

# 作业2

## Chp26

### T1

#### (1)

`loop.s` 程序，每次将 `%dx` 的值减一，只要 `%dx` 的值大于等于0，就跳转到 `.top` 的位置。如果 `%dx` 的值小于0了，就跳出循环。

#### (2)

`%dx` 寄存器的值如下所示：



The screenshot shows a debugger window with a table of register values. The register `dx` is highlighted in blue. The table has two columns: the register name and the thread ID. The values for `dx` are 0, -1, -1, -1, and -1. The thread ID is 0. The instructions are `sub $1,%dx`, `test $0,%dx`, `jgte .top`, and `halt`.

dx	Thread 0
0	
-1	1000 sub \$1,%dx
-1	1001 test \$0,%dx
-1	1002 jgte .top
-1	1003 halt

### T2

#### (1)

`%dx` 寄存器的值如下所示：

dx	Thread 0	Thread 1
3		
2	1000 sub \$1,%dx	
2	1001 test \$0,%dx	
2	1002 jgte .top	
1	1000 sub \$1,%dx	
1	1001 test \$0,%dx	
1	1002 jgte .top	
0	1000 sub \$1,%dx	
0	1001 test \$0,%dx	
0	1002 jgte .top	
-1	1000 sub \$1,%dx	
-1	1001 test \$0,%dx	
-1	1002 jgte .top	
-1	1003 halt	
3	----- Halt;Switch -----	----- Halt;Switch -----
2		1000 sub \$1,%dx
2		1001 test \$0,%dx
2		1002 jgte .top
1		1000 sub \$1,%dx
1		1001 test \$0,%dx
1		1002 jgte .top
0		1000 sub \$1,%dx
0		1001 test \$0,%dx
0		1002 jgte .top
-1		1000 sub \$1,%dx
-1		1001 test \$0,%dx
-1		1002 jgte .top
-1		1003 halt

## (2)

多个线程不会影响计算，因为运行的指令总数远小于中断间隔

## (3)

这段代码没有竞态条件。因为并没有出现两个线程同时进入临界区的情况。

## T3

```
(mpdd)
~/Desktop/Operating System/Homework/ostep-homework-master/threads-intro
$ ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx -s 1 -c
ARG seed 1
ARG numthreads 2
ARG program loop.s
ARG interrupt frequency 3
ARG interrupt randomness True
ARG argv dx=3,dx=3
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace dx
ARG cetrace False
ARG printstats False
ARG verbose False

dx      Thread 0      Thread 1
3
2 1000 sub $1,%dx
3 ----- Interrupt -----
2 1000 sub $1,%dx
2 1001 test $0,%dx
2 1002 jgte .top
2 ----- Interrupt -----
2 1001 test $0,%dx
2 1002 jgte .top
1 1000 sub $1,%dx
2 ----- Interrupt -----
1 1000 sub $1,%dx
1 ----- Interrupt -----
1 1001 test $0,%dx
1 1002 jgte .top
1 ----- Interrupt -----
1 1001 test $0,%dx
1 1002 jgte .top
1 ----- Interrupt -----
0 1000 sub $1,%dx
0 1001 test $0,%dx
1 ----- Interrupt -----
0 1000 sub $1,%dx
0 1001 test $0,%dx
0 1002 jgte .top
0 ----- Interrupt -----
0 1002 jgte .top
0 ----- Interrupt -----
-1 1000 sub $1,%dx
-1 1001 test $0,%dx
-1 1002 jgte .top
-1 ----- Interrupt -----
-1 1001 test $0,%dx
-1 1002 jgte .top
-1 ----- Interrupt -----
-1 1003 halt
-1 ----- Halt;Switch -----
-1 1003 halt

(mppd)
~/Desktop/Operating System/Homework/ostep-homework-master/threads-intro
$
```

中断频率不会改变这个程序的行为，原因如下：

因为两个线程的寄存器是独立的，不存在两个线程访问同一寄存器的情况。因此这段程序不存在临界区，也没有竞态条件。所以中断频率不会改变这个程序的行为。

## T4

变量x的值如下所示，假设内存中数据的初始值是0：

```
2000      Thread 0
0
0 1000 mov 2000, %ax
0 1001 add $1, %ax
1 1002 mov %ax, 2000
1 1003 sub $1, %bx
1 1004 test $0, %bx
1 1005 jgt .top
1 1006 halt
```

## Chp 28

## T1

- 这段汇编代码试图把变量 `flag` 作为锁，`flag=0` 表示锁未被占有，`flag=1` 表示锁已经被占有。临界区的内容是将变量 `count` 加1。最外层有一个循环，`%bx` 是循环变量。
- 但是这段代码对锁的实现是有问题的，因为上锁的过程不是原子的。如果多个线程尝试同时访问变量 `flag`，可能会导致竞态条件。

## T2

### (1)

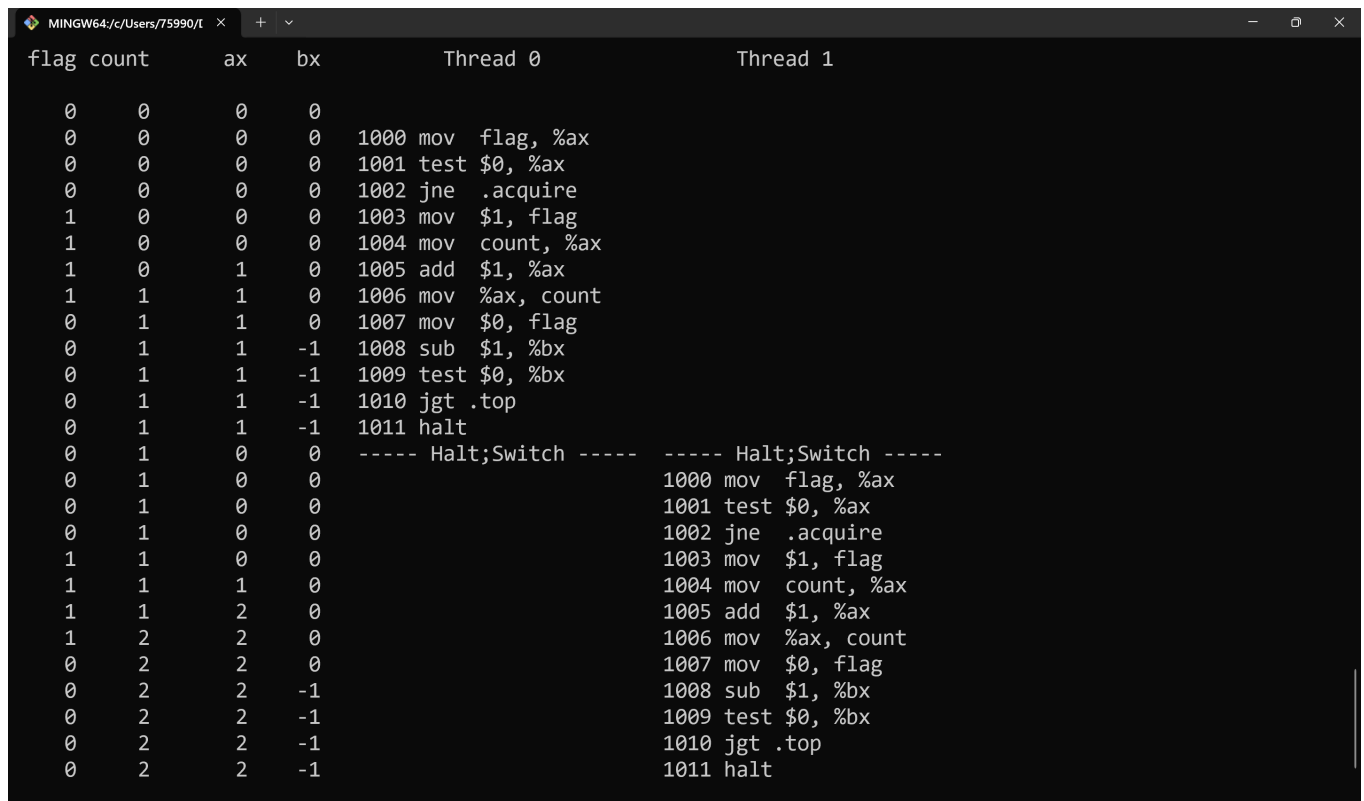
使用默认值运行时，`flag.s` 会按预期工作，并且能产生预期的结果。

### (2)

代码运行后标志最终的值是：

```
flag=0
count=2
```

验证如下：



```
MINGW64/c/Users/75990/I x + v
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 -----
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
```

## T3

### (1)

```
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=2,bx=2
```

代码前后分别运行了两个独立的、一模一样的线程。

每个线程执行了两次循环：首先获取锁 `flag`，然后将变量 `count` 加1，然后释放锁。

## (2)

在默认情况下，中断间隔为50，线程运行没有发生中断，两个线程先后运行，不会产生错误。

代码运行后标志最终的值是：

```
flag=0  
count=4
```

## T4

尝试较大的 `%bx` 和不同的 `-i` 的值如下：

%bx	中断间隔	count
1000	1	1000
1000	2	1000
1000	3	1333
1000	4	1667
1000	5	1750
1000	6	1833
1000	7	1666
1000	8	1600
1000	9	1888
1000	10	1925
1000	20	1779
1000	30	1802
1000	40	1776
1000	50	1985
1000	60	1904
1000	70	1746
1000	80	1800

%bx	中断间隔	count
1000	90	1802
1000	100	1810

- 可以发现当 `-i` 的值较小 (`1~3`) 的时候, `count` 最终的值得明显小于其他的情况, 导致不好的结果。
- 当 `-i` 的值为其他情况时, `count` 最终的值得虽然不同, 但总体分布在 `1600~2000` 之间。

## T5

**获取锁:** 首先将 `%ax` 的值设置为1, 然后原子地交换 `%ax` 和 `mutex` 的值, 再检查 `%ax` 的值 (此时 `%ax` 的值为 `mutex` 原来的值), 如果 `%ax` 的值为0, 则成功获取锁。

代码如下:

```
mov $1, %ax
xchg %ax, mutex
test $0, %ax
jne .acquire
```

**释放锁:** 将 `mutex` 的值设置为0即可

代码如下:

```
mov $0, mutex
```

## T6

### (1)

执行结果如下, 发现代码可以按预期工作:

%bx	中断间隔	count
1000	1	2000
1000	2	2000
1000	3	2000
1000	4	2000
1000	5	2000
1000	6	2000
1000	7	2000

%bx	中断间隔	count
1000	8	2000
1000	9	2000
1000	10	2000
1000	20	2000
1000	30	2000
1000	40	2000
1000	50	2000
1000	60	2000
1000	70	2000
1000	80	2000
1000	90	2000
1000	100	2000

## (2)

有时CPU利用率不高：

主要原因是自旋等待锁被释放的过程也占用了CPU。

量化方法：

1. **自旋占比**：统计进程在 `.acquire` 循环中的指令执行次数与总指令数的比例。比例越高，CPU浪费越多。
2. **吞吐量**：测量单位时间内 `count` 的增量次数。增量越少，说明锁竞争越激烈，效率越低。
3. **中断频率影响**：通过调整 `-i` 值，观察吞吐量和自旋占比的变化。若增大 `-i` 导致吞吐量下降，则表明长中断间隔降低了效率。

## T7

测试1：

```
$ ./x86.py -p test-and-set.s -P 0000011111
```

结果正确：

Thread 0	Thread 1
1000 mov \$1, %ax	
1001 xchg %ax, mutex	
1002 test \$0, %ax	
1003 jne .acquire	
1004 mov count, %ax	
----- Interrupt -----	----- Interrupt -----
	1000 mov \$1, %ax
	1001 xchg %ax, mutex
	1002 test \$0, %ax
	1003 jne .acquire
	1000 mov \$1, %ax
----- Interrupt -----	----- Interrupt -----

测试2：第二个线程先获取锁，第一个线程再获取锁

```
$ ./x86.py -p test-and-set.s -P 1111100000
```

结果正确：



Thread 0

Thread 1

```

----- Interrupt -----
1000 mov  $1, %ax
1001 xchg %ax, mutex
1002 test $0, %ax
1003 jne  .acquire
1000 mov  $1, %ax
----- Interrupt -----

1000 mov  $1, %ax
1001 xchg %ax, mutex
1002 test $0, %ax
1003 jne  .acquire
1004 mov  count, %ax
----- Interrupt -----
```

## Chp 30

### T1

#### 运行时行为

##### 1. 生产者填充流程:

- 缓冲区未滿时，生产者持续填充数据；若滿则阻塞，等待消费者取出数据后唤醒。
- 填充操作通过 `num_full` 跟踪缓冲区占用状态，确保线程安全。

##### 2. 消费者取出流程:

- 缓冲区非空时，消费者持续取出数据；若空则阻塞，等待生产者填充后唤醒。
- 遇到 `END_OF_STREAM` 时停止消费，返回有效消费次数。

##### 3. 同步与竞态避免:

- 使用 `while` 检查条件变量，避免虚假唤醒（如系统信号或意外唤醒）。
- 互斥锁确保对共享资源的原子操作，条件变量优化线程等待效率。

#### 预期结果:

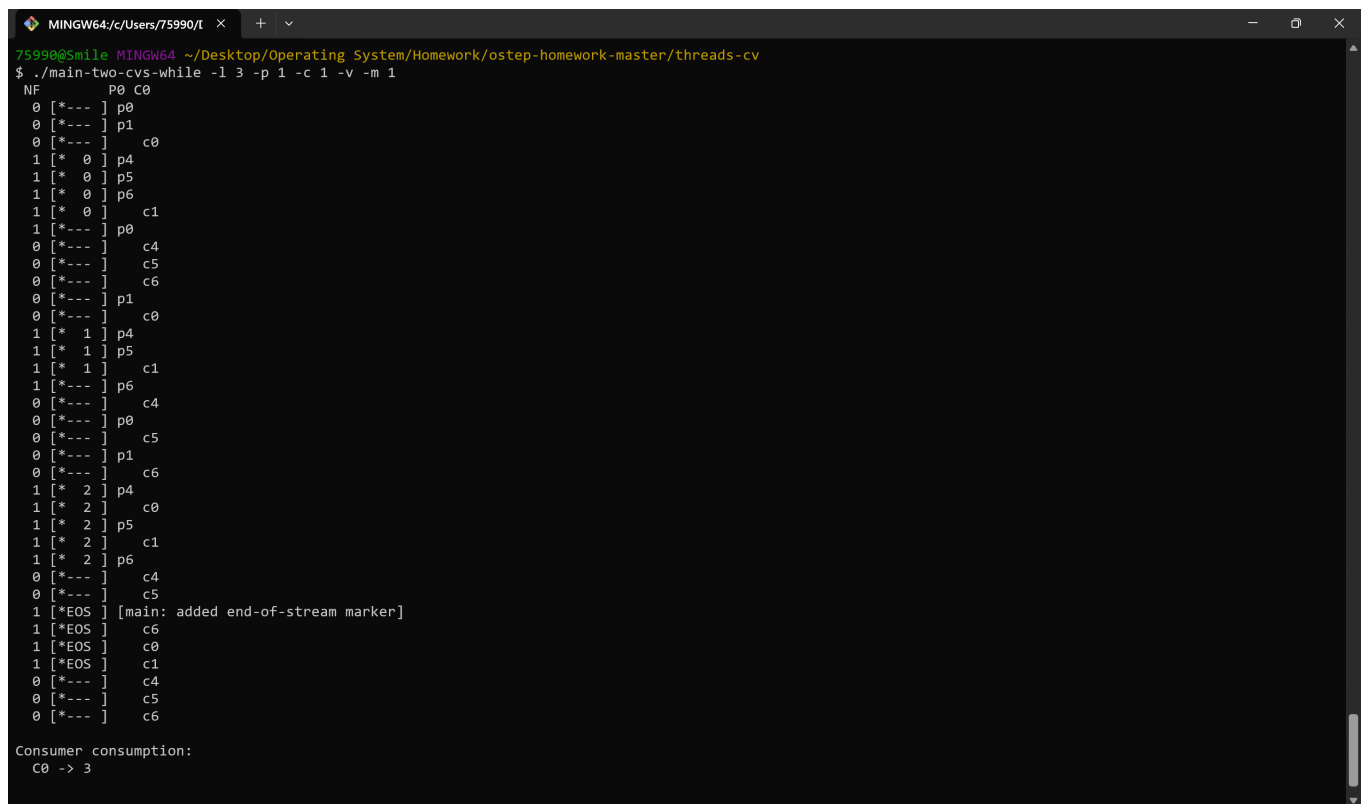
生产者生成的所有数据（除结束标记）均被消费者正确消费，无数据丢失或重复。

## T2

### (1)

运行以下命令：

```
./main-two-cvs-while -l 3 -m 1 -p 1 -c 1 -v
```



```
MINGW64/c/Users/75990/l x + v
75990@Smile MINGW64 ~/Desktop/Operating System/Homework/ostep-homework-master/threads-cv
$ ./main-two-cvs-while -l 3 -p 1 -c 1 -v -m 1
NF          P0 C0
0 [*--- ] p0
0 [*--- ] p1
0 [*--- ] c0
1 [* 0 ] p4
1 [* 0 ] p5
1 [* 0 ] p6
1 [* 0 ] c1
1 [*--- ] p0
0 [*--- ] c4
0 [*--- ] c5
0 [*--- ] c6
0 [*--- ] p1
0 [*--- ] c0
1 [* 1 ] p4
1 [* 1 ] p5
1 [* 1 ] c1
1 [*--- ] p6
0 [*--- ] c4
0 [*--- ] p0
0 [*--- ] c5
0 [*--- ] p1
0 [*--- ] c6
1 [* 2 ] p4
1 [* 2 ] c0
1 [* 2 ] p5
1 [* 2 ] c1
1 [* 2 ] p6
0 [*--- ] c4
0 [*--- ] c5
1 [*EOS ] [main: added end-of-stream marker]
1 [*EOS ] c6
1 [*EOS ] c0
1 [*EOS ] c1
0 [*--- ] c4
0 [*--- ] c5
0 [*--- ] c6
Consumer consumption:
C0 -> 3
```

```
./main-two-cvs-while -l 3 -m 3 -p 1 -c 1 -v
```

```
MINGW64/c/Users/75990/t x + v
75990@Smile MINGW64 ~/Desktop/Operating System/Homework/ostep-homework-master/threads-cv
$ ./main-two-cvs-while -l 3 -p 1 -c 1 -v -m 3
NF      p0 c0
0 [*--- --- ] p0
0 [*--- --- ] p1
1 [u 0 f--- ] p4
1 [u 0 f--- ] p5
1 [u 0 f--- ] p6
1 [u 0 f--- ] p0
1 [u 0 f--- ] p1
2 [u 0 1 f--- ] p4
2 [u 0 1 f--- ] p5
2 [u 0 1 f--- ] p6
2 [u 0 1 f--- ] p0
2 [u 0 1 f--- ] p1
3 [* 0 1 2 ] p4
3 [* 0 1 2 ] p5
3 [* 0 1 2 ] p6
3 [* 0 1 2 ] c0
3 [* 0 1 2 ] c1
2 [f--- u 1 2 ] c4
2 [f--- u 1 2 ] c5
2 [EOS * 1 2 ] c6
3 [EOS * 1 2 ] c0
3 [EOS * 1 2 ] [main: added end-of-stream marker]
3 [EOS * 1 2 ] c1
2 [EOS f--- u 2 ] c4
2 [EOS f--- u 2 ] c5
2 [EOS f--- u 2 ] c6
2 [EOS f--- u 2 ] c0
2 [EOS f--- u 2 ] c1
1 [uEOS f--- --- ] c4
1 [uEOS f--- --- ] c5
1 [uEOS f--- --- ] c6
1 [uEOS f--- --- ] c0
1 [uEOS f--- --- ] c1
0 [ --- *--- --- ] c4
0 [ --- *--- --- ] c5
0 [ --- *--- --- ] c6

Consumer consumption:
c0 -> 3
```

- 当缓冲区大小为1时，生产者线程由于完整运行一个循环后，缓冲区就已经满了，所以生产者一定会休眠，切换到消费者运行。
- 发现当缓冲区的大小超过1时，生产者线程和消费者线程都可能连续运行多个循环而不中断。

## (2)

运行命令：

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

1. 程序开始运行时，生产者会快速填充缓冲区，直到填满（`num_full = 10`）。期间消费者最多消费一次，因为消费者每消费一次就需要睡眠1秒，在这一秒时间内生产者完全有时间将缓冲区填满。
2. 之后，每次消费者唤醒后消费一个值，生产者就会立即填充一个值，`num_full` 始终保持在9 ~ 10之间，（生产者填满后等待消费者睡醒，消费者消费后生产者立即填充）。
3. 生产者一共会生产100次，消费者需要 100 秒才能消费完所有数据（每次消费休眠 1 秒）。
4. 生产者生产完所有东西后，消费者每秒消费1次，`num_full` 的值从10逐渐变成0。

大致运行截图如下所示：

```
MINGW64/c/Users/75990/t x + v
(mppdd)
75990@Smile MINGW64 ~/Desktop/Operating System/Homework/ostep-homework-master/threads-cv
$ ./main-two-cvs-while -p 1 -c 1 -m 10 -l 100 -C 0,0,0,0,0,1 -v
NF P0 C0
0 [*--- --- --- --- --- --- --- --- ---] p0
0 [*--- --- --- --- --- --- --- --- ---] p1
1 [u 0 f--- --- --- --- --- --- --- ---] p4
1 [u 0 f--- --- --- --- --- --- --- ---] p5
1 [u 0 f--- --- --- --- --- --- --- ---] p6
1 [u 0 f--- --- --- --- --- --- --- ---] p0
1 [u 0 f--- --- --- --- --- --- --- ---] p1
2 [u 0 1 f--- --- --- --- --- --- --- ---] p4
2 [u 0 1 f--- --- --- --- --- --- --- ---] p5
2 [u 0 1 f--- --- --- --- --- --- --- ---] p6
2 [u 0 1 f--- --- --- --- --- --- --- ---] p0
2 [u 0 1 f--- --- --- --- --- --- --- ---] p1
3 [u 0 1 2 f--- --- --- --- --- --- --- ---] p4
3 [u 0 1 2 f--- --- --- --- --- --- --- ---] p5
3 [u 0 1 2 f--- --- --- --- --- --- --- ---] p6
3 [u 0 1 2 f--- --- --- --- --- --- --- ---] p0
3 [u 0 1 2 f--- --- --- --- --- --- --- ---] p1
4 [u 0 1 2 3 f--- --- --- --- --- --- --- ---] p4
4 [u 0 1 2 3 f--- --- --- --- --- --- --- ---] p5
4 [u 0 1 2 3 f--- --- --- --- --- --- --- ---] p6
4 [u 0 1 2 3 f--- --- --- --- --- --- --- ---] p0
4 [u 0 1 2 3 f--- --- --- --- --- --- --- ---] p1
5 [u 0 1 2 3 4 f--- --- --- --- --- --- --- ---] p4
5 [u 0 1 2 3 4 f--- --- --- --- --- --- --- ---] p5
5 [u 0 1 2 3 4 f--- --- --- --- --- --- --- ---] p6
5 [u 0 1 2 3 4 f--- --- --- --- --- --- --- ---] p0
5 [u 0 1 2 3 4 f--- --- --- --- --- --- --- ---] p1
6 [u 0 1 2 3 4 5 f--- --- --- --- --- --- --- ---] p4
6 [u 0 1 2 3 4 5 f--- --- --- --- --- --- --- ---] p5
6 [u 0 1 2 3 4 5 f--- --- --- --- --- --- --- ---] c0
6 [u 0 1 2 3 4 5 f--- --- --- --- --- --- --- ---] c1
5 [--- u 1 2 3 4 5 f--- --- --- --- --- --- --- ---] c4
5 [--- u 1 2 3 4 5 f--- --- --- --- --- --- --- ---] c5
5 [--- u 1 2 3 4 5 f--- --- --- --- --- --- --- ---] c6
5 [--- u 1 2 3 4 5 f--- --- --- --- --- --- --- ---] p6
5 [--- u 1 2 3 4 5 f--- --- --- --- --- --- --- ---] p0
5 [--- u 1 2 3 4 5 f--- --- --- --- --- --- --- ---] p1
6 [--- u 1 2 3 4 5 6 f--- --- --- --- --- --- --- ---] p4
6 [--- u 1 2 3 4 5 6 f--- --- --- --- --- --- --- ---] p5
6 [--- u 1 2 3 4 5 6 f--- --- --- --- --- --- --- ---] p6
6 [--- u 1 2 3 4 5 6 f--- --- --- --- --- --- --- ---] p0
6 [--- u 1 2 3 4 5 6 f--- --- --- --- --- --- --- ---] p1
7 [--- u 1 2 3 4 5 6 7 f--- --- --- --- --- --- --- ---] p4
```

```
MINGW64/c/Users/75990/t x + v
10 [ 10 11 12 13 * 4 5 6 7 8 9 ] p2
10 [ 10 11 12 13 * 4 5 6 7 8 9 ] c0
10 [ 10 11 12 13 * 4 5 6 7 8 9 ] c1
9 [ 10 11 12 13 f--- u 5 6 7 8 9 ] c4
9 [ 10 11 12 13 f--- u 5 6 7 8 9 ] c5
9 [ 10 11 12 13 f--- u 5 6 7 8 9 ] p3
9 [ 10 11 12 13 14 * 5 6 7 8 9 ] c6
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] p4
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] p5
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] p6
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] p0
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] p1
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] p2
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] c0
10 [ 10 11 12 13 14 * 5 6 7 8 9 ] c1
9 [ 10 11 12 13 14 f--- u 6 7 8 9 ] c4
9 [ 10 11 12 13 14 f--- u 6 7 8 9 ] c5
9 [ 10 11 12 13 14 f--- u 6 7 8 9 ] c6
9 [ 10 11 12 13 14 f--- u 6 7 8 9 ] p3
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] p4
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] p5
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] p6
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] p0
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] p1
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] p2
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] c0
10 [ 10 11 12 13 14 15 * 6 7 8 9 ] c1
9 [ 10 11 12 13 14 15 f--- u 7 8 9 ] c4
9 [ 10 11 12 13 14 15 f--- u 7 8 9 ] c5
9 [ 10 11 12 13 14 15 f--- u 7 8 9 ] c6
9 [ 10 11 12 13 14 15 f--- u 7 8 9 ] p3
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] p4
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] p5
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] p6
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] p0
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] p1
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] p2
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] c0
10 [ 10 11 12 13 14 15 16 * 7 8 9 ] c1
```

```
MINGW64/c/Users/75990/t  ×  +  ▾
10 [ EOS + 91 92 93 94 95 96 97 98 99 ] [main: added end-of-stream marker]
18 [ EOS + 91 92 93 94 95 96 97 98 99 ] c0
18 [ EOS + 91 92 93 94 95 96 97 98 99 ] c1
9 [ EOS f--- u 92 93 94 95 96 97 98 99 ] c4
9 [ EOS f--- u 92 93 94 95 96 97 98 99 ] c5
9 [ EOS f--- u 92 93 94 95 96 97 98 99 ] c6
9 [ EOS f--- u 92 93 94 95 96 97 98 99 ] c0
9 [ EOS f--- u 92 93 94 95 96 97 98 99 ] c1
8 [ EOS f--- --- u 93 94 95 96 97 98 99 ] c4
8 [ EOS f--- --- u 93 94 95 96 97 98 99 ] c5
8 [ EOS f--- --- u 93 94 95 96 97 98 99 ] c6
8 [ EOS f--- --- u 93 94 95 96 97 98 99 ] c0
8 [ EOS f--- --- u 93 94 95 96 97 98 99 ] c1
7 [ EOS f--- --- u 94 95 96 97 98 99 ] c4
7 [ EOS f--- --- u 94 95 96 97 98 99 ] c5
7 [ EOS f--- --- u 94 95 96 97 98 99 ] c6
7 [ EOS f--- --- u 94 95 96 97 98 99 ] c0
7 [ EOS f--- --- u 94 95 96 97 98 99 ] c1
6 [ EOS f--- --- --- u 95 96 97 98 99 ] c4
6 [ EOS f--- --- --- u 95 96 97 98 99 ] c5
6 [ EOS f--- --- --- u 95 96 97 98 99 ] c6
6 [ EOS f--- --- --- u 95 96 97 98 99 ] c0
6 [ EOS f--- --- --- u 95 96 97 98 99 ] c1
5 [ EOS f--- --- --- --- u 96 97 98 99 ] c4
5 [ EOS f--- --- --- --- u 96 97 98 99 ] c5
5 [ EOS f--- --- --- --- u 96 97 98 99 ] c6
5 [ EOS f--- --- --- --- u 96 97 98 99 ] c0
5 [ EOS f--- --- --- --- u 96 97 98 99 ] c1
4 [ EOS f--- --- --- --- --- u 97 98 99 ] c4
4 [ EOS f--- --- --- --- --- u 97 98 99 ] c5
4 [ EOS f--- --- --- --- --- u 97 98 99 ] c6
4 [ EOS f--- --- --- --- --- u 97 98 99 ] c0
4 [ EOS f--- --- --- --- --- u 97 98 99 ] c1
3 [ EOS f--- --- --- --- --- --- u 98 99 ] c4
3 [ EOS f--- --- --- --- --- --- u 98 99 ] c5
3 [ EOS f--- --- --- --- --- --- u 98 99 ] c6
3 [ EOS f--- --- --- --- --- --- u 98 99 ] c0
3 [ EOS f--- --- --- --- --- --- u 98 99 ] c1
2 [ EOS f--- --- --- --- --- --- --- u 99 ] c4
2 [ EOS f--- --- --- --- --- --- --- u 99 ] c5
2 [ EOS f--- --- --- --- --- --- --- u 99 ] c6
2 [ EOS f--- --- --- --- --- --- --- u 99 ] c0
2 [ EOS f--- --- --- --- --- --- --- u 99 ] c1
1 [ uEOS f--- --- --- --- --- --- --- ] c4
1 [ uEOS f--- --- --- --- --- --- --- ] c5
1 [ uEOS f--- --- --- --- --- --- --- ] c6
1 [ uEOS f--- --- --- --- --- --- --- ] c0
1 [ uEOS f--- --- --- --- --- --- --- ] c1
0 [ --- *f--- --- --- --- --- --- --- ] c4
0 [ --- *f--- --- --- --- --- --- --- ] c5
0 [ --- *f--- --- --- --- --- --- --- ] c6

Consumer consumption:
C0 -> 100

(mppd)
75990@Smile MINGW64 ~/Desktop/Operating System/Homework/ostep-homework-master/threads-cv
```

## T4

结论：程序一共会运行大约**10**秒钟的时间，分析过程如下

### 参数分析

- 生产者：1 个（`-p 1`），生产 10 个值（`-l 10`）。
- 消费者：3 个（`-c 3`），每个消费者在步骤 `c3` 后休眠 1 秒（`-C ...,0,0,0,1,0,0,0`）。
- 缓冲区大小：1（`-m 1`），即同一时间只能容纳 1 个值。

### 关键机制分析

#### 1. 缓冲区行为：

- 缓冲区大小为 1，生产者必须等待消费者消费后才能继续生产。

#### 2. 消费者休眠：

- 每个消费者在 `c3` 步骤（从 `Cond_wait` 返回并持有锁后）休眠 1 秒。
- 休眠期间持有锁，导致其他线程（包括生产者和消费者）无法执行。

#### 3. 线程调度：

- 生产者生产一个值后，通过 `Cond_signal(&fill)` 唤醒一个消费者。
- 被唤醒的消费者，在 `c3` 步骤休眠 1 秒（阻塞所有线程），消费值然后释放锁。
- 生产者立即获取锁并生产下一个值，重复此过程。

## 时间计算

- 每个值的生命周期：
  1. 生产者生产一个值（时间忽略不计）。
  2. 消费者消费该值，并在 **c3** 步骤休眠 1 秒（持有锁）。
  3. 生产者等待 1 秒（消费者休眠期间无法生产）。
  4. 消费者释放锁，生产者立即生产下一个值。
- 总时间：
  - 生产 10 个值，每个值需要 **1 秒**（消费者休眠时间）。
  - 总时间 **≈ 10 秒**。

## T8

本问题的答案是无法构造该场景，原因如下：

这与之前的代码的关键区别在于仅使用一个信号量实现生产者与消费者间的通信。而这样做的明显缺陷是在多消费者场景下，消费者完成消费后发送给生产者的唤醒信号可能被其他消费者截获，导致该消费者检查后无数据可消费而继续休眠，同时生产者未被唤醒，最终三者均陷入休眠。此时程序将停滞。

本问题仅设单一生产者与单一消费者，通信对象唯一且固定，不存在信号混淆，因此不会引发上述问题。

## T9

构造的睡眠字符串如下所示：

```
-P 0,0,0,0,0,0,1  
-C 0
```

**解释：**

当生产者完成所有生产任务之后，此时生产者会睡眠一秒。

在这期间，某个消费者会先消费缓冲区中的值，然后尝试继续消费时，发现缓冲区为空，则通过条件变量发送唤醒的信号。但此时生产者正在睡眠，无法接受信号，所以这个唤醒的信号只能被另一个消费者接受。

另一个消费者醒来时，发现并没有内容可以消费，于是继续睡眠。

最终生产者醒来后顺利结束它自己的线程，而另外两个消费者都在睡眠中，程序运行无法结束，造成问题。

执行程序可以发现，程序一直在运行，并且没有输出内容，这就是两个消费者都在睡眠中所导致的。

```
./main-one-cv-while -p 1 -c 2 -m 1 -P 0,0,0,0,0,0,1 -l 3 -v -t
```

## T10

构造的睡眠字符串如下：

```
./main-two-cvs-if -m 1 -c 2 -p 1 -l 4 -C 0,0,0,2:0,0,0,2 -P 1 -v
```

SHELL

这样会使两个消费者在从 `Cond_wait` 返回后都睡一秒，这样就可能导致两个消费者同时访问临界区，进一步的原因如下：

### 1. 单消费者时的潜在问题

- 虚假唤醒（**Spurious Wakeup**）：

POSIX 条件变量允许线程在没有收到信号（`Cond_signal`）或广播（`Cond_broadcast`）的情况下被唤醒。若消费者被虚假唤醒且 `num_full == 0`，使用 `if` 会跳过重新检查条件，导致 `do_get` 操作空缓冲区，触发断言错误（`ensure(tmp != EMPTY)`）。

### 2. 多消费者时的致命问题

当存在多个消费者时，**条件竞争**会加剧问题：

- 竞争场景示例：

1. 缓冲区初始为空，消费者 C1 和 C2 均阻塞在 `Cond_wait(&fill, &m)`。
2. 生产者填充一个元素，调用 `Cond_signal(&fill)` 唤醒 C1。
3. C1 消费元素后释放锁，`num_full` 变为 0。
4. 此时另一个消费者 C2 被唤醒（例如因虚假唤醒或被 P1 唤醒 C1 时唤醒），使用 `if` 会跳过重新检查条件，导致 `do_get` 操作空缓冲区。

## T11

### 1. 产生的问题

在 `producer` 和 `consumer` 函数中，锁的释放时机错误：

- 生产者：在调用 `do_fill()` 前释放锁（p4 前执行 `Mutex_unlock(&m)`）。

- 消费者：在调用 `do_get()` 前释放锁（`c4` 前执行 `Mutex_unlock(&m)`）。

由于 `do_fill()` 和 `do_get()` 操作共享资源（`buffer`、`fill_ptr`、`use_ptr`、`num_full`），提前释放锁会导致这些操作暴露在竞态条件下。可能导致多个生产者同时向缓冲区写入数据，或多个消费者同时向缓冲区消费数据。

## 2.构造睡眠字符串

```
./main-two-cvs-while-extra-unlock \  
-l 1 -m 1 -p 2 -c 0 \  
-P 0,0,0,0,1,0,0:0,0,0,0,0,0,0 \  
-v
```

产生的问题：两个生产者同时填同一个槽

- 场景：缓冲只有 1 个槽（`-m 1`），有 2 个生产者 P0、P1
- 操作流程：
  1. P0 拿到锁、检查 `num_full < max`，通过后立刻解锁，接着在 `p4` 标记处睡一会儿。
  2. 这时 P1 抢到锁，也检查通过，解锁后马上执行 `do_fill`，把自己的值写进那个空槽里，然后发信号、解锁退出。
  3. P0 睡醒，继续执行自己的 `do_fill`，却不知不觉地覆盖了 P1 刚写入的数据，并把 `num_full` 再次加一，从而让计数永远错乱。

## Chp 31

### T1

代码如下：



```
#include <semaphore.h>

sem_t s;

void *child(void *arg) {
    sleep(1);
    printf("child\n");
    // use semaphore here
    sem_post(&s);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init semaphore here
    sem_init(&s, 0, 0);
    Pthread_create(&p, NULL, child, NULL);
    // use semaphore here
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

编译命令:

```
gcc .\fork-join.c -pthread
```

运行结果:

```
.\a.exe
parent: begin
child
parent: end
```

说明父子线程正确同步

## T2

让两个线程互相等待对方的信号量即可：

```
sem_t s1, s2;

void *child_1(void *arg) {
    printf("child 1: before\n");
    // what goes here?
    sem_post(&s1); // signal child 2
    sem_wait(&s2); // wait for child 2
    printf("child 1: after\n");
    return NULL;
}

void *child_2(void *arg) {
    printf("child 2: before\n");
    // what goes here?
    sem_post(&s2); // signal child 1
    sem_wait(&s1); // wait for child 1
    printf("child 2: after\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("parent: begin\n");
    // init semaphores here
    sem_init(&s1, 0, 0);
    sem_init(&s2, 0, 0);
    Pthread_create(&p1, NULL, child_1, NULL);
    Pthread_create(&p2, NULL, child_2, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("parent: end\n");
    return 0;
}
```

运行结果:

```
parent: begin  
child 2: before  
child 1: before  
child 1: after  
child 2: after  
parent: end
```

## T4

代码实现:

```
typedef struct __rwlock_t {
    sem_t lock;
    sem_t write_lock;
    int reader_number;
} rwlock_t;

// 初始化锁
void rwlock_init(rwlock_t* rw) {
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->write_lock, 0, 1);
    rw->reader_number = 0;
}

void rwlock_acquire_readlock(rwlock_t* rw) {
    sem_wait(&rw->lock); // 获取访问reader_number的锁
    rw->reader_number++;
    if (rw->reader_number == 1) {
        // 第一个读者获取写锁, 防止该锁被写者获取
        sem_wait(&rw->write_lock);
    }
    sem_post(&rw->lock); // 释放访问reader_number的锁
}

void rwlock_release_readlock(rwlock_t* rw) {
    sem_wait(&rw->lock); // 获取访问reader_number的锁
    rw->reader_number--;
    if (rw->reader_number == 0) {
        // 最后一个读者释放写锁
        sem_post(&rw->write_lock);
    }
    sem_post(&rw->lock); // 释放访问reader_number的锁
}

void rwlock_acquire_writelock(rwlock_t* rw) {
    sleep(1);
    sem_wait(&rw->write_lock); // 写者获取写锁
}

void rwlock_release_writelock(rwlock_t* rw) {
```

```
sem_post(&rw->write_lock); // 写者释放写锁  
}
```

运行结果：

```
.\reader-writer.exe 5 5 3
begin
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
write 1
write 2
write 3
write 4
write 5
write 6
write 7
write 8
write 9
write 10
write 11
write 12
write 13
write 14
write 15
end: value 15
```

### 展示饥饿问题：

可以发现，写者等到所有读者读完之后才开始写操作，如果读者的数量过多，可能存在饥饿问题。

## T5

考虑添加一把新的锁：当场上没有写者在等待时，多个读者可以同时获得这把锁。而当场上有写者在等待时，最多只有一个读者能获得这把锁。这样当这个读者执行完读操作后，写者又可以和其他的读者公平竞争。

代码实现：

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "common_threads.h"

typedef struct __rwlock_t {
    sem_t service_queue; // 服务队列, 用于排队控制公平性
    sem_t mutex;          // 保护 reader_count 的互斥量
    sem_t write_lock;     // 写锁
    int reader_count;     // 当前正在读的读者数量
} rwlock_t;

void rwlock_init(rwlock_t* rw) {
    sem_init(&rw->service_queue, 0, 1);
    sem_init(&rw->mutex, 0, 1);
    sem_init(&rw->write_lock, 0, 1);
    rw->reader_count = 0;
}

void rwlock_acquire_readlock(rwlock_t* rw) {
    // 先在服务队列排队
    sem_wait(&rw->service_queue);

    // 再去更新 reader_count
    sem_wait(&rw->mutex);
    if (++rw->reader_count == 1) {
        // 第一个读者占用写锁
        sem_wait(&rw->write_lock);
    }
    sem_post(&rw->mutex);

    // 让下一个请求 (无论读者还是写者) 进入队列
    sem_post(&rw->service_queue);
}

void rwlock_release_readlock(rwlock_t* rw) {
    sem_wait(&rw->mutex);
    if (--rw->reader_count == 0) {
```



```
        // 最后一个读者释放写锁
        sem_post(&rw->write_lock);
    }
    sem_post(&rw->mutex);
}

void rwlock_acquire_writelock(rwlock_t* rw) {
    // 写者也先在服务队列排队
    sem_wait(&rw->service_queue);
    // 然后再真正拿写锁
    sem_wait(&rw->write_lock);
    // 放行服务队列，让下一个请求进来
    sem_post(&rw->service_queue);
}

void rwlock_release_writelock(rwlock_t* rw) {
    sem_post(&rw->write_lock);
}
```

运行结果：

```
.\a.exe 5 5 3
begin
read 0
read 0
read 0
write 1
read 1
read 1
read 1
write 2
write 3
write 4
read 4
write 5
read 5
read 5
read 5
write 6
write 7
write 8
write 9
write 10
read 10
read 10
read 10
write 11
write 12
write 13
write 14
read 14
read 14
write 15
end: value 15
```

可以发现不存在上一题中全部读者读完，写者才能写的情况了。读者和写者的操作分布变得十分均匀。

## T6

### 等待空间设置

我们设计了三个线程等待区域：room1、room2和room3

其中room3是默认隐藏的等待区，采用互斥访问机制，同一时刻仅允许单个线程进入

### 运行机制说明

系统运行流程如下：

所有在特定时间段内申请获取锁的线程首先会被统一安排进入room1等待区。由于信号量s1的初始数值设定为1，第一个到达room1的线程能够顺利执行后续48行代码逻辑，获得进入room2的权限。该线程在room2中会执行关键判断操作：检测room1中是否还存在其他被阻塞的线程。若存在，则随机选择一个被阻塞线程唤醒并允许其进入room2；若不存在，则通知room2中的所有线程可以按顺序进入room3。

当room1中的所有线程都成功转移到room2后，最后一个进入的线程会触发s2信号量的post操作，这将随机唤醒room2中的一个线程。被选中的线程会离开room2进入room3，开始执行核心业务逻辑，完成后释放锁资源。在释放锁的过程中，线程会再次进行判断：若发现room2中仍有被阻塞的线程，则随机唤醒其中一个进入room3；若room2已空，则通知那些被阻塞在代码第42行（即room1入口处）的线程可以依次进入room1。

### 形象化类比

我们可以用校园场景来形象说明这个机制：

- room1 相当于教学楼里的普通教室
- room2 相当于面试前的候考区
- room3 则是正式面试的考场

具体流程如下：

1. 学生们按顺序进入教室，每次仅限一人。进入时需要先握住门把手，在教室花名册上登记姓名，然后才能松开把手。
2. 当一批学生都进入教室后，他们需要集体转移到候考区。转移时，每个学生需要同时握住教室和候考区的门把手，完成两个操作：先在教室花名册上注销自己的信息，然后在候考区名册上登记姓名，最后才能松开两个门把手进入候考区。
3. 进入候考区的学生如果发现教室内还有同学未转移，需要保持候考区的门锁开放状态，自己在候考区等待；如果是最后一位转移的学生，则需要负责开启考场门锁。
4. 当考场门锁开启后，候考区中的某位学生握住考场门把手，在候考区名册上注销自己的信息后，即可进入考场参加面试。

5. 学生完成面试离开考场时，如果候考区还有等待的学生，需要保持考场门锁开放；如果是最后一位面试者，则意味着本轮面试全部结束，系统将重置所有状态，准备迎接下一批进入教室的学生。

```

typedef struct __ns_mutex_t {
    int room1;    // 请求线程列表, lock信号量为锁
    int room2;    // 等待线程列表, s1信号量为锁
    sem_t s1;
    sem_t s2;
    sem_t lock;
} ns_mutex_t;

void ns_mutex_init(ns_mutex_t* m) {
    sem_init(&m->s1, 0, 1);    // s1初始值为1, 允许1个线程进入修改
    room2
    sem_init(&m->s2, 0, 0);    // s2初始值为0, 等待获取room2能进入
    room3的信号
    sem_init(&m->lock, 0, 1); // lock初始值为1, 一次只允许1个线程进
    入修改room1
    m->room1 = 0;
    m->room2 = 0;
}

void ns_mutex_acquire(ns_mutex_t* m) {
    // 进入room1
    sem_wait(&m->lock); // 等待进入room1
    m->room1++;          // 进入room1
    sem_post(&m->lock);

    // 等待点1: room1

    // 离开room1, 进入room2
    // 这里同时占有两个锁
    sem_wait(&m->s1);
    m->room2++;
    sem_wait(&m->lock);
    m->room1--;

    // 将room1中的所有线程放入room2, 并打开room3的锁
    if (m->room1) { // 若room1内还有线程等待, 开启room2的锁, 本线程
        需要将它放入room2, 然后本线程也等待在room2
        sem_post(&m->lock);
        sem_post(&m->s1);
    }
}

```

```

    } else { // 若room1内无线程等待，本线程是最后一个线程，开启room3
的锁，允许room2的进入room3
        sem_post(&m->lock);
        sem_post(&m->s2);
    }

    // 等待点2: room2

    // 离开room2，进入room3（也就是开始执行）
    sem_wait(&m->s2);
    m->room2--;
}

void ns_mutex_release(ns_mutex_t* m) {
    // 本线程执行完后，放行一个来自room2的线程
    if (m->room2) {
        sem_post(&m->s2); // 打开room3的门
    } else {
        sem_post(&m->s1); // 恢复初始值，重新开始上述例程
    }
}

```

运行结果：

```

.\mutex-nostarve.exe
parent: begin
parent: end

```