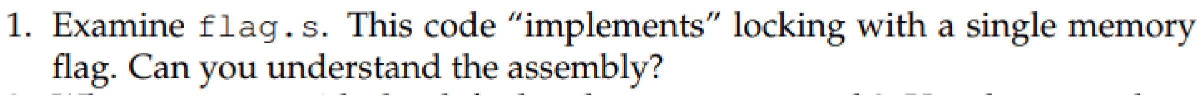
【OS】Day34

Class	Operating System: Three Easy Pieces
Date	@February 8, 2022

【Ch28】Locks Homework

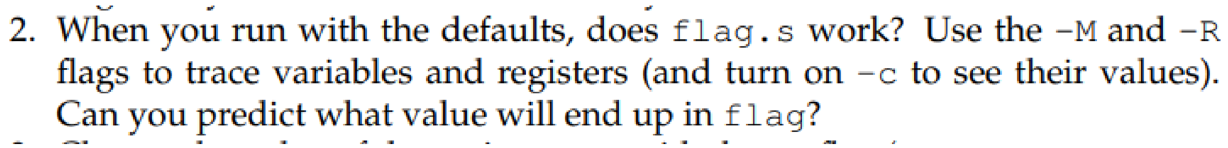
Question 1



1. Examine `flag.s`. This code “implements” locking with a single memory flag. Can you understand the assembly?

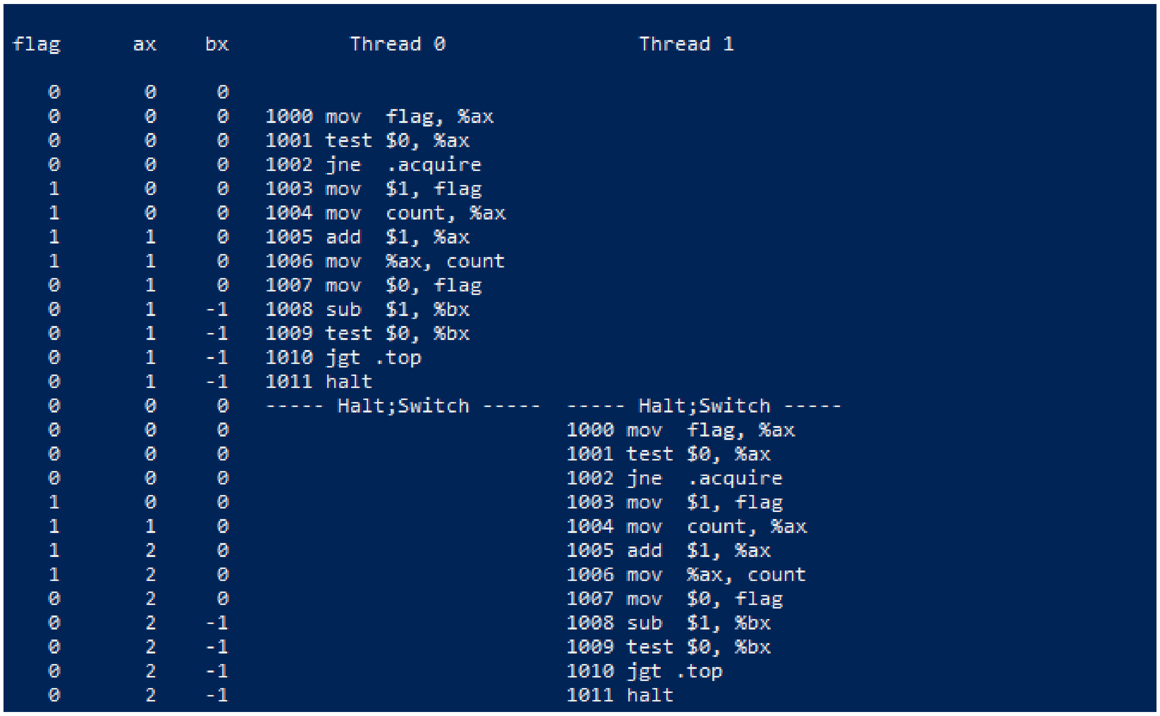
```
.var flag
.var count

.main
.top

.acquire
mov  flag, %ax      # get flag
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again
mov  $1, flag       # store 1 into flag

# critical section
mov  count, %ax     # get the value at the address
add  $1, %ax        # increment it
mov  %ax, count     # store it back

# release lock
mov  $0, flag       # clear the flag now

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Question 2

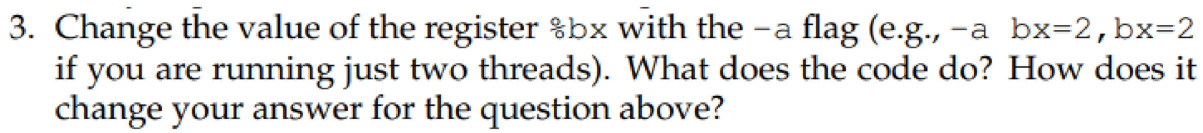

2. When you run with the defaults, does flag.s work? Use the -M and -R flags to trace variables and registers (and turn on -c to see their values). Can you predict what value will end up in flag?

The lock spin waits until the lock is free. In this case, since thread 1 executes after thread 0 finishes, thread 1 acquires the lock without spin-waiting.

flag is 0 when the program finishes.

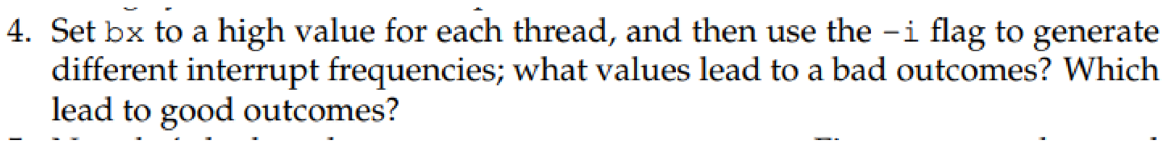

```
flag      ax    bx          Thread 0                Thread 1

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

Question 3


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?

Each thread will run two loops, flag is still 0.

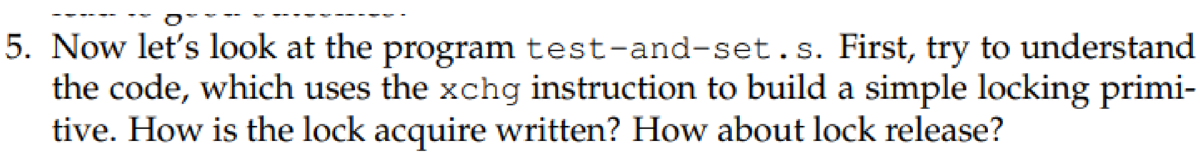
Question 4



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?

```
// bad outcomes
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 1-10,12,13,14,17

// god outcomes
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 11,15,16
```

Question 5



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?

```
.var mutex
.var count

.main
.top

# Test and set
.acquire
mov  $1, %ax
xchg %ax, mutex     # atomic swap of 1 and mutex
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again

# critical section
mov  count, %ax     # get the value at the address
add  $1, %ax        # increment it
mov  %ax, count     # store it back

# release lock
mov  $0, mutex

# see if we're still looping
sub  $1, %bx
test $0, %bx
```

```
jgt .top

halt
```