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

Yes, it uses flag to check critical section is locked or not. If it is 1, means lock, then keep looping, until the flag is 0.

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?

At T0 PC 1003, it put number 1 to flag(lock), and pc 1007 put 0 to the flag (unlock).
Also increment the count to 1
At T1 PC 1003, it put number 1 to flag(lock), and pc 1007 put 0 to the flag (unlock).
Also increment the count to 1

Eventually will 0 in flag when is 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?

Each thread will loop twice to increment the count. The flag will change 0->1->0->1 for first thread, and 0->1->0->1 for second thread.

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?

With bx=4, with -I 10, 11, 12 leads to good outcome, others will lead to bad outcome.

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?

Lock acquire, is by first move 1 into ax, then atomically put mutex value into ax, and put value in ax which is 1 into mutex, Then compare with 0 and value in ax. If it is 0 means lock is free, keep executing. If it is 1 then, jump back to .acquire.

Release the lock, simply put 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, the code is doing as we expected. it indeed lead to an inefficient use of the CPU, when one of the thread holding the lock, but the other thread without the key will keep spinning until the lock is free.

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?

```
./x86.py -p test-and-set.s -M mutex,count -a bx=1,bx=1 -c -i 5 -P 0011
```

Yes, the right thing happen.I also test the performance. With 001111111111

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.

This time it correctly increment the count.

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.

```
./x86.py -p eterson.s -M flag,count,turn -a bx=0,bx=1 -R bx,cx,ax -c -P10
```

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?

It does match to the code in the chapter. Yes, the threads spend much time spin waiting for the lock.

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

More thread will start spin and waiting for the lock to be available.

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?

At pc 1004, the thread yield to the OS to give up the CPU time.

```
./x86.py -p yield.s -c -i 3
./x86.py -p test-and-set.s -c -i 3
```

Save the one instruction.

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 add another while loop before xchg., to check it mutex is free or not. If it is not free, it won't proceed to xchg.