



 **ТЕХНОСФЕРА**

# Многопоточность. Часть I

Антон Кухтичев





Не забудьте  
отметиться на  
портале!!!

Иначе всё плохо будет.



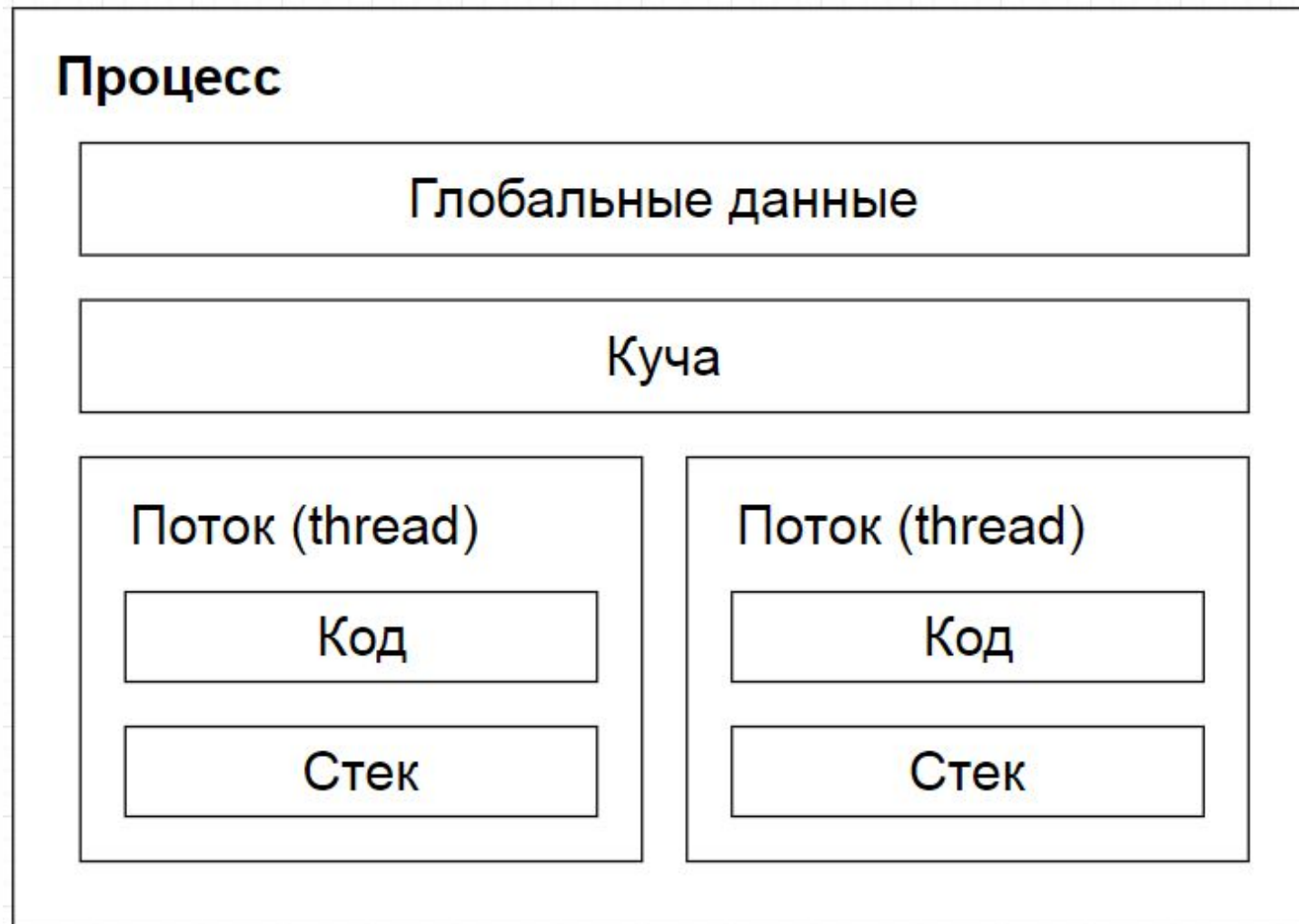
# Содержание занятия

1. Многопоточность (multithreading)
2. `std::thread`
3. `std::atomic`
4. Исключения в потоке
5. Гонки (race condition)
6. Взаимоблокировки (deadlock)



# **Многопоточность (multithreading)**

# Многопоточность (multithreading)



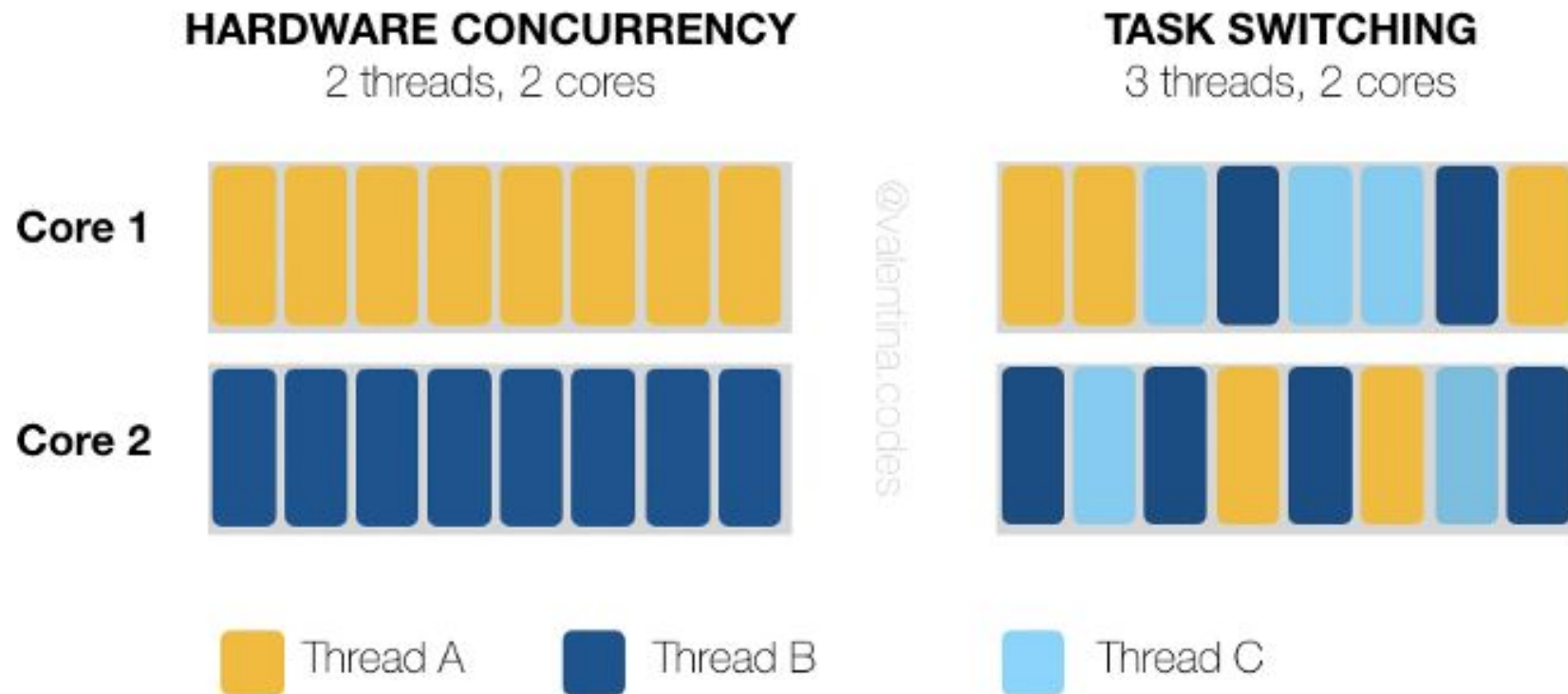
# Многопоточность (multithreading)

Многозадачность – возможность параллельной (или псевдопараллельной) обработки нескольких задач.

- Многозадачность основанная на прерываниях планировщика
- Кооперативная многозадачность – выполняемый код должен уступать процессорное время для других

Чтобы определить количество задач, которые реально можно выполнять в многопоточном режиме на том или ином компьютере, используется функция `std::thread::hardware_concurrency()`.

# Многопоточность (multithreading)





**std::thread**



# std::thread

- Заголовочный файл `#include <thread>`
- `std::thread t(callable_object, arg1, arg2, ...)`
- `t` вызывает `callable_object(arg1, arg2, ...)`
- Аргументы по умолчанию копируются, чтобы избежать копирования нужно использовать `std::ref()`;
- Объект потока можно перенести, но нельзя копировать!

# std::thread

- Если основной поток завершает выполнение, все второстепенные сразу останавливаются без возможности восстановления (`std::terminate`). Чтобы этого не допустить, у родительского потока имеются два варианта для каждого порождённого:
  - Блокирует и ждёт завершения порождённого потока, вызывая на нём метод `join`.
  - Прямо объявляет, что порождённый поток может продолжить выполнение даже после завершения родительского, используя метод `detach`.

---

# std::thread

```
#include <thread>

void threadFunction()
{
    ...
}

std::thread t(threadFunction);

t.join(); или t.detach();
```

## std::this\_thread

```
// идентификатор потока
const std::thread::id id =
    std::this_thread::get_id();

// указание планировщику снять поток с выполнения до следующего раза
std::this_thread::yield();

// усыпить поток на указанное время
std::this_thread::sleep_for(
    std::chrono::seconds(1))
```

`std::thread::id` можно сравнить, можно вывести в поток вывода

## std::future

Ожидание выполнения асинхронной задачи.

```
std::future<int> f =  
    std::async(std::launch::async, []() { return 5 });  
...  
const int result = f.get();  
  
auto f =  
    std::async(std::launch::async, []() { return 5 });  
...  
f.wait();
```

# std::async

Поведение `std::async`, зависит от переданных флагов следующим образом:

- `launch::async` – будет создан объект класса `thread`, с функцией и её аргументами в качестве аргументов нового потока;
- `launch::deferred` – никакого асинхронного вызова не произойдёт. Вместо исполнения функции в новом потоке, она, вместе с аргументами, будет сохранена в `future`, чтобы быть вызванными позже. Это поведение есть ни что иное, как отложенный вызов процедуры;
- `launch::async | launch::deferred` – в этом случае будет выбрано одно из двух поведений описанных выше. Какое из двух? Неизвестно и зависит от реализации.

## std::future

- `T get()` или `T& get()`  
Поток вызвавший `get` блокируется до вычисления значения
- `void wait() const;`  
просто подождать появления значения без его непосредственного получения
- `template<class Rep, class Period>`  
`std::future_status wait_for(const`  
`std::chrono::duration<Rep,Period>& timeout_duration) const;`
- `template< class Clock, class Duration >`  
`std::future_status wait_until(const`  
`std::chrono::time_point<Clock,Duration>& timeout_time) const;`

## std::future

```
auto f =  
    std::async(std::launch::async, []() { return 5 });  
  
auto status = f.wait_for(std::chrono::seconds(1));  
  
if (status == std::future_status::deferred)  
    std::cout << "задача еще не стартовала";  
else if (status == std::future_status::timeout)  
    std::cout << "результата не дождались";  
else if (status == std::future_status::ready)  
    std::cout << "все готово";
```



# std::promise

Позволяет вернуть результат работы из потока.

```
#include <future>
std::future<int> runTask()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();

    auto task = [](std::promise<int>&& p)
    {
        p.set_value(1);
    };

    std::thread thread(task, std::move(promise));
    thread.detach();

    return future;
}
```



**std::atomic**

# Барьеры в памяти

```
bool ready = false;
int data = 0;

int foo() { return 5; }

void produce()
{
    data = foo();
    ready = true;
}

void consume()
{
    while (!ready) ;
    assert(data == 5); // не всегда
}
```

# Барьеры в памяти

```
void produce()
{
    // data = foo();
    // Это долго, выполняю пока это:
    ready = true;
    // А теперь остальное:
    data = foo();
}

void consume()
{
    while (!ready) // ждем
        ;
    assert(data == 5); // не всегда
}
```



# Барьеры в памяти

Барьер – инструкция состоящая из указания двух типов операций работы с памятью:

Барьер гарантирует, что до барьера все операции работы с памятью типа X будут выполнены, а операции типа Y после барьера не начнут выполняться.

Операций работы с памятью две:

1. Чтение (Load)
2. Запись (Store)

# Барьеры в памяти

```
#include <atomic>

enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire, // <-- acquire
    memory_order_release, // <-- release
    memory_order_acq_rel,
    memory_order_seq_cst  // <-- default
};

void atomic_thread_fence(std::memory_order order) noexcept;
```

---

# Барьеры в памяти

```
void produce()
{
    data = foo();
    // Перед тем, как делать Store-операции
    // завершить все операции до барьера
    std::atomic_thread_fence(std::memory_order_release);
    ready = true;
}
```

---

# Барьеры в памяти

```
produce():  
    call    foo()  
    mov     DWORD PTR data[rip], eax  
    mfence  
    mov     BYTE PTR ready[rip], 1  
    ret
```



---

## std::atomic

```
std::atomic<T> value;
```

```
T load(std::memory_order  
       order = std::memory_order_seq_cst) const noexcept;
```

```
void store(T value, std::memory_order  
          order = std::memory_order_seq_cst) noexcept;
```

---

## std::atomic

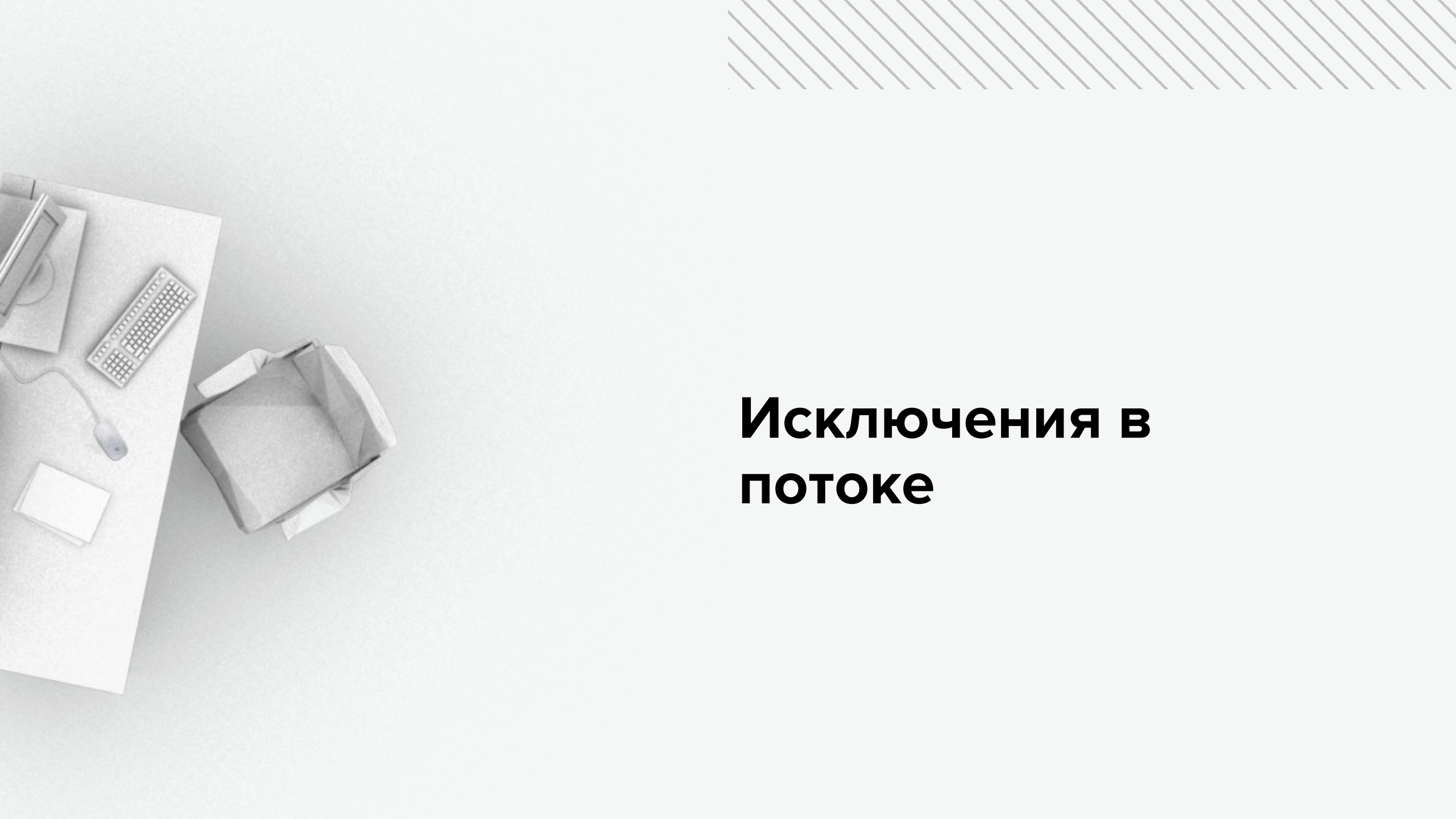
```
std::atomic<int> i = 5;
```

```
i.store(3);
```

```
int j = i.load();
```

```
++i;
```

```
int k = i;
```



# Исключения в потоке

# Исключения в потоке

```
void foo()  
{  
    throw std::runtime_error();  
}
```

```
std::thread t1(foo);  
t1.join();
```

В этом случае поток просто завершится, об исключении мы не узнаем.

# Исключения в потоке

```
auto f = std::async(std::launch::async, foo);

try
{
    f.get();
}
catch (const std::runtime_error& error)
{
    // Получили ошибку
}
```

# Исключения в потоке

```
auto task = ([](std::promise<int>&& p)
{
    try
    {
        foo();
    }
    catch (...)
    {
        p.set_exception(std::current_exception());
    }
})
```

# std::packaged\_task

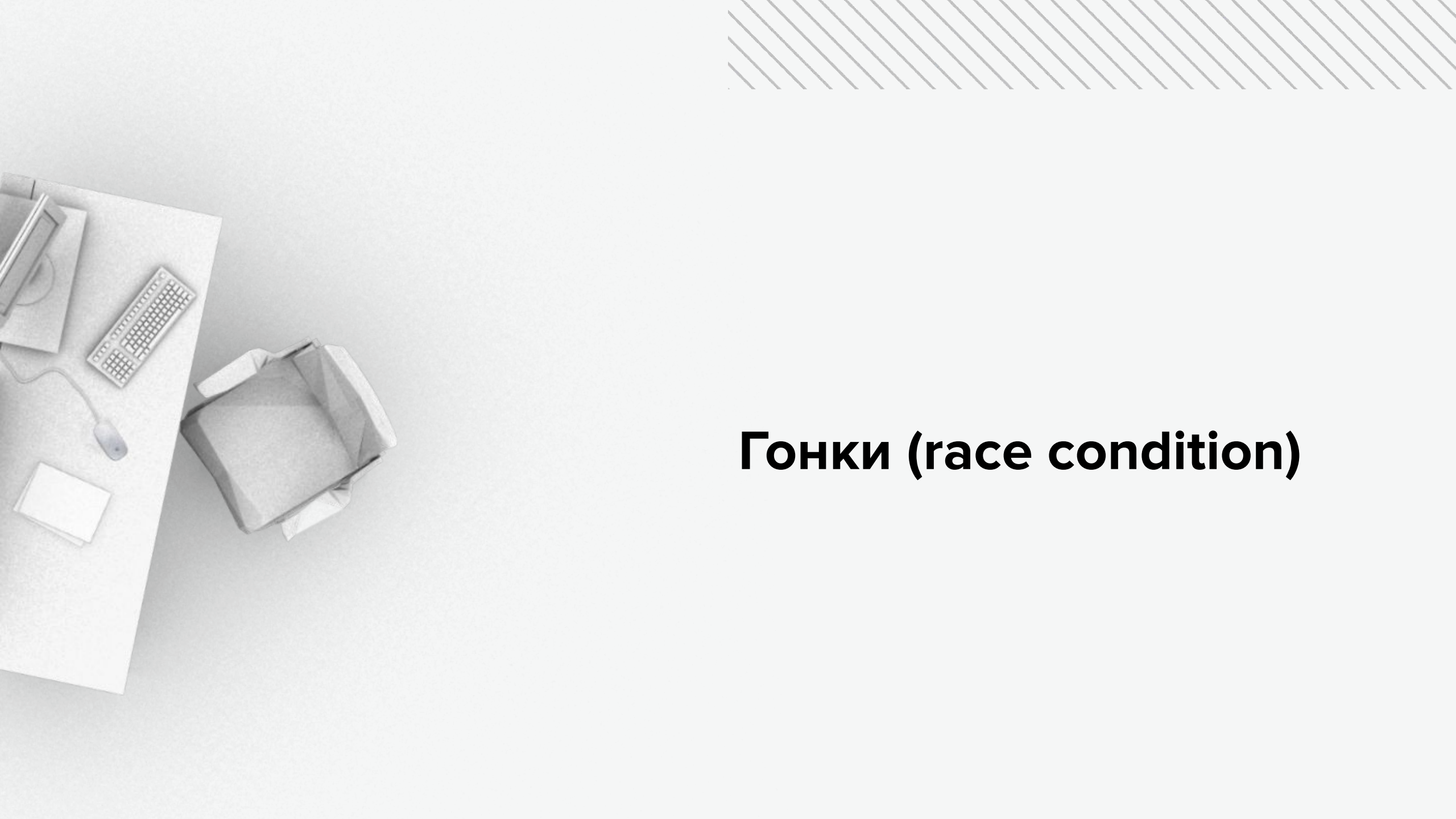
```
std::future<int> runTask()
{
    std::packaged_task<int()> task([]()
    {
        return 1;
    });

    auto future = task.get_future();

    std::thread thread(std::move(task));
    thread.detach();

    return future;
}

auto task = runTask();
task.get();
```



**Гонки (race condition)**



## Гонки (race condition)

```
int i = 17;

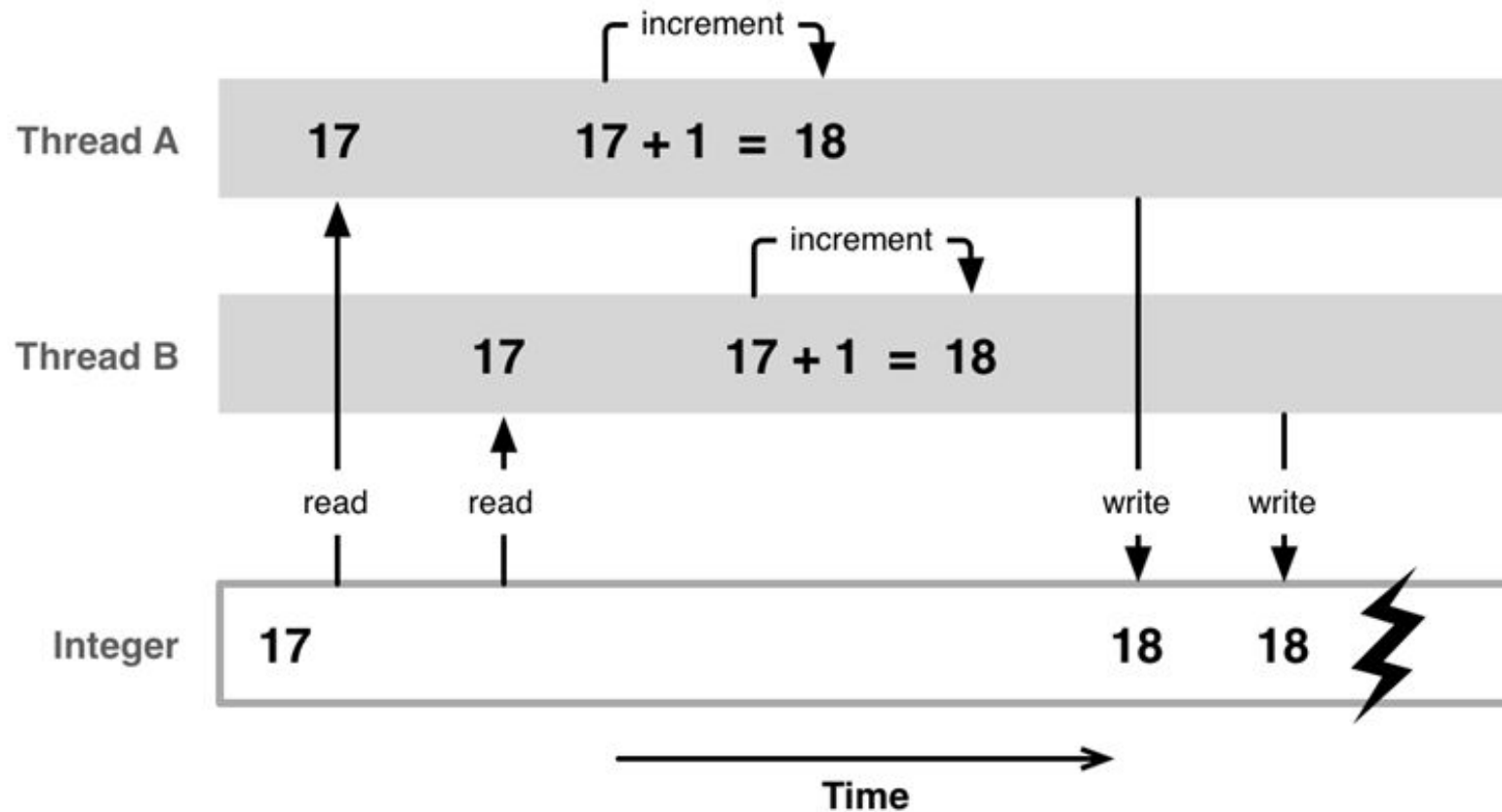
void plus1()
{
    i += 1;
}

std::thread t1(plus1);
std::thread t2(plus1);

t1.join();
t2.join();


std::cout << i; // ???
```

# Гонки (race condition)






# Средства синхронизации

- 
1. Атомарные операции;
  2. Спинлоки (spinlock);
  3. Семафоры (semaphore);
  4. Мьютексы (mutex);
  5. Условные переменные (condition variable);
  6. Критические секции (critical section);
  7. Высокоуровневые очереди и планировщики.



## Спинлоки (spinlock)



База – все блокировки в ядре ОС основаны на спинлоках, которые в свою очередь используют атомарные операции, без этого реализовать безопасное межпроцессорное взаимодействие невозможно.

# Спинлоки (spinlock)

```
int atomicExchange(int* old, int newValue);

// *lock == 0 - никем не захвачен
void spinlock(volatile int* lock)
{
    while (true)
    {
        if (*lock == 0)
        {
            const int old = atomicExchange(lock, 1);
            if (old == 0)
            {
                return;
            }
        }
    }
}
```

# Семафоры (semaphore)

Семафор — это объект, над которым можно выполнить три операции:

1. Инициализация семафора (задать начальное значение счётчика)
2. Захват семафора (ждать пока счётчик станет больше 0, после этого уменьшить счётчик на единицу)
3. Освобождение семафора (увеличить счётчик на единицу)

Реализуется ОС, описан в POSIX, на базе семафора можно реализовать остальные механизмы синхронизации.

---

# Мьютексы (mutex)

```
#include <mutex>
```

```
std::mutex m;
```

```
m.lock();
```

```
m.unlock();
```

```
if (m.try_lock())  
    m.unlock();
```

---

# Мьютексы (mutex)

```
int i = 0;

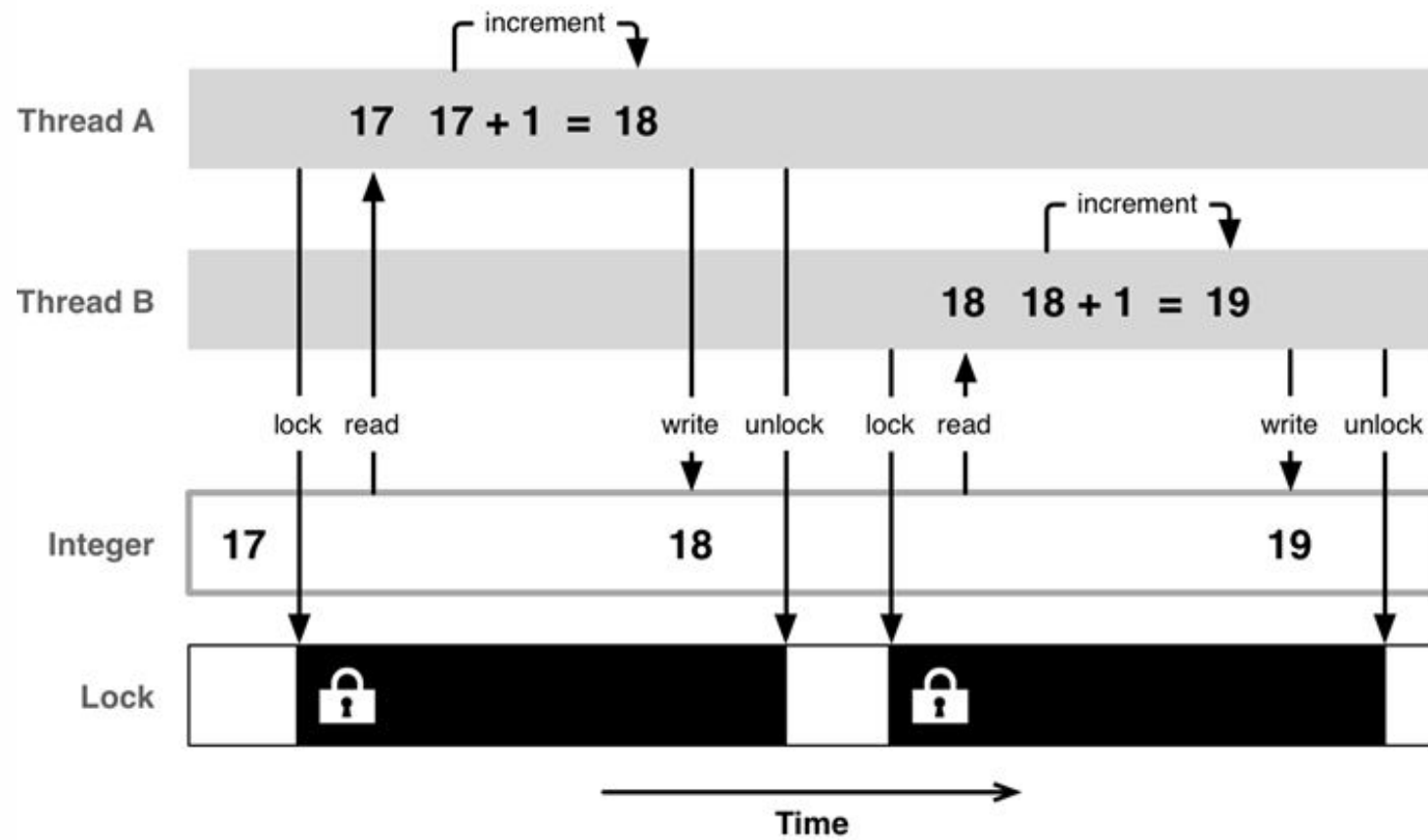
std::mutex mutex;

void plus1()
{
    std::lock_guard<std::mutex> lock(mutex);
    i += 1;
}

std::thread t1(plus1);
std::thread t2(plus1);
```



# Мьютексы (mutex)



## recursive\_mutex

```
std::mutex m;  
m.lock();  
m.lock(); // Неопределенное поведение
```

```
std::recursive_mutex m;  
m.lock();  
m.lock(); // Ок
```

Количество lock и unlock должно совпадать

---

# timed\_mutex

```
#include <mutex>
```

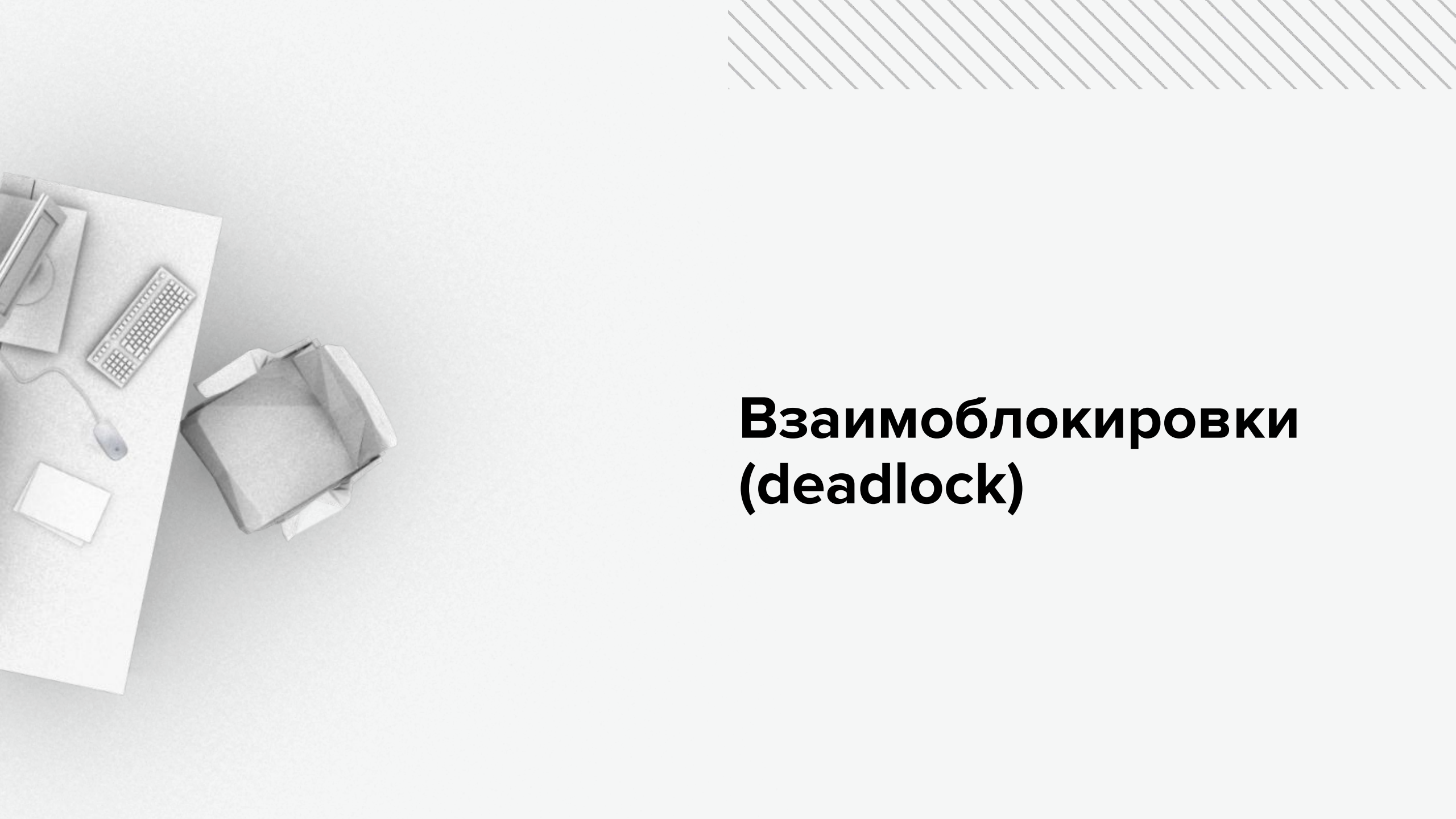
```
std::timed_mutex m;
```

```
m.lock();  
m.unlock();
```

```
if (m.try_lock())  
    m.unlock();
```

```
auto period = std::chrono::milliseconds(100);  
if (m.try_lock_for(period))  
    m.unlock();
```

```
auto now = std::chrono::steady_clock::now();  
m.try_lock_until(now + std::chrono::seconds(1));
```



# **Взаимоблокировки (deadlock)**

## Взаимоблокировки (deadlock)

```
std::mutex m1;  
std::mutex m2;  
  
void t1() // thread 1  
{  
    std::lock_guard<std::mutex> lock1(m1);  
    std::lock_guard<std::mutex> lock2(m2);  
}  
  
void t2() // thread 2  
{  
    std::lock_guard<std::mutex> lock1(m2);  
    std::lock_guard<std::mutex> lock2(m1);  
}
```

## Блокировка в одном и том же порядке

```
void t1() // thread 1
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
}

void t2() // thread 2
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
}
```

# Одновременная блокировка

Иногда дать гарантию на блокировку в одном и том же порядке дать нельзя.

```
class Data
{
    std::mutex m_;
public:
    void apply(const Data& data)
    {
        std::lock_guard<std::mutex> lock1(m_);
        std::lock_guard<std::mutex> lock2(data.m_);
        ...
    }
};
```

# Одновременная блокировка

```
Data d1;
```

```
Data d2;
```

```
d1.apply(d2); // thread 1
```

```
d2.apply(d1); // thread 2
```

```
void apply(const Data& data)
```

```
{
```

```
    using Lock = std::unique_lock<std::mutex>;
```

```
    Lock lock1(m_, std::defer_lock);
```

```
    Lock lock2(data.m_, std::defer_lock);
```

```
    std::lock(lock1, lock2);
```

```
    ...
```

```
}
```



# Условные переменные (condition\_variable)

Средство для обеспечения коммуникации потоков.

```
Data data;
```

```
std::mutex m;
```

```
std::condition_variable dataReady;
```

```
void consumer() // thread 1
```

```
{
```

```
    std::unique_lock<std::mutex> lock(m);
```

```
    while (!data.ready())
```

```
        dataReady.wait(lock);
```

```
}
```

## Условные переменные (condition\_variable)

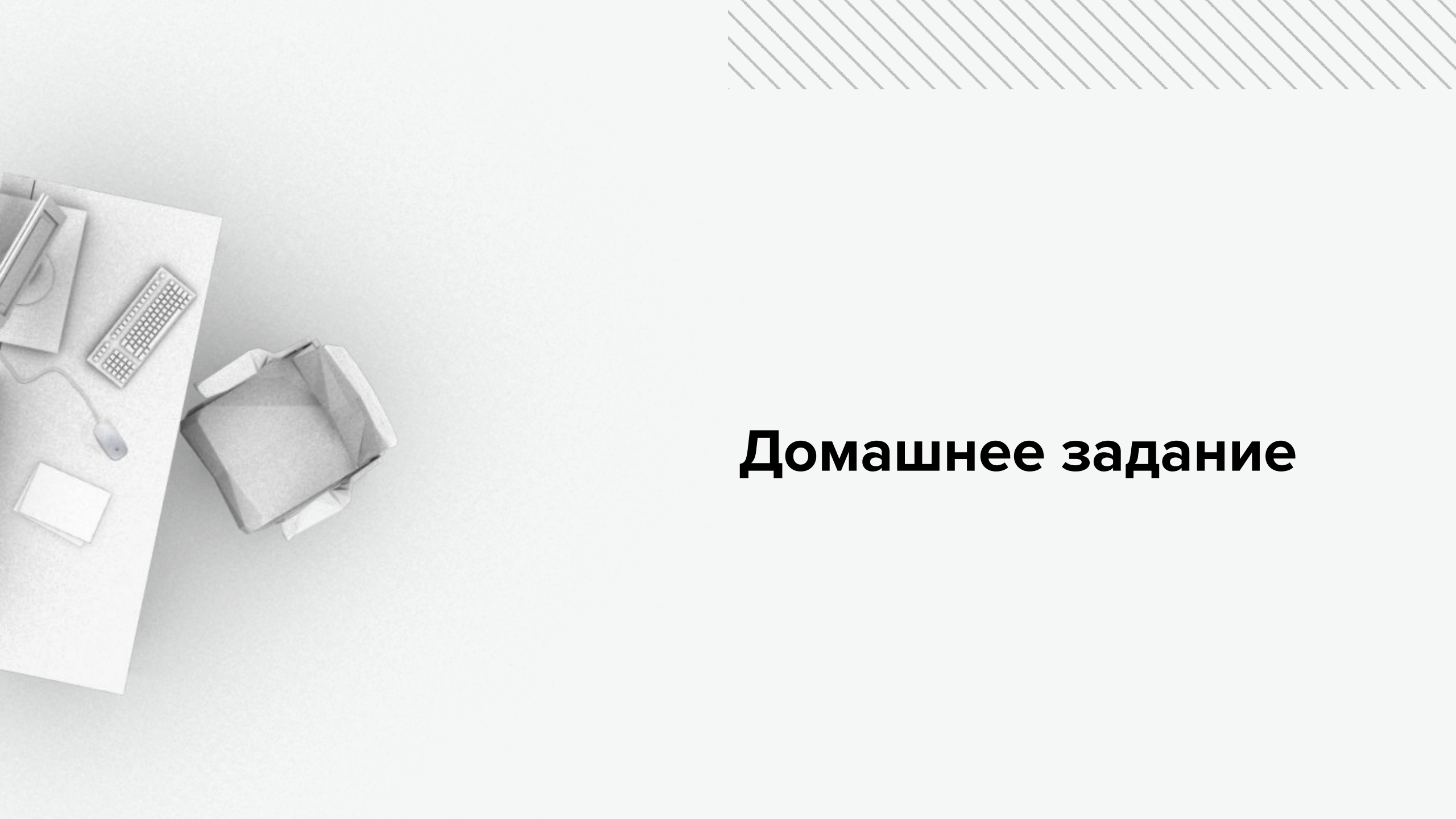
```
void producer() // thread 2
{
    {
        std::lock_guard<std::mutex> lock(m);
        data.prepare();
    }
    dataReady.notify_one();
}
```

# Пул потоков (thread pool)

Создание потока – дорогая операция, поэтому иногда хочется этого избежать.

Идея:

1. Создаем N потоков, каждый поток либо выполняет задачу, либо спит;
2. Новая задача добавляется в очередь, при этом происходит оповещение спящих потоков;
3. Проснувшись поток проверяет, что в очереди есть задача, после чего извлекает её из очереди и выполняет;
4. Если задачи нет, поток засыпает, ожидая оповещения.



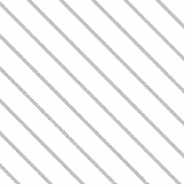
# Домашнее задание

# Домашнее задание (1)

Реализовать пул потоков со следующим интерфейсом:

```
class ThreadPool
{
public:
    explicit ThreadPool(size_t poolSize);

    // pass arguments by value
    template <class Func, class... Args>
    auto exec(Func func, Args... args) -> std::future<decltype(func(args...))>;
};
```



---

## Домашнее задание по уроку #9

Домашнее задание №8

#054

?

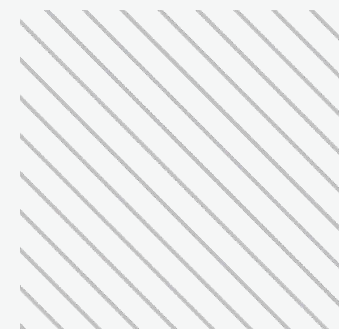
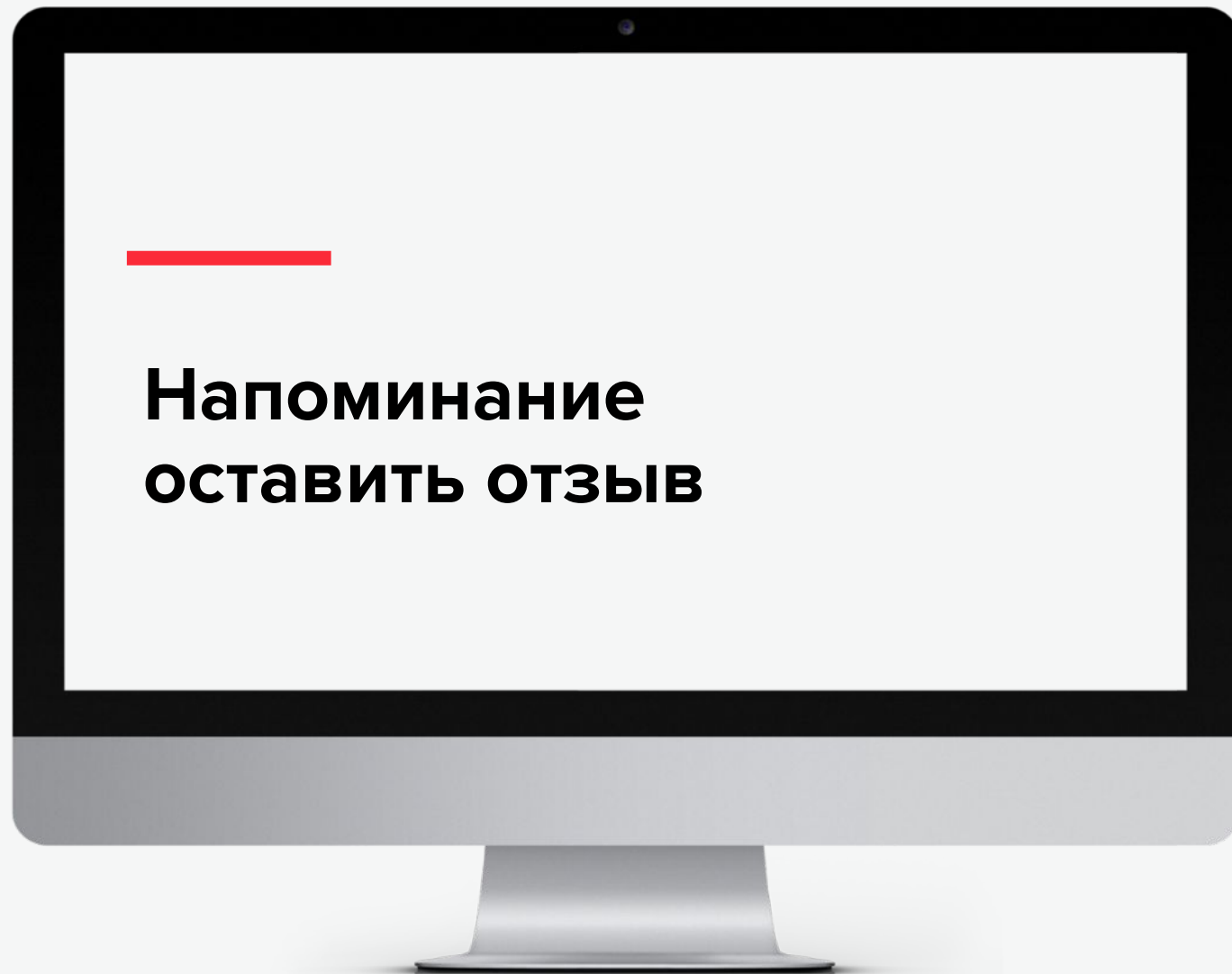
Баллов  
за задание

03.12.20

Срок  
сдачи

Полезная литература в помощь

- Документация стандартной библиотеки



**СПАСИБО  
ЗА ВНИМАНИЕ**

