**Homework Wan Huzaifah bin Wan Azhar**

**Answer:**



* The program flag.s just use one variable as locking mechanism
* If flag = 0, set it to 1 and run critical section. Release lock by flag = 0.
* Increment count by 1 by %bx loop.
* Repeat until loop counter %bx is 0 or lower



* 0 will ends up in flag
* Flag.s work with default argument because interrupt frequency is 50



* Setting bx=2 on both threads will increase the count to 4, as each thread will increment by two
* Flag as locking variable work as intended so far.



* The only interrupt frequency that can generate correct count is 11.
* As in, run all the assembly code atomically, on each thread, without interruption.
* Anything other than 11 will generate incorrect count value.
* This proves that at any given interrupt frequency except 11, there will be a time where both threads can access critical section at the same time and change the count value.
* As such, using flags as locking mechanism does not work, as it does not provide guarantee to mutual exclusion.



* Lock acquire works by swapping the value at mutex and %ax atomically.
* This means that the test-and-swap operation is done in one instruction.
* At the beginning, mutex has the value of 0. By swapping %ax (that has the value of 1) with mutex, %ax is now 0 while mutex is 1.
* Lock is released by changing the value of mutex to 0



* The code is always leads to correct result regardless of interval frequency and loop counter, bx.
* To calculate the efficiency of the program, we need to calculate how many spin-waiting instruction are executed compared to optimal value if there is no spin waiting.
* For the frequency interval, we will compare low frequency interrupt of 2, 5 and 11 (atomically).
* Interrupt interval of 11 is the best case interrupt as it runs all code atomically without interrupt.
* Using Windows’s powershell command “Measure-Object -line”, we are able to get that the program runs 2419 instruction with the best case interrupt interval.
* For interrupt interval of 2, there are 5107 instruction and 3614 instruction for interrupt interval of 5.
* This means that if the interrupt interval is 2, there are 2688 (5107 – 2419) more instruction that are ran by the CPU while there are 1195 more instruction when interrupt interval is at 5.
* Therefore, the CPU wastes 53% and 33% of the CPU cycle spin waiting when interrupt interval is at 2 and 5.
* As such, it is highly inefficient.



* Using -P 01, we can schedule thread 0 to run first and then thread 1 at interrupt frequency of 1.
* The program works perfectly fine even if xchg is running at thread 0 and then xchg is running at thread 1.
* Mutual exclusion still works because when thread 0 xchg with mutex, %ax in thread 0 now become 0 while mutex has the value of 1.
* Then, at thread 1, when it execute xchg instruction, %ax at thread 1 become 1, therefore it did not acquire the lock and spin-wait.



* Mutual exclusion works as intended even using different interrupt frequency
* The algorithm still use spin-waiting.



* Using -P 000001111101, we can show that when both thread attempt to update turn variable, it still provide mutual exclusion, as in, only one thread can run critical section.
* Turn is what provide mutual exclusion, at any point, there will only be one value of turn and only one thread can run.
* It also proves that there can be no deadlock when both threads try to get their turn. This is because they didn’t actually try to grab their turn, they want the other thread to start.
* There are three mechanism to avoid deadlock,
  + First is every thread initialize turn to 0. This means that if all else fail, thread 0 will always start and change the turn.
  + Turn = 1 – Self change the turn to other thread.
  + Flag(self) = 0 break other thread from spin-waiting.



* It is identical to most part. However, in the ticket.s, when it tries to release lock, it uses fetch-and-add instead of just incrementing like in the textbook.
* Yes, so much time spent busy waiting because the thread did not switch because of default interrupt interval of 50.
* So it just waits and waits and waits until thread switch occurs.



* The code still provide mutual exclusion, but much more time is spent busy waiting.
* Actually, the more thread there is, the more time CPU spent spin waiting.



* Test-and-set wastes CPU cycle when another thread get the lock but context switch on the thread does not happen.
* Using -a bx=100,bx=100 on test-and-set.s, we can see that a thread spent entire time spin-waiting.
* The same argument on yield.s did not have that problem as it puts thread to sleep if it did not get the lock.
* Using “Measure-Object -line” on both program tells that there are approximately 2416 (Approx. because halt line are also counted) instruction running on yield.s and 3437 instruction on test-and-set.s.
* Yield.s saves 1021 instruction from running on CPU.



* Test-and-test-and-set.s tests the mutex two times before proceeding into critical section.
* The intent is that if mutex is 0, the it is probable that no one has the lock, so it proceed to get the lock.
* If the thread get the lock, test it and proceed to critical section.
* Instead of looping on 3 instruction uses by test-and-set.s, it tries to decrease the instruction into best cases of 2. But, it can also become worst case of 6.
* As it can be seen, using argument bx=50,bx=50 and -i 7, test-and-test-and-set produce 2413 instruction and test-and-set produce 1892 instruction.
* This is because test-and-test-and-set produce worst-case of checking locks of 6 even though best-case is 2 while checking locks instruction of test-and-set is 4.