

## Excercise Sheet 6

(To be processed until 20.01.)

## Lecture Parallel Processing Winter Term 2025/26

### Exercise 1: Lock-Free Data Structures

For the purpose of debugging and performance analysis, parallel programs often log certain events into a trace file for later analysis (see Sect. 2.9.6 of the lecture). In order to minimize the perturbation, a typical architecture for such a tracing system is to use an event buffer in main memory, where all threads of the parallel program append their events, and an additional monitoring thread that reads the events from the buffer and stores them into a file. Thus, we have a typical producer/consumer problem with many producers and just one consumer thread.

The trace buffer typically is implemented as a lock-free ring buffer, consisting of a fixed size array and two variables containing the index of the next free element for writing and the index of the first element that was not yet consumed (i.e., written to file). If both indices are identical, the buffer is empty.

In the archive [u06eFiles.zip](#)<sup>1</sup> on the lecture's web page, you will find the implementation of an event buffer class, together with a simple test application. The application creates 10 threads that perform 1000 iterations of a loop and append an event to the buffer in each iteration. In addition, a monitoring thread reads these events and prints them to a file `trace.txt`. To make the exercise easier, the code contains a number of simplifications:

- The method `pop_front()` uses busy waiting, if the event buffer is empty.
- The size of the buffer is identical to the total number of created events, so the method `push_back()` never needs to block because of a full buffer.
- The monitor thread knows the the expected number of events and terminates when it has written this number of events to the file.

The code does not contain any synchronisation, so it does not yet work. When you start it, it will not terminate, since the number of events read by the monitor thread is less then the expected number, because some of the entered events are lost due to race conditions. So, in order to stop the program, you have to press `^C`. When you inspect the output file `trace.txt`, you may also notice that some entries in the trace file are zero, although this cannot happen in a correctly synchronized implementation.

Modify the class `EventBuffer`, especially the methods `push_back()` and `pop_front()`, in order to implement a lock-free synchronization for a correct function of the event buffer (c.f. slide 299 of the lecture). Do not modify code outside the class `EventBuffer`!

Hint: the `g++` compiler provides a built-in function

```
int __sync_fetch_and_add(int *ptr, int value)
```

that implements an atomic fetch-and-add operation.

Check the output file `trace.txt` to ensure that all traced events are correct. After your first attempt, you may still see lines in the trace file, where some entries are zero. If so, think about the cause of this problem and fix your code!

### Exercise 2: Sokoban with Lock-Free Data Structures

- a) Using lock-free data structures often is much more efficient than using classical data structures with mutual exclusion.

To show this, compare the performance of your optimized parallel Sokoban solver of Exercise 4 on [Exercise Sheet 5](#) with a version using lock-free implementations of the two `lookup_and_add()` methods. Be sure to use levels

---

<sup>1</sup><http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/pv/u06eFiles.zip>

with more computational complexity for this comparison.<sup>2</sup>

If you are motivated, you can try to implement the lock-free versions on your own, see part **b)** of this exercise. Otherwise, you can just use the code provided in the archive [u06eFiles.zip](#)<sup>3</sup> on the lecture's web page.

Explain your results!

**b) For motivated students:** Try to create lock-free implementations of the methods

- `BFSQueue::lookup_and_add()`
- `DFSDepthMap::lookup_and_set()`

of the Sokoban solver.

Because OpenMP only very recently supports the corresponding atomic operations (and thus, your compiler may not yet include this support), use the built-in gcc functions:

- `bool __sync_bool_compare_and_swap(type *ptr, type oldval, type newval)`
- `type __sync_fetch_and_or(type *ptr, type value)`
- `type __sync_fetch_and_add (type *ptr, type value)`

Here, `type` can be an arbitrary integer type or a pointer type.

See <http://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Atomic-Builtins.html> for more details.

The g++ compiler also supports newer builtin functions that support the C++-11 memory model. These functions are more flexible, but also a bit more complex to use. You can find a documentation of these functions on the web page [https://gcc.gnu.org/onlinedocs/gcc/\\_005f\\_005fatomic-Builtins.html](https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html). If you like, you can use `__atomic_compare_exchange_n()`, `__atomic_fetch_or()`, and `__atomic_fetch_and()` instead of the functions shown above.

---

<sup>2</sup>On the lab computers, use the level `sasquatch-III-3.txt` for the breadth first search and the level `sasquatch-III-5.txt` with maximum depth 53 for the depth first search.

<sup>3</sup><http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/pv/u06eFiles.zip>