Locks

Annalise Tarhan

March 1, 2021

1 Examine flag.s. Can you understand the assembly?

The code is organized as a loop, iterating a number of times equal to the original value of %bx. At the beginning of the loop, the .acquire section loops until it can acquire the lock, which occurs when flag equals zero, and sets flag to one. Then, it increments a counter by moving its value to %ax, as it did with the flag, incrementing it, and moving it back to count. Finally, the lock is released by setting flag to 0, and %bx is decremented. If it is still greater than zero, the loop repeats.

2 When you run with the defaults, does flag.s work? Can you predict what value will end up in flag?

The code works when run with the defaults, because %bx is set to zero and the memory interrupt interval is higher than the number of instructions in the program. If those values were different and one thread was interrupted during .acquire, the program might give incorrect answers. Regardless of execution order, flag is always set to zero before a thread halts.

3 Change the value of the register %bx. What does the code do? How does it change your answer for the previous question?

When both threads' %bx registers are set to 10 or higher, the program gives incorrect results, since it interrupts a thread at least once during .acquire. The flag variable still always ends up set to zero.

4 Set %bx to a high value for each thread, and then use the -i flag to generate different interrupt frequency. What values lead to a bad outcome? Which lead to good outcomes?

For %bx>20, most interrupt intervals gave bad results. The ones that gave good results were the ones that were multiples of 11, which makes sense because there are 11 instructions per loop, and others which were less predictable. Multiples of 15 did well, but I didn't notice any other patterns.

5 Now look at 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?

Acquiring the lock is done by atomically swapping one into the lock, which returns the previous value. If the returned value is zero, it has acquired the lock, otherwise it keeps trying. The release is simpler; all it does is set the lock value to zero, no need to use xchg.

6 Now run the code, changing the value of the interrupt interval 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 always works as expected. It does lead to an inefficient use of the CPU when one thread is stuck trying to acquire the lock for its entire interval, which occurs when the interrupt interval isn't a multiple of the number of instructions, in this case 11. To quantify it, calculate how much longer the code runs for a specific interrupt interval than how long it runs if the two threads run sequentially, which can be measured by setting the interrupt interval to an extremely high value.

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?

It runs correctly, but inefficiently. The threads run the same code, so there is no need to test the reverse case, where the second thread grabs the lock and is then interrupted, but it might also be useful to test what happens when the thread is interrupted during the critical section as well as after the lock is released.

8 Now look at the code in peterson.s, which implements Peterson's algorithm. Study the code and see if you can make sense of it.

Peterson's algorithm uses a two element array called flag to indicate whether each of the two threads is waiting to enter the critical section and a variable called turn to indicate which thread's turn it is. To acquire the lock, a thread sets its flag element to one and sets turn to the other thread's id. Then, it enters a loop, where it stays until one of two things happens. If it sees that the other thread's flag element is zero or that the turn variable is set to its own thread id, it breaks out of the loop and enters the critical section. Once the critical section is finished, it sets its own flag value to zero, then sets the turn variable to the other thread's id.

9 Run the code with different values of -i. What kinds of different behavior do you see?

No matter what the interrupt value is, the code always produces the same result.

10 Can you control the scheduling to 'prove' that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

The most exhaustive way to test would be to schedule interruptions at every possible combination of instructions. Fortunately, that would be overkill. The code can be divided into four main sections: acquire, spin, critical, and release. Testing interruptions at each of these sections four times, with the other thread restarting once at each of its four sections, suffices to prove that the code works.

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. Watch what happens; do the threads spend much time spin-waiting for the lock?

The code in the chapter includes more setup and lacks code for what to do when the lock is acquired, but assuming FetchAndAdd and fetchadd work the same way, the mechanism for locking and unlocking is the same. With the default interrupt value, the threads spend more than half of the time spin-waiting for the lock.

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

Because the threads always run in the same order, it always seems to be a thread's turn when it is restarted. In a less orderly system, a thread would be restarted while a different thread held the current ticket and would spend its entire interval spinning. In this case, the proportion of time spent spinning is the same for two threads or many.

13 Now examine yield.s, in which a yield instruction enables one thread to yield control of the CPU. 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?

Test-and-set.s wastes many cycles spinning when the interrupt occurs before the previously running thread thread has released the lock. In the same scenario, yield.s' newly restarted thread yields immediately and only a few instructions are wasted. The number of instructions saved depends on the interrupt interval - the longer it is, the more wasted instructions before control returns to the thread that holds the lock.

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?

This version waits until it sees that the lock is free before attempting to grab it, spinning with a mov instruction instead of with an xchg instruction. The savings introduced depend on how xchg is implemented.