Homework for Intro to Concurrency

1. **Instruction Run**: ./x86.py -p loop.s -t 1 -i 100 -R dx

   loop.s should subtract 1 from %dx, check to see if the result is negative, and if it isnt, repeat the subtraction until it is.

   Once I ran the "-c" command it cleared up some confusion I had regarding the way the output was formatted: it traces the register but prints it out *next to* a single instruction, not after an entire run through of the code. This is more verbose than I expected but better I guess for detailed inspection.

   So since there was only run through, it makes perfect sense that initial value was 0.

2. **Instruction Run**: ./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx

   Upon first seeing this I immediatley got hung up on the double initialization of the same register. But what I finally realized is that in a multi-threaded program, *every single thread has its own set of registers*. The shared portion of the thread is main memory, not registers or program counter, both of which preserved via context switches.

   Also note the interrupt every one hundred instructions is essentially meaningless here, as it takes far less than a hundred instructions to terminate.

   So to finally answer, the question, I predicted duplicate behavior by the threads, repeating the subtraction operation until %dx goes from 3 to -1...and that is exactly what happened.

3. Instruction run: ./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx

   No, it doesn't. And why would it? We're operating on pre-initialized registers that are unique to each thread, *not* to shared memory locations which are the crux of the issue. All that happens is the threads interleaved: otherwise the behavior is preserved.

4. **Instruction Run**: ./x86.py -p looping-race-nolock.s -t 1 -M 2000

   Looking at the 'cat' terminal output, the code moves the memory from location 2000 into register %ax, adds 1 to it, then moves it back. After this is done the code subtracts 1 from %bx, tests to see if %bx is 0 yet. If it isnt, it jumps back up to the top.

   Even without running the program it is is clear this code is inefficient. Rather than merely keeping the value inside the register and checking the condition there, it moves the value back to memory upon every iteration before checking the condition.

   I predicted that the value would be 1 after the run, since we hadn't explicitly set it with program arguments and the program only loops once. After checking this answer was confirmed.

5. **Instruction Run**: ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000

   Since each register is set separately by itself, i.e. each thread has its own set of registers, setting bx=3 will set the registers in *separate* threads to 3. Since bx is also the loop counter, and the loop decrements one at a time, both threads loop three times then terminate.

   The final result will be double the loop coutner since memory is shared between threads (unlike resiters), or 6.

6. **Instruction Run**: ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0

   The critical section is when the assembly code has moved the memory value into the register. The problem with having multiple threads access this is that they may both move the memory value into the register to be updated with no way of coordinating this updating, alter it in their own ways, and then move it back. This could lead to multiple issues...

   Note that the critical section is conveniently denoted in the comments given to us in the code...

7. **Instruction Run**: ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1

   Since the thread is interrupted every single instruction, it seems to me as though this should be equivalent to a single thread running tbh. In other words, in effect only one add operation will have been performed to x (so the final x value will be 1) whereas two might have been intended.

   and...running the c option confirms my hunch. I would guess that once the interval is set to take up the entire critical section in one go, or i = 3, then the final memory value will indeed be 2. G

   and...again my hunch is confirmed. The final result with $i = 3$ is that $x = 2$.

8. Basically, in order to get a correct result the interrupt frequency needs to be a multiple of 3, or the size of the critical section. Otherwise with the large number of loops, a thread will eventually be interrupted while in the critical section. Evidently, the *maximum* number will always be if the code runs correctly, or 200, and this cannot be exceeded by

the nature of the range condition. That is because the number of looping times is set in separate registers, so all that may be overwritten are the increments.

Of course, the multiple of three rule stops mattering once the interrupt level is set to be large enough so the first thread is allowed to terminate before it can be interrupted.

9. **Instruction Run**: ./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000

   From looking at the code, it seems as though depending upon the value of the register ax for the thread, the code is either a "waiter" (for ax = 0) or a "signaller" (for ax = 1). If it's a waiter thread, it moves the memory value located in address 2000 into register cx, tests it against 1, and if its not equal to that, repeats.

   The signaller essentially is the only way this operation can be terminated...once the signaller is run it finally moves the desired "1" value for the waiter into the 2000 memory address and the waiter can finish.

   With the given parameters, this won't even be a problem since the signaller is called "before" the waiter. Running the code we see that this is indeed the case and that eahc thread only runs once.

   Its final value will be 1: the value moved there by the signaller.

10. **Instruction Run**: ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000

    Fairly straightforward...thread 1 essentially "spins" or locks indefinitely until the signaller is called and the thread is interrupted. I'm not sure what the default interrupt rate is but as -i was not set in the instruction I surmise there exists one, and that was what allowed the signaller, or thread 2, to be called, and for the wait loop to be ended.

    Obviously the program is not efficiently using the CPU since it's redundantly checking address 2000 when nothing has changed, and it does so many times in a loop.