Chapter 20. Boost.Lockfree

Tim Blechmann

Table of Contents

Introduction & Motivation

Introduction & Terminology

The term **non-blocking** denotes concurrent data structures, which do not use traditional synchronization primitives like guards to ensure thread-safety. Maurice Herlihy and Nir Shavit (compare "The Art of Multiprocessor Programming") distinguish between 3 types of non-blocking data structures, each having different properties:

- data structures are **wait-free**, if every concurrent operation is guaranteed to be finished in a finite number of steps. It is therefore possible to give worst-case guarantees for the number of operations.
- data structures are **lock-free**, if some concurrent operations are guaranteed to be finished in a finite number of steps. While it is in theory possible that some operations never make any progress, it is very unlikely to happen in practical applications.
- data structures are **obstruction-free**, if a concurrent operation is guaranteed to be finished in a finite number of steps, unless another concurrent operation interferes.

Some data structures can only be implemented in a lock-free manner, if they are used under certain restrictions. The relevant aspects for the implementation of `boost.lockfree` are the number of producer and consumer threads. **Single-producer** (**sp**) or **multiple producer** (**mp**) means that only a single thread or multiple concurrent threads are allowed to add data to a data structure. **Single-consumer** (**sc**) or **Multiple-consumer** (**mc**) denote the equivalent for the removal of data from the data structure.

Properties of Non-Blocking Data Structures

Non-blocking data structures do not rely on locks and mutexes to ensure thread-safety. The synchronization is done completely in user-space without any direct interaction with the operating system [7]. This implies that they are not prone to issues like priority inversion (a low-priority thread needs to wait for a high-priority thread).

Instead of relying on guards, non-blocking data structures require **atomic operations** (specific CPU instructions executed without interruption). This means that any thread either sees the state before or after the operation, but no intermediate state can be observed. Not all hardware supports the same set of atomic instructions. If it is not available in hardware, it can be emulated in software using guards. However this has the obvious drawback of losing the lock-free property.

Performance of Non-Blocking Data Structures

When discussing the performance of non-blocking data structures, one has to distinguish between **amortized** and **worst-case** costs. The definition of 'lock-free' and 'wait-free' only mention the upper bound of an operation. Therefore lock-free data structures are not necessarily the best choice for every use case. In order to maximise the throughput of an application one should consider high-performance concurrent data structures [8].

Lock-free data structures will be a better choice in order to optimize the latency of a system or to avoid priority inversion, which may be necessary in real-time applications. In general we advise to consider if lock-free data structures are necessary or if concurrent data structures are sufficient. In any case we advice to perform benchmarks with different data structures for a specific workload.

Sources of Blocking Behavior

Apart from locks and mutexes (which we are not using in `boost.lockfree` anyway), there are three other aspects, that could violate lock-freedom:

Atomic Operations
: Some architectures do not provide the necessary atomic operations in natively in hardware. If this is not the case, they are emulated in software using spinlocks, which by itself is blocking.

Memory Allocations
: Allocating memory from the operating system is not lock-free. This makes it impossible to implement true dynamically-sized non-blocking data structures. The node-based data structures of `boost.lockfree` use a memory pool to allocate the internal nodes. If this memory pool is exhausted, memory for new nodes has to be allocated from the operating system. However all data structures of `boost.lockfree` can be configured to avoid memory allocations (instead the specific calls will fail). This is especially useful for real-time systems that require lock-free memory allocations.

Exception Handling
: The C++ exception handling does not give any guarantees about its real-time behavior. We therefore do not encourage the use of exceptions and exception handling in lock-free code.

Data Structures

`boost.lockfree` implements three lock-free data structures:

`boost::lockfree::queue`
: a lock-free multi-producer/multi-consumer queue

`boost::lockfree::stack`
: a lock-free multi-producer/multi-consumer stack

`boost::lockfree::spsc_queue`
: a wait-free single-producer/single-consumer queue (commonly known as ringbuffer)

Data Structure Configuration

The data structures can be configured with Boost.Parameter-style templates:

`boost::lockfree::fixed_sized`

Configures the data structure as **fixed sized**. The internal nodes are stored inside an array and they are addressed by array indexing. This limits the possible size of the queue to the number of elements that can be addressed by the index type (usually 2**16-2), but on platforms that lack double-width compare-and-exchange instructions, this is the best way to achieve lock-freedom.

`boost::lockfree::capacity`
Sets the **capacity** of a data structure at compile-time. This implies that a data structure is fixed-sized.

`boost::lockfree::allocator`
Defines the allocator. `boost.lockfree` supports stateful allocator and is compatible with Boost.Interprocess allocators.

---

[7] Spinlocks do not directly interact with the operating system either. However it is possible that the owning thread is preempted by the operating system, which violates the lock-free property.

[8] Intel's Thread Building Blocks library provides many efficient concurrent data structures, which are not necessarily lock-free.
