Exercise Sheet #10

Deadline: 24.06.2024, 12:00h

Rate us! Please evaluate the course via this link on the 21st of June 2024 from 8:00 to 10:00h.

Sign up for the exam! If you want to take the exam, send a quick, informal email to
nevermann@itp.uni-frankfurt.de.
by the **17th of July 2024**.

Problem 1 (*Multithreading and Mutex*) (20 points)

Multithreading is often a valuable tool to speed up the computation of a program. While threads *cannot* make the computer run faster (i.e. they do not speed up the execution of a block of code), they *can* increase the efficiency of the computer by using time that would otherwise be wasted. The reason that time is wasted in the execution of a program is often caused by the hardware. The CPU can execute calculations more quickly than the time it takes to input or output from or to memory and hard storage.
Your task in this exercise is to add up numbers in an array. In particular, we consider the sum

$$\mathcal{S} := \sum_{n=1}^{N} n, \qquad N \in \mathbb{N}. \tag{1}$$

According to the young Carl Friedrich Gauß, this sum can be easily evaluated analytically to

$$\mathcal{S} = \frac{N(N+1)}{2}. \tag{2}$$

We will now verify this numerically.

(a) Write a function that sums the elements in a given array, i.e. a function with the signature

```
long long calculateSum(int arr[], int N){...}
```

where we return a `long long` to avoid integer overflows (we will test this with large arrays later). In the `main` function, create an `int` array with $N = 1\,000\,000$ entries $1, 2, \ldots, N$, representing the terms in (1). Using

`calculateSum`, verify (2). Time the execution of the summation and output the sum and the execution time to the console.

Hint: To time your program recall the code presented on sheet #3.

(⑥ points)

(b) We now want to implement a multithreaded version of that program. For that first define a global variable[1] `long long sum = 0;` and write a function

```
void calculateSum(int arr[], int start, int end){...}
```

that calculates a partial sum of the array from a start index `int start` to an end index `int end` and adds the result to `sum`. Then create a total of $n_{\text{threads}} = 5$ threads, that calculate the partial sums for chunks of the array of equal size n_{threads}/N using `calculateSum`. Await all threads to finish using the `join()` method and print the sum and the execution time of the summation to the console. Run the code a couple of times. You should see some incorrect results that also change between different executions of the program.

(⑧ points)

(c) The reason you see incorrect and changing results in part (b) is a so-called *race condition*. This occurs, when two or more threads access shared data (in our case the global variable `sum`) and try to change it at the same time. Both threads 'race' to change the data. To overcome that, we can use a `mutex`[2]. Create a `mutex` and lock it before accessing the shared variable `sum`. Then unlock it again. Now your code should yield the correct result.

(④ points)

(d) Did the multithreading speed up the execution? Compare the execution times between part (a) and part (c).

(② points)

Hint: Remember to compile with flags `-std=c++11 -pthread`.

[1] A global variable is defined in the global scope, i.e. outside any function.

[2] Mutex stands for ***mut****ual* ***ex****clusion*.

Problem 2 (Advanced: *Implementing a Mutex*) (10 points)

In this problem we will implement our own `mutex`. The following steps will guide you through the implementation.

(1) Create a class `Mutex` with a private member variable `locked` which indicates if the `mutex` is locked or not. Since a race condition could occur in the process of locking the `mutex`, we need to use an atomic variable[3] provided by `std::atomic`. Include the header file and define the atomic variable using `std::atomic<bool> locked;`.

(2) In the constructor of the `Mutex` class, initialize `locked` to `false`.

(3) Implement a public method to lock the `mutex`. The code is given below

```
void lock()
{
    while (locked.exchange(true, std::memory_order_acquire))
    {
        // Wait for the lock to be released
    }
}
```

In this, `exchange()` atomically exchanges the value of `locked` with the given value `true` and returns the previous value. `std::memory_order_acquire` specifies that any memory access that occurs after the acquire operation must not be reordered by the thread scheduling algorithm before the acquire operation.

(4) Implement a public method to unlock the `mutex`. The code is given below

```
void unlock() {
    locked.exchange(false, std::memory_order_release);
}
```

This completes the `Mutex` class. Now define and initialize a shared global variable `int sharedVariable = 0;` and define a function
`void incrementVariable(){...}`
that increments the variable in a `for`-loop to 100000. In the `main` function, create two threads that both execute `incrementVariable()` and call `join()` on them. Output the final value of `sharedVariable`. Create a `Mutex` object and lock / unlock it in the required places. Compare the result with and without using a `mutex`.

[3] For an atomic variable it is ensured that operations on the variable are executed atomically, i.e. without interference from other threads.