Advanced Parallel Programming Exercise 5

Marvin Kaster

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer term 2025
26.06.2025

Please solve the following tasks by 26.06.2025. The results are not graded, but a solution is discussed on 26.06.2025.

Task 1: Implement a thread-safe array

Implement a thread-safe lock-based array. You can write a thread-safe wrapper around `std::array`. Your implementation should have the interface shown below. It shall support the concurrent writing on two different indices of the array and concurrent reading of the same index. Please do not use a global lock. Why is it not easily possible to create a thread-safe vector using the same technique?

```cpp
#include <cstddef>

template<typename T, std::size_t N>
class ThreadSafeArray {
private:

public:
    ThreadSafeArray() = default;

    ThreadSafeArray(const ThreadSafeArray &) = delete;

    ThreadSafeArray &operator=(const ThreadSafeArray &) = delete;

    void set(std::size_t index, const T &value);

    T get(std::size_t index) const;

    T operator[](std::size_t index) const;

    std::size_t size() const;

};
```

Listing 1: Thread-safe array

Solution:

```cpp
#include <array>
#include <cstddef>
#include <memory>
#include <mutex>
#include <shared_mutex>


template<typename T, std::size_t N>
class ThreadSafeArray {
private:
    std::array<T, N> data{};
    mutable std::array<std::shared_mutex, N> locks;
```

```

public:
    ThreadSafeArray() = default;

    ThreadSafeArray(const ThreadSafeArray &) = delete;
    ThreadSafeArray &operator=(const ThreadSafeArray &) = delete;

    void set(std::size_t index, const T& value) {
        if (index >= N) {
            throw std::out_of_range("Index out of bounds");
        }
        auto cpy = value;
        std::unique_lock lock(locks[index]);
        std::swap(data[index], cpy);
    }

    T get(std::size_t index) const {
        if (index >= N) {
            throw std::out_of_range("Index out of bounds");
        }
        std::shared_lock lock(locks[index]);
        return data[index];
    }

    T operator[](std::size_t index) const {
        return get(index);
    }

    std::size_t size() const {
        return N;
    }

};
```

Listing 2: Thread-safe array

A `std::vector` could trigger a resizing when a new element is pushed to it. We must ensure that the resizing is not occuring while any other thread accesses an element of the vector.

Task 2: Implement a thread safe queue

Implement a thread-safe lock-based queue. You can write a thread-safe wrapper around `std::queue`. Your implementation should have the interface shown below. Implement the `wait_and_pop()` method using `std::condition_variable`. What is the advantage of returning the elements with a `std::shared_ptr` instead of a copy of the value? Is there an alternative?

```
#pragma once

#include <memory>

template<typename T>
class ThreadSafeQueue {

  public:
  ThreadSafeQueue() = default;

  ThreadSafeQueue(const ThreadSafeQueue &t) = delete;

  /**
  * Removes the first element of the queue and returns it if the queue is not empty
  * @return shared_ptr to the first element or null if the queue is empty
  */
  [[nodiscard]] std::shared_ptr<T> try_pop();

  /**
  * Removes the first element of the queue and returns it. If the queue is empty, this call blocks until
    at least one element is in the queue
  * @return Shared_ptr to the first element
```

```
  */
  [[nodiscard]] std::shared_ptr<T> wait_and_pop();

  /**
  * Pushes the value to the end of the queue
  * @param value The value
  */
  void push(T value);

  /**
  * Returns true if the queue is empty
  * @return True if the queue is empty
  */
  [[nodiscard]] bool empty();
};

```

Listing 3: ThreadSafeQueue.h

Solution:

The advantage of using `std::shared_ptr` is that we can combine the function for checking if the queue is not empty and popping. Would a thread call empty() and then pop(), a different thread could have already taken the new element, leaving the queue empty. An alternative would be to throw an exception if the stack is empty.

Another advantage is the exception safety. If we would return-by-value, we can only use types that have an exception-safe copy- and/or move-constructor. Otherwise, an exception could be thrown when we return the object from the queue. This is a problem because we already removed the object from the queue. The object would be lost. An alternative would be to pass a reference to our `wait_and_pop()` and `try_and_pop()` function and store the object in the reference. This moves the constructor call to the outside of the function.

```
#pragma once

#include <condition_variable>
#include <memory>
#include <mutex>
#include <queue>

template<typename T>
class ThreadSafeQueue {
  private:
  std::queue<std::shared_ptr<T>> queue;
  std::mutex mutex;
  std::condition_variable cond;

  public:
  ThreadSafeQueue() = default;

  ThreadSafeQueue(const ThreadSafeQueue &t) = delete;

  /**
  * Removes the first element of the queue and returns it if the queue is not empty
  * @return shared_ptr to the first element or null if the queue is empty
  */
  [[nodiscard]] std::shared_ptr<T> try_pop() {
    std::lock_guard<std::mutex> lock(mutex);
    if (queue.empty()) {
      return {};
    }
    auto ptr = queue.front();
    queue.pop();
    return ptr;
  }

  /**
```

```
  * Removes the first element of the queue and returns it. If the queue is empty, this call blocks until
    at least one element is in the queue
  * @return Shared_ptr to the first element
  */
  [[nodiscard]] std::shared_ptr<T> wait_and_pop() {
    std::unique_lock<std::mutex> lock(mutex);
    cond.wait(lock, [&]() { return !queue.empty(); });
    auto ptr = queue.front();
    queue.pop();
    return ptr;
  }

  /**
  * Pushes the value to the end of the queue
  * @param value The value
  */
  void push(T value) {
    std::shared_ptr<T> ptr{std::make_shared<T>(std::move(value))};
    std::lock_guard<std::mutex> lock(mutex);
    queue.push(ptr);
    cond.notify_one();
  }

  /**
  * Returns true if the queue is empty
  * @return True if the queue is empty
  */
  [[nodiscard]] bool empty() {
    std::lock_guard<std::mutex> lock(mutex);
    return queue.empty();
  }
};
```

Listing 4: ThreadSafeQueue.h

Task 3: Implement a thread-safe stack

Implement a thread-safe lock-based stack. You can write a thread-safe wrapper around `std::stack`. Your implementation should have the interface shown below and can throw an exception if the user attempts to `pop()` on an empty stack.

```
#include <memory>
#include <mutex>
#include <stack>


template<typename T>
class ThreadSafeStack {
private:
    std::stack<T> data{};
    mutable std::mutex mutex{};

public:

    ThreadSafeStack() = default;

    ThreadSafeStack(const ThreadSafeStack<T>& other);

    void push(T new_value);

    std::shared_ptr<T> pop();

    void pop(T &value);

    bool empty() const;

```

```
};
```

Listing 5: Thread-safe stack

Solution:

```
#include <memory>
#include <mutex>
#include <stack>


template<typename T>
class ThreadSafeStack {
private:
    std::stack<T> data{};
    mutable std::mutex mutex{};

public:

    ThreadSafeStack() = default;

    ThreadSafeStack(const ThreadSafeStack<T>& other)  {
        std::lock_guard<std::mutex> lock(other.mutex);
        data = other.data;
    }

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(mutex);
        data.push(new_value);
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(mutex);
        if (data.empty())
            throw std::out_of_range("Stack is empty");

        std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
        data.pop();
        return res;
    }

    void pop(T &value) {
        std::lock_guard<std::mutex> lock(mutex);
        if (data.empty()) {
            throw std::out_of_range("Stack is empty");
        }

        value = data.top();
        data.pop();
    }

    bool empty() const  {
        std::lock_guard<std::mutex> lock(mutex);
        return data.empty();
    }

};
```

Listing 6: Thread-safe stack