Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Benedikt Naumann, benedikt.naumann@tu-darmstadt.de

# Advanced Multithreading in C++ (WS 21/22)

# Exercise 3

Please solve the following tasks by December 14th, 2021. The results are not graded, but a solution is discussed on December 14th, 2021.

## Task 1

Consider the following code:

Listing 1: Two threads incrementing shared variable

```
1  using namespace std;
2
3  int sharedValue = 0;
4
5  void IncrementSharedValue10000000Times()
6  {
7      int count = 0;
8      while (count < 10000000)
9      {
10         sharedValue++;
11         count++;
12     }
13 }
14
15 int main(int argc, char* argv[])
16 {
17     sharedValue = 0;
18     thread thread2(IncrementSharedValue10000000Times);
19     IncrementSharedValue10000000Times();
20     thread2.join();
21     printf("sharedValue=%d\n", sharedValue);
22
23     return 0;
24 }
```

**a)** What would be the expected output of this program?
Is it consistent between executions? If not, give a reason why.

**b)** Reimplement the given program in a thread-safe way using `std::atomic`.

## Task 2

Briefly explain C++ memory ordering relationships, namely *synchronizes-with* and *happens-before*.

## Task 3

Given two threads *i* and *j*, you need to write a program that can guarantee mutual exclusion between the two without any additional hardware support. The simplest and the most popular way to do this is by

using Peterson Algorithm for mutual Exclusion. Considering the following sample implementation for Peterson lock, answer the subsequent question.

Listing 2: Sample implementation of Peterson lock

```
class PetersonLock {
  private:
    // indexed by thread ID, 0 or 1
    std::atomic<bool> interested[2];
    // who's yielding priority?
    std::atomic<int> turn;

  public:
    PetersonLock()
    {
        turn.store(0, std::memory_order_release);
        interested[0].store(false, std::memory_order_release);
        interested[1].store(false, std::memory_order_release);
    }

    void lock()
    {
        int me = threadID; // either 0 or 1
        int he = 1 - me; // the other thread
        interested[me].exchange(true, std::memory_order_acq_rel);
        turn.store(me, std::memory_order_release);
        while (interested[he].load(std::memory_order_acquire)
            && turn.load(std::memory_order_acquire) == me)
           continue; // spin
    }

    void unlock()
    {
         int me = threadID;
         interested[me].store(false,std::memory_order_release);
    }
}
```

**a)** Using your answer given to the previous question, validate the following conditions:

- If thread 0 successfully acquires the lock, then thread 1 will not do so;
- If thread 0 acquires the lock and then releases it, then thread 1 will successfully acquire the lock;
- If thread 0 fails to acquire the lock, then thread 1 does so.

**b)** If these conditions are violated by the given implementation, how can we modify the code to uphold the conditions?