Assignment 6

Friday, August 3, 2018 2:01 PM

Tania Akter
V00810640

7.1

a

Thread t1 is going to set the value of x and y to 1 and 2 respectively. Thread t2 is supposed to print the value the y and x.

However, when t1 runs, x might have been set but y might not have been set yet, which would mean printing the value of the y would give the value as 0, since y would not have been set by t1 yet.

The above case is a data race. The way to fix it is to introduce mutex in the main function, and lock the mutex when the data are being set in t1, and also acquire the mutex when the data are being printed.

b

The behaviour of the program can be either of the two things: The t1 can run first and set the values of x and y, and then t2 is run, printing y to be 2 and x to be 1.
Or,
T2 can run first and print x and y to be 0, and the value of x and y are set to 1 and 2 respectively in t1 after.

That being said, there are no data races in this program, just out of order of the thread execution.

g.

The behaviour of the program can be either of two:

If the thread t1 runs before t2, then x will be set to 42 and and the assert in t2 will be true. However, if t1 did not run before t2, the while condition in thread t2 will wait till thread t1 is executed, and the done is set to true first before it moves on to the assert.

There are no data races in the program.

i

The program will always do nothing because the threads depend on the other threads setting the x and y values. Hence, the if statements are always false, causing the threads to do nothing.

There are no data races since there were no data writes being made.

L

The first ideal case is when thread t1 finishes before thread t2, and the counter is incremented sequentially in t1 and t2 respectively, giving a perfect incremented counter value of 200000, which is printed at the end.

The program does not guarantee any execution order, or locking mechanism. Here, thread t1 and thread t2 can run concurrently, and the value of the counter read for incrementing might not be the same because the other thread might have already changed the value. The value of the counter could be between 100000 - 200000. This causes data race.

To fix the data race, we should add mutex inside the for loop in each of the threads, lock before increment and unlock after increment.

m

The expected behaviour of the program is that the w.x gets set in t1 and w.y gets set in t2. It can happen in any order.

There are no data races, since the threads are changing different members of the object.

Friday, August 3, 2018 2:56 PM

7.10

Initially : x , a, y ,b = 0

X	a	Y	B	Case explanation
1	0	1	1	Thread 1 runs completely before thread 2
1	1	1	1	Thread 1 runs till x =1, thread 2 starts and finishes execution, and then thread 2 finishes
1	1	1	1	Thread 1 does x =1, thread 2 does y =1, thread 1 does a = y, thread 2 does b =x
1	1	1	0	Thread 2 runs completely before thread 1
1	1	1	1	Thread 1 runs till y=1, thread 2 runs to completion, and then thread 2 does b=x
1	1	1	1	Thread 2 does y=1, thread 1 does x =1, thread 2 does b =x, thread 1 does a=y

There are no cases where we can get 0 for both a and b.

Friday, August 3, 2018 3:09 PM

7.12

a.

Sometimes, because thread 1 might have assigned x=2, making thread 2 pass the if condition, but thread 1 might not have finished assigning y=1, and when thread 2 checks for y==1 in this condition, it will fail.

b
Always

Because even if thread 2 runs before, it will wait for the value of y to be 1, which means that thread 1 has to run for the thread 2's while loop to go through. Since, x is set to 1 before y =1 in thread 1, the assert in thread 2 will always be true.

C
Sometimes

If thread 1 finishes completely before thread 2, the asserts in thread 2 will be true.

However, in the case where thread 1 has run x =1 and y=1, and thread 2 starts execution and passes the while loop, the assert for x==1 will be true, but since z was not set to 1 in thread 1 yet, the assert for z will fail in thread 2

Friday, August 3, 2018 3:38 PM

Answer to PART C:

With 8 threads, the compute_julia_set function takes 29658512microseconds

With 4 threads, the compute_julia_set function takes 44574864 microseconds

With 2 threads, the compute_julia_set function takes 84850141 microseconds

With 1 threads, the compute_julia_set function takes 169476134 microseconds

The above values make sense because the thread pool has lower thread count to run the scheduled tasks, which means to complete the entire task, more time will be required, since all the thread in the pool will be in use.

It can be also seen that the execution time for test_julia_set is also doubling with making the number of threads in the thread pool to be half each time, which can be correlated to the explanation given above about all the threads being in use and having to wait for them to be idle to do more jobs