

# Параллелизм. Потоки

# Дополнительные принципы ООП.

- 1. Параллелизм.**
- 2. Устойчивость.**
- 3. Типизация.**

**Самим рассмотреть историю развития ОС и программирования в контексте развития архитектуры: от первых терминальных систем к многозадачным сетевым ОС.**

# Параллелизм это:

**Свойство нескольких абстракций одновременно находится в активном состоянии, т. е. выполнять некоторые операции «одновременно».**

## **«Виды» параллелизма:**

- 1. Разделение вычислений на потоки.**
- 2. Разделение вычислений на процессы.**
- 3. Разделение вычислений на потоки для графических процессоров.**

# Класс «thread»

```
#include <iostream>
#include <thread>
#include <chrono>

long int car[2] = {0,0};

void car1(){
    while(1){
        if (car[0] > 999999999)
            car[0] = 0;
        else
            car[0] = car[0] + 1;
    }
}

void car2(int x) {
    while(1) {
        if (car[1] > 999999999)
            car[1] = 0;
        else
            car[1] = car[1] + 1;
    }
}
```

Напишем 2 подпрограммы, которые будут Моделировать гонки автомобилей и будут работать в параллельном режиме. Обе будут работать с единым объектом. В данном случае car = {0,0}. Где car[0] координата первой машинки car[1] — координата второй машинки. Каждая из подпрограмм изменяет только «свою» переменную.

# Класс «thread». Реализуем main.

```
int main() {  
    std::cout << "start main";  
    std::thread first(car1);  
    std::thread second (car2,0);  
    first.join();  
    second.join();  
    return 0;  
}
```

Обратите внимание, что в конструкторы класса thread, можно передавать входные параметры для используемых функций, например для сигнатуры функции `void car2(int x)` передано значение `x=0`. Посмотрите обязательно, что из себя представляет конструктор.

## Класс «thread». Запуск.

**После запуска, вы ничего не увидите. Но теоретически будут созданы 3 потока: 1 поток - основной, 2 других созданы основным first, second.**

**Давайте попробуем проследить, как они работают, для этого создадим третий дополнительный поток, который будет проверять выполнять чтение из значение переменных, которые устанавливаю два других.**

# Класс «thread». Добавим show.

```
void show(){  
    while(1){ std::this_thread::sleep_for(std::chrono::seconds(1));  
        std::cout << "\n";  
        std::cout << car[0] << " -- " << car[1];  
    }  
}
```

**В main:**

```
std::thread show_thread(show);
```

```
.....
```

```
show_thread.join();
```

```
....
```

**Обратите внимание, что тут добавлена функция `sleep_for()` из библиотеки `<chrono>`. Она устанавливает время «зависания» для потока из которого вызвана. В этой библиотеке есть и другие функции, ознакомьтесь. Это сделано, чтобы можно было следить за изменением значений в массиве `car`.**



# Класс «thread». Результат запуска.

```
lec_thread
start main
340942933 -- 356800278
683107248 -- 717640017
25321977 -- 78889426
367947966 -- 440017382
710632369 -- 801103354
52531031 -- 161284570
395314782 -- 521463908
739173898 -- 882190716
█

lec_thread
start main
349944945 -- 350930097
703785787 -- 707047534
58570932 -- 62477574
412793444 -- 418249593
766444393 -- 773458725
120662271 -- 129444918
█
```

```
lec_thread
start main
349469872 -- 344537074
701362966 -- 691794518
52147977 -- 38919644
403385211 -- 386430996
754575885 -- 734078385
105610662 -- 81324294
█

lec_thread
start main
354259925 -- 355413030
710783406 -- 711666766
67677856 -- 68655292
424449393 -- 425169148
781342884 -- 782000342
█
```

# Класс «thread». Результат запуска.

Результаты каждый раз разные. Поэкспериментируйте обязательно с различными вариантами задания времени съема. Также можете добавить `sleep_for` и функции, которые изменяют переменные.

Добавьте в `main` после запусков потоков несколько строк, которые демонстрировали бы, где сейчас находится вычисления в основном потоке.

....

```
std::cout << "end main";  
return 0;  
}
```

Ваши эксперименты показывают, что при таком использовании `thread`, вы никогда не завершите основной поток.

## **Класс «thread». Варианты запуска созданных потоков.**

**Запуск созданных объектов-потоков можно осуществлять в двух режимах:**

**object\_thread.join() - запуск в «синхронном» режиме.**

**object\_thread.detach() - запуск в «асинхронном» режиме.**

**Синхронно — выполнение операций последовательно.**

**Асинхронно — «одновременно»**

# Класс «thread». Запуск в асинхронном режиме

**first.detach();**

**second.detach();**

**show\_thread.detach();**

**Приведет к тому, что процесс завершится и потеряет доступ к созданным потокам.**

**Запустите:**

```
first.detach();  
second.detach();  
show_thread.join();
```

```
first.detach();  
second.detach();  
while(1){  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
    std::cout << "\n";  
    std::cout << car[0] << " -- " << car[1];  
}
```

## Класс «thread». Вывод.

**Посмотрите при различных запусках вы получаете помимо различных результатов, так и сами выводы в консоль отличаются. То будут ошибки записи, то ошибки чтения, то разбивания процесса вывода на экран на несколько строк. Что происходит понять бывает невозможно.**

**Но все эти проблемы связанные с одновременным обращением нескольких сущности к одним ресурсам. Для решения такого рода проблем применяется «синхронизация».**

# Класс «thread». Синхронизация.

**Синхронизация — это организация процесса последовательного выполнения операций.**

**Существуют различные технологии, которые как правило, связаны с «глобальными/общими» переменными.**

**Рассмотрим механизм синхронизации на основе использования mutex.**

**std::mutex**

**- объект ядра ОС, по сути позволяет блокировать свое состояние одному потоку и возвращать свой статус другим потокам.**

# Класс «thread». Использование mutex.

```
void car1(std::mutex &m){
    while(1){
        m.lock();
        if (car[0] > 999999999)
            car[0] = 0;
        else
            car[0] = car[0] + 1;
        // std::this_thread::sleep_for(std::chrono::seconds(1));
        m.unlock();
    }
}

void car2(int x, std::mutex &m) {
    while(1) {
        m.lock();
        if (car[1] > 999999999)
            car[1] = 0;
        else
            car[1] = car[1] + 1;
        // std::this_thread::sleep_for(std::chrono::seconds(1));
        m.unlock();
    }
}

void show(std::mutex &m){
    while(1){
        m.lock();
        //std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "\n";
        std::cout << car[0] << " -- " << car[1];
        m.unlock();
    }
}

int main() {
    std::cout << "start main";
    std::mutex m;
    std::thread first(car1, std::ref(m));
    std::thread second (car2,0, std::ref(m));
    std::thread show_thread(show, std::ref(m));
    first.join();
    second.join();
    show_thread.join();
    return 0;
}
```

Поэкспериментируйте с  
закомментированными строчками

# Класс «thread». Объект-функтор.

```
class Car{
    long int pos;
public:
    Car():pos(0){};
    void operator()(){
        while(1){
            if (pos > 999999999)
                pos = 0;
            else
                pos++;
        }
    };
};

Int main() {
    .....
    Car car_obj;
    std::thread thread_car(car_obj);
    thread_car.join()
    ....
}
```