Parallel and Distributed Computing
DD2443 - Pardis24
Exercises for Lecture 10

Name: Casper Kristiansson

September 26, 2024

Exercise 1

Question

Consider the program in Figure 1. You may assume that the two `get_this_in_order` statements in this program are executed in program order using some machinery outside the scope of this exercise. What possible pairs of values $(r1, r2)$ are output by this program? Explain your reasoning carefully.

```
1  public class SynchronizedBarriers {
2      int x, y;
3
4      void actor() {
5          synchronized(this) {
6              x = 1;
7          }
8          synchronized(this) {
9              y = 1;
10         }
11     }
12
13     void observer() {
14         int r1, r2;
15         r1 = get_this_in_order(y);
16         r2 = get_this_in_order(x);
17         output(r1, r2);
18     }
19 }
```

Figure 1: Code for Exercise 1

Answer

The possible pairs of the values for (r1, r2) outputted by the program can be deterred by how the observer() threads read both the value x and y and how they are in relation to the actor() thread.

- (0,0): This happens if the observer() runs before actor() and writes to both x and y values where the initial value of the variables is 0.
- (1,0): This happens for the situation where observer() reads the value y after the actor() function has updated the specific value to 1 before the variable x is.
- (1,1): For the last situations this happens when the observers run after the actor function and therefore both the value of the variables will be 1.

Exercise 2

Question

Consider the code snippet in Figure 2. The claim is that this code snippet behaves as a mutex lock for the critical section `a = x; b = 1;`. Explain carefully why that is.

```
1 int x;
2 volatile boolean busyFlag;
3 while (!CAS(lock.busyFlag, false, true));
4 a = x;
5 b = 1;
6 lock.busyFlag = false;
```

Figure 2: Code snippet for Exercise 2

Answer

For the provided code the code snipped will act as a mutex lock using a compare-and-swap operation to handle the specific critical section. The while statement ensures that only one of the threads can successfully set the busyFlag and then processed with the critical section. After executing the specific critical statement the thread will rest the busyFlag to default which then will allow another thread to acquire the specific lock and then enter the critical section.

Exercise 3

Question

Consider the program in Figure 3. Is the outcome $(r1, r2) = (1, 0)$ possible? Explain carefully as usual.

```
class VolatileArray {
    volatile int[] arr = new int[2];

    void actor() {
        int[] a = arr;
        a[0] = 1;
        a[1] = 1;
    }

    void observer(int r1, int r2) {
        int[] a = arr;
        r1 = a[1];
        r2 = a[0];
    }
}
```

Figure 3: Code for exercise 3

Answer

Yes the outcome $(r1, r2) = (1, 0)$ is possible because:

- First off because arr reference is declared as volatile it can ensure that the array is visible to the other threads. But it does now allow synchronization for the specific accesses of the specific indexes.
- For the actor method the writes cannot be guaranteed to occur in the specific order as they will be visible to the other threads. This means that the actor-specific functions might write the different values to the different indexes in a different order.
- For the specific observer method where we copy the array values the visibility of the order cannot be guaranteed. This means that the specific thread could see the updated value of the second access while the first access provides an old value.

Because of these reasons the outcome that $(r1, r2) = (1, 0)$ is possible due to the lack of synchronization between the array elements.

Exercise 4

Question

The exercise is here: https://fzn.fr/teaching/bertinoro15/javamm.html. Answer the questions and provide explanations. Document the machine and Java runtime you used. Do the bonus exercise as well. Fix `Bonus.java` such that you believe race-free behaviour is guaranteed (it suffices to run the fixed version of Bonus a reasonable number of times, according to your own version of "reasonable"). Explain your changes. Instrument the original version of Bonus as well as the amended one to measure running time (you can use `System.nanoTime()` for this), to get an idea of the overhead involved. Explain what you've done. What overhead did you get (take a suitable average)?

Answer

Why "unsurprisingly"?

The final counter value is not 10000000 because the two different threads increment the shared counter without synchronization. This means that the threads might be reading and updating the counter simultaneously and therefore overwriting each other values.

Do we (observe the output 10000000)?

No, the program hangs in the Dekker algorithm because, without any proper synchronization, the variable turn which is accessed between the threads waits infinitely.

Can you reproduce the above behavior on your machine?

Yes, when running the program with two threads we could see this behavior happening and the execution gets stuck. This is because we have a busy-waiting loop that isn't properly synchronized between the threads.

What should this program print? Should it terminate? Why?

The program should print the value val twice and then terminate after setting the condition to false. However, because the variable condition is not optimized and synchronized.

What does this program print? Does it terminate?

The program printed the value val twice which in my case was 849475563 and then the program did not terminate and constantly was just stuck.

Declare the variable condition as volatile and observe the outcome. Is it the expected one?

Yes, with the help of using volatile for the condition variable than the program will terminate and print the expected values.

Declare the variable turn as volatile and observe the outcome. Is it the expected one?

By declaring the variable will still not help with the value to get to 10000000 because of the race condition that happens with the flag array.

Modify Dekker.java accordingly and observe the result

To solve this and modify the Dekker algorithm the correct way we can utilize AtomicIntegerArray for the flag array to make sure that a race condition wont happen. Doing this will allow that the program counter to reach the value 10000000 as expected.

Bonus Exercise

The program might have a behavior due to memory reordering that is not expected. When the first thread writes x=1 it will than set the y=1 in a loop. When thread 2 than reads both x and y the expected values are y and x equals to 1. But due to optimization the write to x can be reordered so that the thread 2 might see the values y equal to 1 and x equal to 0. I read up on this in regards to global code motion but I wasn't able to reproduce this. This can easily be fixed declaring the variables x and y as volatile.

When adding the volatile keyword to the variables there was a lot more overhead and the execution time was about 4x the amount of time to execute the code. Without using the volatile keyword the average execution time was 956,258.4 ns while for the volatile keyword was 4,025,929.3 ns.

- **Machine**: Apple Macbook Pro 16 M1
- **Java Runtime**: Java 17