Parallel and Distributed Computing
DD2443 - Pardis24
Exercises for Lecture 2

Name: Casper Kristiansson

August 29, 2024

Exercise 1

Question

Consider the class `VolatileCounters` in Figure 1. The "volatile" declaration ensures that variable reads and writes occur in a one-at-a-time sequential order (as one might expect) when accessed by each of the parallel threads. Assume `actor1` and `actor2` are invoked to run in parallel.

1. **What are the relevant events?**
2. **Sketch the state machine graph (only parts, it is quite large). Include all events you have identified in 1.**
3. **Sketch a few traces of the state machine.**
4. **List the relevant intervals for the program.**
5. **Which are the possible final values for `x`? Explain your reasoning very carefully.**

Note that this is more subtle than it looks. Are you sure that you have the right solution?

Answer

1.

For this program, the relevant events are regarding the reading and writing to the variable x. Since it is declared using volatile to make sure that reading and writing to it happens in sequential order. The actions can be described as:

- Read Event: This happens when we want to read the variable [1]. For example, we can assign Read1(x) for actor1 and Read2(x) for actor 2 when reading the variable x.
- Write Event: These are the events when we write to the variable x. This can be assigned as Write1(x, newValue) for actor1 and Write2(x, newValue) for actor2.

2.

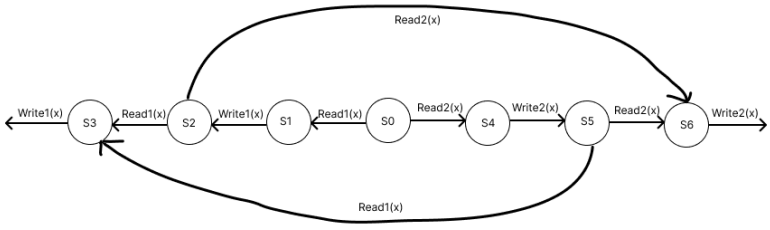

Figure 1: State Machine Graph

In the different states, we have the following:

1. State 1: x=0, Read1=false, Read2=false
2. State 1: x=0, Read1=true, Read2=false
3. State 2: x=1, Read1=false, Read2=false
4. State 3: x=1, Read1=true, Read2=false
5. State 4: x=0, Read1=false, Read2=true

6. State 5: x=1, Read1=false, Read2=false
7. State 6: x=1, Read1=false, Read2=true

Note that this is only showing the real beginning of the state machine.

3.

Here are two different traces of how it could look like for the incrementing between the two other actors.

- Read1(x=0), Write1(x=1)
- Read1(x=1), Write1(x=3)
- Read1(x=2), Write1(x=4)
- ...
- Read2(x=19), Write2(x=20)

- Read1(x=0), Write1(x=1)
- Read2(x=1), Write2(x=2)
- Read1(x=3), Write1(x=4)
- ...
- Read2(x=19), Write2(x=20)

4.

There are different types of intervals the program can take. These are the following different intervals that are interesting:

1. The interval between actor1 and actor2: It this scenario the read and write will change between actor1 and actor2 after each has finished a sequence of (read-write). This means that first actor1 will increase the counter and than actor2 will increase it etc.
2. Full increment for actor1: This means that actor1 will finish its incrementing before actor2 starts incrementing.
3. Full increment for actor2: This means that actor2 will finish its incrementing before actor1 starts incrementing.

5.

With the usage of volatile every read and write of the variable x will be seen as a chunk and lock the variable. This means that if actor1 is in the process of reading/writing to a variable another actor is not able to access it until it gets realsed. Because of the usage of volatile in this situation, the only possible final value of x is that it is 20. It ensures that variable reads and writes occur in a one-at-a-time sequential order.

Exercise 2

Question

HSLS Exercise 2.3 (Flaky Computer Corporation) Use the method presented in class and in the textbook to solve this.

Programmers at the Flaky Computer Corporation designed the protocol shown in Fig. 2.16 to achieve n-thread mutual exclusion. For each question, either sketch a proof or display an execution where it fails.

- **Does this protocol satisfy mutual exclusion?**
- **Is this protocol starvation-free?**
- **Is this protocol deadlock-free?**
- **Is this protocol livelock-free?**

Answer

1.

Yes, this function with lock and unlock does provide mutual exclusion. This is because when entering the function lock a thread will write to the object its id. It is then only that id will be able to enter the while true loop when busy is false. While all threads will constantly change the variable turn to me there will always only be one that can enter and set busy to true. Doing this allows for mutual exclusion.

2.

No this doesn't provide a starvation-free program. This is because currently, a random thread will be the one setting busy to true. Because with the current scenario, the threads will over and over overwrite the turn variable to me. Meaning if it times it right and the turn is equal to a specific thread and busy becomes false than that thread will enter. If with bad luck this could leave a thread not being able to execute code.

3.

Yes, this program is deadlock-free because it doesn't have a scenario where the current executing thread relies on a thread that might be waiting for its turn. But this will only happen if the critical section doesn't rely on another locking mechanism which in this case it doesn't.

4.

No, with this current type of lock mechanism, there might be a scenario where the threads might constantly change the turn variable to me while the other is trying to check if turn == me. This means that it constantly changes and is busy and its turn is then mismatched leading to that it isn't able to enter its critical zone. This means that the mechanism is not livelock-free.

Exercise 3

Question

HSLS Exercise 2.8 (Fast path lock) Use the method presented in class and in the textbook to solve this.

In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock. Scientists at Cantaloupe-Melon University have devised the following "wrapper" for an arbitrary lock, shown in Fig. 2.18. They claim that if the base `Lock` class provides mutual exclusion and is starvation-free, so does the `FastPath` lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

Answer

With this type of lock where we will utilize three different paths right, down, or stop a thread can acquire a thread in a constant number of threads. The protocol works as follows; First, it will check if right is already set to True which it isn't, and will continue to the down path while setting go right to true. This will then in turn if another thread enters the protocol it will go right which often involves that the thread can't acquire the lock right away and needs to wait.

With the usage of a FastPath, it will allow threads to either acquire the lock right away or via the right path. Utilizing this method will mean that the algorithm is both Mutual Exclusion and starvation-free. This is said as long as the Base lock functions correctly.

References

[1] *Volatile class (system.threading) — microsoft learn*, https://learn.microsoft.com/en-us/dotnet/api/system.threading.volatile?view=net-8.0, (Accessed on 08/29/2024).