Concurrency: Introduction

Annalise Tarhan

February 19, 2021

1 Run loop.s with the following arguments. What will %dx be during the run?

The loop in loop.s decrements the value stored in %dx by one every iteration until the value is less than zero. How many iterations there are and %dx's value during each depend on what value is stored there initially. The -c flag shows that the inital value is zero, so the value is decremented, compared to zero, and since it's value is less than zero, the program halts.

2 Run the same code with the following arguments, which specify two threads and initialize %dx to 3. What values will %dx see? Does the presence of multiple threads affect your calculations? Is there a race in this code?

There are two %dx values because each thread is associated with its own set of registers. Each %dx will start at 3, then be decremented to 2, then 1, then 0, then -1, the same as if only one thread were executing. There is no race in this code because the threads are operating on different sets of registers.

3 Run again with the following arguments, which make the interrupt small/random; use different seeds to see different interleavings. Does the interrupt frequency change anything?

The interrupt frequency doesn't change anything, besides the extra time from the overhead of the context switches. Even though the interrupts happen while each thread is still running, their registers are separate.

4 Now run looping-race-nolock.s with the following arguments, which accesses a shared variable. What is the value of the shared variable throughout the run?

The number of times the code loops, as well as the value of the shared variable, depend on the initial values of the shared variable and what is stored in %bx. It seems that registers and the shared variable are initialized to zero, so the loop will only run once, incrementing the shared variable to one.

5 Run with multiple iterations/threads. Why does each thread loop three times? What is the final value of the shared variable?

Each thread loops three times because each decrements its %bx value three times, from three to zero. The final value of the shared variable is six, because each thread increments it three times.

6 Run with random interrupt intervals. Can you tell by looking at thread interleaving what the final value of the shared variable 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, if you know when the interrupts occur, you can tell what the final value will be. The timing of the interrupts affects the final value of the shared variable in that if it occurs after a tread's first mov instruction but before it's second mov, the other thread will access a 'stale' value and both threads' increments will only add up to a single increment. This means the critical section consists of three instructions: mov, add, mov.

7 Now examine fixed interrupt intervals. What will the final value of the shared variable be? What about when you change the interrupt intervals? For which interrupt intervals does the program give the 'correct' answer?

For a single iteration, the program gives the correct result of two when the interrupt interval is three or greater. Otherwise, it gives one, because both threads increment the same original value.

8 Run the same for more loops. What interrupt intervals lead to a correct outcome? Which intervals are surprising?

The interrupt intervals that maintain the integrity of the critical section are those that are multiples of three. This isn't just because the size of the critical section is three instructions; the number of remaining instructions is also three. (If there was only one additional instruction, for example, the critical section would be split on the second iteration.)

9 Run wait-for-me.s with the following arguments. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

The first thread's %ax value is one, so it is the designated signaller. When its turn comes, it sets the value of the shared variable to one and halts. The other thread is the waiter, so on its turn, it enters a loop and repeatedly (maybe) checks the value of the shared variable. When it sees the value is one, it halts. The final value of the shared variable is one.

10 Now switch the inputs. How do the threads behave? What is thread 0 doing? How would changing the interrupt interval change the trace outcome? Is the program using the CPU efficiently?

This version is a terrible use of resources. Designating the first thread as the waiter guarantees the first thread will loop for the entire interrupt interval, waiting for the second thread to modify the shared variable. Shortening the interrupt interval causes it to loop for less time.