Assignment 2 2INCO

Robert Jong-A-Lock 0726356
Tom Buskens 1378120
October 9, 2018

1. Overview

There were 2 main challenges while writing this program:

- Reduce chance of blocking mutex lock
- Limit amount threads

2. Reduce chance of locks waiting

The exercise dictated that an buffer made out of an array from a type with 128 bits is toggled by multiple threads. This buffer has to be protected to prevent race conditions. This protection is implemented by using a mutex lock around the computations that have to be atomic. This made sure that threads have to wait on each other to access the buffer. To limit the amount of times the threads have to wait a few optimizations were implemented.

- First every element in the buffer array has its own mutex lock. This splits the entire buffer in many smaller parts that can be accessed concurrently.
- The amount of time the thread needs to lock an element of the buffer is reduced by first generating all the necessary masks. These masks are then used in the critical section to toggle all necessary bits with one bitwise exclusive or computation.
- When a mask only contains zeros it is not necessary to request an element for the buffer. Because no bits will be toggled as a result of it. Therefore these masks are just skipped.
- At last the order in which the masks are used to toggle the bits is different for every thread. Each thread loops through all elements but starts at a different point in the buffer. The points where threads start are evenly distributed over the buffer.

3. Limit amount threads

Another aspect of the exercise was to limit the amount of threads that run at the same time. Besides it would be preferable that threads would be restarted as fast as possible when they are finished. This was accomplish by keeping track of the state of each thread. Initially all threads have the state uninitialized. They could also be busy when they were altering the buffer and be finished when they were done with that.
The main thread has a semaphore which is used to block the main thread. During initial execution the semaphore will not block and the threads are started until the maximum amount of threads is reached. At this point the main thread will be blocked by the semaphore. Whenever a thread is started its state is updated from uninitialized to busy.
At some point one of the threads will finish and put its state to finished and increase the semaphore. Now the main thread will wake up and look through all the states to find the thread that woke him up. Whenever the main thread finds it, it will join with the thread and start a new one. After this the main thread will block again.
This concept keeps working when multiple threads finish at the same time because the semaphore will unlock for each thread that finishes.
A mutex could not be used to accomplish this behavior because it can happen that multiple threads finish at the same time and the behavior of unlocking a mutex twice is not defined.