Потоки

Программа - набор команд для процессора.

Процесс - запущенная программа.

Поток - элемент выполнения процесса, наименьшая единица обработки с точки зрения ОС.

Один процесс может иметь **множество** потоков.

Потоки

Поток породивший другой поток называется родительский, порожденный - дочерний.

При завершении родительского потока - дочерние тоже завершаются.

Каждый поток имеет:

- PID
- Состояние (запущен, не запущен, приостановлен)
- Приоритет
- Указатели на память
- ….

Асинхронность

Стиль программирования при котором все тяжеловесные задачи исполняются в отличном от вызывающего потоке, и результат выполнения которых может быть получен, вызывающим потоком, когда он того пожелает(при условии, что результат доступен).

future

Предоставляет доступ к разделяемому состоянию:

- данные
- флаг готовности

Является получателем значение и не может его самостоятельно выставлять.

После того как значение вычислено, устанавливается флаг готовности. С этого момента значение может быть получено в любой момент.

async

- выполняет функцию f асинхронно
- возвращает **std::future -** результат этого вызова функции **f**

```
template< class Function, class... Args>
std::future<typename std::result_of<Function(Args...)>::type>
    async( Function&& f, Args&&... args );

template< class Function, class... Args >
std::future<typename std::result_of<Function(Args...)>::type>
    async( std::launch policy, Function&& f, Args&&... args );
```

async ***std::launch*** **policy**

- ***aunch::async*** – создается объект класса *thread*, с функцией и её аргументами в качестве аргументов нового потока. Т.е. *async* инкапсулирует создание потока, получение *future* и предоставляет однострочную запись для выполнения такого кода.
- ***launch::deferred*** – функция вместе с аргументами, будет сохранена в ***future***, чтобы быть вызванными позже. Когда будет вызван метод *get*(или *wait,* но не *wait_for!*) на *future*, которое вернул *async*. При этот вызываемый объект выполнится в потоке, который вызывал *get*!
- ***launch::async*** | ***launch::deferred*** - в этом случае будет выбрано одно из двух поведений описанных выше. Какое из двух? Неизвестно и зависит от имплементации.

future & async

- ***future<T>***, созданный с помощью ***async***, вызывает ***get()*** в своем деструкторе

- если результат вызова функции ***async*** не сохранить в переменную, то программа может выполняться последовательно

Вопрос: скомпилируется ли этот код?

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>

using namespace std;

int sumVectors(vector<int> &first, vector<int> &second)
{
    return accumulate(first.begin(), first.end(), 0)
            + accumulate(second.begin(), second.end(), 0);

}

int main() {

    cout << sumVectors({1, 1, 1, 1}, {3, 3, 3});

    return 0;
}
```

Ответ.

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>

using namespace std;

int sumVectors(const vector<int> &first, const vector<int> &second)
{
    return accumulate(first.begin(), first.end(), 0)
            + accumulate(second.begin(), second.end(), 0);

}

int main() {

    cout << sumVectors({1, 1, 1, 1}, {3, 3, 3});

    return 0;
}
```

Нет. Так как в функцию передаются временные объекты и поэтому аргументы должны быть константны.

Данный код можно распараллелить.

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>

using namespace std;

int sumVectors(const vector<int> &first, const vector<int> &second)
{
    return accumulate(first.begin(), first.end(), 0)
            + accumulate(second.begin(), second.end(), 0);

}

int main() {

    cout << sumVectors({1, 1, 1, 1}, {3, 3, 3});

    return 0;
}
```

Считать сумму элементов первого вектора и второго вектора одновременно в двух потоках

Пример использования async

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
#include <future>

using namespace std;

int sumVectors(const vector<int> &first, const vector<int> &second)
{
    future<int> f = async([&first]{ return accumulate(first.begin(), first.end(), 0);});

    int result =  accumulate(second.begin(), second.end(), 0);
    return result + f.get();
}

int main() {

    cout << sumVectors({1, 1, 1, 1}, {3, 3, 3});

    return 0;
}
```

Потоки

- Работа с потокам осуществляется по средствам класса std::thread (доступного из заголовочного файла <thread>)
- Может работать с регулярными функциями, лямбдами и функторами.
- Позволяет вам передавать любое число параметров в функцию потока.

Создание потока

Создание объекта типа thread, в конструктор передается имя функци threadFunction.

```
void threadFunction()
{
	cout << "Hello from thread" << endl;
}

int main()
{
   thread thr(threadFunction);

   thr.join();

   return 0;
}
```

Join и Detach

- **join** - блокирует вызывающий поток до тех пор, пока поток (не выполнит свою работу
- **detach** - делает процесс фоновым

Передача аргументов

```
void threadFunctionArgs(int x, double &y, std::string &name)
{
    cout << "x = " << x << "y = " << y
         << "name -" << name << endl;

    y++;
}

int main()
{
    double value = 13.54;
    string s_value = "tmp";

    thread thr(threadFunctionArgs, 5, ref(value), ref(s_value));
    thr.join();

    cout << "value " << value << endl;
    return 0;
}
```

По умолчанию аргументы передаются по значению. Для передачи по ссылке используется std::ref, std::cref

Функции работы с потоками

- **get_id**: возвращает id текущего потока
- yield: говорит планировщику выполнять другие потоки, может использоваться при активном ожидании
- **sleep_for**: блокирует выполнение текущего потока в течение установленного периода
- **sleep_until**: блокирует выполнение текущего потока, пока не будет достигнут указанный момент времени

Мьютекс

Мьютекс - механизм использующийся для синхронизации потоков.

Может находится в одном из двух состояний:

- locked()
- unlocked()

Используется для защиты данных от одновременного доступа из нескольких потоков.

Мьютекс

Опишем класс TVector для безопасной работы с потоками.

```
template <typename T>
class TVector
{

public:

    void add(T element)
    {...}

    void print()
    {...}

private:

    std::mutex _lock; //мьютекс для защиты данных
    std::vector<T> _elements;
};
```

Реализация


```
template <typename T>

class TVector

{

public:
    void add(T element)
    {
        _lock.lock(); //берем мьютекс перед тем как обратиться к вектору
        _elements.push_back(element);
        _lock.unlock(); //освобождаем мьютекс после
    }

    void print()
    {
        _lock.lock();
        for(auto e: _elements)
          std::cout << e << std::endl;
        _lock.unlock();
    }

…

};
```

Реализуем функцию для копирования вектора


```
template <typename T>
class TVector
{

public:
…
    void add(T element)
    {
        _lock.lock();
        _elements.push_back(element);
        _lock.unlock();
    }

    void add(const std::vector<T> &vect)
    {
       for(auto item:vect)
       {
           _lock.lock();
           add(item);
           _lock.unlock();
       }
    }

…
};
```

Реализуем функцию для копирования вектора


```
template <typename T>
class TVector
{

public:
...
    void add(T element)
    {
        _lock.lock();
        _elements.push_back(element);
        _lock.unlock();
    }

    void add(const std::vector<T> &vect)
    {
       for(auto item:vect)
       {
           _lock.lock();
           add(item);
           _lock.unlock();
       }
    }

...
};
```

2. Пытаемся получить мьютекс. Мьютекс уже в состоянии locked

1. Получаем мьютекс.

Мьютекс в состоянии locked

Dead Lock

```
template <typename T>
class TVector
{

public:
...
    void add(T element)
    {
        _lock.lock();

_elements.push_back(element);
        _lock.unlock();
    }

    void add(const std::vector<T>
&vect)
    {
       for(auto item:vect)
       {
          _lock.lock();
          add(item);
          _lock.unlock();
       }
    }

...
};
```

2. Пытаемся получить мьютекс. Мьютекс уже в состоянии locked

1. Получаем мьютекс. Мьютекс в состоянии locked

При выполнении этой программы произойдет **deadlock** (взаимоблокировка, т.е. заблокированный поток так и останется ждать).

Причиной является то, что контейнер пытается получить мьютекс несколько раз до его освобождения (вызова unlock), что невозможно

Улучшения

std::recursive_mutex

позволяет получать тот же мьютекс несколько раз. Максимальное количество получения мьютекса не определено, но если это количество будет достигнуто, то **lock** бросит исключение **std::system_error**.

std::lock_guard

когда объект создан, он пытается получить мьютекс (вызывая **lock()**), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая **unlock()**)

Улучшенная версия

```
template <typename T>
class TVector
{
public:
    void add(T element)
    {
        std::lock_guard<std::recursive_mutex> locker(_lock);
        _elements.push_back(element);
    }

    void add(const std::vector<T> &vect)
    {
       for(auto item:vect)
       {
          std::lock_guard<std::recursive_mutex> locker(_lock);
          add(item);
       }
    }

    void print()
    {
         std::lock_guard<std::recursive_mutex> locker(_lock);

         for(auto e: _elements)
             std::cout << e << std::endl;
    }
private:
    std::recursive_mutex  _lock; //мьютекс для защиты данных
    std::vector<T> _elements;
};
```