

第 28 章

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？
执行指令 ./x86.py -p flag.s

```
1 .var flag
2 .var count
3
4 .main
5 .top
6
7 .acquire
8 mov flag, %ax      # get flag
9 test $0, %ax      # if we get 0 back: lock is free!
10 jne .acquire      # if not, try again
11 mov $1, flag      # store 1 into flag
12
13 # critical section
14 mov count, %ax    # get the value at the address
15 add $1, %ax      # increment it
16 mov %ax, count    # store it back
17
18 # release lock
19 mov $0, flag      # clear the flag now
20
21 # see if we're still looping
22 sub $1, %bx
23 test $0, %bx
24 jgt .top
25
26 halt
```

```
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False

Thread 0      Thread 1

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

这里用 flag 储存锁的开关状态，对临界区代码进行保护。
但是在执行过程中由于中断间隔的设置，仍然可能会出现

线程1	线程2
mov flag, %ax test \$0, %ax	
	mov flag, %ax test \$0, %ax mov \$1, flag
mov \$1, flag ...	

在不恰当的中断时，可以看到线程 1 和线程 2 可能同时对 flag 赋值 1，都持有锁并进入临界区，锁并没有其他应该的作用。

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标志最终会变成什么值吗？

使用默认值运行时，flags 会按预期工作，可以产生预期的正确结果。

可以使用指令来追踪变量和寄存器的值 ./x86.py -p flag.s -R ax,bx -M flag,count -c

中间间隔为 50，两个线程分别获取锁，占有锁，释放锁，线程运行中没有发生中断，所以最终的 flag 应该为 0，count 会变为 2，ax 会变为 2，bx 会变为 -1

```
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
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
```

3. 使用 -a 标志更改寄存器 %bx 的值(例如,如果只运行两个线程,就用 -a bx=2,bx=2)。代码是做什么的？对这段代码问上面的问题,答案如何？

-a 可以更改寄存器 %bx 的值，将两个线程的寄存器 bx 值设置为 2，运行两个线程，每个线程都会重复执行两次（因为 bx 初始值为 2，减去两次才不满足大于 0），那么最终的 count 在每个线程中都会 +2，最终值应该为 4。在题 1 中已分析过线程执行的代码存在竞态条件，但由于 50 条指令才发生中断，此处两个线程仍然是先后运行的，不会产生错误，由于两个线程都是完整地进行获得锁，释放锁的过程，所以最终 flag=0 执行 ./x86.py -p flag.s -t 2 -a bx=2,bx=2 -M count -R ax,bx -M flag,count -c 指令查看运行情况

flag	count	ax	bx	Thread 0	Thread 1
0	0	0	2		
0	0	0	2	1000 mov flag, %ax	
0	0	0	2	1001 test \$0, %ax	
0	0	0	2	1002 jne .acquire	
1	0	0	2	1003 mov \$1, flag	
1	0	0	2	1004 mov count, %ax	
1	0	1	2	1005 add \$1, %ax	
1	1	1	2	1006 mov %ax, count	
0	1	1	2	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	0	1	1000 mov flag, %ax	
0	1	0	1	1001 test \$0, %ax	
0	1	0	1	1002 jne .acquire	
1	1	0	1	1003 mov \$1, flag	
1	1	1	1	1004 mov count, %ax	
1	1	2	1	1005 add \$1, %ax	
1	2	2	1	1006 mov %ax, count	
0	2	2	1	1007 mov \$0, flag	
0	2	2	0	1008 sub \$1, %bx	
0	2	2	0	1009 test \$0, %bx	
0	2	2	0	1010 jgt .top	
0	2	2	0	1011 halt	
0	2	0	2	----- Halt;Switch -----	----- Halt;Switch -----
0	2	0	2		1000 mov flag, %ax
0	2	0	2		1001 test \$0, %ax
0	2	0	2		1002 jne .acquire
1	2	0	2		1003 mov \$1, flag
1	2	2	2		1004 mov count, %ax
1	2	3	2		1005 add \$1, %ax
1	3	3	2		1006 mov %ax, count
0	3	3	2		1007 mov \$0, flag
0	3	3	1		1008 sub \$1, %bx
0	3	3	1		1009 test \$0, %bx
0	3	3	1		1010 jgt .top
0	3	0	1		1000 mov flag, %ax
0	3	0	1		1001 test \$0, %ax
0	3	0	1		1002 jne .acquire
1	3	0	1		1003 mov \$1, flag
1	3	3	1		1004 mov count, %ax
1	3	4	1		1005 add \$1, %ax
1	4	4	1		1006 mov %ax, count
0	4	4	1		1007 mov \$0, flag
0	4	4	0		1008 sub \$1, %bx
0	4	4	0		1009 test \$0, %bx
0	4	4	0		1010 jgt .top
0	4	4	0		1011 halt

结果和我们预测的一致

4. 对每个线程将 bx 设置为高值,然后使用-i 标志生成不同的中断频率。什么值导致产生不好的结果?什么值导致产生良好的结果?

将 bx 设置为高值, 单个线程执行循环的次数增多, 指令的数目也会增多。

不同的中断频率可能会让两个线程在一些特定的位置发生线程切换, 可能会出现题目 1 中出现的问題, 会产生同时对临界区进行写入, 导致最终的结果小于预期。两个线程结束后 count 应该为 2bx

使用如下指令运行查看结果

./x86.py -p flag.s -t 2 -a bx=100,bx=100 -M count -c -i ?

./x86.py -p flag.s -t 2 -a bx=1000,bx=1000 -M count -c -i ?

? 为相应的中断间隔

bx	中断间隔	结果 (count)
100	20	179
100	30	182
100	40	176
100	50	185
100	60	194
100	70	174
100	80	178
100	90	182
100	100	181
1000	20	1779
1000	30	1802
1000	40	1776
1000	50	1985
1000	60	1904
1000	70	1746
1000	80	1800
1000	90	1802
1000	100	1810

根据结果可以看到，中断间隔越小，线程切换越多，出现对临界区同时写入的情况就越多，最终结果就会越小。在实际运行时，在一些中断时，中断发生在设置 flag 的值，就会产生不好的结果，如果发生在不设置 flag 的值，结果就会较好。而中断间隔越小，发生中断的可能就越大，发生在设置 flag 的值的可能性就越大。

总体来说，中断频率越高，bx 越大，越容易产生不好的结果，如果中断频率设置较好，恰好在每次循环结束时发生，或者大到足以让一个线程完整进行完，就会保证最终的结果是正确的。

5. 现在让我们看看程序 test-and-set.s。首先尝试理解使用 xchg 指令构建简单锁原语的代

码。获取锁怎么写?释放锁如何写?

```
1  .var mutex
2  .var count
3
4  .main
5  .top
6
7  .acquire
8  mov $1, %ax
9  xchg %ax, mutex    # atomic swap of 1 and mutex
10 test $0, %ax       # if we get 0 back: lock is free!
11 jne .acquire       # if not, try again
12
13 # critical section
14 mov count, %ax      # get the value at the address
15 add $1, %ax         # increment it
16 mov %ax, count      # store it back
17
18 # release lock
19 mov $0, mutex
20
21 # see if we're still looping
22 sub $1, %bx
23 test $0, %bx
24 jgt .top
25
26 halt
```

和之前不同的是 xchg 这个原子操作，将 ax 寄存器与 mutex 内存中的值进行交换。

之前的自旋锁中，由于取 flag 和交换 ax 与 flag 的值这两个操作是分开进行的，所以在中断时可能出现同时对 flag 写入的情况。

而我们这里通过一条原子操作解决了这个问题。

获得锁：

```
mov $1, %ax
xchg %ax, mutex    # 原子操作:交换 ax 寄存器与内存 mutex 空间的值(mutex 设为 1)
test $0, %ax       #
jne .acquire       # 如果(%ax)!=0 则自旋等待,即原 mutex 值不为 0
```

首先将 ax 赋值为 1，然后交换 mutex 与 ax 的值，如果此时 mutex=1，那么 ax 与 mutex 交换后就为 ax=1，将 ax 的值与 0 比较，如果 ax!=0，那么就会跳转回.acquire，自旋等待直到 ax=0，跳出循环，也就是 mutex=0 时，ax 与 mutex 交换后 ax=0，将 ax 的值与 0 比较，此时就会跳出循环。

释放锁：

```
mov $0, mutex
```

将已经获得锁的线程的锁释放。

6.现在运行代码,再次更改中断间隔(-i)的值,并确保循环多次。代码是否总能按预期工作?有时会导致 CPU 使用率不高吗?如何量化呢?

执行如下指令

```
./x86.py -p test-and-set.s -t 2 -a bx=100,bx=100 -M count -c -i ?
./x86.py -p test-and-set.s -t 2 -a bx=1000,bx=1000 -M count -c -i ?
#? 为相应的中断间隔
```

bx	中断间隔	结果 (count)
100	20	200
100	30	200
100	40	200
100	50	200
100	60	200
100	70	200
100	80	200
100	90	200
100	100	200
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

由结果可以看到，代码总能按照预期工作。

由于 xchg 的原子操作，我们的锁能够提供互斥，两个线程不会同时进入临界区。

有时候 cpu 的利用率不高，主要原因是当一个线程得到锁，其他线程会自旋等待而不是进入休眠，让 cpu 进行其他工作，直到锁被释放，线程获得锁，开始执行本条指令的内容。

量化计算：完成一次完整的循环计算需要六条指令，加上获取释放锁的指令一共需要 11 条，也就是只有 55% 的时间 cpu 用于关键的运算。其他线程在某一些线程获得锁后都会陷入自旋

等待，完成一次循环计算就需要更多的时间，也就是说 cpu 的最大利用率可以达到 55%。

7. 使用-P 标志生成锁相关代码的特定测试。例如,执行一个测试计划,在第一个线程中获取锁,但随后尝试在第二个线程中获取锁。正确的事情发生了吗?你还应该测试什么?

./x86.py -p test-and-set.s -R ax,bx -M mutex,count -a bx=5 -P 000011111111 -c

mutex	count	ax	bx	Thread 0	Thread 1	1	3	3	4	1006	mov	%ax, count
0	0	0	5			0	3	3	4	1007	mov	\$0, mutex
0	0	1	5	1000 mov \$1, %ax		0	3	3	3	1008	sub	\$1, %bx
1	0	0	5	1001 xchg %ax, mutex		0	3	3	3	1009	test	\$0, %bx
1	0	0	5	1002 test \$0, %ax		0	3	3	3	1010	jgt	.top
1	0	0	5	1003 jne .acquire		0	3	1	3	1000	mov	\$1, %ax
1	0	0	5	1003 jne .acquire		1	3	0	3	1001	xchg	%ax, mutex
1	0	1	5	----- Interrupt -----	1000 mov \$1, %ax	1	3	1	4	----- Interrupt -----		
1	0	1	5	1001 xchg %ax, mutex		1	3	1	4	1001	xchg	%ax, mutex
1	0	1	5	1002 test \$0, %ax		1	3	1	4	1002	test	\$0, %ax
1	0	1	5	1003 jne .acquire		1	3	1	4	1003	jne	.acquire
1	0	1	5	1000 mov \$1, %ax		1	3	1	4	1000	mov	\$1, %ax
1	0	1	5	1001 xchg %ax, mutex		1	3	0	3	----- Interrupt -----		
1	0	1	5	1002 test \$0, %ax		1	3	0	3	1002	test	\$0, %ax
1	0	1	5	1003 jne .acquire		1	3	0	3	1003	jne	.acquire
1	0	0	5	----- Interrupt -----	1003 jne .acquire	1	3	3	3	1004	mov	count, %ax
1	0	0	5	1004 mov count, %ax		1	3	4	3	1005	add	\$1, %ax
1	0	1	5	1005 add \$1, %ax		1	4	4	3	1006	mov	%ax, count
1	1	1	5	1006 mov %ax, count		0	4	4	3	1007	mov	\$0, mutex
0	1	1	5	1007 mov \$0, mutex		0	4	4	2	1008	sub	\$1, %bx
0	1	1	5	----- Interrupt -----	1009 test \$0, %bx	0	4	4	2	1009	test	\$0, %bx
0	1	1	5	1000 mov \$1, %ax		0	4	1	4	----- Interrupt -----		
1	1	0	5	1001 xchg %ax, mutex		1	4	0	4	1001	xchg	%ax, mutex
1	1	0	5	1002 test \$0, %ax		1	4	0	4	1002	test	\$0, %ax
1	1	0	5	1003 jne .acquire		1	4	0	4	1003	jne	.acquire
1	1	1	5	1004 mov count, %ax		1	4	4	4	1004	mov	count, %ax
1	1	2	5	1005 add \$1, %ax		1	4	4	2	----- Interrupt -----		
1	2	2	5	1006 mov %ax, count		1	4	4	2	1010	jgt	.top
0	2	2	5	1007 mov \$0, mutex		1	4	1	2	1000	mov	\$1, %ax
0	2	1	5	----- Interrupt -----	1001 xchg %ax, mutex	1	4	1	2	1001	xchg	%ax, mutex
0	2	1	4	1008 sub \$1, %bx		1	4	1	2	1002	test	\$0, %ax
0	2	1	4	1009 test \$0, %bx		1	4	1	2	1003	jne	.acquire
0	2	1	4	1010 jgt .top		1	4	1	2	1000	mov	\$1, %ax
0	2	1	4	1000 mov \$1, %ax		1	4	1	2	1001	xchg	%ax, mutex
0	2	2	5	----- Interrupt -----	1002 test \$0, %ax	1	4	1	2	1002	test	\$0, %ax
0	2	2	4	1008 sub \$1, %bx		1	4	4	4	----- Interrupt -----		
0	2	2	4	1009 test \$0, %bx		1	4	5	4	1005	add	\$1, %ax
0	2	2	4	1010 jgt .top		1	5	5	4	1006	mov	%ax, count
0	2	1	4	1000 mov \$1, %ax		0	5	5	4	1007	mov	\$0, mutex
1	2	0	4	1001 xchg %ax, mutex		0	5	5	3	1008	sub	\$1, %bx
1	2	0	4	1002 test \$0, %ax		0	5	1	2	----- Interrupt -----		
1	2	0	4	1003 jne .acquire		0	5	1	2	1003	jne	.acquire
1	2	2	4	1004 mov count, %ax		0	5	1	2	1000	mov	\$1, %ax
1	2	1	4	----- Interrupt -----	1001 xchg %ax, mutex	1	5	0	2	1001	xchg	%ax, mutex
1	2	1	4	1001 xchg %ax, mutex		1	5	0	2	1002	test	\$0, %ax
1	2	1	4	1002 test \$0, %ax		1	5	0	2	1003	jne	.acquire
1	2	1	4	1003 jne .acquire		1	5	5	2	1004	mov	count, %ax
1	2	1	4	1000 mov \$1, %ax		1	5	6	2	1005	add	\$1, %ax
1	2	2	4	----- Interrupt -----	1006 mov %ax, count	1	6	6	2	1006	mov	%ax, count
1	2	3	4	1005 add \$1, %ax		1	6	5	3	----- Interrupt -----		
1	6	5	3	1009 test \$0, %bx								
1	6	5	3	1010 jgt .top								
1	6	1	3	1000 mov \$1, %ax								
1	6	1	3	1001 xchg %ax, mutex								
1	6	6	2	----- Interrupt -----	1007 mov \$0, mutex							
0	6	6	2	1007 mov \$0, mutex								
0	6	6	1	1008 sub \$1, %bx								
0	6	6	1	1009 test \$0, %bx								
0	6	6	1	1010 jgt .top								
0	6	1	1	1000 mov \$1, %ax								
1	6	0	1	1001 xchg %ax, mutex								
1	6	0	1	1002 test \$0, %ax								
1	6	0	1	1003 jne .acquire								
1	6	1	3	----- Interrupt -----	1003 jne .acquire							
1	6	1	3	1002 test \$0, %ax								
1	6	1	3	1003 jne .acquire								
1	6	1	3	1000 mov \$1, %ax								
1	6	1	3	1001 xchg %ax, mutex								
1	6	0	1	----- Interrupt -----	1004 mov count, %ax							
1	6	6	1	1004 mov count, %ax								
1	6	7	1	1005 add \$1, %ax								
1	7	7	1	1006 mov %ax, count								
0	7	7	1	1007 mov \$0, mutex								
0	7	7	0	1008 sub \$1, %bx								
0	7	7	0	1009 test \$0, %bx								
0	7	7	0	1010 jgt .top								
0	7	7	0	1011 halt								
0	7	1	3	----- Halt/Switch -----	1011 halt							
0	7	1	3	1002 test \$0, %ax		0	9	9	1	1010	jgt	.top
0	7	1	3	1003 jne .acquire		0	9	1	1	1000	mov	\$1, %ax
0	7	1	3	1000 mov \$1, %ax		1	9	0	1	1001	xchg	%ax, mutex
1	7	0	3	1001 xchg %ax, mutex		1	9	0	1	1002	test	\$0, %ax
1	7	0	3	1002 test \$0, %ax		1	9	0	1	1003	jne	.acquire
1	7	0	3	1003 jne .acquire		1	9	9	1	1004	mov	count, %ax
1	7	7	3	1004 mov count, %ax		1	9	10	1	1005	add	\$1, %ax
1	7	8	3	1005 add \$1, %ax		1	10	10	1	1006	mov	%ax, count
1	8	8	3	1006 mov %ax, count		0	10	10	1	1007	mov	\$0, mutex
0	8	8	3	1007 mov \$0, mutex		0	10	10	0	1008	sub	\$1, %bx
0	8	8	2	1008 sub \$1, %bx		0	10	10	0	1009	test	\$0, %bx
0	8	8	2	1009 test \$0, %bx		0	10	10	0	1010	jgt	.top
0	8	8	2	1010 jgt .top		0	10	10	0	1011	halt	
0	8	1	2	1000 mov \$1, %ax								
1	8	0	2	1001 xchg %ax, mutex								
1	8	0	2	1002 test \$0, %ax								
1	8	0	2	1003 jne .acquire								
1	8	8	2	1004 mov count, %ax								
1	8	9	2	1005 add \$1, %ax								
1	9	2	2	1006 mov %ax, count								
0	9	9	2	1007 mov \$0, mutex								
0	9	9	1	1008 sub \$1, %bx								

-p 可以用来引起线程 0 和线程 1 的执行，我们先让线程 0 执行 4 句，此时线程 0 获取锁，

然后线程 1 执行 8 句，没有锁，陷入自旋，线程 1 将这段代码执行了两次，线程 1 的第二段执行过程那里可以看到，当线程 0 释放锁之后，线程 1 再次获取锁成功，随后成功进入临界区执行，这说明这个锁在释放之后，其他线程可以获取锁，锁的功能正常。

第 30 章

1. 我们的第一个问题集中在 main-two-cvs-while.c (有效的解决方案) 上。首先，研究代码。你认为你了解当你运行程序时会发生什么吗？

```
//这是生产者消费者问题的解决方案

void do_fill(int value) {
    // ensure empty before usage
    ensure(buffer[fill_ptr] == EMPTY, "error: tried to fill a non-empty buffer");
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % max;
    num_full++;
}

int do_get() {
    int tmp = buffer[use_ptr];
    ensure(tmp != EMPTY, "error: tried to get an empty buffer");
    buffer[use_ptr] = EMPTY;
    use_ptr = (use_ptr + 1) % max;
    num_full--;
    return tmp;
}

void *producer(void *arg) {
    int id = (int) arg;
    // make sure each producer produces unique values
    int base = id * loops;
    int i;
    for (i = 0; i < loops; i++) {
        p0;
        Mutex_lock(&m);
        while (num_full == max) {
            p2;
            Cond_wait(&empty, &m);
            p3;
        }
        do_fill(base + i);
        Cond_signal(&fill);
        p5;
        Mutex_unlock(&m);
        p6;
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) {
        c0;
        Mutex_lock(&m);
        while (num_full == 0) {
            c2;
            Cond_wait(&fill, &m);
            c3;
        }
        tmp = do_get();
        c4;
        Cond_signal(&empty);
        c5;
        Mutex_unlock(&m);
        c6;
        consumed_count++;
    }

    // return consumer_count-1 because END_OF_STREAM does not count
    return (void *) (long long) (consumed_count - 1);
}
```

do_fill 函数将给定的值填充到缓冲区中，而 do_get 函数从缓冲区中获取一个值。pthread_cond_t 和 pthread_mutex_t 类型的全局变量 empty、fill 和 m 分别用于条件变量和互斥锁。

让生产者生产数据到缓冲区中（如果缓冲区未满），消费者从缓冲区中取数据。程序运行时，希望生产者会在缓冲区满时等待，并且一旦缓冲区有空间，则立即解锁等待放置数据。消费者会在缓冲区为空的时候等待，并且缓冲区一旦有数据，则立刻解除锁等待消费数据。

2. 指定一个生产者和一个消费者运行，并让生产者产生一些元素。缓冲区大小从 1 开始，然后增加。随着缓冲区大小增加，程序运行结果如何改变？当使用不同的缓冲区大小(例如 -m 10)，生产者生产不同的产品数量(例如 -l 100)，修改消费者的睡眠字符串(例如 -C 0,0,0,0,0,1)，full_num 的值如何变化？

先使用如下指令生成可执行文件

```
make main-two-cvs-while
```

再执行类似如下指令获取结果

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

在终端的运行结果中，NF 表示 num_full,缓冲区中数据的数据量。中间是缓冲区的情况。--表示该缓冲区没有数据。后面 P0 列表示生产者 (producer) 0 执行到哪一行代码。对应第一题中每一行代码后面的注释，C0 列也同理，表示消费者 (consumer) 0 执行到哪一行代码。

1) 只改变缓冲区

执行下面的指令

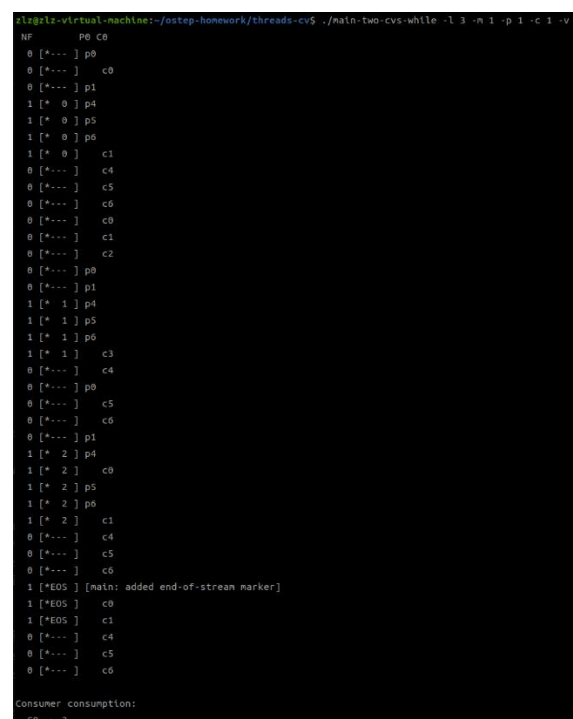
```
./main-two-cvs-while -l 3 -m 1 -p 1 -c 1 -v
```

```
./main-two-cvs-while -l 3 -m 2 -p 1 -c 1 -v
```

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

运行截图：

Buffer=1



```
sl@sliz-virtual-machine:~/astep-homework/threads-cvs$ ./main-two-cvs-while -l 3 -m 1 -p 1 -c 1 -v
NF      P0 C0
0 [*---] p0
0 [*---] c0
0 [*---] p1
1 [* 0] p4
1 [* 0] p5
1 [* 0] p6
1 [* 0] c1
0 [*---] c4
0 [*---] c5
0 [*---] c6
0 [*---] c0
0 [*---] c1
0 [*---] c2
0 [*---] p0
0 [*---] p1
1 [* 1] p4
1 [* 1] p5
1 [* 1] p6
1 [* 1] c3
0 [*---] c4
0 [*---] p0
0 [*---] c5
0 [*---] c6
0 [*---] p1
1 [* 2] p4
1 [* 2] c0
1 [* 2] p5
1 [* 2] p6
1 [* 2] c1
0 [*---] c4
0 [*---] c5
0 [*---] c6
1 [*EOS] [main: added end-of-stream marker]
1 [*EOS] c0
1 [*EOS] c1
0 [*---] c4
0 [*---] c5
0 [*---] c6
Consumer consumption:
C0 -> 3
```

Buffer=2

```

zlg@zlg-virtual-machine:~/ostep-homework/threads-cv$ ./main-two-cvs-while -l 3 -n 2 -p 1 -c 1 -v
NF          p0 c0
0 [*--- ---] p0
0 [*--- ---] c0
0 [*--- ---] p1
1 [u 0 f---] p4
1 [u 0 f---] p5
1 [u 0 f---] p6
1 [u 0 f---] c1
1 [u 0 f---] p0
0 [--- *---] c4
0 [--- *---] c5
0 [--- *---] c6
0 [--- *---] p1
1 [f--- u 1] p4
1 [f--- u 1] p5
1 [f--- u 1] p6
1 [f--- u 1] p0
1 [f--- u 1] c0
1 [f--- u 1] p1
2 [ 2 * 1] p4
2 [ 2 * 1] p5
2 [ 2 * 1] p6
2 [ 2 * 1] c1
2 [u 2 f---] c4
1 [u 2 f---] c5
1 [u 2 f---] c6
2 [* 2 EOS] [main: added end-of-stream marker]
2 [* 2 EOS] c0
2 [* 2 EOS] c1
1 [f--- uEOS] c4
1 [f--- uEOS] c5
1 [f--- uEOS] c6
1 [f--- uEOS] c0
1 [f--- uEOS] c1
0 [*--- ---] c4
0 [*--- ---] c5
0 [*--- ---] c6
Consumer consumption:
c0 -> 3

```

Buffer=3

```

zlg@zlg-virtual-machine:~/ostep-homework/threads-cv$ ./main-two-cvs-while -l 3 -n 3 -p 1 -c 1 -v
NF          p0 c0
0 [*--- --- ---] c0
0 [*--- --- ---] p0
0 [*--- --- ---] p1
1 [u 0 f--- ---] p4
1 [u 0 f--- ---] p5
1 [u 0 f--- ---] p6
1 [u 0 f--- ---] c1
1 [u 0 f--- ---] p0
0 [--- *--- ---] c4
0 [--- *--- ---] c5
0 [--- *--- ---] c6
0 [--- *--- ---] p1
1 [--- u 1 f---] p4
1 [--- u 1 f---] p5
1 [--- u 1 f---] p6
1 [--- u 1 f---] p0
1 [--- u 1 f---] p1
1 [--- u 1 f---] c0
2 [f--- u 1 2] p4
2 [f--- u 1 2] p5
2 [f--- u 1 2] p6
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

```

程序运行的结果基本不变，随着 m 的增加，生产者和消费者每次防止数据的位置和消费的数据有所不同。但消费者还是能在缓冲区为 0 时等待，一旦不为 0 就开始消费。生产者放置数据没有异常，消费者消费数据也没有异常。消费者成功地消费了三个数据。

2) 设置休眠序列并且改变固定缓冲区的大小

可以设置缓冲区的大小为 10，即 -m 10,生产的总数设置为 100，即 -l 100,同时设置消费者的睡眠情况为 -C 0, 0, 0, 0, 0, 0, 1（在 c6 处进入睡眠）时，输入如下指令

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

```

slizviz-virtual-machine:~/ostep-homework/threads-cv$ ./main-two-cvs-while -p 1 -c 1 -m 10 -l 100 -C 0,0,0,0,0,1
NF                                p0 c0
0 [*--- --- --- --- --- --- --- --- ] p0
0 [*--- --- --- --- --- --- --- --- ] c0
0 [*--- --- --- --- --- --- --- --- ] p1
1 [u 0 f--- --- --- --- --- --- --- ] p4
1 [u 0 f--- --- --- --- --- --- --- ] p5
1 [u 0 f--- --- --- --- --- --- --- ] p6
1 [u 0 f--- --- --- --- --- --- --- ] c1
0 [ --- *--- --- --- --- --- --- --- ] c4
0 [ --- *--- --- --- --- --- --- --- ] p0
0 [ --- *--- --- --- --- --- --- --- ] c5
0 [ --- *--- --- --- --- --- --- --- ] c6
0 [ --- *--- --- --- --- --- --- --- ] p1
1 [ --- u 1 f--- --- --- --- --- --- ] p4
1 [ --- u 1 f--- --- --- --- --- --- ] p5
1 [ --- u 1 f--- --- --- --- --- --- ] p6
1 [ --- u 1 f--- --- --- --- --- --- ] p0
1 [ --- u 1 f--- --- --- --- --- --- ] p1
2 [ --- u 1 2 f--- --- --- --- --- --- ] p4
2 [ --- u 1 2 f--- --- --- --- --- --- ] p5
2 [ --- u 1 2 f--- --- --- --- --- --- ] p6
2 [ --- u 1 2 f--- --- --- --- --- --- ] p0
2 [ --- u 1 2 f--- --- --- --- --- --- ] p1
3 [ --- u 1 2 3 f--- --- --- --- --- --- ] p4
3 [ --- u 1 2 3 f--- --- --- --- --- --- ] p5
3 [ --- u 1 2 3 f--- --- --- --- --- --- ] p6
3 [ --- u 1 2 3 f--- --- --- --- --- --- ] p0
3 [ --- u 1 2 3 f--- --- --- --- --- --- ] p1
4 [ --- u 1 2 3 4 f--- --- --- --- --- --- ] p4
4 [ --- u 1 2 3 4 f--- --- --- --- --- --- ] p5
4 [ --- u 1 2 3 4 f--- --- --- --- --- --- ] p6
4 [ --- u 1 2 3 4 f--- --- --- --- --- --- ] p0
4 [ --- u 1 2 3 4 f--- --- --- --- --- --- ] p1
5 [ --- u 1 2 3 4 5 f--- --- --- --- --- --- ] p4
5 [ --- u 1 2 3 4 5 f--- --- --- --- --- --- ] p5
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

```

一开始生产者会将缓冲区填满。在此之后会进入如下的交替模式。一旦出现缓冲区出现空缺，消费者就会唤醒生产者将空缺填满。

```

9 [f--- u 1 2 3 4 5 6 7 8 9 ] p4
9 [f--- u 1 2 3 4 5 6 7 8 9 ] p5
9 [f--- u 1 2 3 4 5 6 7 8 9 ] p6
9 [f--- u 1 2 3 4 5 6 7 8 9 ] p0
9 [f--- u 1 2 3 4 5 6 7 8 9 ] p1
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] p4
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] p5
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] p6
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] p0
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] p1
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] p2
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] c0
10 [ 10 * 1 2 3 4 5 6 7 8 9 ] c1
9 [ 10 f--- u 2 3 4 5 6 7 8 9 ] c4
9 [ 10 f--- u 2 3 4 5 6 7 8 9 ] c5
9 [ 10 f--- u 2 3 4 5 6 7 8 9 ] c6
9 [ 10 f--- u 2 3 4 5 6 7 8 9 ] p3
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] p4
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] p5
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] p6
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] p0
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] p1
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] p2
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] c0
10 [ 10 11 * 2 3 4 5 6 7 8 9 ] c1
9 [ 10 11 f--- u 3 4 5 6 7 8 9 ] c4
9 [ 10 11 f--- u 3 4 5 6 7 8 9 ] c5
9 [ 10 11 f--- u 3 4 5 6 7 8 9 ] c6
9 [ 10 11 f--- u 3 4 5 6 7 8 9 ] p3

```

剩下的会按照这个模式交替执行直到 100 个全部生产完毕后，消费者会一次性把所有内容

消费完。

5. 我们来看一些 timings。 对于一个生产者，三个消费者，大小为 1 的共享缓冲区以及每个消费者在 **c3 点暂停一秒**，您认为需要执行多长时间？（./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0 -l 10 -v -t）

消费者在 c3 点暂停 1s，因为只有一个大小为 1 的缓冲区，所以肯定是生产者先进行生产，然后唤醒消费者消费，所以正常情况下应该是 10 次唤醒，如果不考虑多个消费者同时醒着，因为 num_full=0 而在 while 中会额外循环一次，正常需要 10s，再考虑到最终处理 EOF 标记时 3 个消费者各自需要 1s，所以理想情况下需要 13s

```
144@ali-virtual-machine: /root/.homeworl/threads-cv$ ./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0 -l 10 -v -t
MF      p0 c0 c1 c2
0 [*...] p0
0 [*...] c0
0 [*...] c0
0 [*...] c0
0 [*...] p1
1 [* 0] p4
1 [* 0] p5
1 [* 0] p6
1 [* 0] c1
1 [* 0] p0
0 [*...] c4
0 [*...] c5
0 [*...] c6
0 [*...] c1
0 [*...] c2
0 [*...] c0
0 [*...] c1
0 [*...] c2
0 [*...] p1
1 [* 1] p4
1 [* 1] p5
1 [* 1] p6
1 [* 1] c1
1 [* 1] p0
0 [*...] c4
0 [*...] c5
0 [*...] c6
0 [*...] c3
0 [*...] c0
0 [*...] c2
0 [*...] p1
1 [* 2] p4
1 [* 2] p5
1 [* 2] p6
1 [* 2] c1
0 [*...] c4
0 [*...] p0
0 [*...] c5
0 [*...] c6
0 [*...] c3
0 [*...] c0
```

```
0 [*...] c2
0 [*...] p1
1 [* 3] p4
1 [* 3] p5
1 [* 3] p6
1 [* 3] c1
1 [* 3] p0
0 [*...] c4
0 [*...] c5
0 [*...] c6
0 [*...] c3
0 [*...] c0
0 [*...] c2
0 [*...] p1
1 [* 4] p4
1 [* 4] p5
1 [* 4] p6
1 [* 4] c1
1 [* 4] p0
0 [*...] c4
0 [*...] c5
0 [*...] c6
0 [*...] c3
0 [*...] c0
0 [*...] c2
0 [*...] p1
1 [* 5] p4
1 [* 5] p5
1 [* 5] p6
1 [* 5] c1
1 [* 5] p0
0 [*...] c4
0 [*...] c5
0 [*...] c6
0 [*...] c3
0 [*...] c0
0 [*...] c2
0 [*...] p1
1 [* 6] p4
1 [* 6] p5
1 [* 6] p6
```



```

int base = id * loops;
int i;
for (i = 0; i < loops; i++) {    p0;
    Mutex_lock(&m);                p1;
    while (num_full == max) {    p2;
        Cond_wait(&cv, &m);      p3;
    }
    do_fill(base + i);            p4;
    Cond_signal(&cv);              p5;
    Mutex_unlock(&m);              p6;
}
return NULL;
}

```

//消费者

```

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) { c0;
        Mutex_lock(&m);            c1;
        while (num_full == 0) {    c2;
            Cond_wait(&cv, &m);    c3;
        }
        tmp = do_get();            c4;
        Cond_signal(&cv);          c5;
        Mutex_unlock(&m);          c6;
        consumed_count++;
    }
}

```

与之前代码的区别是生产者和消费者之间只用一个信号量来处理。其中最明显的问题是在多个消费者的情况下，消费者消费结束时唤醒生产者的信号可能被另一个消费者接收，这时新的消费者检查之后没有可以用的数据，就会重新陷入睡眠，生产者也没有被唤醒，此时三个都在休眠，就会导致没有线程运行。

但是本题目中只有一个生产者和一个消费者，消费者不会唤醒其他消费者，智慧唤醒生产者，所以不会出现问题，也就是构造字符串无法实现。

9. 现在将消费者数量更改为两个。为生产者消费者配置睡眠字符串，从而使代码运行出现问题。

在两个消费者的情况下，就有可能出现第8题所描述的情况。在多个消费者的情况下，消费者消费结束时唤醒生产者的信号可能被另一个消费者接收，这时新的消费者检查之后没有可以用的数据，就会重新陷入睡眠，生产者也没有被唤醒，此时三个都在休眠，就会导致没有线程运行。

我们可以在生产者结束所有生产后，将其中的一个消费者唤醒，这个消费者消费完这个缓冲区的资源后，可以唤醒的就只有剩下的另一个消费者了，此时缓冲区无内容，三个对象就都会陷入休息，程序停止运行。

我们可以构造如下的序列：0, 0, 0, 0, 0, 0, 1

执行时需要重新 make 文件 make main-one-cv-while

然后执行指令 ./main-one-cv-while -p 1 -c 2 -m 1 -P 0,0,0,0,0,1 -l 3 -v -t

```
flag@iz-virtuall-machine:~/osdep-homework/threads-cv$ ./main-one-cv-while -p 1 -c 2 -m 1 -P 0,0,0,0,0,1 -l 3 -v -t
NF      p0 c0 c1
0 [*--- ] p0
0 [*--- ] c0
0 [*--- ] p1
0 [*--- ] c0
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
0 [*--- ] p0
0 [*--- ] p1
1 [* 1 ] p4
1 [* 1 ] p5
1 [* 1 ] p6
1 [* 1 ] c3
0 [*--- ] c4
0 [*--- ] c5
0 [*--- ] c6
0 [*--- ] c3
0 [*--- ] c0
0 [*--- ] c2
0 [*--- ] c1
0 [*--- ] c2
0 [*--- ] p0
0 [*--- ] p1
1 [* 2 ] p4
1 [* 2 ] p5
1 [* 2 ] p6
1 [* 2 ] c3
0 [*--- ] c4
0 [*--- ] c5
0 [*--- ] c6
0 [*--- ] c3
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
```

最终程序会停止运行，但是程序没有结束，这是因为 c0 唤醒 c1，经过 c1 检查后发现没有可以消费的对象，陷入休眠，三个对象都在休眠

10. 现在查看 main-two-cvs-if.c。您是否可以配置一些参数让代码运行出现问题？ 再次考虑只有一个消费者的情况，然后再考虑有一个以上消费者的情况。

先查看 main-two-cvs-if.c 生产者和消费者的代码

```
void *producer(void *arg) {
    int id = (int) arg;
    // make sure each producer produces unique values
    int base = id * loops;
    int i;
    for (i = 0; i < loops; i++) {
        p0;
        Mutex_lock(&m);
        p1;
        if (num_full == max) {
            p2;
            Cond_wait(&empty, &m);
            p3;
        }
    }
}
```



```

        do_fill(base + i);          p4;
        Cond_signal(&fill);         p5;
        Mutex_unlock(&m);           p6;
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) { c0;
        Mutex_lock(&m);             c1;
        if (num_full == 0) {         c2;
            Cond_wait(&fill, &m);   c3;
        }
        tmp = do_get();              c4;
        Cond_signal(&empty);         c5;
        Mutex_unlock(&m);            c6;
        consumed_count++;
    }
    // return consumer_count-1 because END_OF_STREAM does not count
    return (void *) (long long) (consumed_count - 1);
}

```

这里可能出现错误的原因是有单个消费者和多个消费者，消费者休眠后，生产者生产后唤醒了改消费者，如果还未来得及上锁，发生了中断，另一个消费者进入消费了这个值，这个消费者再次唤醒后执行 c4 就会发生错误。

只要消费者 c1 在 c0 处，而生产者刚好完成生产，就有可能出现这种问题。

执行时需要重新 make 文件 make main-two-cvs-if

然后执行如下代码

./main-two-cvs-if -m 1 -c 2 -p 1 -l 10 -v

```

zllz@zllz-virtual-machine:~/ostep-homework/threads-cv$ ./main-two-cvs-if -m 1 -c 2 -p 1 -l 10 -v
NF      P0 C0 C1
0 [*--- ] p0
0 [*--- ]  c0
0 [*--- ]  c0
0 [*--- ]  c1
0 [*--- ]  c2
0 [*--- ] p1
1 [*  0 ] p4
1 [*  0 ] p5
1 [*  0 ] p6
1 [*  0 ]  c1
1 [*  0 ] p0
0 [*--- ]  c4
0 [*--- ]  c5
0 [*--- ]  c6
0 [*--- ]  c3
error: tried to get an empty buffer

```

```

0 [*--- ]      c0//消费者 C1 已经在 c0 处
0 [*--- ]      c1
0 [*--- ]      c2
0 [*--- ] p1
1 [* 0 ] p4
1 [* 0 ] p5
1 [* 0 ] p6      //生产结束，唤醒 c0，但是消费者 C1 先运行了
1 [* 0 ]      c1//C1 抢先消费了值
1 [* 0 ] p0
0 [*--- ]      c4
0 [*--- ]      c5
0 [*--- ]      c6
0 [*--- ]      c3 //C0 准备消费，将发生错误

```

11. 最后查看 main-two-cvs-while-extra-unlock.c。在向缓冲区添加或取出元素时释放锁时会出现什么问题？给定睡眠字符串来引起这类问题的发生？会造成什么不好的结果？查看 main-two-cvs-while-extra-unlock.c 中的生产者和消费者函数的实现。

```

void *producer(void *arg) {
    int id = (int) arg;
    // make sure each producer produces unique values
    int base = id * loops;
    int i;
    for (i = 0; i < loops; i++) {
        p0;
        Mutex_lock(&m);
        p1;
        while (num_full == max) {
            p2;
            Cond_wait(&empty, &m);
            p3;
        }
        Mutex_unlock(&m);
        do_fill(base + i);
        p4;
        Mutex_lock(&m);
        Cond_signal(&fill);
        p5;
        Mutex_unlock(&m);
        p6;
    }
    return NULL;
}

```

```

void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) {
        c0;
        Mutex_lock(&m);
        c1;
        while (num_full == 0) {
            c2;
            Cond_wait(&fill, &m);
            c3;
        }
    }
}

```

```

    }
    Mutex_unlock(&m);
    tmp = do_get();          c4;
    Mutex_lock(&m);
    Cond_signal(&empty);     c5;
    Mutex_unlock(&m);        c6;
    consumed_count++;
}

```

在向缓冲区添加或取出元素时释放锁时可能会出现問題, 对缓冲区的访问可能会出现并发现象。

如果两个生产者都在进行 do_fill, 对同一个缓冲区进行操作, 对 buff[fill_ptr]赋值 value, 还没有进行 fill_ptr+1, 此时另一个生产者运行, 对同一个 buff[fill_ptr]进行赋值, 就会导致对同一个缓冲区进行两次赋值, 第一次赋值会被覆盖而没有被消费。

也有可能出現两个消费者同时对同一片区域进行消费, 导致没有数据可以消费的情况。为了构造这样一个序列, 我们可以设置生产者和消费者线程的睡眠字符串, 使它们在适当的时候释放锁, 然后其他线程可以同时执行。我们以 main-two-cvs-while-extra-unlock.c 中的生产者和消费者函数为例, 构造如下睡眠字符串:

对于生产者, 我们在 p4 和 p5 之间添加一个睡眠点, 以模拟在填充缓冲区时释放锁。

对于消费者, 我们在 c4 和 c5 之间添加一个睡眠点, 以模拟在获取数据时释放锁。

因此, 我们可以使用如下睡眠字符串来构造这样一个序列: -P 0,0,0,1,0,0,0 -C 0,0,0,1,0,0,0

仍然需要先 make 文件 make main-two-cvs-while-extra-unlock

然后执行指令 ./main-two-cvs-while-extra-unlock -l 1 -m 2 -p 1 -c 1 -P 1,0,0,0,0,0,0 -C 0 -v

这将使得生产者在执行 p4 和 p5 之间的时候睡眠 1 秒, 而消费者在执行 c4 和 c5 之间的时候睡眠 1 秒。这样, 就有可能出现多个生产者或消费者同时访问缓冲区的情况, 从而导致并发问题。

```

1 -P 1,0,0,0,0,0,0 -C 0 -v
NF          P0 C0
0 [*-- -- ] p0
0 [*-- -- ] c0
0 [*-- -- ] c1
0 [*-- -- ] c2
0 [*-- -- ] p1
1 [u 0 f-- ] p4
1 [u 0 f-- ] p5
1 [u 0 f-- ] p6
1 [u 0 f-- ] c3
0 [ -- *-- ] c4
1 [f-- uEOS ] [main: added end-of-stream marker]
1 [f-- uEOS ] c5
1 [f-- uEOS ] c6
1 [f-- uEOS ] c0
1 [f-- uEOS ] c1
0 [*-- -- ] c4
0 [*-- -- ] c5
0 [*-- -- ] c6

Consumer consumption:
C0 -> 1

```

第 31 章

1. 第一个问题就是实现和测试 fork/join 问题的解决方案，如本文所述。即使在文本中描述了此解决方案，重新自己实现一遍也是值得的。even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one。有关详细信息，请参见 fork-join.c。将添加 sleep(1) 到 child 函数内以确保其正常工作。

修改前：

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include "common_threads.h"
5
6 sem_t s;
7
8 void *child(void *arg) {
9     printf("child\n");
10    // use semaphore here
11    return NULL;
12 }
13
14 int main(int argc, char *argv[]) {
15     pthread_t p;
16     printf("parent: begin\n");
17     // init semaphore here
18     Pthread_create(&p, NULL, child, NULL);
19     // use semaphore here
20     printf("parent: end\n");
21     return 0;
22 }
```

修改后：

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include "common_threads.h"
5 #include <semaphore.h>
6
7 sem_t s;
8
9 void *child(void *arg) {
10    sleep(1);
11    printf("child\n");
12    sem_post(&s);
13    return NULL;
14 }
15
16 int main(int argc, char *argv[]) {
17     pthread_t p;
18     printf("parent: begin\n");
19     sem_init(&s, 0, 0);
20     Pthread_create(&p, NULL, child, NULL);
21     sem_wait(&s);
22     printf("parent: end\n");
23     return 0;
24 }
25
```

加入了信号量并且在 child 中停顿 1s

用指令进行编译 gcc -o fork-join fork-join.c -Wall -pthread

运行结果如下：

```
zllz@zllz-virtual-machine:~/ostep-homework/threads-sema$ ./fork-join
parent: begin
child
parent: end
```

在运行过程中，parent: begin 与 child 中间有明显的停顿

2. 现在，我们通过研究集合点问题 rendezvous problem 来对此进行概括。问题如下：您有两个线程，每个线程将要在代码中进入集合点。任何一方都不应在另一方进入之前退出代码的这一部分。该任务使用两个信号量，有关详细信息，请参见 rendezvous.c。

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include "common_threads.h"
4
5 // If done correctly, each child should print their "before" message
6 // before either prints their "after" message. Test by adding sleep(1)
7 // calls in various locations.
8
9 sem_t s1, s2;
10
11 void *child_1(void *arg) {
12     printf("child 1: before\n");
13     // what goes here?
14     printf("child 1: after\n");
15     return NULL;
16 }
17
18 void *child_2(void *arg) {
19     printf("child 2: before\n");
20     // what goes here?
21     printf("child 2: after\n");
22     return NULL;
23 }
24
25 int main(int argc, char *argv[]) {
26     pthread_t p1, p2;
27     printf("parent: begin\n");
28     // init semaphores here
29     pthread_create(&p1, NULL, child_1, NULL);
30     pthread_create(&p2, NULL, child_2, NULL);
31     pthread_join(p1, NULL);
32     pthread_join(p2, NULL);
33     printf("parent: end\n");
34     return 0;
35 }
36

```

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include "common_threads.h"
5 #include <semaphore.h>
6
7 // If done correctly, each child should print their "before" message
8 // before either prints their "after" message. Test by adding sleep(1)
9 // calls in various locations.
10
11 sem_t s1, s2;
12
13 void *child_1(void *arg) {
14     printf("child 1: before\n");
15     sleep(1);
16     sem_post(&s1);
17     sem_wait(&s2);
18     printf("child 1: after\n");
19     return NULL;
20 }
21
22 void *child_2(void *arg) {
23     printf("child 2: before\n");
24     sleep(1);
25     sem_post(&s2);
26     sem_wait(&s1);
27     printf("child 2: after\n");
28     return NULL;
29 }
30
31 int main(int argc, char *argv[]) {
32     pthread_t p1, p2;
33     printf("parent: begin\n");
34     sem_init(&s1, 0, 0);
35     sem_init(&s2, 0, 0);
36     pthread_create(&p1, NULL, child_1, NULL);
37     pthread_create(&p2, NULL, child_2, NULL);
38     pthread_join(p1, NULL);
39     pthread_join(p2, NULL);
40     printf("parent: end\n");
41     return 0;
42 }
43

```

在每个线程到达集合点时，发送一个信号量（post）表明自己的状态。每一个线程接收到对方发出的信号量(wait)后才继续往下进行，这就完成了集合。其中任意一个线程的 post 和 wait 可以互换顺序，但不能都换。

执行如下指令编译

gcc -o rendezvous rendezvous.c -Wall -pthread

```

z1z@z1z-virtual-machine:~/ostep-homework/threads-sema$ ./rendezvous
parent: begin
child 1: before
child 2: before
child 1: after
child 2: after
parent: end

```

可以看到 child1 和 child2 同时进入后才会 after

4. 现在按照文本中所述，解决读者写者问题。首先，不用考虑进程饥饿。有关详细信息，请参见 reader-writer.c 中的代码。将 sleep () 调用添加到您的代码中，以证明它可以按预期工作。你能证明饥饿问题的存在吗？

这道题目可以仿照课本中的实现，实现相对复杂

修改前：

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include "common_threads.h"
5
6 //
7 // Your code goes in the structure and functions below
8 //
9
10 typedef struct __rwlock_t {
11 } rwlock_t;
12
13
14 void rwlock_init(rwlock_t *rw) {
15 }
16
17 void rwlock_acquire_readlock(rwlock_t *rw) {
18 }
19
20 void rwlock_release_readlock(rwlock_t *rw) {
21 }
22
23 void rwlock_acquire_writelock(rwlock_t *rw) {
24 }
25
26 void rwlock_release_writelock(rwlock_t *rw) {
27 }
28
29 //
30 // Don't change the code below (just use it!)
31 //
32
33 int loops;
34 int value = 0;
35
36 rwlock_t lock;
37
38 void *reader(void *arg) {
39     int i;
40     for (i = 0; i < loops; i++) {
41         rwlock_acquire_readlock(&lock);
42         printf("read %d\n", value);
43         rwlock_release_readlock(&lock);
44     }
45     return NULL;
46 }
47
48 void *writer(void *arg) {
49     int i;
50     for (i = 0; i < loops; i++) {
51         rwlock_acquire_writelock(&lock);
52         value++;
53         printf("write %d\n", value);
54         rwlock_release_writelock(&lock);
55     }
56     return NULL;
57 }
58
59 int main(int argc, char *argv[]) {
60     assert(argc == 4);
61     int num_readers = atoi(argv[1]);
62     int num_writers = atoi(argv[2]);
63     loops = atoi(argv[3]);
64
65     pthread_t pr[num_readers], pw[num_writers];
66
67     rwlock_init(&lock);
68
69     printf("begin\n");
70
71     int i;
72     for (i = 0; i < num_readers; i++)
73         Pthread_create(&pr[i], NULL, reader, NULL);
74     for (i = 0; i < num_writers; i++)
75         Pthread_create(&pw[i], NULL, writer, NULL);
76
77     for (i = 0; i < num_readers; i++)
78         Pthread_join(pr[i], NULL);
79     for (i = 0; i < num_writers; i++)
80         Pthread_join(pw[i], NULL);
81
82     printf("end: value %d\n", value);
83
84     return 0;
85 }

```


修改后:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include "common_threads.h"
5 #include <semaphore.h>
6 //
7 // Your code goes in the structure and functions below
8 //
9
10 typedef struct __rwlock_t {
11     sem_t lock;
12     sem_t write_lock;
13     int reader_number;
14 } rwlock_t;
15
16 //初始化锁
17 void rwlock_init(rwlock_t *rw) {
18     sem_init(&rw->lock, 0, 1);
19     sem_init(&rw->write_lock, 0, 1);
20     rw->reader_number = 0;
21 }
22
23 void rwlock_acquire_readlock(rwlock_t *rw) {
24     sleep(1);
25     sem_wait(&rw->lock); //获取访问reader_number的锁
26     rw->reader_number++;
27     if (rw->reader_number == 1) {
28         //第一个读者获取写锁, 防止该锁被写者获取
29         sem_wait(&rw->write_lock);
30     }
31     sem_post(&rw->lock); //释放访问reader_number的锁
32 }
33
34 void rwlock_release_readlock(rwlock_t *rw) {
35     sem_wait(&rw->lock); //获取访问reader_number的锁
36     rw->reader_number--;
37     if (rw->reader_number == 0) {
38         //最后一个读者释放写锁
39         sem_post(&rw->write_lock);
40     }
41     sem_post(&rw->lock); //释放访问reader_number的锁
42 }
43
44 void rwlock_acquire_writelock(rwlock_t *rw) {
45     sleep(1);
46     sem_wait(&rw->write_lock); //写者获取写锁
47 }
48
49 void rwlock_release_writelock(rwlock_t *rw) {
50     sem_post(&rw->write_lock); //写者释放写锁
51 }
52 //
53 //
54 // Don't change the code below (just use it!)
55 //
56
```

```
57 int loops;
58 int value = 0;
59
60 rwlock_t lock;
61
62 void *reader(void *arg) {
63     int i;
64     for (i = 0; i < loops; i++) {
65         rwlock_acquire_readlock(&lock);
66         printf("read %d\n", value);
67         rwlock_release_readlock(&lock);
68     }
69     return NULL;
70 }
71
72 void *writer(void *arg) {
73     int i;
74     for (i = 0; i < loops; i++) {
75         rwlock_acquire_writelock(&lock);
76         value++;
77         printf("write %d\n", value);
78         rwlock_release_writelock(&lock);
79     }
80     return NULL;
81 }
82
83 int main(int argc, char *argv[]) {
84     assert(argc == 4);
85     int num_readers = atoi(argv[1]);
86     int num_writers = atoi(argv[2]);
87     loops = atoi(argv[3]);
88     pthread_t pr[num_readers], pw[num_writers];
89     rwlock_init(&lock);
90     printf("begin\n");
91     int i;
92     for (i = 0; i < num_readers; i++)
93         Pthread_create(&pr[i], NULL, reader, NULL);
94     for (i = 0; i < num_writers; i++)
95         Pthread_create(&pw[i], NULL, writer, NULL);
96
97     for (i = 0; i < num_readers; i++)
98         Pthread_join(pr[i], NULL);
99     for (i = 0; i < num_writers; i++)
100         Pthread_join(pw[i], NULL);
101
102     printf("end: value %d\n", value);
103     return 0;
104 }
105
```

执行如下指令编译代码

gcc -o reader-writer reader-writer.c -Wall -pthread

执行如下指令测试代码

./reader-writer 5 5 10


```

begin
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
read 8
read 8
read 8
read 8
write 9
read 9
write 10
write 11
read 11
read 11
read 11
read 11
write 12
write 13
write 14
write 15
read 15
read 15
read 15
read 15
read 15
write 16
write 17
write 18
write 19
write 20
read 20
write 21
write 22
write 23
write 24
read 24
read 24
read 24
write 25
write 26
write 27
write 28
read 28
read 28
write 29
read 29
write 30
read 30
read 30
read 30
write 31
write 32
read 32
write 33
write 34
read 34
write 35
read 35
write 36
write 37
write 38
write 39
write 40
read 40
read 40
read 40
read 40
write 40
write 41
write 42
write 43
write 44
write 45
read 45
read 45
read 45
read 45
read 45
write 46
write 47
read 47
read 47
read 47
read 47
read 47
write 48
write 49
write 50
end: value 50

```

运行很快，几乎无停顿，结果符合预期，但是读写的操作分布不均，过于集中。

5.让我们再次看一下读者写者问题，但这一次需要考虑进程饥饿。 您如何确保所有读者和写者运行？ 有关详细信息，请参见 `reader-writer-nostarve.c`。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include "common_threads.h"
5
6 //
7 // Your code goes in the structure and functions below
8 //
9
10 typedef struct __rwlock_t {
11 } rwlock_t;
12
13
14 void rwlock_init(rwlock_t *rw) {
15 }
16
17 void rwlock_acquire_readlock(rwlock_t *rw) {
18 }
19
20 void rwlock_release_readlock(rwlock_t *rw) {
21 }
22
23 void rwlock_acquire_writelock(rwlock_t *rw) {
24 }
25
26 void rwlock_release_writelock(rwlock_t *rw) {
27 }
28
29 //
30 // Don't change the code below (just use it!)
31 //
32
33 int loops;
34 int value = 0;
35
36 rwlock_t lock;
37
38 void *reader(void *arg) {
39     int i;
40     for (i = 0; i < loops; i++) {
41         rwlock_acquire_readlock(&lock);
42         printf("read %d\n", value);
43         rwlock_release_readlock(&lock);
44     }
45     return NULL;
46 }
47
48 void *writer(void *arg) {
49     int i;
50     for (i = 0; i < loops; i++) {
51         rwlock_acquire_writelock(&lock);
52         value++;
53         printf("write %d\n", value);
54         rwlock_release_writelock(&lock);
55     }
56     return NULL;
57 }
58
59 int main(int argc, char *argv[]) {
60     assert(argc == 4);
61     int num_readers = atoi(argv[1]);
62     int num_writers = atoi(argv[2]);
63     loops = atoi(argv[3]);
64
65     pthread_t pr[num_readers], pw[num_writers];
66
67     rwlock_init(&lock);
68
69     printf("begin\n");
70
71     int i;
72     for (i = 0; i < num_readers; i++)
73         Pthread_create(&pr[i], NULL, reader, NULL);
74     for (i = 0; i < num_writers; i++)
75         Pthread_create(&pw[i], NULL, writer, NULL);
76
77     for (i = 0; i < num_readers; i++)
78         Pthread_join(pr[i], NULL);
79     for (i = 0; i < num_writers; i++)
80         Pthread_join(pw[i], NULL);
81
82     printf("end: value %d\n", value);
83
84     return 0;
85 }

```

本题需要解决读者写者锁中写者可能饿死的问题。写者可能饿死，是因为读者数量不受限制，同一时刻可以有多个读者进行读，而只要有读者，写者就不能获取写锁。

为了解决这个问题，可以再增加一个信号量实现一个锁 `write_waiting_lock`，一旦有写者准备进行写操作，尝试获取该锁，获取该锁后可以使新的读者不能进行读，直到写者获取写锁为止，这样就可以在获取锁后限制读者的数量，写者能够保证在等待当前数量的读者读取数据后，可以进行写操作

修改后：

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <semaphore.h>
5 // Your code goes in the structure and functions below
6 //
7 //
8 //
9
10 typedef struct _rwlock_t {
11     sem_t lock;
12     sem_t write_lock;
13     sem_t write_waiting;
14     // 读锁定义等待锁
15     // 与读者获取写锁时先获取等待锁，禁止新读者加入
16     int reader_number;
17 } rwlock_t;
18
19
20 void rwlock_init(rwlock_t *rw) {
21     sem_init(&rw->lock, 0, 1);
22     sem_init(&rw->write_lock, 0, 1);
23     sem_init(&rw->write_waiting, 0, 1);
24     rw->reader_number = 0;
25 }
26
27 void rwlock_acquire_readlock(rwlock_t *rw) {
28     sleep(1);
29     sem_wait(&rw->write_waiting); // 读者读取时必须先获得等待锁以证明此时没有写者存在，也就是自己不是用读写者的新变量
30     sem_wait(&rw->lock);
31     rw->reader_number++;
32     if (rw->reader_number == 1) {
33         sem_wait(&rw->write_lock);
34     }
35     sem_post(&rw->lock);
36     sem_post(&rw->write_waiting); // 读者获取了读锁，归还等待锁
37 }
38
39
40 void rwlock_release_readlock(rwlock_t *rw) {
41     sem_wait(&rw->lock);
42     rw->reader_number--;
43     if (rw->reader_number == 0) {
44         sem_post(&rw->write_lock);
45     }
46     sem_post(&rw->lock);
47 }
48
49
50 void rwlock_acquire_writelock(rwlock_t *rw) {
51     sleep(1);
52     sem_wait(&rw->write_waiting); // 写者先获得等待锁，禁止新读者加入
53     sem_wait(&rw->write_lock);
54     sem_post(&rw->write_waiting); // 写者已经获取写权限，归还等待锁
55 }
56

```

```

58 void rwlock_release_writelock(rwlock_t *rw) {
59     sem_post(&rw->write_lock);
60 }
61
62 //
63 // Don't change the code below (just use it!)
64 //
65
66 int loops;
67 int value = 0;
68
69 rwlock_t lock;
70
71 void *reader(void *arg) {
72     int i;
73     for (i = 0; i < loops; i++) {
74         rwlock_acquire_readlock(&lock);
75         printf("read %d\n", value);
76         rwlock_release_readlock(&lock);
77     }
78     return NULL;
79 }
80
81 void *writer(void *arg) {
82     int i;
83     for (i = 0; i < loops; i++) {
84         rwlock_acquire_writelock(&lock);
85         value++;
86         printf("write %d\n", value);
87         rwlock_release_writelock(&lock);
88     }
89     return NULL;
90 }
91
92 int main(int argc, char *argv[]) {
93     assert(argc == 4);
94     int num_readers = atoi(argv[1]);
95     int num_writers = atoi(argv[2]);
96     loops = atoi(argv[3]);
97
98     pthread_t pr[num_readers], pw[num_writers];
99
100     rwlock_init(&lock);
101
102     printf("begin\n");
103
104     int i;
105     for (i = 0; i < num_readers; i++)
106         pthread_create(&pr[i], NULL, reader, NULL);
107     for (i = 0; i < num_writers; i++)
108         pthread_create(&pw[i], NULL, writer, NULL);
109
110     for (i = 0; i < num_readers; i++)
111         pthread_join(pr[i], NULL);
112     for (i = 0; i < num_writers; i++)
113         pthread_join(pw[i], NULL);
114
115     printf("end: value %d\n", value);
116
117     return 0;
118 }

```

执行编译代码

gcc -o reader-writer-nostarve reader-writer-nostarve.c -Wall -pthread

执行运行指令

./reader-writer-nostarve 5 5 10

```

begin      read 22
read 0     write 23
read 0     read 23
read 0     read 23
read 0     write 24
write 1    read 24
read 1     write 25
write 2    read 25
write 3    read 25
write 4    write 26
write 5    write 27
write 6    write 28
read 6     read 28
read 6     read 28
read 6     read 28
read 6     write 29
read 6     read 29
write 7    write 30
write 8    read 30
write 9    write 31
write 10   read 31
write 11   write 32
write 12   read 32
read 12    read 32
write 13   read 32
read 13    write 33
write 14   write 34
write 15   write 35
read 15    write 36
read 15    read 36
read 15    read 36
write 16   write 37
write 17   write 38
read 17    read 38
write 18   read 38
read 18    write 39
write 19   write 40
write 20   read 40
read 20    write 41
read 20    write 42
read 20    read 42
write 21   read 42
write 22   write 43
read 43
write 44
read 44
write 45
read 45
read 45
read 45
read 45
read 45
write 46
write 47
write 48
write 49
write 50
read 50
read 50
read 50
end: value 50

```

读取和写入均衡很多，解决了饿死的问题。

6. 使用信号量构建一个没有饥饿的互斥量，其中任何试图获取该互斥量的线程都将最终获得它。 有关更多信息，请参见 `mutex-nostarve.c` 中的代码。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5 #include "common_threads.h"
6
7 //
8 // Here, you have to write (almost) ALL the code. Oh no!
9 // How can you show that a thread does not starve
10 // when attempting to acquire this mutex you build?
11 //
12
13 typedef __ns_mutex_t {
14 } ns_mutex_t;
15
16 void ns_mutex_init(ns_mutex_t *m) {
17 }
18
19 void ns_mutex_acquire(ns_mutex_t *m) {
20 }
21
22 void ns_mutex_release(ns_mutex_t *m) {
23 }
24
25
26 void *worker(void *arg) {
27     return NULL;
28 }
29
30 int main(int argc, char *argv[]) {
31     printf("parent: begin\n");
32     printf("parent: end\n");
33     return 0;
34 }

```

这道题目参考了网上的做法，实现了一个无饥饿（starvation-free）的互斥量，使用了信号量来构建。在这个互斥量中，任何试图获取锁的线程最终都会获得它，从而避免了饥饿现象。下面是这个互斥量的原理：

互斥量中有三个房间：room1、room2 和 room3。

room1 用于存放等待获取锁的线程。

room2 用于存放等待进入 room3 的线程。

room3 是一个隐式的房间，同一时间只能有一个线程进入，相当于一个互斥锁。

过程：

当线程想要获取互斥量时，首先进入 room1，表示它想要获取锁。

之后，线程会检查是否有其他线程正在等待获取锁（即 room1 中是否还有线程），如果有，则会随机唤醒一个线程，使其进入 room2。

线程进入 room2 后，会再次检查 room1 是否还有线程在等待。如果有，线程会继续等待在 room2，直到 room1 中没有线程为止。

当 room1 中没有线程时，表示当前线程是最后一个等待获取锁的线程，它会唤醒一个在 room2 中等待的线程，使其进入 room3，并释放锁。

线程在执行完临界区代码后，会再次检查 room2 是否还有线程在等待。如果有，线程会唤醒一个线程进入 room3，否则会重置整个互斥量的状态，使得下一批线程可以获取锁。

这样，就保证了任何试图获取锁的线程最终都能成功获取，并且避免了饥饿现象的发生。

实现代码的主要函数：

ns_mutex_acquire: 获取互斥量的函数。线程首先进入 room1，然后等待其他线程释放锁，并通知线程可以进入 room2。进入 room2 后，再次等待其他线程释放锁，并通知可以进入 room3，最终获取锁。

ns_mutex_release: 释放互斥量的函数。线程执行完临界区代码后，检查是否还有其他线程在等待获取锁，如果有，则唤醒一个线程进入 room3，否则重新开始互斥量的状态。

这种方式确保了线程获取锁的公平性，避免了某些线程长时间无法获取锁的情况，从而实现了无饥饿的效果。

修改后：

```

1 #include "common_threads.h"
2 #include <pthread.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6
7 //
8 // Here, you have to write (almost) ALL the code, oh no!
9 // How can you show that a thread does not starve
10 // when attempting to acquire this mutex you build?
11 //
12
13 typedef struct __ns_mutex_t
14 {
15     int room1; // 请求线程列表, lock信号量为锁
16     int room2; // 等待线程列表, s1信号量为锁
17     sem_t s1;
18     sem_t s2;
19     sem_t lock;
20 } ns_mutex_t;
21
22 void ns_mutex_init(ns_mutex_t *m)
23 {
24     sem_init(&m->s1, 0, 1); // s1初始值为1, 允许1个线程进入修改room2
25     sem_init(&m->s2, 0, 0); // s2初始值为0, 等待获取room2能进入room3的信号
26     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
27     m->room1 = 0;
28     m->room2 = 0;
29 }
30
31 void ns_mutex_acquire(ns_mutex_t *m)
32 {
33     // 进入room1
34     sem_wait(&m->lock); // 等待进入room1
35     m->room1++; // 进入room1
36     sem_post(&m->lock);
37
38     // 等待点1: room1
39
40     // 离开room1, 进入room2
41     // 这里同时占有两个锁
42     sem_wait(&m->s1);
43     m->room2++;
44     sem_wait(&m->lock);
45     m->room1--;
46
47     // 将room1中的所有线程放入room2, 并打开room3的锁
48     if (m->room1)
49     {
50         sem_post(&m->lock);
51         sem_post(&m->s1);
52     }
53
54     else
55     {
56         // 若room1内无线程等待, 本线程是最后一个线程, 离开room3的锁, 允许room2的进入room3
57         sem_post(&m->lock);
58         sem_post(&m->s2);
59     }
60
61     // 等待点2: room2
62
63     // 离开room2, 进入room3 (也就是开始执行)
64     sem_wait(&m->s2);
65     m->room2--;
66
67     void ns_mutex_release(ns_mutex_t *m)
68     {
69         // 本线程执行完后, 放一个来自room3的线程
70         if (m->room2)
71         {
72             sem_post(&m->s2); // 打开room3的门
73         }
74         else
75         {
76             sem_post(&m->s1); // 恢复初始值, 重新开始上述流程
77         }
78     }
79
80     void *worker(void *arg)
81     {
82         return NULL;
83     }
84
85     int main(int argc, char *argv[])
86     {
87         printf("parent: begin\n");
88         printf("parent: end\n");
89         return 0;
90     }

```

执行如下命令

gcc -o mutex-nostarve mutex-nostarve.c -Wall -pthread

```

zllz@zllz-virtual-machine:~/ostep-homework/threads-sema$ ./mutex-nostarve
parent: begin
parent: end

```