1.Let's examine a simple program, "loop.s". First, just read and understand it. Then, run it with these arguments (./x86.py -p loop.s -t 1 -i 100 -R dx) This specifies a single thread, an interrupt every 100 instructions, and tracing of register %dx. What will %dx be during the run? Use the -c flag to check your answers; the answers, on the left, show the value of the register (or memory value) after the instruction on the right has run.

At start the register ax has 0 in it. At instruction 1000, we decrement 1 on dx, thus, we got -1. At 1001 we run the test, only -1 is less than the 0 is set to 1 bit. At 1002, since -1 is less than 0 then jgte will keep executing reach instruction 1003, which end the thread.

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

2.Same code, different flags: (./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx) This specifies two threads, and initializes each %dx to 3. What values will %dx see? Run with -c to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?
The value in %dx will start from 3->2->1->0->-1 then end the one thread, same thing happen to thread2. No the presence of multiple threads didn't affect my calculations. And there is no race in this code.

3. Run this: ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx This makes the interrupt interval small/random; use different seeds (-s) to see different interleavings. Does the interrupt frequency change anything?

No it didn't change anything because two threads didn't share same variable(memory), and two threads don't share registers.

4.Now, a different program, looping-race-nolock.s, which accesses a shared variable located at address 2000; we'll call this variable value. Run it with a single thread to confirm your understanding: ./x86.py -p looping-race-nolock.s -t 1 -M 2000 What is value (i.e., at memory address 2000) throughout the run? Use -c to check.

The value is one since the only increment one time at ax register, then move back memory 2000.

```
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

5.Run with multiple iterations/threads: ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 Why does each thread loop three times? What is final value of value?

The reason for each thread three times because we set initial bx value to 3, thus, it need three iteration to decrement it's value until not grater to 0.

The final value will be 6, since first thread will increment ax register 3 times and put back to memory 2000. Then switch to second thread, it move 3 in 2000 to ax, and increment 3 times. Finally move back memory 2000 with value of 6.

6.Run with random interrupt intervals: ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0 with different seeds (-s 1, -s 2, etc.) Can you tell by looking at the thread interleaving what the final value of value will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?

Yes, for -s 0 the final value is 2, -s 2 is 1, -s 2 is 2.
Yes, the timing of the interrupt does matter.
As long as, the interrupt does not happen between:
? 1000 mov 2000, %ax
? 1001 add $1, %ax
? 1002 mov %ax, 2000
Because instruction 1000, instruction 1001, and instruction 1002 are critical section.

7. Now examine fixed interrupt intervals: ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 What will the final value of the shared variable value be? What about when you change -i 2, -i 3, etc.? For which interrupt intervals does the program give the "correct" answer?

The final value of the shared variable is 1, because the interrupts happen at critical sections.
-I 3 will have correct answer, because with -i3 the thread avoid interrupt at critical section.

8.Run the same for more loops (e.g., set -a bx=100). What interrupt intervals (-i) lead to a correct outcome? Which intervals are surprising?

The interrupt intervals number is multiple of 3 can lead to correct answer. -i 5 give me 160 really surprise me.

9.One last program: wait-for-me.s. Run: ./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000 This sets the %ax register to 1 for thread 0, and 0 for thread 1, and watches %ax and memory location 2000. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

For thread 0 since it's ax register value 1 then it start execute .signaller, then move 1 to memory 2, then halt. Switch to thread 1, since it's ax value is 0, then jump to .wait instructions, move 2000 content(which is 1) to cx, and check cx is equal to 1 or not. Since it is equal to 1 halt the thread.

The final value for memory 2000 is 1.

10.Now switch the inputs: ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000 How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., -i 1000, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?

The thread 0 will jump in to .wait since it's ax value is 0, and from here it start go to infinity loop, that constantly check memory 2000 value is 1 or not. If it is not 1 jump back to .wait. Until interrupt happen. Switch to thread 1, and thread 1 set memory 2000 to 1, then halt. Switch thread 0 again, but his time when it's check memory is 1 it halt.

For this situation it should change the interrupt interval frequency higher to avoid a lot of cpu time on infinity loop.

No this program not efficiently using the CPU.