Advanced Parallel Programming
Exercise 8

Tim Beringer

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Summer term 2025
17.07.2025

Please solve the following tasks by 17.07.2025. The results are not graded, but a solution is discussed on 17.07.2025.

Task 1

Please implement a single-producer, single-consumer lock-free queue. You may use the interface provided as a starting point for your implementation.

```
#pragma once
#include <atomic>
#include <memory>

template<typename T>
class LockFreeQueue
{
private:
    struct Node
    {
        std::shared_ptr<T> data;
        Node* next;

        Node():  next(nullptr) {}
    };

    std::atomic<Node*> head;
    std::atomic<Node*> tail;

    Node* pop_head();

public:
    LockFreeQueue();
    LockFreeQueue(const LockFreeQueue& other)=delete;
    LockFreeQueue& operator=(const LockFreeQueue& other)=delete;
    ~LockFreeQueue();

    void pop(std::shared_ptr<T>& res);
    void push(T new_value);
};
```

Listing 1: Single-producer single-consumer queue interface

Solution:

```
#pragma once

#include <atomic>
#include <memory>

template<typename T>
class LockFreeQueue {
```

```
private:
    struct Node {
        std::shared_ptr<T> data;
        Node* next;

        Node() : next(nullptr) {}
    };

    std::atomic<Node*> head;
    std::atomic<Node*> tail;

    Node* pop_head() {
        Node* const old_head = head.load();
        if (old_head == tail.load()) {
            return nullptr;
        }
        head.store(old_head->next);
        return old_head;
    }

public:
    LockFreeQueue() : head(new Node{}), tail(head.load()) {}
    LockFreeQueue(const LockFreeQueue& other) = delete;
    LockFreeQueue& operator=(const LockFreeQueue& other) = delete;

    ~LockFreeQueue() {
        while (Node* const old_head = head.load()) {
            head.store(old_head->next);
            delete old_head;
        }
    }

    void pop(std::shared_ptr<T>& res) {
        Node* old_head = pop_head();
        if (!old_head) {
            res = std::shared_ptr<T>();
            return;
        }
        res = old_head->data;
        delete old_head;
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
        Node* p = new Node;
        Node* const old_tail = tail.load();
        old_tail->data.swap(new_data);
        old_tail->next = p;
        tail.store(p);
    }
};
```

Listing 2: Thread-safe queue with conditional variables

Task 2: Progress Conditions

Progress conditions are useful to the liveness property. Two progress conditions are interesting to us:

Wait-free: A method is wait-free if it guarantees that every call to it finishes its execution in a finite number of steps. It is *bounded wait-free* if there is a bound on the number of steps a method call can take.

Lock-free: A method is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps. Clearly, any wait-free method is also lock-free, but not vice versa. Lock-free algorithms admit the possibility that some threads could starve.

- Consider the following rather unusual implementation of a method `m`. In every execution history, the i^{th} time a thread calls `m`, the call returns after 2^i steps. Is this method wait-free, bounded wait-free, or neither?

- Is the following property equivalent to saying that object x is lock-free?

  For every infinite execution history of x, an infinite number of method calls are completed.

Solution:

- This method is wait-free because for any given i, 2^i is a finite number. However, this method is not bounded wait-free because 2^i does not converge.
- Yes. Suppose not, then there exist at least a method call of x which is not lock-free. That means for any call to this method, it takes infinite number of steps. Select a history that contains only a finite number of calls to this method, for example, two calls. Then the history is an infinite history, but it completes at maximum two calls, a contradiction.