Chapter 46. Boost.Lockfree

Boost.Lockfree provides thread-safe and lock-free containers. Containers from this library can be accessed from multiple threads without having to synchronize access.

In version 1.56.0, Boost.Lockfree provides only two containers: a queue of type `boost::lockfree::queue` and a stack of type `boost::lockfree::stack`. For the queue, a second implementation is available: `boost::lockfree::spsc_queue`. This class is optimized for use cases where exactly one thread writes to the queue and exactly one thread reads from the queue. The abbreviation spsc in the class name stands for single producer/single consumer.

Example 46.1. Using `boost::lockfree::spsc_queue`

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

boost::lockfree::spsc_queue<int> q{100};
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  consume();
  std::cout << sum << '\n';
}
```

Example 46.1 uses the container `boost::lockfree::spsc_queue`. The first thread, which executes the function `produce()`, adds the numbers 1 to 100 to the container. The second thread, which executes `consume()`, reads the numbers from the container and adds them up in **`sum`**. Because the container `boost::lockfree::spsc_queue` explicitly supports concurrent access from two threads, it isn't necessary to synchronize the threads.

Please note that the function `consume()` is called a second time after the threads terminate. This is required to calculate the total of all 100 numbers, which is 5050. Because `consume()` accesses the queue in a loop, it is possible that it will read the numbers faster than they are inserted by `produce()`. If the queue is empty, `pop()` returns `false`. Thus, the thread executing `consume()` could terminate because `produce()` in the other thread couldn't fill the queue fast enough. If the thread executing `produce()` is terminated, then it's clear that all of the numbers

were added to the queue. The second call to `consume()` makes sure that numbers that may not have been read yet are added to **`sum`**.

The size of the queue is passed to the constructor. Because `boost::lockfree::spsc_queue` is implemented with a circular buffer, the queue in Example 46.1 has a capacity of 100 elements. If a value can't be added because the queue is full, `push()` returns `false`. The example doesn't check the return value of `push()` because exactly 100 numbers are added to the queue. Thus, 100 elements is sufficient.

Example 46.2. `boost::lockfree::spsc_queue` with `boost::lockfree::capacity`

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

using namespace boost::lockfree;

spsc_queue<int, capacity<100>> q;
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  while (q.consume_one([](int i){ sum += i; }))
    ;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  q.consume_all([](int i){ sum += i; });
  std::cout << sum << '\n';
}
```

Example 46.2 works like the previous example, but this time the size of the circular buffer is set at compile time. This is done with the template `boost::lockfree::capacity`, which expects the capacity as a template parameter. **`q`** is instantiated with the default constructor – the capacity cannot be set at run time.

The function `consume()` has been changed to use `consume_one()`, rather than `pop()`, to read the number. A lambda function is passed as a parameter to `consume_one()`. `consume_one()` reads a number just like `pop()`, but the number isn't returned through a reference to the caller. It is passed as the sole parameter to the lambda function.

When the threads terminate, `main()` calls the member function `consume_all()`, instead of `consume()`. `consume_all()` works like `consume_one()` but makes sure that the queue is empty after the call. `consume_all()` calls the lambda function as long as there are elements in the queue.

Once again, Example 46.2 writes `5050` to standard output.

Example 46.3. `boost::lockfree::queue` with variable container size

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

boost::lockfree::queue<int> q{100};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

Example 46.3 executes `consume()` in two threads. Because more than one thread reads from the queue, the class `boost::lockfree::spsc_queue` must not be used. This example uses `boost::lockfree::queue` instead.

Thanks to `std::atomic`, access to the variable **`sum`** is also now thread safe.

The size of the queue is set to 100 – this is the parameter passed to the constructor. However, this is only the initial size. By default, `boost::lockfree::queue` is not implemented with a circular buffer. If more items are added to the queue than the capacity is set to, it is automatically increased. `boost::lockfree::queue` dynamically allocates additional memory if the initial size isn't sufficient.

That means that `boost::lockfree::queue` isn't necessarily lock free. The allocator used by `boost::lockfree::queue` by default is `boost::lockfree::allocator`, which is based on `std::allocator`. Thus, this allocator determines whether `boost::lockfree::queue` is lock free without constraints.

Example 46.4. `boost::lockfree::queue` with constant container size

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

```
using namespace boost::lockfree;

queue<int, fixed_sized<true>> q{10000};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

Example 46.4 uses a constant size of 10,000 elements. In this example, the queue doesn't allocate additional memory when it is full. 10,000 is a fixed upper limit.

The queue's capacity is constant because `boost::lockfree::fixed_sized` is passed as a template parameter. The capacity is passed as a parameter to the constructor and can be updated at any time using `reserve()`. If the capacity must be set at compile time, `boost::lockfree::capacity` can be passed as a template parameter to `boost::lockfree::queue`. `boost::lockfree::capacity` includes `boost::lockfree::fixed_sized`.

In Example 46.4, the queue has a capacity of 10,000 elements. Because `consume()` inserts 10,000 numbers into the queue, the upper limit isn't exceeded. If it were exceeded, `push()` would return `false`.

`boost::lockfree::queue` is similar to `boost::lockfree::spsc_queue` and also provides member functions like `consume_one()` and `consume_all()`.

The third class, `boost::lockfree::stack`, is similar to the other ones. As with `boost::lockfree::queue`, `boost::lockfree::fixed_size` and `boost::lockfree::capacity` can be passed as template parameters. The member functions are similar, too.

Exercise

Remove the class `boost::lockfree::spsc_queue` from Example 46.1 and implement the program with `std::queue`.