MINISTRY FOR EDUCATION AND SCIENCE OF RUSSIA

____

SAINT PETERSBURG ELECTROTECHNICAL UNIVERSITY «LETI»

____

Laboratory work № 3
Computational Systems

Student
Group 4300
Bathaie N.

Teacher
PhD, associate prof.
of CE department
Paznikov A.A.

Saint Petersburg
ETU «LETI»
2018

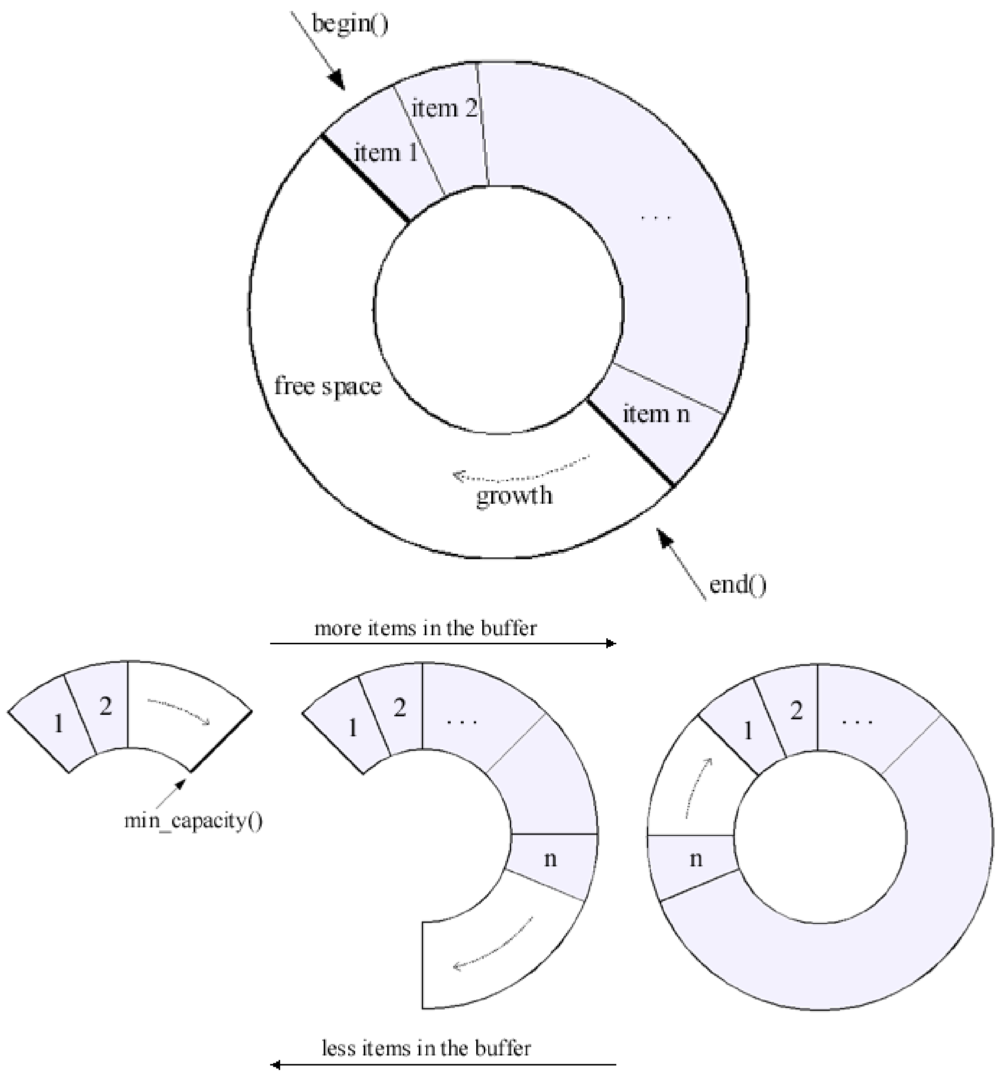
A SURVEY OF THE PERFORMANCE OF DIFFERENT LOCK TYPES ON CONCURRENT CIRCULAR BUFFER.

1. Circular buffer

A circular buffer is a memory allocation scheme where memory is reused (reclaimed) when an index, incremented modulo the buffer size, writes over a previously used location.

A circular buffer makes a bounded queue when separate indices are used for inserting and removing data. The queue can be safely shared between threads (or processors) without further synchronization so long as one processor enqueues data and the other dequeues it. (Also, modifications to the read/write pointers must be atomic, and this is a non-blocking queue--an error is returned when trying to write to a full queue or read from an empty queue).

Note that a circular buffer with n elements is usually used to implement a queue with n-1 elements--there is always one empty element in the buffer. Otherwise, it becomes difficult to distinguish between a full and empty queue--the read and write pointers would be identical in both cases.



2. Additional functions

To have a better assessment, two function is added to the queue which are as follows.

2.1. Resize Function:

To use this function for a couple of times during each experiment it increases the size of the buffer by nopers / 10u. The default queue size for this experiment was also nopers / 10u.

```
template<typename TItem, typename TLock>
void concurrent_queue<TItem, TLock>::resize_queue()
{
    TItem *old_items =  items;
    auto new_capacity = capacity + nopers / 10u;
    items = new TItem[new_capacity];
    auto j = 0;
    for(auto i = head; i < tail; i++)
        items[j++] = old_items[i % capacity];
    head = 0u;
    tail = capacity;
    capacity = new_capacity;
    #ifdef _DEBUG
        std::cout << "queue resized, new capacity: "
            << capacity<< std::endl;
    #endif
    delete[] old_items;
}
```

2.2. Read Function

To compare the performance of read/write buffers with normal mutexes this function added. Within this function we just read data and no change is done to the queue.

```
template<typename TItem, typename TLock>
TItem concurrent_queue<TItem, TLock>::read_queue()
{
    std::shared_lock<TLock> ilock(lock);
    if (head == tail)
        //cond_.wait(ilock);
        throw std::runtime_error("queue is empty");
    auto item = items[head % capacity];
    return item;
}
```

3. Code

```
#include <iostream>
#include <random>
#include <algorithm>
#include <thread>
#include <mutex>
#include <chrono>
#include <fstream>
#include <shared_mutex>
#include <boost/thread/locks.hpp>
#include <boost/thread/shared_mutex.hpp>
//#include <condition_variable>
const auto nruns = 10;
const auto nopers = 1000000;
// #define _DEBUG
class thread_raii
{
public:
    thread_raii(std::thread&& _t): t{std::move(_t)} { }
    thread_raii(thread_raii &&thr_raii): t{std::move(thr_raii.t)} {
}
    ~thread_raii()
    {
        if (t.joinable())
            t.join();
    }
    std::thread& get()
    {
        return t;
    }
    void join()
    {
        if (t.joinable())
            t.join();
    }
private:
    std::thread t;
};
///......................
class spinlock
{
public:
    spinlock()
    {
        pthread_spin_init(&slock, 0);
    }
    ~spinlock()
    {
        pthread_spin_destroy(&slock);
    }
    void lock()
    {
        pthread_spin_lock(&slock);
    }
    void unlock()
```

```
    {
        pthread_spin_unlock(&slock);
    }
private:
    pthread_spinlock_t slock;
};
///..................
template<typename TItem, typename TLock>
class concurrent_queue
{
public:
    concurrent_queue(unsigned _capacity =
concurrent_queue::def_capacity):
        capacity{_capacity}, items{new TItem[_capacity]}
    {}
    ~concurrent_queue()
    {
        delete[] items;
    }
    TItem read_queue();
    TItem dequeue();
    void enqueue(TItem &item);
    unsigned get_capacity()
    {
        return capacity;
    }
    void print();
private:
    void resize_queue();
    static const unsigned def_capacity = nopers / 10u;
    unsigned capacity = def_capacity;
    unsigned head = 0;
    unsigned tail = 0;
    TLock lock;
    TItem *items;
    //std::condition_variable cond_;
};
///..................
template<typename TItem, typename TLock>
void concurrent_queue<TItem, TLock>::resize_queue()
{
    TItem *old_items =  items;
    auto new_capacity = capacity + nopers / 10u;
    items = new TItem[new_capacity];
    auto j = 0;
    for(auto i = head; i < tail; i++)
        items[j++] = old_items[i % capacity];
    head = 0u;
    tail = capacity;
    capacity = new_capacity;
    #ifdef _DEBUG
        std::cout << "queue resized, new capacity: "
            << capacity<< std::endl;
    #endif
    delete[] old_items;
}
```

```
template<typename TItem, typename TLock>
TItem concurrent_queue<TItem, TLock>::dequeue()
{
    std::unique_lock<TLock> ilock(lock);
    if (head == tail)
        //cond_.wait(ilock);
        throw std::runtime_error("queue is empty");
    auto item = items[head % capacity];
    head++;
    return item;
}
template<typename TItem, typename TLock>
void concurrent_queue<TItem, TLock>::enqueue(TItem &item)
{
    std::unique_lock<TLock> ilock(lock);
    if (tail - head == capacity)
        resize_queue();
        //throw std::runtime_error("queue is full");
    items[tail % capacity] = item;
    tail++;
    ilock.unlock();
    //cond_.notify_one();
}
template<typename TItem, typename TLock>
TItem concurrent_queue<TItem, TLock>::read_queue()
{
    std::shared_lock<TLock> ilock(lock);
    if (head == tail)
        //cond_.wait(ilock);
        throw std::runtime_error("queue is empty");
    auto item = items[head % capacity];
    return item;
}
template<typename TItem, typename TLock>
void concurrent_queue<TItem, TLock>::print()
{
    for (auto i = 0u; i < capacity; i++)
        std::cout << items[i] << " ";
    std::cout << std::endl;
}
static auto rand_gen()
{
    static const auto maxrand = 100;
    static std::random_device rnd_device;
    static std::mt19937 mersenne_engine{rnd_device()};
    static std::uniform_int_distribution<int> dist{1, maxrand};
    return dist(mersenne_engine);
}
int main(int argc, const char *argv[])
{
    try
    {
        const auto max_threads =
std::thread::hardware_concurrency();
        std::ofstream speedupfile{"throughput"};
        if (!speedupfile.is_open())
```

```
        {
            std::cerr << "can't open file" << std::endl;
            return 1;
        }
        auto get_time = std::chrono::steady_clock::now;
        decltype(get_time()) start, end;
        for (auto nthr = 2u; nthr <= max_threads; nthr++)
        {
            start = get_time();
            for (auto i = 0; i < nruns; i++)
            {
                #ifdef _DEBUG
                    std::cout << "run number: " << i+1 << std::endl;
                #endif
                std::vector<thread_raii> threads;
                ///concurrent_queue<int, std::mutex> cqueue;
                ///concurrent_queue<int, spinlock> cqueue;
                concurrent_queue<int, std::shared_timed_mutex>
cqueue;
                ///concurrent_queue<int, boost::shared_mutex>
cqueue;
// Warm-up
                for (auto i = 0u; i < cqueue.get_capacity() / 2;
i++)
                {
                    auto item = rand_gen();
                    cqueue.enqueue(item);
                }
                auto thread_func = [&cqueue, &nthr]()
                {
                    for (auto i = 0u; i < nopers; i++)
                    {
                        auto oper = rand_gen();
                        if (oper <= 17)
                        {
                            auto item = rand_gen();
                            cqueue.enqueue(item);
                        }
                        else if (oper <= 97)
                        {
                            cqueue.read_queue();
                        }
                        else
                        {
                            cqueue.dequeue();
                        }
                    }
                };
                for (auto thr_id = 0u; thr_id < nthr; thr_id++)
                    threads.emplace_back(thread_raii
                {
                    std::thread{thread_func}});
                for (auto &thr: threads)
                    thr.join();
            // cqueue.print();
            }
```

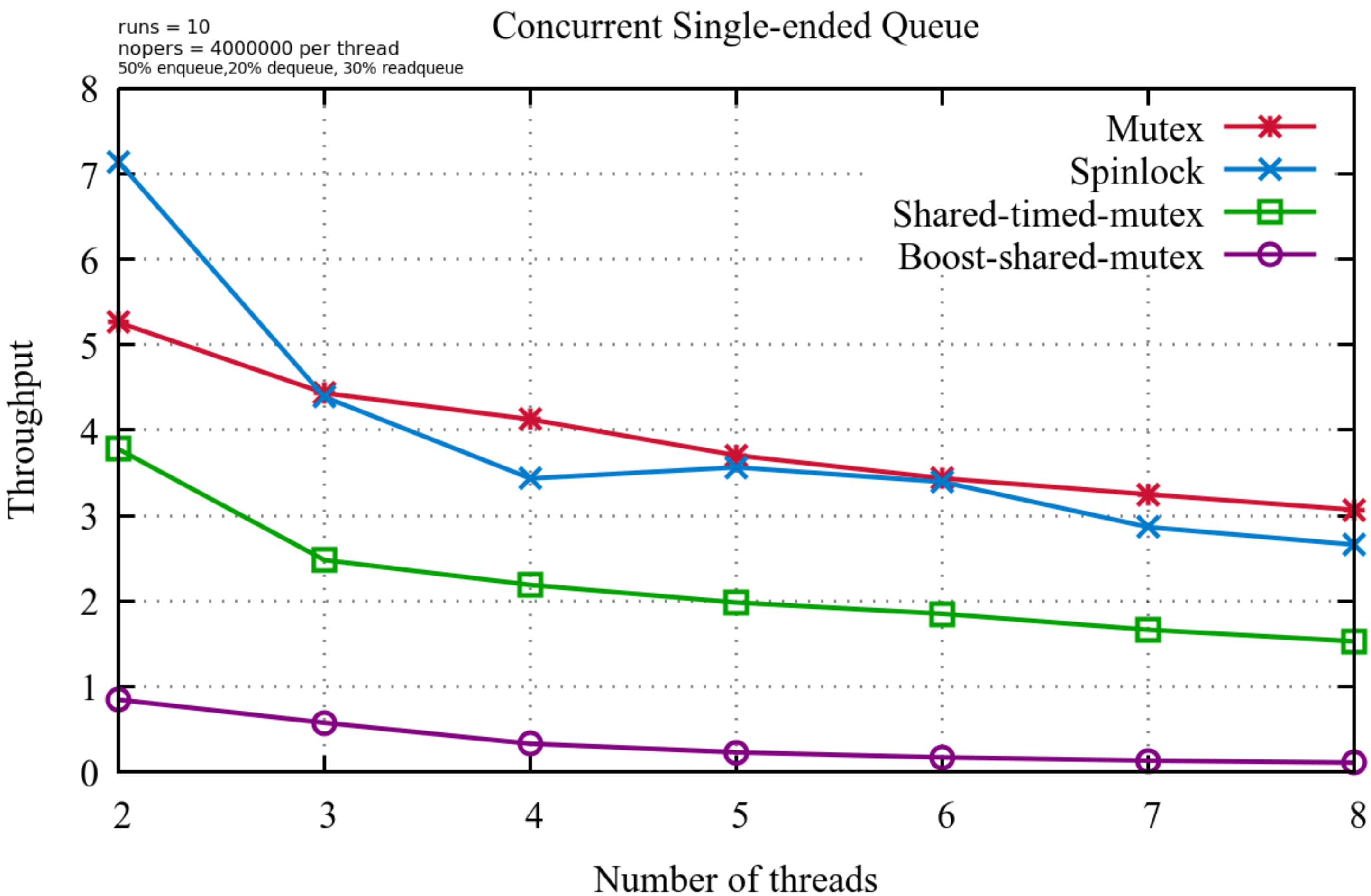
```
            end = get_time();
            const auto elapsed = std::chrono::duration_cast
                                <std::chrono::milliseconds>(end -
start).count();
            const auto par_time = double(elapsed) / nruns;
            const auto throughput = (nthr * nopers) / (par_time *
1000);
            std::cout << "Threads: " << nthr << " elapsed time: "
                      << par_time << " ms throughput: " <<
throughput <<
                      std::endl;
            speedupfile << nthr << "\t" << throughput << std::endl;
        }
        speedupfile.close();
    }
    catch (std::runtime_error &e)
    {
        std::cerr << "Caught a runtime_error exception: "
                  << e.what () << std::endl;
    }
    return 0;
}
```

4. Conclusion

The analysis of the output of this code carried out for the following type of locks.

- Mutex.
  - ➢ std::mutex used for implementation.
- Spinlock
  - ➢ Class spinlock is defined within the code in which pthread_spinlock_t were used.
- Shared-timed-mutex
  - ➢ Std::shared_timed_mutex were used to implement.
- Shared-mutex
  - ➢ Boost::shared_mutex was used.

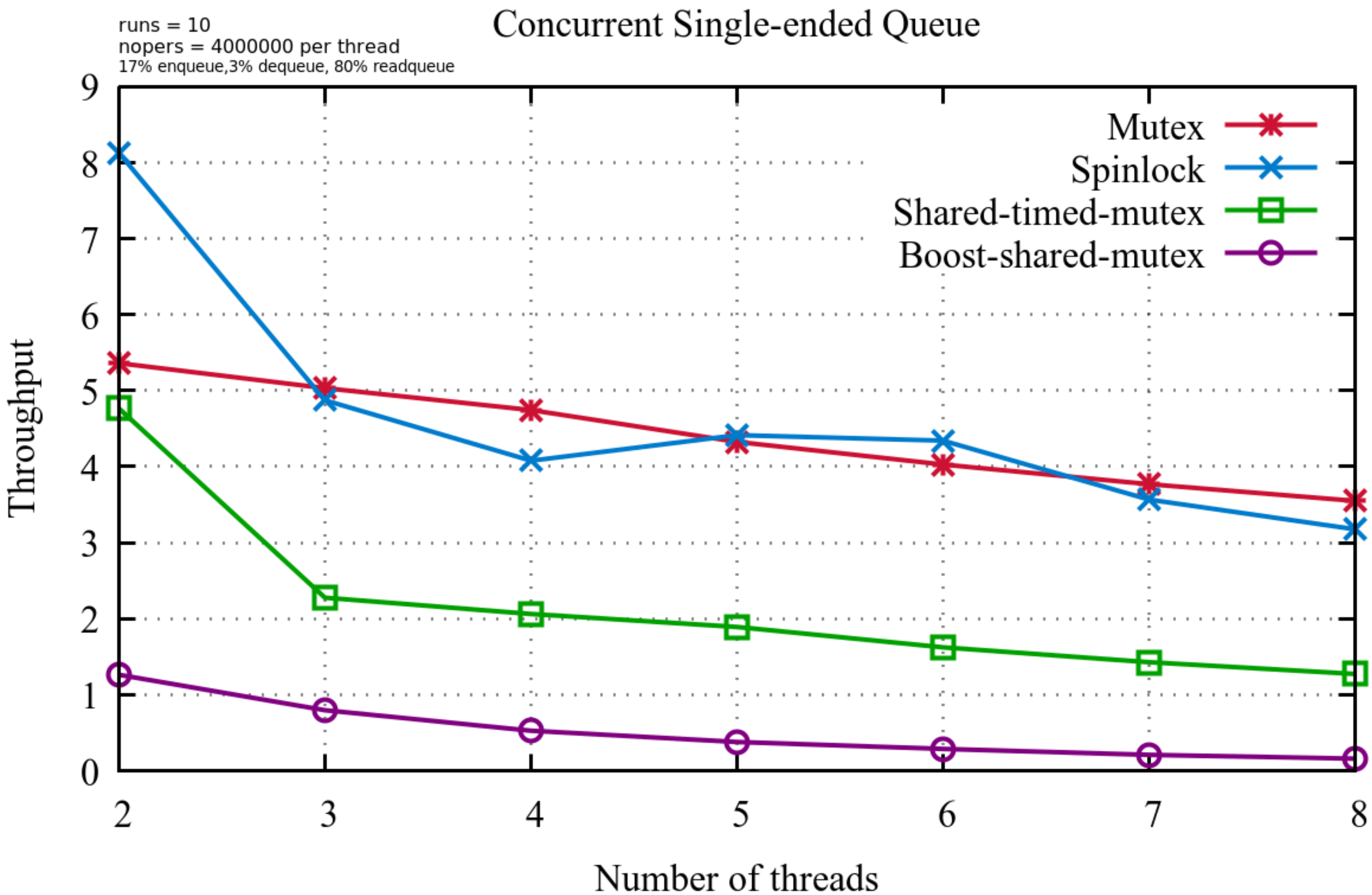
The output graph is as bellow.



The Output when each thread performs 10 run of 4000000 operations of which 50% is enqueuer, 30% is read-queue and 20% is dequeuer.

Another experiment also carried on at which the threads mostly perform read-queue function (80 percent), to see how much using read/write buffer can boost up our throughput, however the result shows the same decrease pitch in throughput of the threads which overall higher value which make a sense but expectation was to see the throughput increasing for more number of thread when we use shared_locks.

The resulted graph for this experiment is as below.



The Output when each thread performs 10 run of 4000000 operations of which 17% is enqueuer, 80% is read-queue and 3% is dequeuer.

The overall change that happen to the throughputs of the threads (using any type of lock) shows that the throughput will be decrease when the size of the critical section grows.

We also can conclude that we have to be cautious because not all the time using shared_mutex can increase our throughput, and due to our experiment it is slower than normal mutex or spinlock. So it is highly depended in our code to see what type of lock is the best suited.