会看到 c0 进入等待后，一直在等待；最后出不来了。

```
NF      P0 C0 C1
0 [*--- ] p0
0 [*--- ] c0
0 [*--- ] c0
0 [*--- ] p1
1 [* 0 ] p4
1 [* 0 ] p5
1 [* 0 ] p6
1 [* 0 ] c1
0 [*--- ] c4
0 [*--- ] c5
0 [*--- ] c6
0 [*--- ] c1
0 [*--- ] c0
0 [*--- ] c2
0 [*--- ] c1
0 [*--- ] c2
1 [*EOS ] [main: added end-of-stream marker]
1 [*EOS ] c3
0 [*--- ] c4
0 [*--- ] c5
0 [*--- ] c6
0 [*--- ] c3
0 [*--- ] c2
```

Question10

1 个消费者始终是正确的，两个之后就可能发生情况。c3 准备，但是没有数据可以启动。

Question11

问题应该会出现锁的提早释放导致同时进行 do_fill 和 do_get 操作。但是很可惜我这里没有找到能验证这个的命令。但还有其他问题，比如消费不均，这个命令很好找，如`./main-two-cvs-while-extra-unlock -p 1 -c 2 -m 10 -l 21 -v -C 0,0,0,0,1,0,0,0,0,0,0,0,0,0`即可。