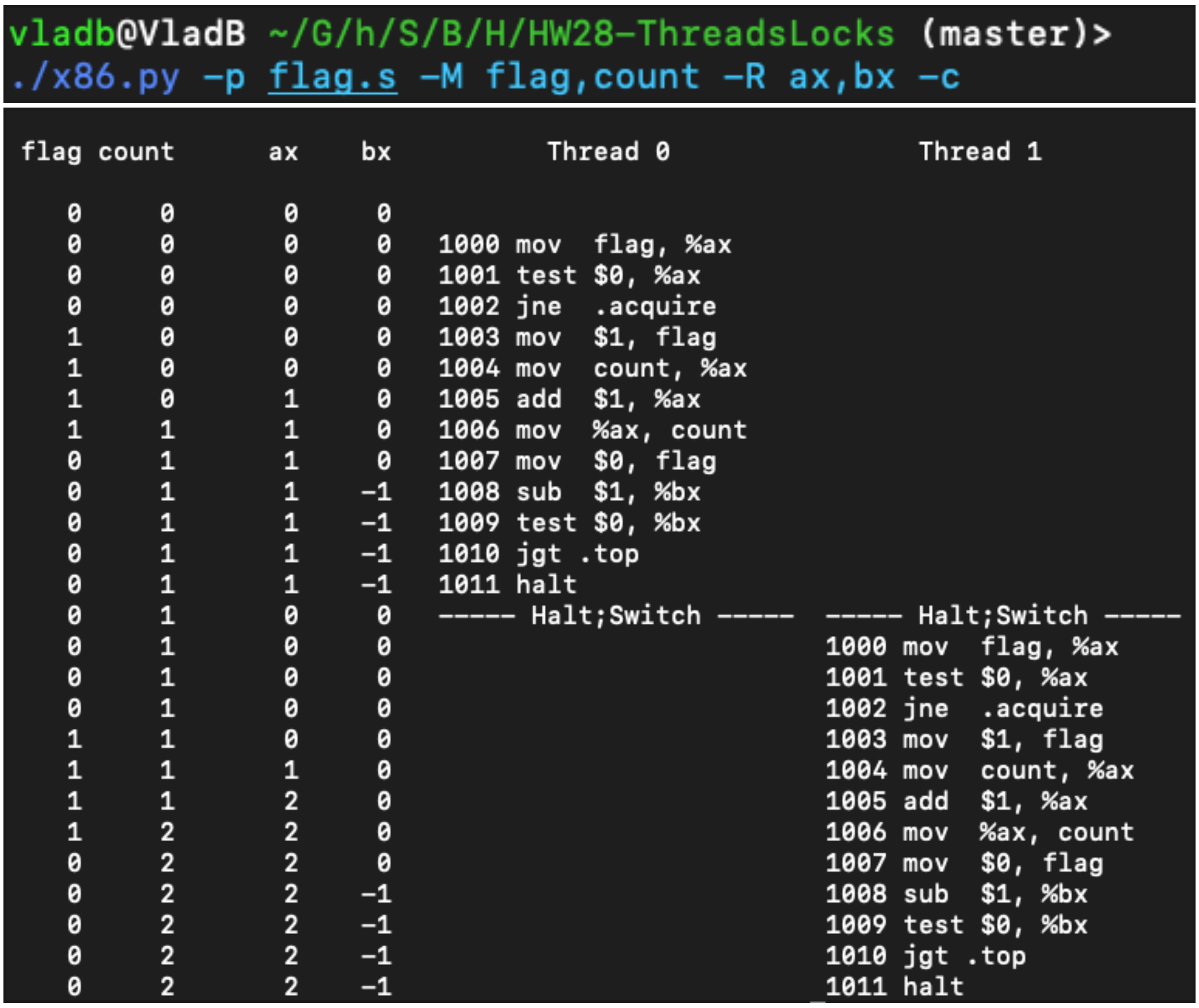
#uni/semester3/Betriebssysteme/chapter28/simulator-fragen

1. Examine flag.s. This code “implements” locking with a single memory flag. Can you understand the assembly?
   ➡ Yes, it shows the function if a simple spin lock with a flag.
   ➡ It has a race condition. If an interrupt occurs right before line 11. And next thread also acquires the lock and enters critical section and then interrupts. The first thread runs again and it also enters the critical section.

2. When you run with the defaults, does flag.s work? Use the -M and -R flags to trace variables and registers (and turn on -c to see their values). Can you predict what value will end up in flag?

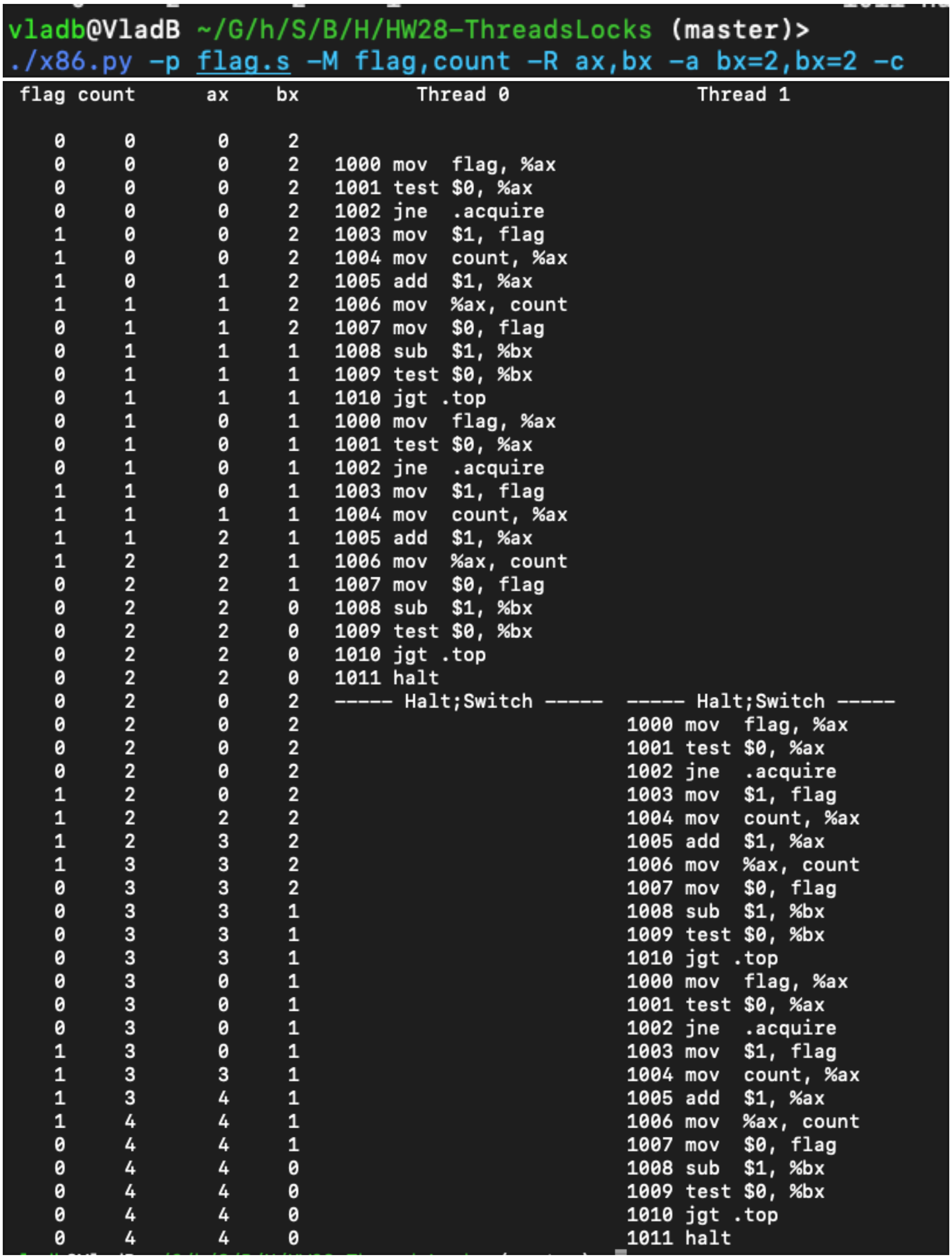


```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)>
./x86.py -p flag.s -M flag,count -R ax,bx -c
```

```
 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
```

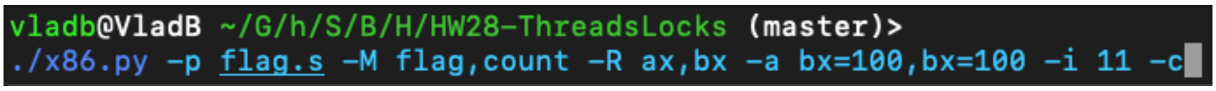
3. Change the value of the register %bx with the -a flag (e.g., -a bx=2,bx=2 if you are running just two threads). What does the code do? How does it change your answer for the question above?

➡ What changes is the value of count, namely 4, because each threads runs 2 times. Thus count is increased 4 times. Flag is still set to 0 at the end of a thread.



```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)>
./x86.py -p flag.s -M flag,count -R ax,bx -a bx=2,bx=2 -c
 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
```

4. Set bx to a high value for each thread, and then use the -I flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?



```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)>
./x86.py -p flag.s -M flag,count -R ax,bx -a bx=100,bx=100 -i 11 -c
```

➡ There are in total 11 operations. So -I 11 works, because it finishes one entire run, so no race condition can happen. Intervals < 11, lead to bad outcome. Intervals % 11 = 0, lead to good outcome.

➡ Good outcome means count=200 and flag is 0 at the end.

5. Now let's look at the program test-and-set.s. First, try to understand the code, which uses the xchg instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?

   ➡ In line 9 it sets mutex to old value of ax (which was 1, in line 8) and gets old value of mutex into ax. This happens atomically. If old value of mutex was 0, the it can enter critical section. If not equal zero, it goes spinning.

   ➡ In line 19 it moves 0 into mutex.

6. Now run the code, changing the value of the interrupt interval (-i) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?

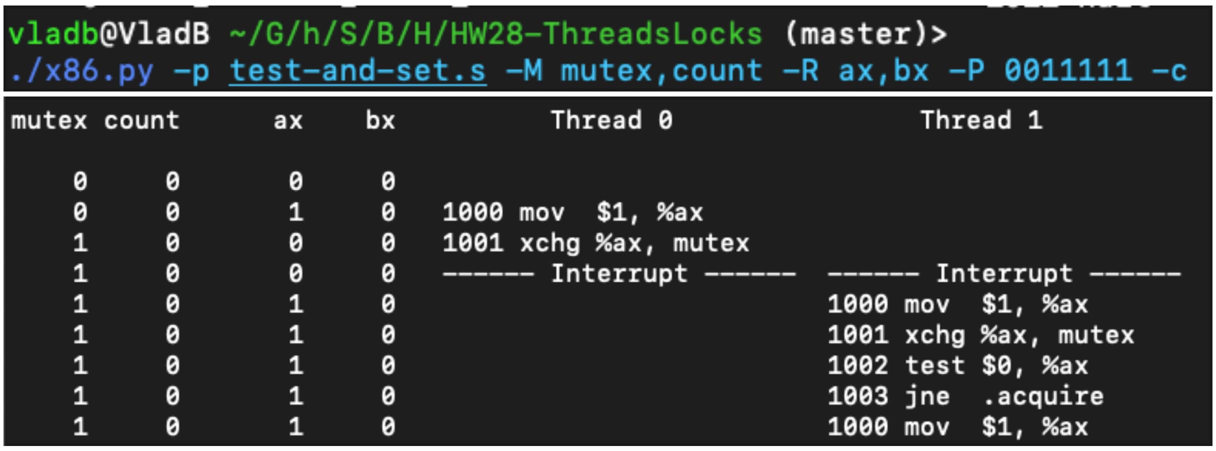
   ➡ Yes, this time all intervals lead to a good outcome. Mutex is 0 at the end.

   ➡ Inefficient use of the CPU means spinning and checking for the mutex variable. To quantify this for a thread, I would count all the times lines 8,9,10 and 11 were executed. Then I would do:

   ➡ count - 4 * loops

   ➡ Every loop will go through this lines, but not every loop will go multiple times through them. So counting these extra executions, shows the inefficiency.

7. Use the -P flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?



```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)>
./x86.py -p test-and-set.s -M mutex,count -R ax,bx -P 0011111 -c
```

```
mutex count      ax    bx           Thread 0                Thread 1

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

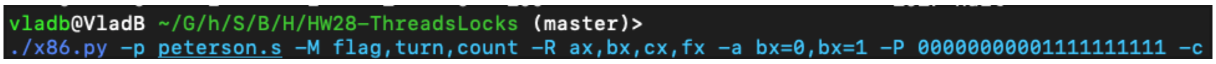
➡ The right thing happens, Thread 1 doesn't enter critical section, because Thread 0 grabbed the lock first.

➡ One should also test fairness and performance of the implementation of lock.

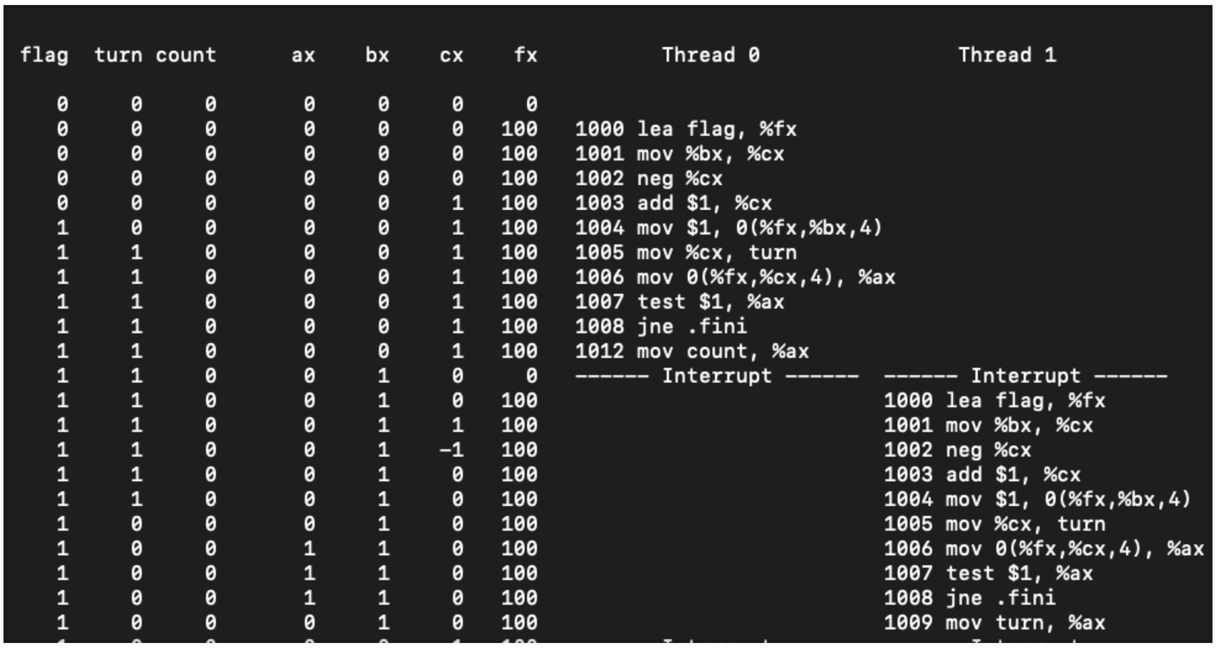
8. Now let's look at the code in peterson.s, which implements Peterson's algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.
   ➡ Most important lines are 33-36. This second condition allows thread 0 to enter the critical section, although flag of thread 1 is 1. This check 'turn == 1 - self' avoids a deadlock. Imagine that T0 runs and interrupt comes after line 26. T1 sets its flag to 1 and also changes turn. T1 can't enter critical section, because T0's flag is 1, and turn equals to T0. It then goes back to T0, which enters the critical section BECAUSE T1 changed turn to T0.

9. Now run the code with different values of -i. What kinds of different behavior do you see? Make sure to set the thread IDs appropriately (using -a bx=0,bx=1 for example) as the code assumes it.
   ➡ T0 finsishes by a lot ahead of T1
   ➡ On interval 17 (count of all operations in the code), both threads don't even run .spin2
   ➡ Once you do for a thread: Interval is set to 5
      ➡ mov $1, 0(%fx,%bx,4)   # flag[self] = 1
      ➡ The other thread's .spin2 is going to run
   ➡ On interval 1, T1 runs line 28 till 36, three times
   ➡ As interval increases, there are less jumps in the code.

10. Can you control the scheduling (with the -P flag) to "prove" that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

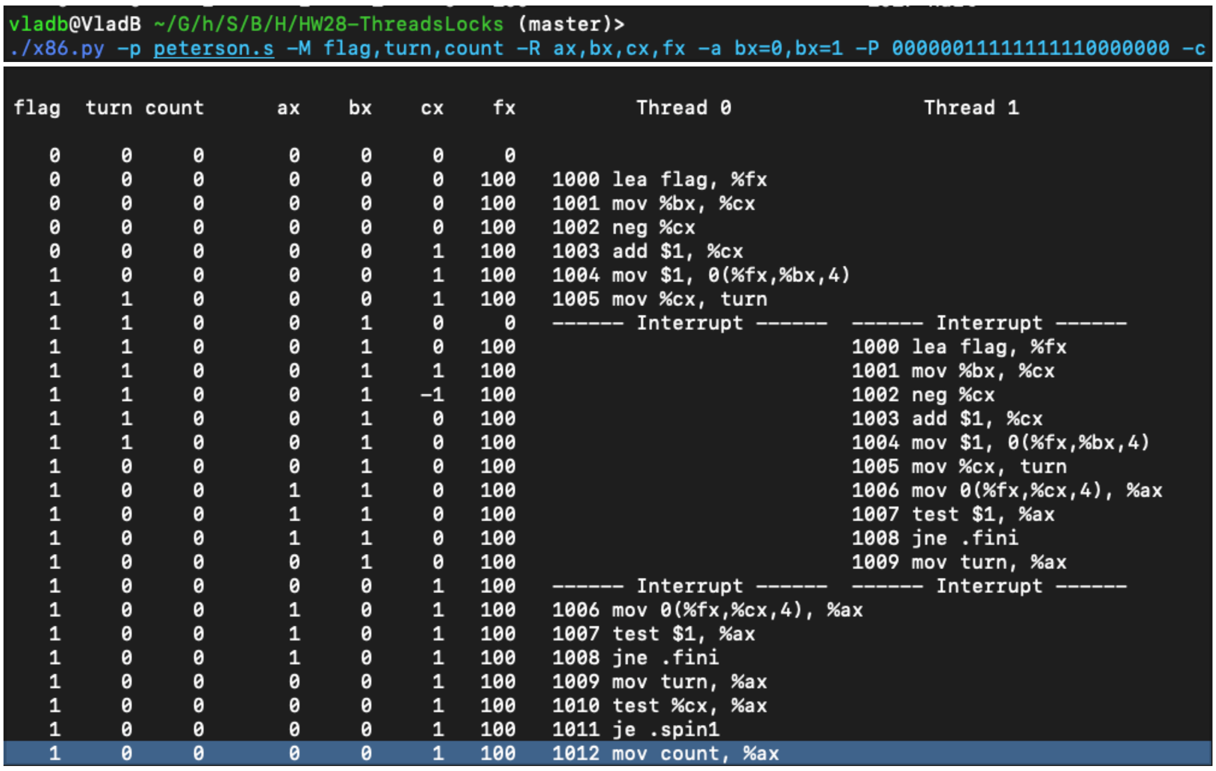


```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)>
./x86.py -p peterson.s -M flag,turn,count -R ax,bx,cx,fx -a bx=0,bx=1 -P 00000000001111111111 -c
```



```
flag  turn  count       ax     bx     cx     fx          Thread 0                Thread 1

   0     0     0         0      0      0      0
   0     0     0         0      0      0    100   1000 lea flag, %fx
   0     0     0         0      0      0    100   1001 mov %bx, %cx
   0     0     0         0      0      0    100   1002 neg %cx
   0     0     0         0      0      1    100   1003 add $1, %cx
   1     0     0         0      0      1    100   1004 mov $1, 0(%fx,%bx,4)
   1     1     0         0      0      1    100   1005 mov %cx, turn
   1     1     0         0      0      1    100   1006 mov 0(%fx,%cx,4), %ax
   1     1     0         0      0      1    100   1007 test $1, %ax
   1     1     0         0      0      1    100   1008 jne .fini
   1     1     0         0      0      1    100   1012 mov count, %ax
   1     1     0         0      1      0      0   ------ Interrupt ------  ------ Interrupt ------
   1     1     0         0      1      0    100                           1000 lea flag, %fx
   1     1     0         0      1      1    100                           1001 mov %bx, %cx
   1     1     0         0      1     -1    100                           1002 neg %cx
   1     1     0         0      1      0    100                           1003 add $1, %cx
   1     1     0         0      1      0    100                           1004 mov $1, 0(%fx,%bx,4)
   1     0     0         0      1      0    100                           1005 mov %cx, turn
   1     0     0         1      1      0    100                           1006 mov 0(%fx,%cx,4), %ax
   1     0     0         1      1      0    100                           1007 test $1, %ax
   1     0     0         1      1      0    100                           1008 jne .fini
   1     0     0         0      1      0    100                           1009 mov turn, %ax
```

➡ mutual exclusion works. For each thread there are 10 executed operations. T0 last operation is in critical section. T1 last operation is in .spin2



```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)>
./x86.py -p peterson.s -M flag,turn,count -R ax,bx,cx,fx -a bx=0,bx=1 -P 00000011111111110000000 -c
```

```
flag  turn  count       ax     bx     cx     fx          Thread 0                Thread 1

   0     0     0         0      0      0      0
   0     0     0         0      0      0    100   1000 lea flag, %fx
   0     0     0         0      0      0    100   1001 mov %bx, %cx
   0     0     0         0      0      0    100   1002 neg %cx
   0     0     0         0      0      1    100   1003 add $1, %cx
   1     0     0         0      0      1    100   1004 mov $1, 0(%fx,%bx,4)
   1     1     0         0      0      1    100   1005 mov %cx, turn
   1     1     0         0      1      0      0   ------ Interrupt ------  ------ Interrupt ------
   1     1     0         0      1      0    100                           1000 lea flag, %fx
   1     1     0         0      1      1    100                           1001 mov %bx, %cx
   1     1     0         0      1     -1    100                           1002 neg %cx
   1     1     0         0      1      0    100                           1003 add $1, %cx
   1     1     0         0      1      0    100                           1004 mov $1, 0(%fx,%bx,4)
   1     0     0         0      1      0    100                           1005 mov %cx, turn
   1     0     0         1      1      0    100                           1006 mov 0(%fx,%cx,4), %ax
   1     0     0         1      1      0    100                           1007 test $1, %ax
   1     0     0         1      1      0    100                           1008 jne .fini
   1     0     0         0      1      0    100                           1009 mov turn, %ax
   1     0     0         0      0      1    100   ------ Interrupt ------  ------ Interrupt ------
   1     0     0         1      0      1    100   1006 mov 0(%fx,%cx,4), %ax
   1     0     0         1      0      1    100   1007 test $1, %ax
   1     0     0         1      0      1    100   1008 jne .fini
   1     0     0         0      0      1    100   1009 mov turn, %ax
   1     0     0         0      0      1    100   1010 test %cx, %ax
   1     0     0         0      0      1    100   1011 je .spin1
   1     0     0         0      0      1    100   1012 mov count, %ax
```

➡ deadlock prevention works. T0 runs for 6 operations. T1 for 10 operations. T0 for 7 operations. At the end T0 arrives in the critical section and doesn't end up spinning, which means that the deadlock was avoided.

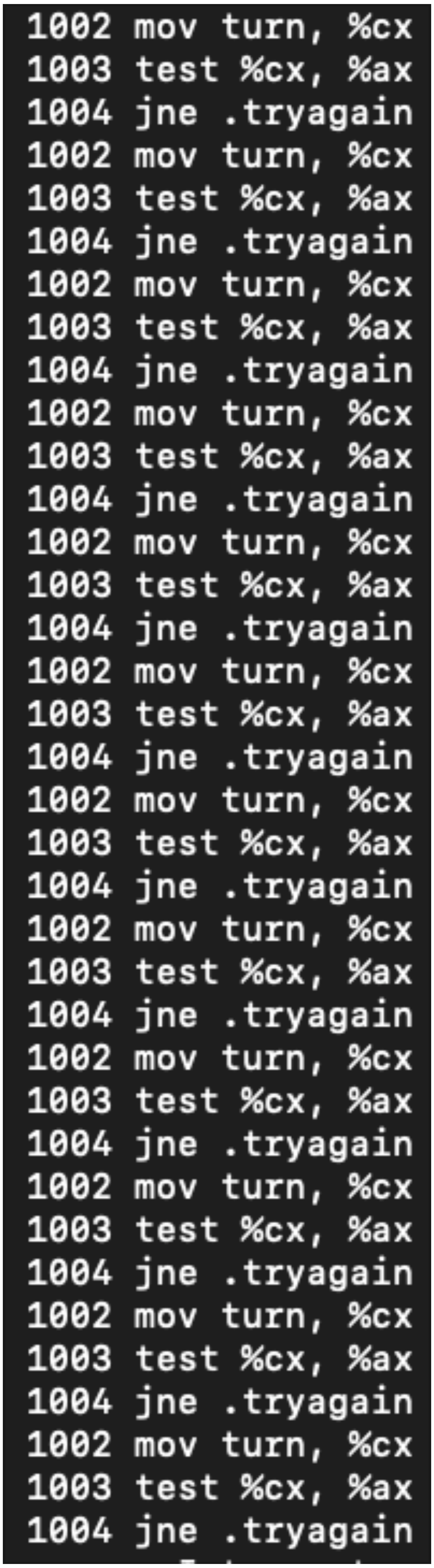
11. Now study the code for the ticket lock in ticket.s. Does it match the code in the chapter? Then run with the following flags: -a bx=1000,bx=1000 (causing each thread to loop through the critical section 1000 times). Watch what happens; do the threads spend much time spin-waiting for the lock?

➡ Yes, it it as in the the chapter!

```
./x86.py -p ticket.s -M ticket,turn,count -R ax,bx,cx -a bx=1000,bx=1000 -c
```

➡ After releasing the lock, the turn is incremented. This means that the same thread cannot obtain the lock again and it starts spinning. This forces thread 0 to giving thread 1 the lock and vice versa.

➡ Yes, some loops spend some time.



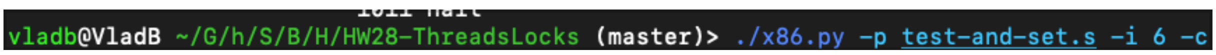
```
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
1002 mov turn, %cx
1003 test %cx, %ax
1004 jne .tryagain
```

12. How does the code behave as you add more threads?

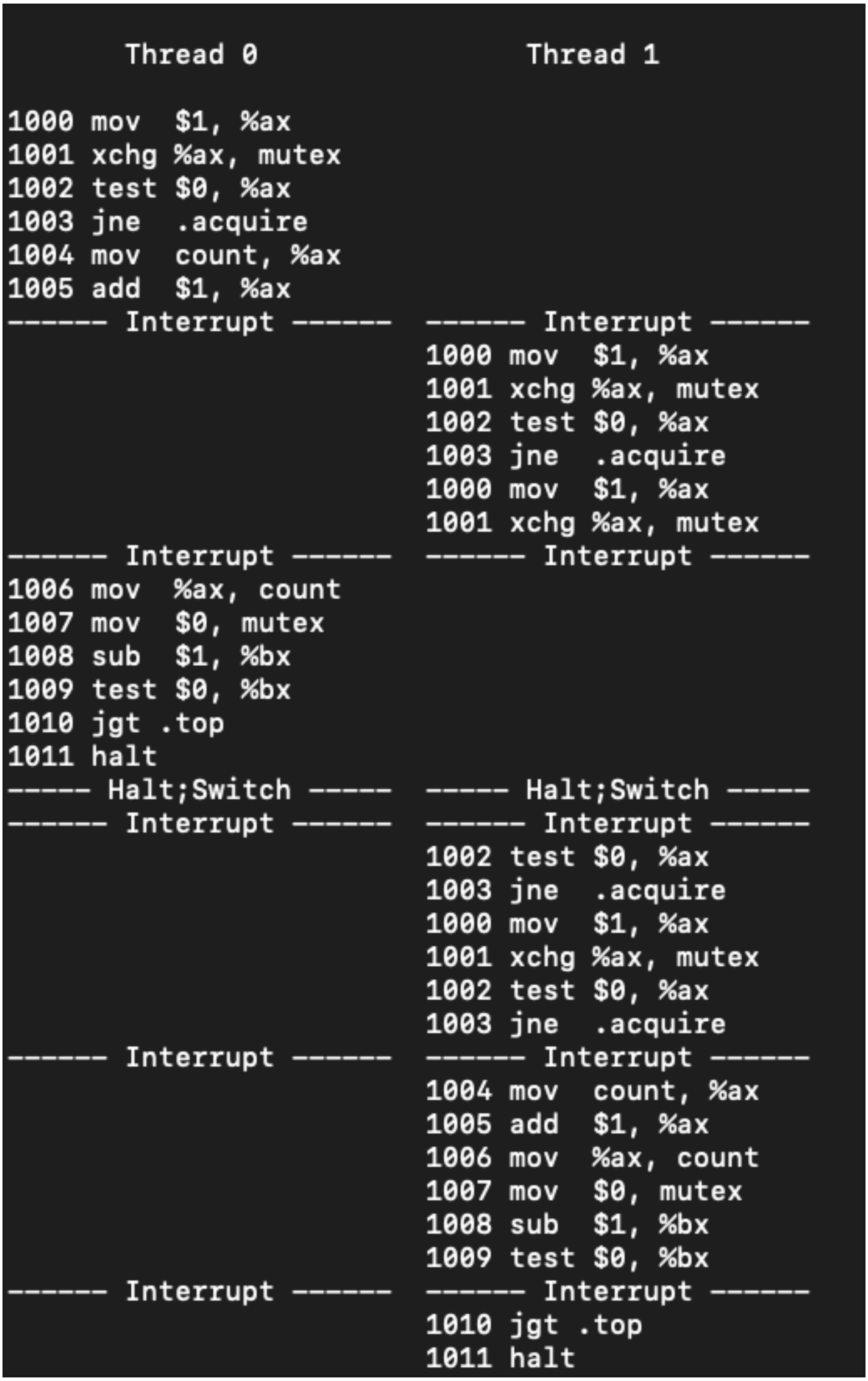
```
./x86.py -p ticket.s -M ticket,turn,count -R ax,bx,cx -t 10 -i 4 -c
```

➜ spinning still happening

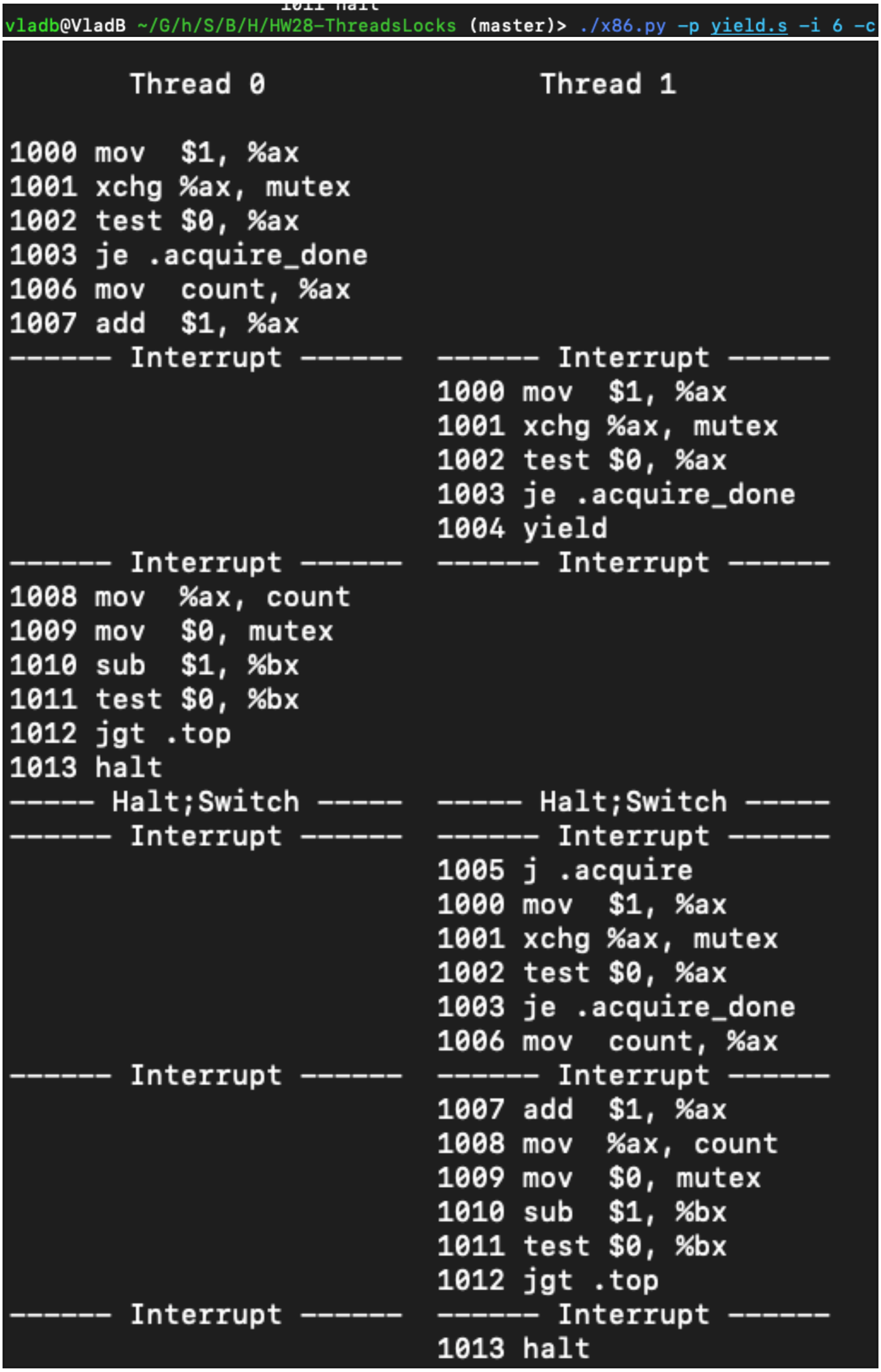
13. Now examine yield.s, in which a yield instruction enables one thread to yield control of the CPU (realistically, this would be an OS primitive, but for the simplicity, we assume an instruction does the task). Find a scenario where test-and-set.s wastes cycles spinning, but yield.s does not. How many instructions are saved? In what scenarios do these savings arise?



```
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)> ./x86.py -p test-and-set.s -i 6 -c
```



```
       Thread 0                Thread 1

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



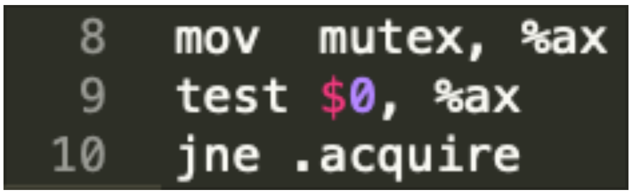
```
1011 halt
vladb@VladB ~/G/h/S/B/H/HW28-ThreadsLocks (master)> ./x86.py -p yield.s -i 6 -c

       Thread 0                Thread 1

1000 mov  $1, %ax
1001 xchg %ax, mutex
1002 test $0, %ax
1003 je .acquire_done
1006 mov  count, %ax
1007 add  $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1000 mov  $1, %ax
                         1001 xchg %ax, mutex
                         1002 test $0, %ax
                         1003 je .acquire_done
                         1004 yield
------ Interrupt ------  ------ Interrupt ------
1008 mov  %ax, count
1009 mov  $0, mutex
1010 sub  $1, %bx
1011 test $0, %bx
1012 jgt .top
1013 halt
----- Halt;Switch -----  ----- Halt;Switch -----
------ Interrupt ------  ------ Interrupt ------
                         1005 j .acquire
                         1000 mov  $1, %ax
                         1001 xchg %ax, mutex
                         1002 test $0, %ax
                         1003 je .acquire_done
                         1006 mov  count, %ax
------ Interrupt ------  ------ Interrupt ------
                         1007 add  $1, %ax
                         1008 mov  %ax, count
                         1009 mov  $0, mutex
                         1010 sub  $1, %bx
                         1011 test $0, %bx
                         1012 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1013 halt
```

➡ Two operations were saved.

➡ It saves operations in the scenario where test-and-set.s spins 3 or more times. (Or going through 1000,1001,1002,1003 ➡ 3 or more times)

14. Finally, examine test-and-test-and-set.s. What does this lock do? What kind of savings does it introduce as compared to test-and-set.s?

    ➡ it adds these lines:

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

    ➡ If mutex is 1 (lock is hold) then it starts directly spinning (this is the common case, where no much is done). It avoids doing expensive call like xchg, even if mutex and ax both are set to 1. In this case xchg doesn't do anything, so it makes sense to avoid it on this case.