# Практическое задание №1

**Запуск потоков посредством класса thread. Ожидание завершения – join(). Отключение объекта от системного потока – detach(). Определение количества исполняемых ядер. Измерение и задание интервалов времени – chrono. Блокировка потока на указанный интервал. Функции для работы с текущим потоком, пространство имен this\_thread. std::mutex. std::lock\_guard.**

## Задание 1. Запуск потоков с ожиданием завершения

Позаимствуйте тексты, например, на <http://do-you-speak.ru/elementary-pre-intermediate> и создайте для каждого текста файл - \*.txt. Задайте спецификации существующих на диске (в текущей директории или в отдельной поддиректории) текстовых файлов, например:   
const char\* filespec[] = {"1.txt", "2.txt", "3.txt", "4.txt"…};  
или std::string filespec[] = {"1.txt", "2.txt", "3.txt", "4.txt"…};  
Обработку каждого файла требуется сделать в отдельном потоке. Каждый поток должен прочитать содержимое файла, перевести все буквы в верхний регистр (а цифры и другие символы оставить прежними) и записать в новый файл. Имена новых файлов можно формировать с помощью заданного префикса или суффикса,   
например: 1.txt -> \_1.txt

Поэкспериментируйте с количеством одновременно запускаемых потоков, каждый раз измеряя время выполняемого фрагмента. Время выполнения хотелось бы выводить в привычном для пользователя виде, например:

2s::10ms::150mks

Подумайте, какие часы Вы будете использовать для измерения времени? А может быть, стоит сравнить использование разных часов (каких?)

В общем случае удобно было бы иметь возможность задавать количество запускаемых потоков.

1.а Последовательно запускаем потоки. Очередной поток запускаем только после завершения текущего

Измерьте время работы фрагмента.

1.б Запускаем столько потоков, сколько файлов требуется обработать

Измерьте время работы фрагмента.

1.в Запускаем столько потоков, сколько ядер в системе

Измерьте время работы фрагмента.

## Задание 2. Запуск обособленных потоков. Блокировка потока на заданный интервал времени

Каждый поток должен «издать» заданное количество звуков с заданной частотой.

В ОС Windows для этого используется системная функция Beep(), другие ОС предоставляют свои системные средства.

Соответствие:

Beep(261, <интервал\_в\_мс>); // соответствует ДО

Beep(293, интервал\_в\_мс); // РЕ

Beep(329, интервал\_в\_мс); // МИ

Beep(349, интервал\_в\_мс); // ФА

Beep(392, интервал\_в\_мс); // СОЛЬ

Beep(440, интервал\_в\_мс); // ЛЯ

Beep(493, интервал\_в\_мс); // СИ

На крайний случай можно «пискнуть» с помощью  
std::cout<<’\a’;  
к сожалению, изменить частоту звука в этом случае невозможно.

После запуска потоков требуется отключить потоки от соответствующих объектов.   
**Но!** при завершении первичного потока в ОС Windows все остальные потоки принудительно завершаются => нужно каким-то образом заставить первичный поток завершиться после того, как завершились все вторичные. Например, отправить его на длительное время «в спячку» - std::this\_thread::sleep\_for()

2.a

Сыграйте гамму

2.б

Попробуйте сыграть аккорд??? Подсказка: функция Beep() использует системный динамик для генерации звука и не возвращает управление, пока не истек заданный интервал

## Задание 3.this\_thread

В первичном потоке выведите на экран алфавит (‘A’ – ‘Z’) таким образом, чтобы скорость вывода плавно увеличивалась.

## Задание 4.Параллельная реализация std::transform(). Определение числа вычислительных ядер. Измерение времени.

Требуется посредством алгоритма std::transform() преобразовать элементы любого базового контейнера (vector, list, deque) с элементами любого арифметического типа (short, int, long long, float, double) в элементы другого по правилу: cont2[i] = abs(cont1[i]);

Выведите пользователю информацию о количестве вычислительных ядер и предоставьте пользователю возможность задавать количество потоков.

Подсказка: количество вторичных потоков может быть на единицу меньше заданного пользователем значения, так как первичный поток тоже может выполнить свою часть работы, пока вторичные потоки выполняют свою.

Определите размер диапазона элементов контейнера источника для каждого вторичного потока. Остаток элементов должен трансформировать первичный поток.

Измерьте время выполнения полной трансформации всего контейнера при задании разного количества потоков. Важно! при этом нужно понимать, что оценка является скорее качественной, чем количественной, а о точности судить трудно, так как для достоверной оценки нужно учитывать большое количество других факторов (в системе параллельно выполняется большое количество системных потоков, пропускная способность общей шины данных, кеширование для каждого вычислительного ядра…)

# Задание 5. Потокобезопасный стек. std::mutex. std::lock\_guard

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 его «модифицировать» =>???
3. все методы на время чтения/записи должны защищать свои операции мьютексом
4. Подсказка: метод push() не может 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