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 [1]. 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 [2].

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.

[1] 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.
[2] Intel's Thread Building Blocks library provides many efficient concurrent data structures, which are not necessarily lock-free.

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-produced/multi-consumer queue

`boost::lockfree::stack`
: a lock-free multi-produced/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.

Examples

Queue

The `boost::lockfree::queue` class implements a multi-writer/multi-reader queue. The following example shows how integer values are produced and consumed by 4 threads each:

```
#include <boost/thread/thread.hpp>
#include <boost/lockfree/queue.hpp>
#include <iostream>

#include <boost/atomic.hpp>

boost::atomic_int producer_count(0);
boost::atomic_int consumer_count(0);

boost::lockfree::queue<int> queue(128);

const int iterations = 10000000;
const int producer_thread_count = 4;
const int consumer_thread_count = 4;

void producer(void)
{
    for (int i = 0; i != iterations; ++i) {
        int value = ++producer_count;
        while (!queue.push(value))
            ;
    }
}

boost::atomic<bool> done (false);
void consumer(void)
{
    int value;
    while (!done) {
        while (queue.pop(value))
            ++consumer_count;
    }

    while (queue.pop(value))
        ++consumer_count;
}

int main(int argc, char* argv[])
{
    using namespace std;
    cout << "boost::lockfree::queue is ";
    if (!queue.is_lock_free())
        cout << "not ";
    cout << "lockfree" << endl;

    boost::thread_group producer_threads, consumer_threads;

    for (int i = 0; i != producer_thread_count; ++i)
        producer_threads.create_thread(producer);

    for (int i = 0; i != consumer_thread_count; ++i)
        consumer_threads.create_thread(consumer);

    producer_threads.join_all();
```

```
    done = true;

    consumer_threads.join_all();

    cout << "produced " << producer_count << " objects." << endl;
    cout << "consumed " << consumer_count << " objects." << endl;
}
```

The program output is:

```
produced 40000000 objects.
consumed 40000000 objects.
```

Stack

The `boost::lockfree::stack` class implements a multi-writer/multi-reader stack. The following example shows how integer values are produced and consumed by 4 threads each:

```
#include <boost/thread/thread.hpp>
#include <boost/lockfree/stack.hpp>
#include <iostream>

#include <boost/atomic.hpp>

boost::atomic_int producer_count(0);
boost::atomic_int consumer_count(0);

boost::lockfree::stack<int> stack(128);

const int iterations = 1000000;
const int producer_thread_count = 4;
const int consumer_thread_count = 4;

void producer(void)
{
    for (int i = 0; i != iterations; ++i) {
        int value = ++producer_count;
        while (!stack.push(value))
            ;
    }
}

boost::atomic<bool> done (false);

void consumer(void)
{
    int value;
    while (!done) {
        while (stack.pop(value))
            ++consumer_count;
    }

    while (stack.pop(value))
        ++consumer_count;
}

int main(int argc, char* argv[])
{
    using namespace std;
    cout << "boost::lockfree::stack is ";
```

```
    if (!stack.is_lock_free())
        cout << "not ";
    cout << "lockfree" << endl;

    boost::thread_group producer_threads, consumer_threads;

    for (int i = 0; i != producer_thread_count; ++i)
        producer_threads.create_thread(producer);

    for (int i = 0; i != consumer_thread_count; ++i)
        consumer_threads.create_thread(consumer);

    producer_threads.join_all();
    done = true;

    consumer_threads.join_all();

    cout << "produced " << producer_count << " objects." << endl;
    cout << "consumed " << consumer_count << " objects." << endl;
}
```

The program output is:

```
produced 4000000 objects.
consumed 4000000 objects.
```

Waitfree Single-Producer/Single-Consumer Queue

The `boost::lockfree::spsc_queue` class implements a wait-free single-producer/single-consumer queue. The following example shows how integer values are produced and consumed by 2 separate threads:

```
#include <boost/thread/thread.hpp>
#include <boost/lockfree/spsc_queue.hpp>
#include <iostream>

#include <boost/atomic.hpp>

int producer_count = 0;
boost::atomic_int consumer_count (0);

boost::lockfree::spsc_queue<int, boost::lockfree::capacity<1024> > spsc_queue;

const int iterations = 10000000;

void producer(void)
{
    for (int i = 0; i != iterations; ++i) {
        int value = ++producer_count;
        while (!spsc_queue.push(value))
            ;
    }
}

boost::atomic<bool> done (false);

void consumer(void)
{
    int value;
    while (!done) {
        while (spsc_queue.pop(value))
            ++consumer_count;
    }

    while (spsc_queue.pop(value))
        ++consumer_count;
}

int main(int argc, char* argv[])
{
    using namespace std;
    cout << "boost::lockfree::queue is ";
    if (!spsc_queue.is_lock_free())
        cout << "not ";
    cout << "lockfree" << endl;

    boost::thread producer_thread(producer);
    boost::thread consumer_thread(consumer);

    producer_thread.join();
    done = true;
    consumer_thread.join();

    cout << "produced " << producer_count << " objects." << endl;
    cout << "consumed " << consumer_count << " objects." << endl;
}
```

The program output is:

```
produced 10000000 objects.
consumed 10000000 objects.
```

Rationale

Data Structures

The implementations are implementations of well-known data structures. The queue is based on Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Michael Scott and Maged Michael, the stack is based on Systems programming: coping with parallelism by R. K. Treiber and the spsc_queue is considered as 'folklore' and is implemented in several open-source projects including the linux kernel. All data structures are discussed in detail in "The Art of Multiprocessor Programming" by Herlihy & Shavit.

Memory Management

The lock-free `boost::lockfree::queue` and `boost::lockfree::stack` classes are node-based data structures, based on a linked list. Memory management of lock-free data structures is a non-trivial problem, because we need to avoid that one thread frees an internal node, while another thread still uses it. `boost.lockfree` uses a simple approach not returning any memory to the operating system. Instead they maintain a **free-list** in order to reuse them later. This is done for two reasons: first, depending on the implementation of the memory allocator freeing the memory may block (so the implementation would not be lock-free anymore), and second, most memory reclamation algorithms are patented.

ABA Prevention

The ABA problem is a common problem when implementing lock-free data structures. The problem occurs when updating an atomic variable using a `compare_exchange` operation: if the value A was read, thread 1 changes it to say C and tries to update the variable, it uses `compare_exchange` to write C, only if the current value is A. This might be a problem if in the meanwhile thread 2 changes the value from A to B and back to A, because thread 1 does not observe the change of the state. The common way to avoid the ABA problem is to associate a version counter with the value and change both atomically.

`boost.lockfree` uses a `tagged_ptr` helper class which associates a pointer with an integer tag. This usually requires a double-width `compare_exchange`, which is not available on all platforms. IA32 did not provide the `cmpxchg8b` opcode before the pentium processor and it is also lacking on many RISC architectures like PPC. Early X86-64 processors also did not provide a `cmpxchg16b` instruction. On 64bit platforms one can work around this issue, because often not the full 64bit address space is used. On X86_64 for example, only 48bit are used for the address, so we can use the remaining 16bit for the ABA prevention tag. For details please consult the implementation of the `boost::lockfree::detail::tagged_ptr` class.

For lock-free operations on 32bit platforms without double-width `compare_exchange`, we support a third approach: by using a fixed-sized array to store the internal nodes we can avoid the use of 32bit pointers, but instead 16bit indices into the array are sufficient. However this is only possible for fixed-sized data structures, that have an upper bound of internal nodes.

Interprocess Support

The `boost.lockfree` data structures have basic support for Boost.Interprocess. The only problem is the blocking emulation of lock-free atomics, which in the current implementation is not guaranteed to be interprocess-safe.

Reference

Header <boost/lockfree/policies.hpp>

```
namespace boost {
  namespace lockfree {
    template<bool IsFixedSized> struct fixed_sized;
    template<size_t Size> struct capacity;
    template<typename Alloc> struct allocator;
  }
}
```

Struct template fixed_sized

boost::lockfree::fixed_sized

Synopsis

```
// In header: <boost/lockfree/policies.hpp>

template<bool IsFixedSized>
struct fixed_sized : public boost::parameter::template_keyword< tag::fixed_sized, ↵
boost::mpl::bool_< IsFixedSized > >
{
};
```

Description

Configures a 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. This implies that a data structure is bounded.

Struct template capacity

boost::lockfree::capacity

Synopsis

```
// In header: <boost/lockfree/policies.hpp>

template<size_t Size>
struct capacity : public boost::parameter::template_keyword< tag::capacity, boost::mpl::size_t< ↵
Size > >
{
};
```

Description

Sets the **capacity** of a data structure at compile-time.

This implies that a data structure is bounded and fixed-sized.

Struct template allocator

boost::lockfree::allocator

Synopsis

```
// In header: <boost/lockfree/policies.hpp>

template<typename Alloc>
struct allocator :
  public boost::parameter::template_keyword< tag::allocator, Alloc >
{
};
```

Description

Defines the **allocator** type of a data structure.

Header <boost/lockfree/queue.hpp>

```
namespace boost {
  namespace lockfree {
    template<typename T, ... Options> class queue;
  }
}
```

Class template queue

boost::lockfree::queue

Synopsis

```
// In header: <boost/lockfree/queue.hpp>

template<typename T, ... Options>
class queue {
public:
  // types
  typedef T                                   value_type;
  typedef implementation_defined::allocator allocator;
  typedef implementation_defined::size_type size_type;

  // construct/copy/destruct
  queue(void);
  template<typename U>
    explicit queue(typename node_allocator::template rebind< U >::other const &);
  explicit queue(allocator const &);
  explicit queue(size_type);
  template<typename U>
    queue(size_type,
          typename node_allocator::template rebind< U >::other const &);
  queue(queue const &) = delete;
  queue(queue &&) = delete;
  queue& operator=(const queue &) = delete;
  ~queue(void);

  // public member functions
  bool is_lock_free(void) const;
  void reserve(size_type);
  void reserve_unsafe(size_type);
  bool empty(void);
  bool push(T const &);
  bool bounded_push(T const &);
  bool unsynchronized_push(T const &);
  bool pop(T &);
  template<typename U> bool pop(U &);
  bool unsynchronized_pop(T &);
  template<typename U> bool unsynchronized_pop(U &);
  template<typename Functor> bool consume_one(Functor &);
  template<typename Functor> bool consume_one(Functor const &);
  template<typename Functor> size_t consume_all(Functor &);
  template<typename Functor> size_t consume_all(Functor const &);
};
```

Description

The queue class provides a multi-writer/multi-reader queue, pushing and popping is lock-free, construction/destruction has to be synchronized. It uses a freelist for memory management, freed nodes are pushed to the freelist and not returned to the OS before the queue is destroyed.

Policies:

- boost::lockfree::fixed_sized, defaults to `boost::lockfree::fixed_sized<false>`
  Can be used to completely disable dynamic memory allocations during push in order to ensure lockfree behavior.
  If the data structure is configured 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, optional
  If this template argument is passed to the options, the size of the queue is set at compile-time.
  It this option implies `fixed_sized<true>`

- boost::lockfree::allocator, defaults to `boost::lockfree::allocator<std::allocator<void>>`
  Specifies the allocator that is used for the internal freelist

Requirements:

- T must have a copy constructor
- T must have a trivial assignment operator
- T must have a trivial destructor

queue public construct/copy/destruct

1.
```
queue(void);
```

Construct queue.

2.
```
template<typename U>
  explicit queue(typename node_allocator::template rebind< U >::other const & alloc);
```

3.
```
explicit queue(allocator const & alloc);
```

4.
```
explicit queue(size_type n);
```

Construct queue, allocate n nodes for the freelist.

5.
```
template<typename U>
  queue(size_type n,
        typename node_allocator::template rebind< U >::other const & alloc);
```

6.
```
queue(queue const &) = delete;
```

7.
```
queue(queue &&) = delete;
```

8.
```
queue& operator=(const queue &) = delete;
```

9.
```
~queue(void);
```

Destroys queue, free all nodes from freelist.

queue public member functions

1.
```
bool is_lock_free(void) const;
```

Warning

It only checks, if the queue head and tail nodes and the freelist can be modified in a lock-free manner. On most platforms, the whole implementation is lock-free, if this is true. Using c++0x-style atomics, there is no possibility to provide a completely accurate implementation, because one would need to test every internal node, which is impossible if further nodes will be allocated from the operating system.

Returns: true, if implementation is lock-free.

2.
```
void reserve(size_type n);
```

Allocate n nodes for freelist

Note

thread-safe, may block if memory allocator blocks

Requires: only valid if no capacity<> argument given

3.
```
void reserve_unsafe(size_type n);
```

Allocate n nodes for freelist

Note

not thread-safe, may block if memory allocator blocks

Requires: only valid if no capacity<> argument given

4.
```
bool empty(void);
```

Check if the queue is empty

Note

The result is only accurate, if no other thread modifies the queue. Therefore it is rarely practical to use this value in program logic.

Returns: true, if the queue is empty, false otherwise

5.
```
bool push(T const & t);
```

Pushes object t to the queue.

Note

Thread-safe. If internal memory pool is exhausted and the memory pool is not fixed-sized, a new node will be allocated from the OS. This may not be lock-free.

Postconditions: object will be pushed to the queue, if internal node can be allocated

Returns: true, if the push operation is successful.

6.
```
bool bounded_push(T const & t);
```

Pushes object t to the queue.

Note

Thread-safe and non-blocking. If internal memory pool is exhausted, operation will fail

Postconditions: object will be pushed to the queue, if internal node can be allocated
Returns: true, if the push operation is successful.
Throws: if memory allocator throws

7.
```
bool unsynchronized_push(T const & t);
```

Pushes object t to the queue.

Note

Not Thread-safe. If internal memory pool is exhausted and the memory pool is not fixed-sized, a new node will be allocated from the OS. This may not be lock-free.

Postconditions: object will be pushed to the queue, if internal node can be allocated
Returns: true, if the push operation is successful.
Throws: if memory allocator throws

8.
```
bool pop(T & ret);
```

Pops object from queue.

Note

Thread-safe and non-blocking

Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if queue was empty.

9.
```
template<typename U> bool pop(U & ret);
```

Pops object from queue.

Note

Thread-safe and non-blocking

Requires: type U must be constructible by T and copyable, or T must be convertible to U
Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if queue was empty.

10.
```
bool unsynchronized_pop(T & ret);
```

Pops object from queue.

Note

Not thread-safe, but non-blocking

Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if queue was empty.

11.
```
template<typename U> bool unsynchronized_pop(U & ret);
```

Pops object from queue.

Note

Not thread-safe, but non-blocking

Requires: type U must be constructible by T and copyable, or T must be convertible to U
Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if queue was empty.

12.
```
template<typename Functor> bool consume_one(Functor & f);
```

consumes one element via a functor

pops one element from the queue and applies the functor on this object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: true, if one element was consumed

13.
```
template<typename Functor> bool consume_one(Functor const & f);
```

consumes one element via a functor

pops one element from the queue and applies the functor on this object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: true, if one element was consumed

14.
```
template<typename Functor> size_t consume_all(Functor & f);
```

consumes all elements via a functor

sequentially pops all elements from the queue and applies the functor on each object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: number of elements that are consumed

15.
```
template<typename Functor> size_t consume_all(Functor const & f);
```

consumes all elements via a functor

sequentially pops all elements from the queue and applies the functor on each object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: number of elements that are consumed

Header <boost/lockfree/spsc_queue.hpp>

```
namespace boost {
  namespace lockfree {
    template<typename T, ... Options> class spsc_queue;
  }
}
```

Class template spsc_queue

boost::lockfree::spsc_queue

Synopsis

```
// In header: <boost/lockfree/spsc_queue.hpp>

template<typename T, ... Options>
class spsc_queue {
public:
  // types
  typedef T                                    value_type;
  typedef implementation_defined::allocator allocator;
  typedef implementation_defined::size_type size_type;

  // construct/copy/destruct
  spsc_queue(void);
  template<typename U>
    explicit spsc_queue(typename allocator::template rebind< U >::other const &);
  explicit spsc_queue(allocator const &);
  explicit spsc_queue(size_type);
  template<typename U>
    spsc_queue(size_type,
               typename allocator::template rebind< U >::other const &);
  spsc_queue(size_type, allocator_arg const &);

  // public member functions
  bool push(T const &);
  bool pop(T &);
  size_type push(T const *, size_type);
  template<size_type size> size_type push(T const (&));
  template<typename ConstIterator>
    ConstIterator push(ConstIterator, ConstIterator);
  size_type pop(T *, size_type);
  template<size_type size> size_type pop(T(&));
  template<typename OutputIterator> size_type pop(OutputIterator);
  template<typename Functor> bool consume_one(Functor &);
  template<typename Functor> bool consume_one(Functor const &);
  template<typename Functor> size_type consume_all(Functor &);
  template<typename Functor> size_type consume_all(Functor const &);
};
```

Description

The spsc_queue class provides a single-writer/single-reader fifo queue, pushing and popping is wait-free.

Policies:

- `boost::lockfree::capacity<>`, optional
  If this template argument is passed to the options, the size of the ringbuffer is set at compile-time.

- `boost::lockfree::allocator<>`, defaults to `boost::lockfree::allocator<std::allocator<T>>`
  Specifies the allocator that is used to allocate the ringbuffer. This option is only valid, if the ringbuffer is configured to be sized at run-time

Requirements:

- T must have a default constructor

- T must be copyable

spsc_queue public construct/copy/destruct

1.
```
spsc_queue(void);
```

Constructs a spsc_queue

Requires: spsc_queue must be configured to be sized at compile-time

2.
```
template<typename U>
  explicit spsc_queue(typename allocator::template rebind< U >::other const & alloc);
```

3.
```
explicit spsc_queue(allocator const & alloc);
```

4.
```
explicit spsc_queue(size_type element_count);
```

Constructs a spsc_queue for element_count elements

Requires: spsc_queue must be configured to be sized at run-time

5.
```
template<typename U>
  spsc_queue(size_type element_count,
             typename allocator::template rebind< U >::other const & alloc);
```

6.
```
spsc_queue(size_type element_count, allocator_arg const & alloc);
```

spsc_queue public member functions

1.
```
bool push(T const & t);
```

Pushes object t to the ringbuffer.

Note

Thread-safe and wait-free

Requires: only one thread is allowed to push data to the spsc_queue
Postconditions: object will be pushed to the spsc_queue, unless it is full.
Returns: true, if the push operation is successful.

2.
```
bool pop(T & ret);
```

Pops one object from ringbuffer.

Note

Thread-safe and wait-free

Requires: only one thread is allowed to pop data to the spsc_queue

Postconditions: if ringbuffer is not empty, object will be copied to ret.
Returns: true, if the pop operation is successful, false if ringbuffer was empty.

3.
```
size_type push(T const * t, size_type size);
```

Pushes as many objects from the array t as there is space.

> **Note**
>
> Thread-safe and wait-free

Requires: only one thread is allowed to push data to the spsc_queue
Returns: number of pushed items

4.
```
template<size_type size> size_type push(T const (&) t);
```

Pushes as many objects from the array t as there is space available.

> **Note**
>
> Thread-safe and wait-free

Requires: only one thread is allowed to push data to the spsc_queue
Returns: number of pushed items

5.
```
template<typename ConstIterator>
  ConstIterator push(ConstIterator begin, ConstIterator end);
```

Pushes as many objects from the range [begin, end) as there is space .

> **Note**
>
> Thread-safe and wait-free

Requires: only one thread is allowed to push data to the spsc_queue
Returns: iterator to the first element, which has not been pushed

6.
```
size_type pop(T * ret, size_type size);
```

Pops a maximum of size objects from ringbuffer.

> **Note**
>
> Thread-safe and wait-free

Requires: only one thread is allowed to pop data to the spsc_queue
Returns: number of popped items

7.
```
template<size_type size> size_type pop(T(&) ret);
```

Pops a maximum of size objects from `spsc_queue`.

Note

Thread-safe and wait-free

Requires: only one thread is allowed to pop data to the spsc_queue
Returns: number of popped items

8.
```
template<typename OutputIterator> size_type pop(OutputIterator it);
```

Pops objects to the output iterator it

Note

Thread-safe and wait-free

Requires: only one thread is allowed to pop data to the spsc_queue
Returns: number of popped items

9.
```
template<typename Functor> bool consume_one(Functor & f);
```

consumes one element via a functor

pops one element from the queue and applies the functor on this object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: true, if one element was consumed

10.
```
template<typename Functor> bool consume_one(Functor const & f);
```

consumes one element via a functor

pops one element from the queue and applies the functor on this object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: true, if one element was consumed

11.
```
template<typename Functor> size_type consume_all(Functor & f);
```

consumes all elements via a functor

sequentially pops all elements from the queue and applies the functor on each object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: number of elements that are consumed

12.
```
template<typename Functor> size_type consume_all(Functor const & f);
```

consumes all elements via a functor

sequentially pops all elements from the queue and applies the functor on each object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: number of elements that are consumed

Header <boost/lockfree/stack.hpp>

```
namespace boost {
  namespace lockfree {
    template<typename T, ... Options> class stack;
  }
}
```

Class template stack

boost::lockfree::stack

Synopsis

```
// In header: <boost/lockfree/stack.hpp>

template<typename T, ... Options>
class stack {
public:
  // types
  typedef T                                    value_type;
  typedef implementation_defined::allocator allocator;
  typedef implementation_defined::size_type size_type;

  // construct/copy/destruct
  stack(void);
  template<typename U>
    explicit stack(typename node_allocator::template rebind< U >::other const &);
  explicit stack(allocator const &);
  explicit stack(size_type);
  template<typename U>
    stack(size_type,
          typename node_allocator::template rebind< U >::other const &);
  stack(stack const &) = delete;
  stack(stack &&) = delete;
  stack& operator=(const stack &) = delete;
  ~stack(void);

  // public member functions
  bool is_lock_free(void) const;
  void reserve(size_type);
  void reserve_unsafe(size_type);
  bool push(T const &);
  bool bounded_push(T const &);
  template<typename ConstIterator>
    ConstIterator push(ConstIterator, ConstIterator);
  template<typename ConstIterator>
    ConstIterator bounded_push(ConstIterator, ConstIterator);
  bool unsynchronized_push(T const &);
  template<typename ConstIterator>
    ConstIterator unsynchronized_push(ConstIterator, ConstIterator);
  bool pop(T &);
  template<typename U> bool pop(U &);
  bool unsynchronized_pop(T &);
  template<typename U> bool unsynchronized_pop(U &);
  template<typename Functor> bool consume_one(Functor &);
  template<typename Functor> bool consume_one(Functor const &);
  template<typename Functor> size_t consume_all(Functor &);
  template<typename Functor> size_t consume_all(Functor const &);
  bool empty(void) const;
};
```

Description

The stack class provides a multi-writer/multi-reader stack, pushing and popping is lock-free, construction/destruction has to be synchronized. It uses a freelist for memory management, freed nodes are pushed to the freelist and not returned to the OS before the stack is destroyed.

Policies:

- `boost::lockfree::fixed_sized<>`, defaults to `boost::lockfree::fixed_sized<false>`
  Can be used to completely disable dynamic memory allocations during push in order to ensure lockfree behavior.

If the data structure is configured 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 stack 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<>`, optional
  If this template argument is passed to the options, the size of the stack is set at compile-time.
  It this option implies `fixed_sized<true>`

- `boost::lockfree::allocator<>`, defaults to `boost::lockfree::allocator<std::allocator<void>>`
  Specifies the allocator that is used for the internal freelist

Requirements:

- T must have a copy constructor

`stack` public construct/copy/destruct

1.
```
stack(void);
```

Construct stack.

2.
```
template<typename U>
  explicit stack(typename node_allocator::template rebind< U >::other const & alloc);
```

3.
```
explicit stack(allocator const & alloc);
```

4.
```
explicit stack(size_type n);
```

Construct stack, allocate n nodes for the freelist.

5.
```
template<typename U>
  stack(size_type n,
        typename node_allocator::template rebind< U >::other const & alloc);
```

6.
```
stack(stack const &) = delete;
```

7.
```
stack(stack &&) = delete;
```

8.
```
stack& operator=(const stack &) = delete;
```

9.
```
~stack(void);
```

Destroys stack, free all nodes from freelist.

Note

not thread-safe

`stack` public member functions

1. ```
   bool is_lock_free(void) const;
   ```

   

   Warning

   It only checks, if the top stack node and the freelist can be modified in a lock-free manner. On most platforms, the whole implementation is lock-free, if this is true. Using c++0x-style atomics, there is no possibility to provide a completely accurate implementation, because one would need to test every internal node, which is impossible if further nodes will be allocated from the operating system.

   Returns: true, if implementation is lock-free.

2. ```
   void reserve(size_type n);
   ```

   Allocate n nodes for freelist

   

   Note

   thread-safe, may block if memory allocator blocks

   Requires: only valid if no capacity<> argument given

3. ```
   void reserve_unsafe(size_type n);
   ```

   Allocate n nodes for freelist

   

   Note

   not thread-safe, may block if memory allocator blocks

   Requires: only valid if no capacity<> argument given

4. ```
   bool push(T const & v);
   ```

   Pushes object t to the stack.

   

   Note

   Thread-safe. If internal memory pool is exhausted and the memory pool is not fixed-sized, a new node will be allocated from the OS. This may not be lock-free.

   Postconditions: object will be pushed to the stack, if internal node can be allocated
   Returns: true, if the push operation is successful.
   Throws: if memory allocator throws

5. ```
   bool bounded_push(T const & v);
   ```

   Pushes object t to the stack.
   ```

Note

Thread-safe and non-blocking. If internal memory pool is exhausted, the push operation will fail

Postconditions: object will be pushed to the stack, if internal node can be allocated
Returns: true, if the push operation is successful.

6.
```
template<typename ConstIterator>
  ConstIterator push(ConstIterator begin, ConstIterator end);
```

Pushes as many objects from the range [begin, end) as freelist node can be allocated.

Note

Operation is applied atomically

Thread-safe. If internal memory pool is exhausted and the memory pool is not fixed-sized, a new node will be allocated from the OS. This may not be lock-free.

Returns: iterator to the first element, which has not been pushed
Throws: if memory allocator throws

7.
```
template<typename ConstIterator>
  ConstIterator bounded_push(ConstIterator begin, ConstIterator end);
```

Pushes as many objects from the range [begin, end) as freelist node can be allocated.

Note

Operation is applied atomically

Thread-safe and non-blocking. If internal memory pool is exhausted, the push operation will fail

Returns: iterator to the first element, which has not been pushed
Throws: if memory allocator throws

8.
```
bool unsynchronized_push(T const & v);
```

Pushes object t to the stack.

Note

Not thread-safe. If internal memory pool is exhausted and the memory pool is not fixed-sized, a new node will be allocated from the OS. This may not be lock-free.

Postconditions: object will be pushed to the stack, if internal node can be allocated
Returns: true, if the push operation is successful.
Throws: if memory allocator throws

9.
```
template<typename ConstIterator>
  ConstIterator unsynchronized_push(ConstIterator begin, ConstIterator end);
```

Pushes as many objects from the range [begin, end) as freelist node can be allocated.

Note

Not thread-safe. If internal memory pool is exhausted and the memory pool is not fixed-sized, a new node will be allocated from the OS. This may not be lock-free.

Returns: iterator to the first element, which has not been pushed
Throws: if memory allocator throws

10.
```
bool pop(T & ret);
```

Pops object from stack.

Note

Thread-safe and non-blocking

Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if stack was empty.

11.
```
template<typename U> bool pop(U & ret);
```

Pops object from stack.

Note

Thread-safe and non-blocking

Requires: type T must be convertible to U
Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if stack was empty.

12.
```
bool unsynchronized_pop(T & ret);
```

Pops object from stack.

Note

Not thread-safe, but non-blocking

Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if stack was empty.

13.
```
template<typename U> bool unsynchronized_pop(U & ret);
```

Pops object from stack.

Note

Not thread-safe, but non-blocking

Requires: type T must be convertible to U
Postconditions: if pop operation is successful, object will be copied to ret.
Returns: true, if the pop operation is successful, false if stack was empty.

14. ```
template<typename Functor> bool consume_one(Functor & f);
```

consumes one element via a functor

pops one element from the stack and applies the functor on this object

**Note**

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: true, if one element was consumed

15. ```
template<typename Functor> bool consume_one(Functor const & f);
```

consumes one element via a functor

pops one element from the stack and applies the functor on this object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: true, if one element was consumed

16. ```
template<typename Functor> size_t consume_all(Functor & f);
```

consumes all elements via a functor

sequentially pops all elements from the stack and applies the functor on each object

**Note**

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: number of elements that are consumed

17. ```
template<typename Functor> size_t consume_all(Functor const & f);
```

consumes all elements via a functor

sequentially pops all elements from the stack and applies the functor on each object

Note

Thread-safe and non-blocking, if functor is thread-safe and non-blocking

Returns: number of elements that are consumed

18.
```
bool empty(void) const;
```

Note

It only guarantees that at some point during the execution of the function the stack has been empty. It is rarely practical to use this value in program logic, because the stack can be modified by other threads.

Returns: true, if stack is empty.

Appendices

Supported Platforms & Compilers

`boost.lockfree` has been tested on the following platforms:

- g++ 4.4, 4.5 and 4.6, linux, x86 & x86_64
- clang++ 3.0, linux, x86 & x86_64

Future Developments

- More data structures (set, hash table, dequeue)
- Backoff schemes (exponential backoff or elimination)

References

1. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Michael Scott and Maged Michael, In Symposium on Principles of Distributed Computing, pages 267–275, 1996.
2. M. Herlihy & Nir Shavit. The Art of Multiprocessor Programming, Morgan Kaufmann Publishers, 2008