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

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

## Задание 1. Параллельные алгоритмы std

* Сформируйте std::vector<int> - вектор должен быть достаточно большим!
* Проинкриментируйте значения всех элементов последовательным for\_each() или transform(). Измерьте время работы
* Проинкриментируйте значения всех элементов параллельным for\_each()или transform(). Попробуйте разные политики выполнения. Измерьте время работы.
* Сравните результаты
* Поэкспериментируйте с размером вектора

## Задание 2. thread\_local, future, async

Дан простой целевой класс, например:

class A{

…

};

Модифицируйте этот класс таким образом, чтобы выполнить приведенное ниже задание:

* запустите несколько потоков. Каждый поток может любым способом (на стеке, в heap-е, с потоковым временем жизни) создавать любое количество объектов целевого типа A. Потоковые функции могут быть разные (сигнатура одинаковая, а реализации разные) или это может быть одна и та же потоковая функция, в которой в зависимости от разных условий выполнения или случайным образом будет создаваться разное количество объектов типа А, например:  
  {  
   A a1;  
   A\* pA = new A();  
   thread\_local A a2;  
   static A a3;  
   …  
  }
* каждому потоку при запуске передать его «имя», чтобы можно было увидеть соответствие
* пока запущенные потоки работают, делаем ПОЛЕЗНУЮ работу, например, выводим заголовок
* требуется дождаться пока все потоки завершаться, чтобы КРАСИВО (в соответствии с тем порядком, в котором они запускались) вывести информацию:   
  «имя» потока каким по порядку был запущен количество созданных объектов

Подсказки:

1. Чтобы увидеть результат, функции нужно запускать в отдельном потоке (не оставлять на откуп реализации)

## Задание 3. Асинхронный запуск потоков. Получение отложенного результата. Параллельная реализация std::accumulate()

Задача: сформировать среднее значение для контейнеров (vector, deque) с элементами любого типа (int, float,…)

1. Создайте и заполните значениями контейнер с элементами типа int (количество элементов и значения можно формировать случайным образом).  
   Замечание: количество элементов должно быть достаточно большим, чтобы оценить результаты распараллеливания задачи
2. Измерьте время работы std::accumulate() для суммирования элементов от begin() до end()
3. Решите ту же задачу посредством параллельного суммирования:
   1. Исходя из количества вычислительных ядер, запустите посредством std::async()разумное количество потоков, каждый из которых должен получать сумму заданного диапазона
   2. Полученные частичные результаты по мере готовности требуется тоже просуммировать
   3. Измерьте общее время
   4. Попробуйте задать количество потоков   
      меньше, чем количество вычислительных ядер,   
      больше чем количество вычислительных ядер
4. Сравните полученные результаты

## Задание 4. Обработка исключений посредством future

Реализуйте в отдельном дочернем потоке прием целых значений от пользователя.

Если пользователь вводит НЕ целое, должно быть сгенерировано исключение.

Обработку исключения требуется реализовать в родительском потоке.

Подсказка: можно использовать функцию std::**stoi**() – С++11, которая генерирует исключения:

* **std::invalid\_argument**, если преобразование не может быть выполнено
* **std::out\_of\_range**, если преобразованное значение будет выходить за границы диапазона значений типа результата

## Задание 5. Потокобезопасная кольцевая очередь фиксированного размера на базе условных переменных

Имитируем формирование и обработку «заявок» посредством многопоточной очереди. Потоки-писатели заносят очередную «заявку» в конец очереди (при этом, если в очереди нет места, писатель должен ждать, пока место появится). Потоки-читатели должны изъять заявку из начала очереди и обработать (при этом, если в очереди заявок нет, читатель должен ждать, пока какой-нибудь писатель заявку сформирует)

* Требуется реализовать класс  
    
  constexpr size\_t size= <размер>;  
  class Queue{  
  char buf[size]; // будет использоваться как кольцевой буфер (будем заносить туда коды символов (для дальнейшей визуализации удобно хранить коды символов в верхнем регистре)   
    
  … //другие переменные класса для организации работы с кольцевым буфером  
  // мьютекс для синхронизации и читателей, и писателей (или два разных мьютекса: один для читателей, другой для писателей??? Проблемы???)  
  //условная переменная для сигнализации о том, что в очередь поставлена новая заявка => можно обрабатывать  
  // условная переменная для сигнализации о том, что в очереди освободилось место => можно заносить новую заявку
* Ограничения: можно только добавлять в очередь и извлекать из нее заявки (нет непосредственного доступа к первому и последнему элементам).   
  Если в буфере нет места, то писатель (push()) должен ждать, пока место появится.  
  Если в буфере пусто, то читатель (pop()) должен ждать, пока в очереди что-нибудь появится
* Методы push() и pop() мы будем запускать в отдельных потоках
* для потокобезопасной очереди реализовать операции копирования и присваивания
* деструктор? – что делать с заявками, которые еще присутствуют в очереди?

# Задание 6. Thread-safe Singleton

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 7. Пул потоков

Разработаем простой пул потоков, который позволяет:

* создавать разумное количество потоков
* и многократно их использовать, пока существует пул
* при уничтожении самого пула потоки также удаляются.

Ограничения (упрощения) нашей реализации:

* все задания выполняют одну и ту же работу над разными наборами данных
* количество «работ»/заданий известно заранее (и превышает количество аппаратных потоков)
* как задавать время жизни пула?
* Чтобы можно было хоть как-то попробовать измерить время работы, задания будем добавлять в пул все сразу и централизовано в функции main (в жизни обычно задания формируются разными клиентами в произвольном месте, в любое время и в произвольном порядке)
* При выполнении задания хотелось бы исключить вывод на консоль/файл, так как вывод должен выполняться исключительно последовательно (=> придется защищать вывод мьютексом) и он (вывод) будет занимать «львиную долю» времени => эксперимент не будет «чистым» => в качестве задания можно использовать создание вектора заданного размера, формирование значений элементов (rand()) и какие-то действия, например, вычисление суммы
* Интересно было бы «посчитать» сколько заданий выполнил каждый поток

Подсказки:

1. Функция main(): создание, постановка в очередь для обслуживания, остановка и обработка результатов работы заданий:
   1. Формирование количества заданий
   2. Создание пула
   3. Резервирование памяти под результаты работы каждого задания
   4. Формирование заданий и добавление в очередь заданий
   5. Имитация деятельности первичного потока (например, std::this\_thread::sleep\_for(<таймаут>);)
   6. Останавливаем все работающие потоки в пуле (устанавливаем признак, который анализируется в потоковой функции)
   7. Выводим результат – при для завершенных заданий результаты должны быть сформированы, а для тех заданий, которые еще ждут обслуживания в очереди пула, естественно, нет
2. Класс, реализующий задание - myTask:
   1. Возможные данные: номер потока (идентификатор), адрес – куда «положить» результат, количество чисел, над которыми нужно произвести действие
   2. Конструктор
   3. Перегруженный оператор operator(), который будет собственно выполнять полезную работу (например, генерировать случайным образом заданное количество значений и выполнять над ними какое-нибудь действие (суммировать) )
3. Класс, реализующий пул потоков:
   1. Данные:
      * Совокупность работающих в пуле потоков (например: std::vector<std::thread>)
      * Очередь заданий
      * Мьютекс для синхронизации доступа к очереди заданий из разных потоков
      * Признак – «нужно завершать поток». По хорошему этот флаг должен быть атомарным. Но при наших ограничениях скорее всего проблем не будет
   2. Методы:
      * Конструктор
        1. Определение разумного количества потоков
        2. Запуск потоков
        3. Не забыли про признак остановки потоков
      * Деструктор - ???
      * Защищенная потоковая функция

Цикл -   
while (если не установлен флаг)  
{  
//под защитой мьютекса  
  
//если очередь пустая, освобождаем мьютекс и «отдыхаем» (std::this\_thread::yield(); или std::this\_thread::wait\_for(<маленький интервал>))  
  
//иначе, если в очереди есть задание, достаем очередное задание в локальную переменную, удаляем задание из очереди, только после этого освобождаем мьютекс и запускаем задание  
}

* + - Public метод add\_task() принимает параметры задания и под защитой мьютекса помещает задание в очередь

Было бы очень интересно измерить и сравнить время выполнения задачи при реализации посредством пула потоков и при запуске для каждого задания своего потока