**Cpt 8**

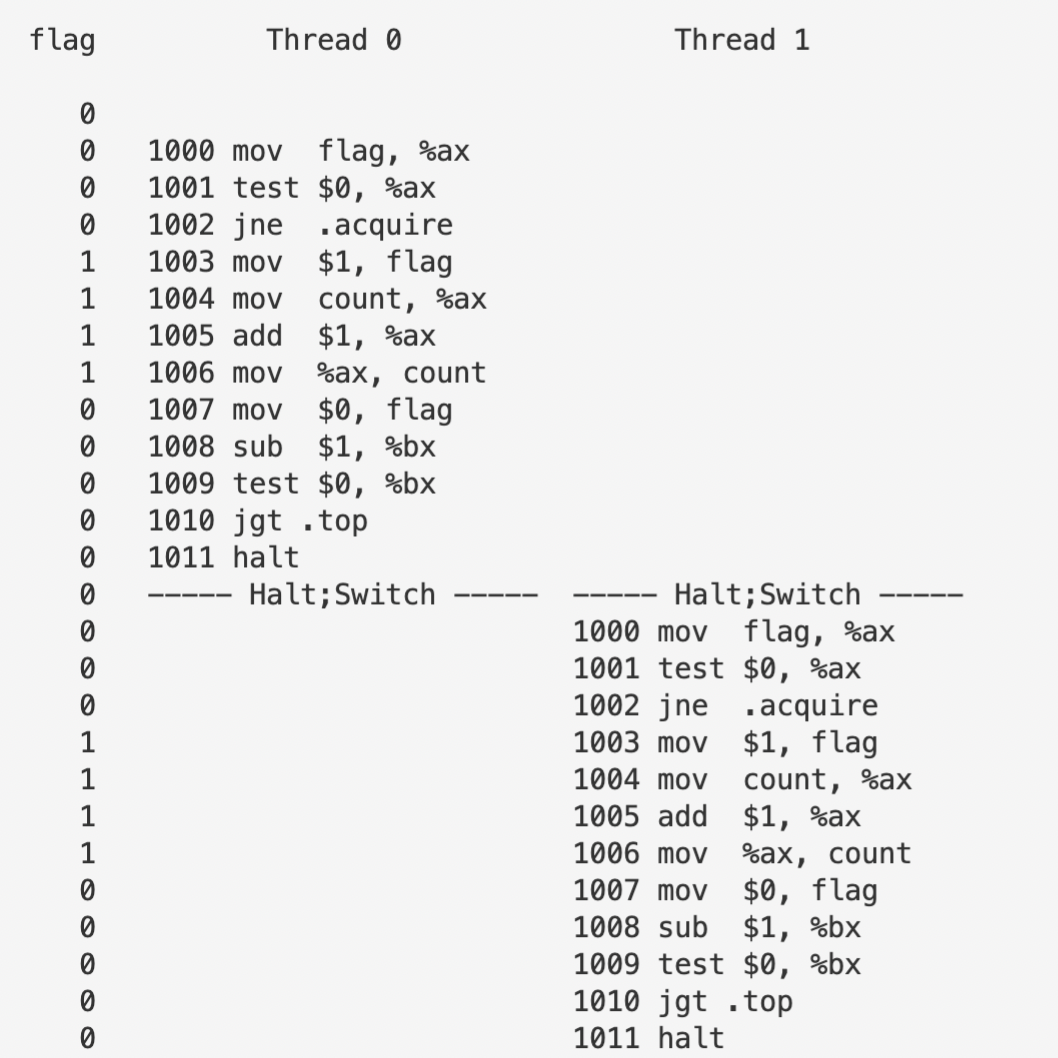
**Homework (Simulation)**

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

The first three lines tests if the lock is locked according to the value stored in flag. If we get 0, which means the lock is free, we can jump out of the loop and move on. If we get 1, it means the lock is occupied and we’ll have to loop and wait until it’s free. Critical section part adds the value in the targeted address by 1, and then set flag to 0 to set it free. Finally, we subtract the value stored in register %bx by 1 to see if the thread has finished its task. If not, we need to go to the top of the code and do everything all over again.

1. **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?**

With default settings, flag.s can do its job as we would expect it. Variable flag should end up being 0 because thread 1 should release the lock by setting flag 0 once it has finished its job. The result is shown in the screenshot below.

****

1. **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?**

This time the code will be executed twice in both threads. It still does its job properly, leaving the value in flag a satisfying 0.

1. **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 outcome? Which lead to good outcomes?**

Take the command below as an example:

python3 x86.py -p flag.s -a bx=5 -i 50 -M flag -c

It is shown in the output that thread 1 will be spinning, waiting for the lock to be opened for the entire second time slice due to its cutting in when thread 0 is trying to do its job and has had the lock locked. Since we suppose bx has a value high enough, the interrupt frequency should be synching with a complete cycle of the code, which means that it should be either equal to the length of the cyclic segment, which is 11, or a multiple of it. This should guarantee us that thread 1 will not cut in while thread 0 is executing, thus will not lead to thread 1’s spinning through the whole time slice. This could be proved by the outcome of the commands below:

python3 x86.py -p flag.s -a bx=5 -i 55 -M flag -c

python3 x86.py -p flag.s -a bx=5 -i 11 -M flag -c

1. **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?**

This program lets mutex exchange atomically with register ax which has constant value 1 so that mutex will always be 1 when the acquire part ends, regardless of its previous value. In this way, if mutex was 1 before the acquire part executed, i.e. the lock is occupied, it will still be 1 and the program will spin until it’s 0. On the other hand, if it was 0, the program will automatically set it 1, which represents the locking action. This embodies the meaning behind the naming of the program. The program releases the lock by manually setting it 0 when it has finished its job, which is pretty much the same as what flag.s does.

1. **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?**

Since the program still doesn’t fix the problem of thread 1 randomly cutting in when thread 0 has the lock, when interrupt interval isn’t a multiple of 5 thread 1 still could possibly spin throughout the whole interval. The output of the following command will prove this:

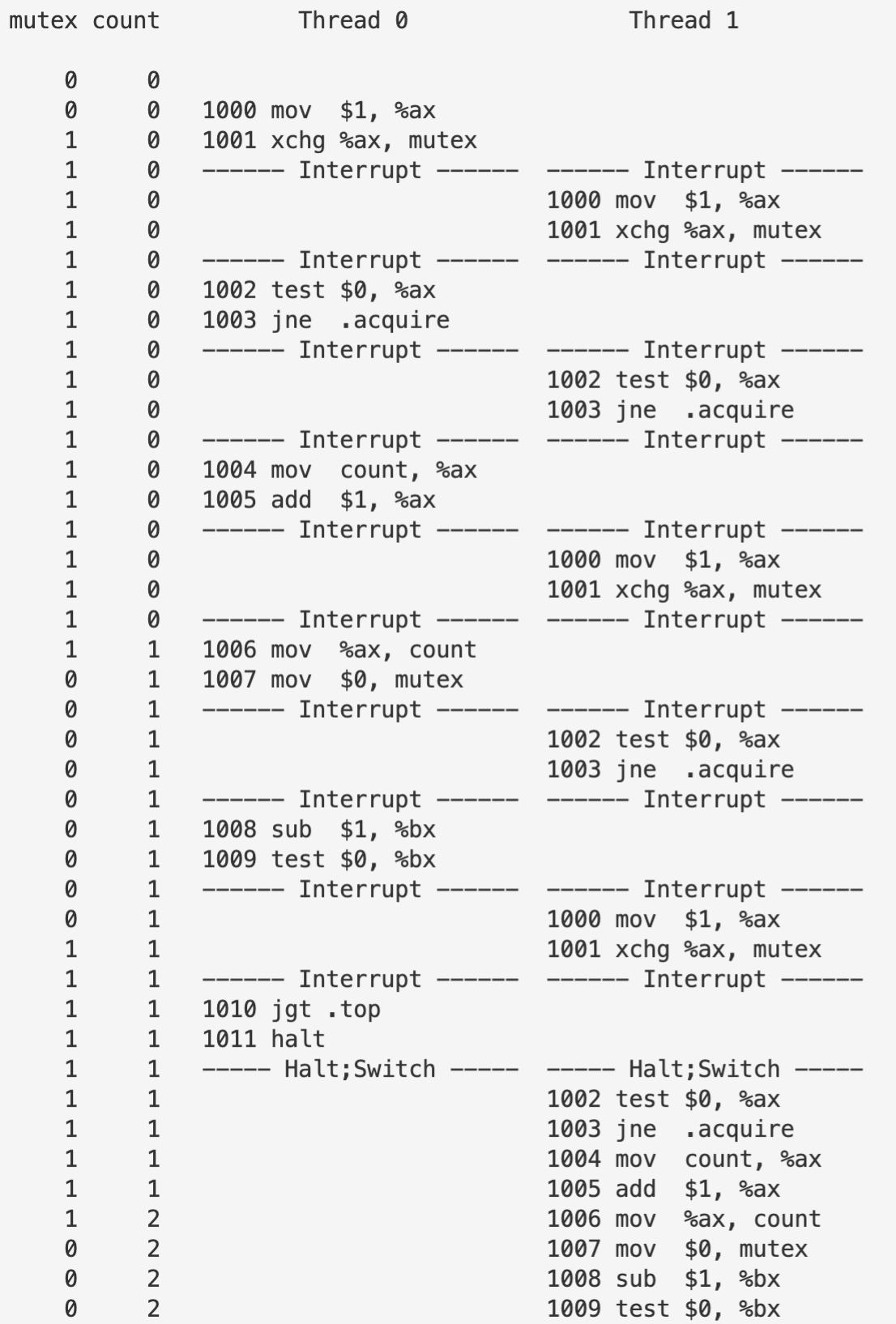
python3 x86.py -p test-and-set.s -a bx=5 -i 50 -M mutex -c

1. **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?**

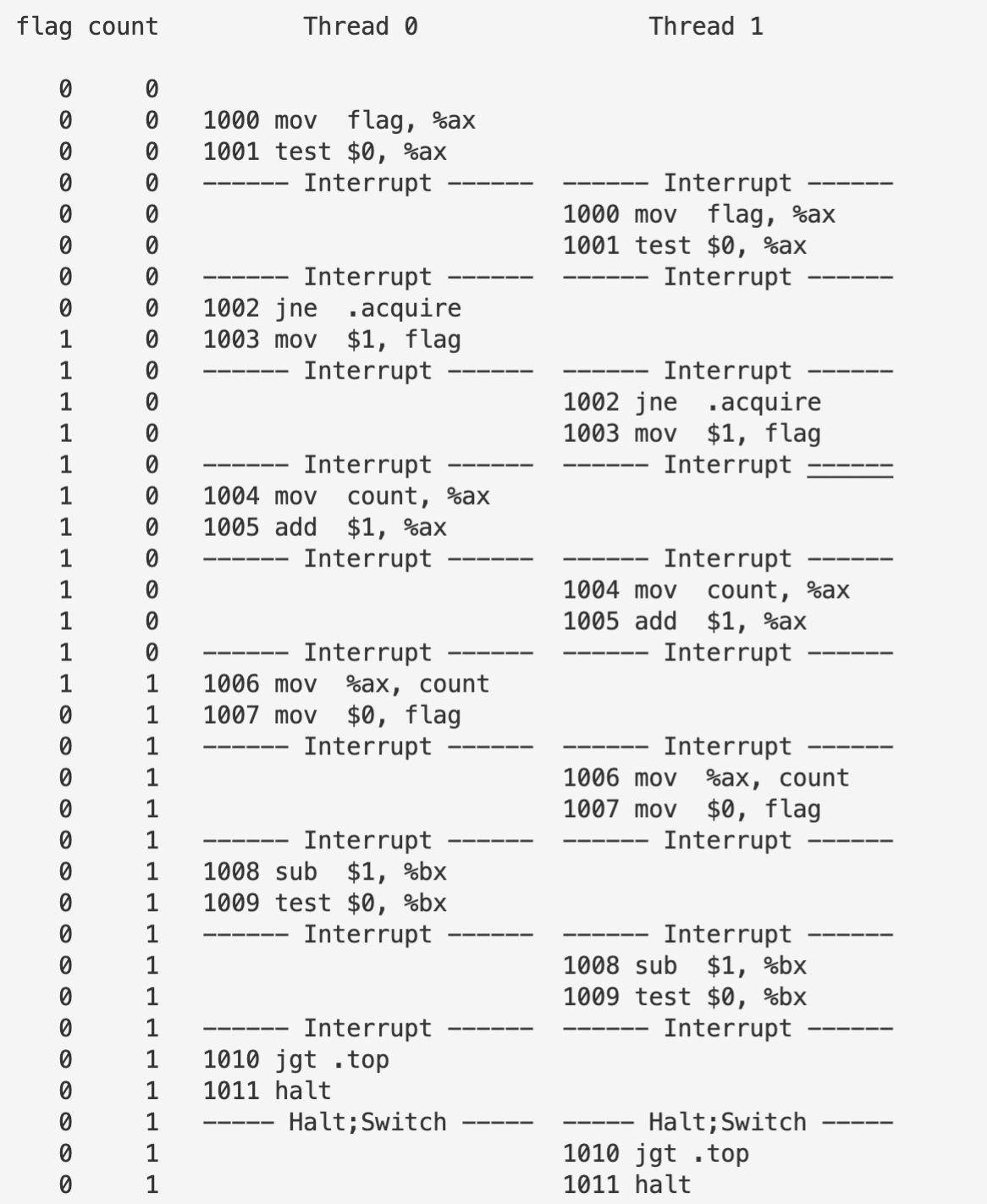
Try the command below:

python3 x86.py -p test-and-set.s -P 0011 -M mutex,count -c

and we get the result shown in the following screenshot:



We can see that the program is still running as we would expect it to do, with variable count ending at 2 as it should be, though not very efficient. However, if we set exactly the same parameters for flag.s, the output looks like this:



Rather than ending up 2 as we expect, the final value of count is 1. This is because what a certain thread does to acquire the lock, unlike test-and-set.s, is not atomic. Consequently thread 1 could sneak in without the permission of the lock variable rather than waiting for thread 0 to free the lock, and thread 1 would ignore what thread 0 does to variable count and set its value a conclusive 1, as we have seen in the case above.

1. **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.**

Basically Peterson’s algorithm assigned two different flags for both threads. Thread 0 first tries to acquire the lock by setting its own flag 1, then it concedes by setting variable turn the identifier for its rival. If the rival doesn’t have the lock, then thread 0 will be able to carry out its task and after finishing it’s expected to release the lock by setting its flag 0 and making sure the rival’s identifier is indeed what is stored in variable turn. If the rival has the lock, thread 0 should test if variable turn indicates that it’s its turn or not. If it is not, thread 0 will have to spin.

1. **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.**

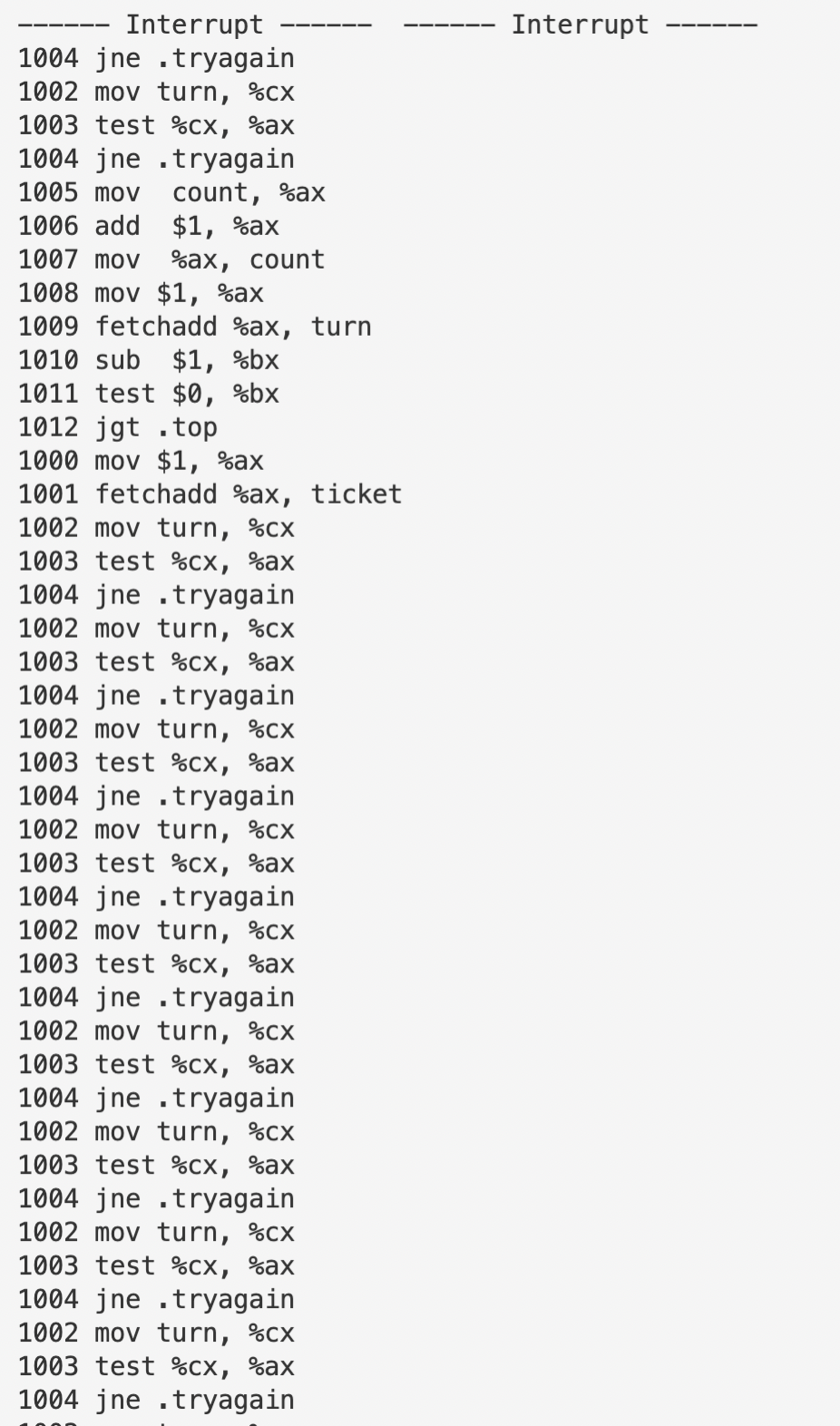
If we set interrupt interval 2, we can see that even though the two threads run alternatively when both of them are trying to gain control, this does not invoke any race condition, which is because they have separate locks. Also, Peterson’s algorithm ensures that one of the threads is sure to be working by using variable turn.

1. **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.**

Mutual exclusion and deadlock avoidance could both be proven by setting interrupt interval small enough. Thus simply setting -P 01, or -i 1 will do the trick.

1. **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?**

The output shows that both threads indeed spent a huge amount of time spinning, which is because once a single thread uses its chance that is guaranteed by its ticket up, it will have to wait until the other thread changes the value of turn for it, which is to say it has no choice but to spin-wait throughout the rest of the interrupt interval.

****

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

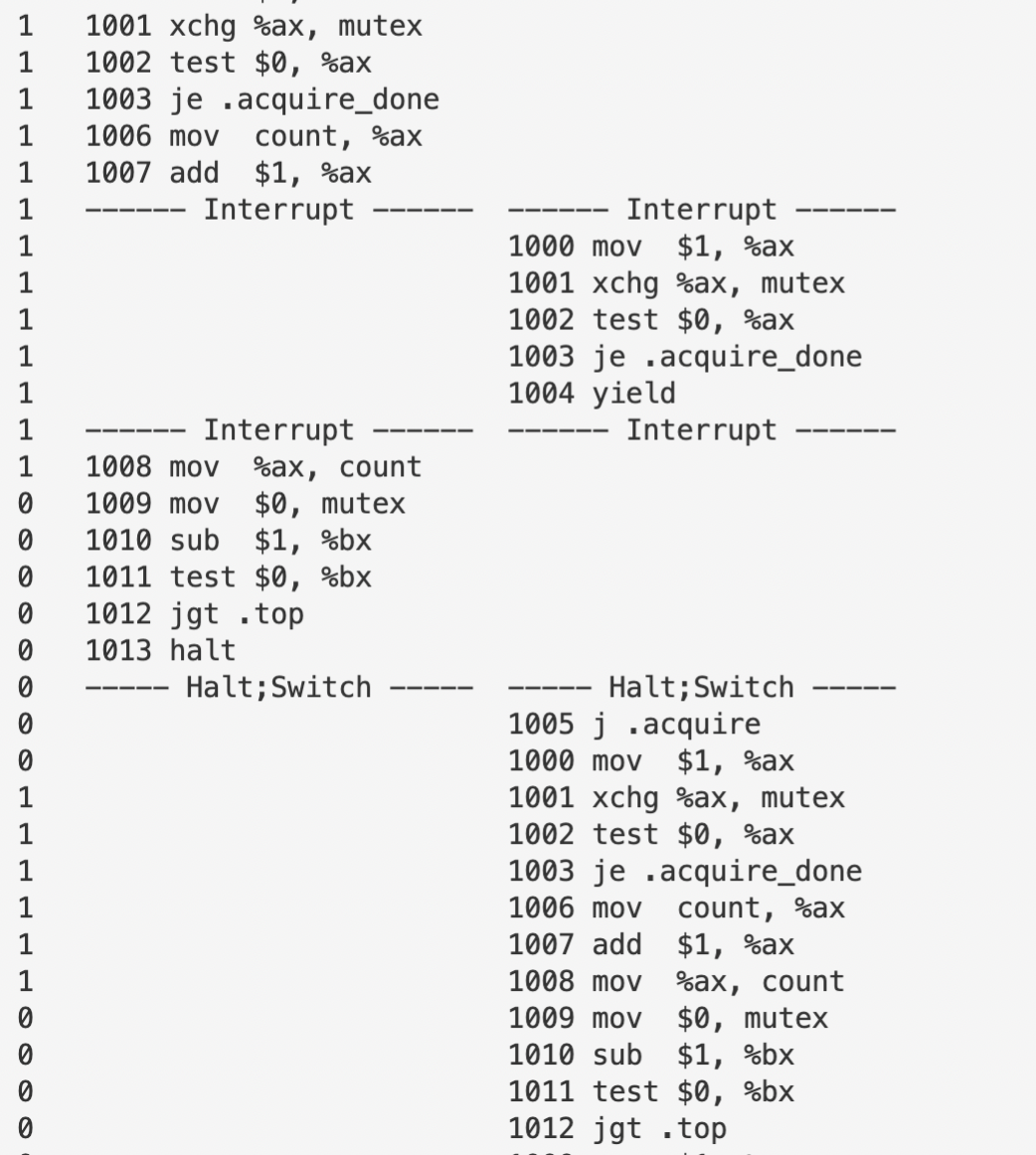
The picture here is pretty much the same, simply because the problem is not about how many threads are running at the same time, but that any thread has to wait until someone else changes variable turn. So, unless we change the interrupt interval, adding more threads does not essentially change anything at all.

1. **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?**

Try, for instance:

python3 x86.py -p yield.s -a bx=5 -i 50 -M mutex -c

which we have already scrutinized in question 6. By contrast, with yield.s, the two threads actually spend little time spinning, which should be attributed to the use of yield that enables a specific thread to concede its control over CPU once it finds out the other has had gripped the lock, thus it doesn’t have to spin-waiting all the way through the interval.



1. **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?**

Just as its naming suggests, this program tests if the lock is occupied twice before it enters the critical section. In comparison with the previous version test-and-set.s, this program first fetches the value of mutex and checks if it’s 0 or not before it does the atomic swap. If nor, the thread will begin spin-waiting already. This maneuver can avoid, to some extent, unnecessary data exchange.