# Практическое задание по теме: «Параллельное программирование с блокировками»

## Shared\_mutex. Параллельные алгоритмы std, thread\_local, once\_flag, call\_once, mutex, async, future, conditional\_variable

# Задание 1. Потокобезопасный стек. std::mutex. std::lock\_guard. std::shared\_mutex. std::lock(). std::scoped\_lock.

Разработаем класс threadsafe\_stack:

1. чтобы не тратить время на рутинную работу, в качестве хранилища данных используем std::vector
2. Для того чтобы предотвратить состояние гонки при добавлении/изъятии данных из стека, следует:
   * на время модификации защитить всю структуру данных целиком от возможности доступа к данным одновременно из двух и более потоков => должен быть общий мьютекс, который может использовать любой поток, обращающийся к стеку => делаем мьютекс членом класса threadsafe\_stack
   * Так как мы будем использовать мьютекс в константных методах, нужно позволить вызывать его неконстантные методы =>???
3. все методы на время чтения/записи должны защищать свои операции мьютексом
4. Подсказка: метод push() не может (не должен) принимать в качестве параметра ссылку, так как другой поток может изменить значение по адресу
5. Проблема: разработчики std::vector разделили две операции: получение значения последнего элемента - **T& back()**;   
   и удаление последнего элемента – **void pop\_back();**  
   Если бы метод был реализован: **T pop\_back ();** (объединение двух операций) и в процессе формирования возвращаемого значения (при копировании) генерируется исключение, то возникает ситуация, когда из стека данные уже удалены, а возвращаемое значение еще не сформировано => данные потеряны!  
   Но! для потокобезопасного стека такое разделение операций порождает гонку:

* + мьютекс не «спасает» в ситуации (то есть возникает гонка):  
    if(!**s.empty())**  
    {  
     res **= s.top();**  
     **s.pop();**  
     do\_smth(res);  
    }  
    , так как между вызовом empty() и top() другой поток может вызвать pop()  
    аналогично гонка может возникнуть между вызовом top() и pop()

|  |  |
| --- | --- |
| Поток 1 | Поток 2 |
| if(!s.empty()) {  res = s.top();  s.pop();  do\_smth(res); } | if(!s.empty()) {  res = s.top();  s.pop();  do\_smth(res); } |

одно значение прочитано вообще не будет, а другое будет обработано дважды!  
Решение – попробовать безопасно совместить эти две операции (можно реализовать оба приведенных ниже варианта, чтобы у пользователя был выбор):

**Вариант 1: void pop(T& val);** //клиент предоставляет адрес для «выталкиваемого» из стека значения => если при копировании генерируется исключение, то оригинал остается в стеке.  
**Вариант 2:** возвращать указатель на динамическую копию выталкиваемого элемента. Затратно, но безопасно, так как при копировании указателей **не** может быть сгенерировано исключение. Остается проблема управления динамической памятью, которую можно решить посредством std::shared\_ptr<T> - **std::shared\_ptr<T> pop();**

1. При реализации конструктора копирования нужно защитить от модификаций только источник
2. При реализации operator= нужно учесть возможность взаимных блокировок => нужно защитить как источник, так и приемник.
3. Запустите несколько потоков, которые будут формировать значения в стеке – push() (поставщики) с заданным периодом и один поток, который будет с заданным периодом доставать элемент с верхушки и обрабатывать
4. При попытке достать элемент с верхушки ПУСТОГО стека должно быть сгенерировано исключение. Но! исключение сгенерировано в дочернем потоке, а обработать логично в родительском потоке

# Задание 2. Пинг-понг на семафорах. semaphore, jthread

Пусть два потока играют в пинг-понг. Игру может остановить только судья (нажатием любой клавиши). Реализуем простейший случай, когда игроков всего двое (а мячик один). Правило очень простое: чтобы отбить удар (pong), нужно, чтобы другой этот удар сделал (ping). И по правилам всегда есть тот, кто начинает первым (ping). Тот, кто бьет, выводит “ping”, тот, кто отбивает - “pong”.

Подсказка 1– для организации взаимодействия достаточно использовать двоичные семафоры;

Подсказка 2: подумайте, как следует инициализировать семафоры?

# Задание 3. Thread-safe Singleton thread\_local

Создайте класс, реализующий паттерн проектирования Синглтон.

Какие операции с таким объектом нужно запретить? и как?

1 вариант - один объект в рамках многопоточного приложения

2 вариант – для каждого потока один экземпляр

## Задание 4. Thread\_local, osyncstream. Исследуем реализацию thread\_local – переменных.

Дано:  
thread\_local std::string s("hello from "); //вне всяких {} определяем строку

Запускаем потоки, в которые в качестве параметра передаем порядковый номер потока (порядок, в котором потоки были запущены). В потоковой функции модифицируем s следующим образом: добавляем (+=) номер запуска. Выводим результирующую строку и адрес s. Наблюдаем результат, если результат не очень нравится, вспоминаем про osyncstream.