Современные средства разработки ПО

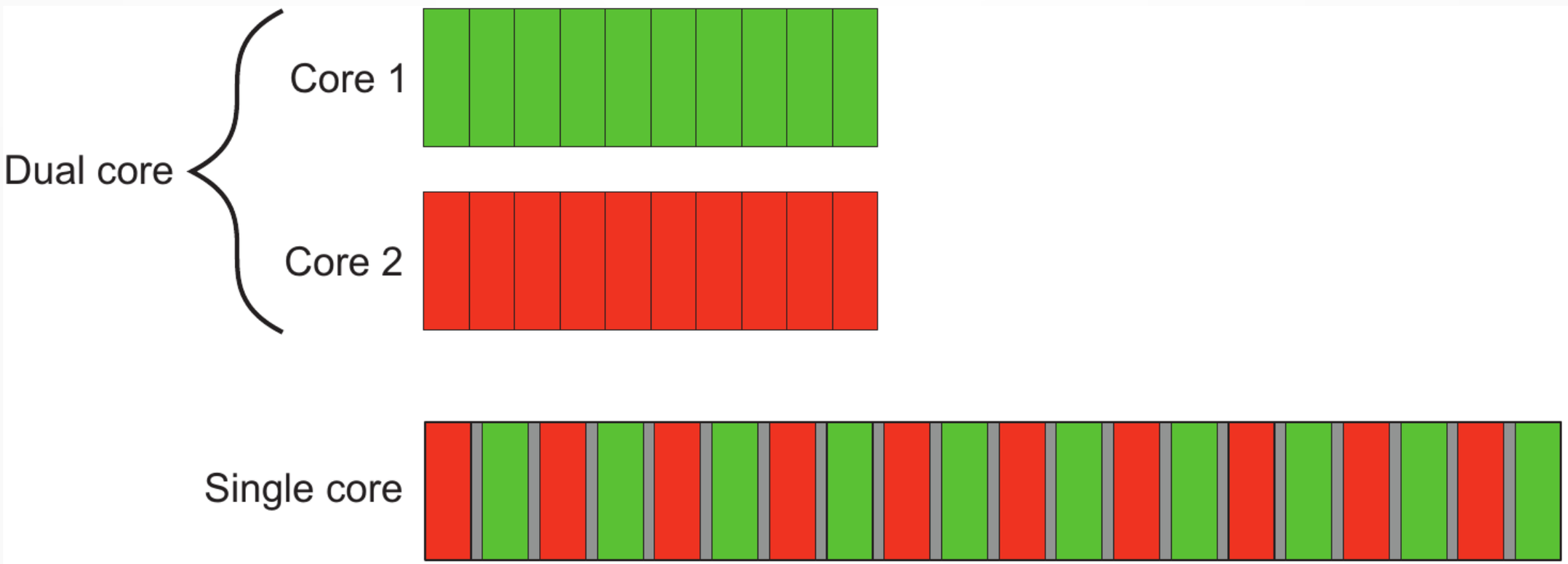
Управление потоками. Разделение данных.

Фетисов Михаил Вячеславович
fetisov.michael@bmstu.ru

Параллелизм и многопоточность Определения

- **Искусственный параллелизм** (**переключение задач**) — имитация параллельного выполнения задач на компьютерах с единственным процессором (ядром).
- **Истинный параллелизм** (**аппаратный параллелизм**) — возможность использовать несколько процессоров (ядер) для достижения действительно параллельного выполнения задач.
- **Многопоточность** — искусственный или истинный параллелизм, реализованный в рамках одного процесса (адресного пространства)

Два подхода к параллелизму
Схема

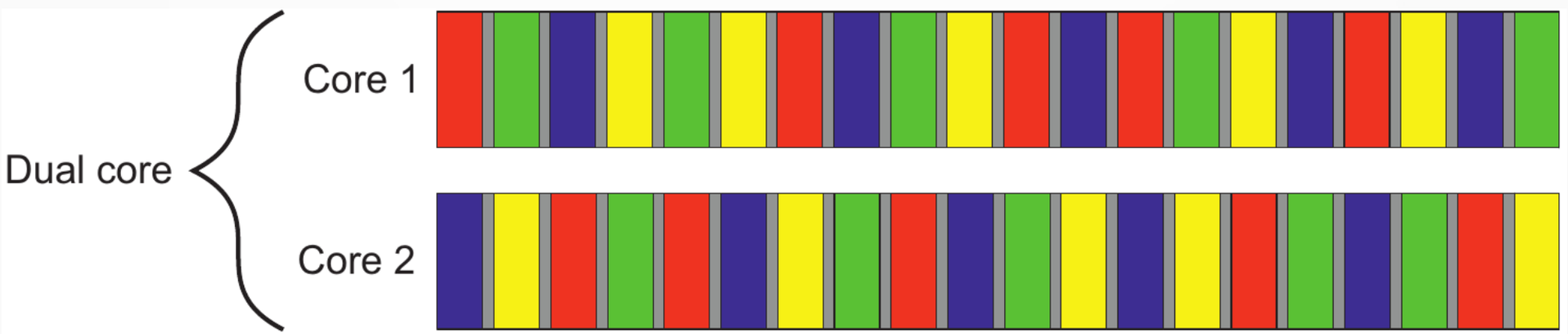


Два подхода к параллелизму
Пояснение

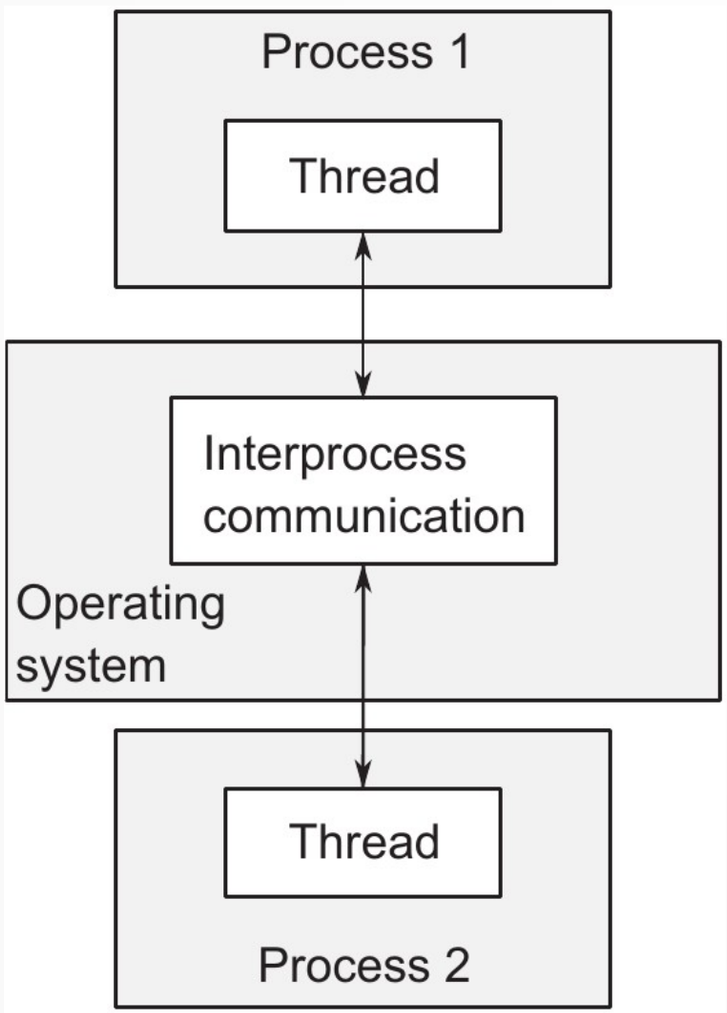
- На рисунке показан идеализированный случай: компьютер, исполняющий ровно две задачи, каждая из которых разбита на десять одинаковых этапов.
- На двухъядерной машине каждая задача может исполняться в своем ядре.
- На одноядерной машине с переключением задач этапы той и другой задачи чередуются. Однако между ними существует крохотный промежуток времени (на рисунке эти промежутки изображены в виде серых полосок, разделяющих более широкие этапы выполнения).
  - Чтобы обеспечить чередование, система должна произвести контекстное переключение при каждом переходе от одной задачи к другой, а на это требуется время.
  - Чтобы переключить контекст, ОС должна сохранить состояние процессора и счетчик команд для текущей задачи, определить, какая задача будет выполняться следующей, и загрузить в процессор состояние новой задачи.
  - Не исключено, что затем процессору потребуется загрузить команды и данные новой задачи в кэш-память; в течение этой операции никакие команды не выполняются, что вносит дополнительные задержки.

Объединение подходов к параллелизму
На двухъядерном компьютере

- В системе с истинным параллелизмом количество задач может превышать число ядер, тогда будет применяться механизм переключения задач.

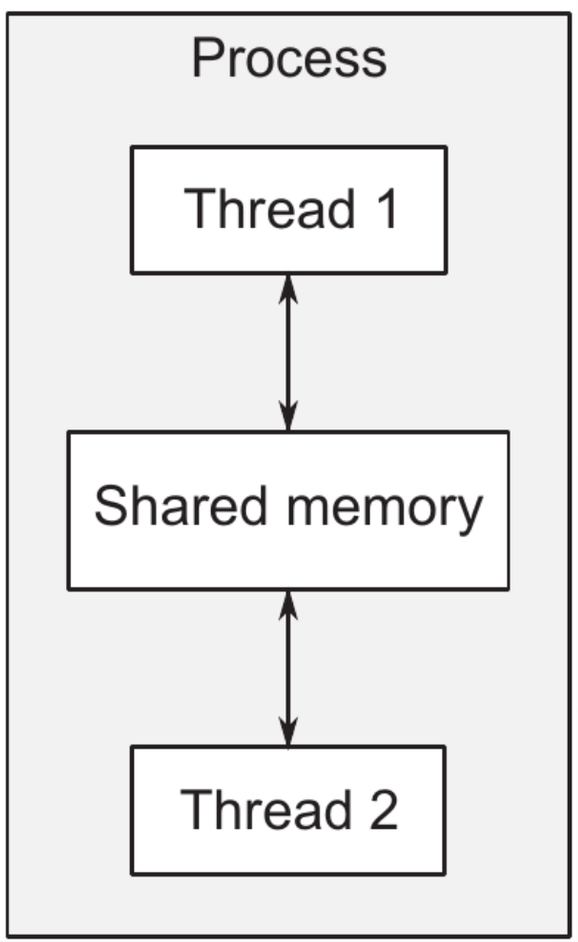


Межпроцессный параллелизм
Плюсы и минусы



- Плюсы:
  - разделение обязанностей,
  - высокая надёжность,
  - в шаге от распределённой системы.
- Минусы:
  - используется больше ресурсов,
  - взаимодействие занимает много времени.

Многопоточность
Плюсы и минусы



- Плюсы:
  - разделение обязанностей,
  - высокая производительность,
  - потребляется меньше ресурсов системы.
- Минусы:
  - слабая надёжность.

Многопоточность
Когда используется

- Для разделения обязанностей
- Для повышения производительности
- Для экономии ресурсов
- Когда мы можем контролировать надёжность программы (качество + детерминизм)

Многопоточность требует изменения подхода к проектированию

- Если требуется, чтобы программа выигрывала от увеличения вычислительной мощности, то ее необходимо проектировать как набор параллельных задач.

Проектирование многопоточности
Два подхода

- **Распараллеливание по задачам** — разделение программы на задачи и запуск их параллельно
- **Распараллеливание по данным** — выполнение одинаковых операций с различными данными

Многопоточность
Когда она вредна

- Когда затраты на распараллеливание не компенсируют получаемый эффект (однако возможны приёмы для компенсации вреда):
  - **слишком мелкие и быстро выполняемые задачи** (может помочь *гетерогенный параллелизм*, *асинхронность*);
  - **задач слишком много** (может помочь *пул потоков*, *асинхронность*).
- Не нужно стремиться распараллеливать всю программу — только наиболее очевидные её части.

Параллелизм и многопоточность в C++

Hello, Concurrent World!

```cpp
#include <iostream>
int main()
{
    std::cout << "Hello, World!" << std::endl;
}
```

```cpp
#include <iostream>
#include <thread>   // 1 - заголовочный файл для поддержкт многопоточности
void hello()        // 2 - в каждом потоке должна быть начальная функция
{
    std::cout << "Hello, Concurrent World!" << std::endl;
}
int main()
{
    std::thread t(hello); // 3 - запуск потока (функции hello)
    t.join();             // 4 - ожидание завершение потока t
}
```

Hello From...

```cpp
#include <iostream>
#include <thread>
void helloFunction() {
    std::cout << "Hello from a function." << std::endl;
}
class HelloFunctionObject{
public:
    void operator()() const {
        std::cout << "Hello from a function object." << std::endl;
    }
};
int main(){
    std::thread t1{helloFunction};
    std::thread t2{HelloFunctionObject()};
    std::thread t3{[]{std::cout << "Hello from a lambda." << std::endl;}};
    t1.join();
    t2.join();
    t3.join();
}
```

Некоторые проблемы при распараллеливании

```cpp
#include <iostream>
#include <cstdio>
#include <thread>
void f(int n,std::string const& s) {
    for(int i=0; i < n; ++i)
        std::cout << i << ": " << s << std::endl;
}
void oops(int n) {
    char buffer[1024];
    sprintf(buffer, "%i", n);
    std::thread t(f, n, buffer); // std::string(buffer)
    t.detach();                  // t.join();
}
int main() {
    oops(30);
    std::cout << "All done!" << std::endl;
}
```

Многопоточность
Гонки

```cpp
#include <iostream>
#include <list>
int main() {
    std::list<unsigned long> li;
    for(unsigned long i = 0; i < 100000000; ++i)
        li.push_back(i);
    std::cout << "Кол-во элементов в списке: " << li.size() << std::endl;
}
```

Многопоточность
Гонки.

```
#include <iostream>
#include <thread>
#include <list>
std::list<unsigned long> li;
void add_range(unsigned long from, unsigned long to) {
    for(unsigned long i = from; i < to; ++i)
        li.push_back(i);
}
int main() {
    std::thread t1(add_range,0,1000);
    std::thread t2(add_range,1000,2000);
    std::thread t3(add_range,2000,3000);
    std::thread t4(add_range,3000,4000);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    std::cout << "Кол-во элементов в списке: " << li.size() << std::endl;
}
```

Гонка данных
Определение

- **Гонка данных** — это ситуация, когда по меньшей мере два потока имеют одновременный доступ к некоторой переменной и хотя бы один из потоков производит её запись

Многопоточность
Гонки. Использование мютекса.

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <list>
std::list<unsigned long> li;
std::mutex               li_mutex;
void add_range(unsigned long from, unsigned long to) {
    li_mutex.lock();
    for(unsigned long i = from; i < to; ++i)
        li.push_back(i);
    li_mutex.unlock();
}
int main() {
    std::thread t1(add_range,0,1000);
    std::thread t2(add_range,1000,2000);
    std::thread t3(add_range,2000,3000);
    std::thread t4(add_range,3000,4000);
    t1.join(); t2.join(); t3.join(); t4.join();
    std::cout << "Кол-во элементов в списке: " << li.size() << std::endl;
}
```

Более корректное применение мьютекса с использованием RAII

```
#include <iostream>
#include <thread>
#include <mutex>
#include <list>
std::list<unsigned long> li;
std::mutex                li_mutex;
void add_range(unsigned long from, unsigned long to) {
    std::lock_guard<std::mutex> guard(li_mutex);
    for(unsigned long i = from; i < to; ++i)
        li.push_back(i);
}
int main() {
    std::thread t1(add_range,0,1000);
    std::thread t2(add_range,1000,2000);
    std::thread t3(add_range,2000,3000);
    std::thread t4(add_range,3000,4000);
    t1.join(); t2.join(); t3.join(); t4.join();
    std::cout << "Кол-во элементов в списке: " << li.size() << std::endl;
}
```

Мьютекс
Определение

- **Мьютекс** (англ. *mutex*, от *mutual exclusion* — «взаимное исключение») — примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода.
- В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом.
- Если другому потоку будет нужен доступ к данным, защищённым мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобождён.

Мьютекс
Проблема. Взаимоблокировка

- **Мёртвая блокировка**, **взаимоблокировка**, **дедлок** (от англ. *deadlock*) — это состояние, в котором каждый из двух или более потоков заблокирован в ожидании ресурса, занятого другим потоком, и до своей разблокировки не может освободить ресурс, которого ожидает другой поток.
- Результат мёртвой блокировки — полная остановка работы потоков, а чаще всего и всей программы. Навечно.

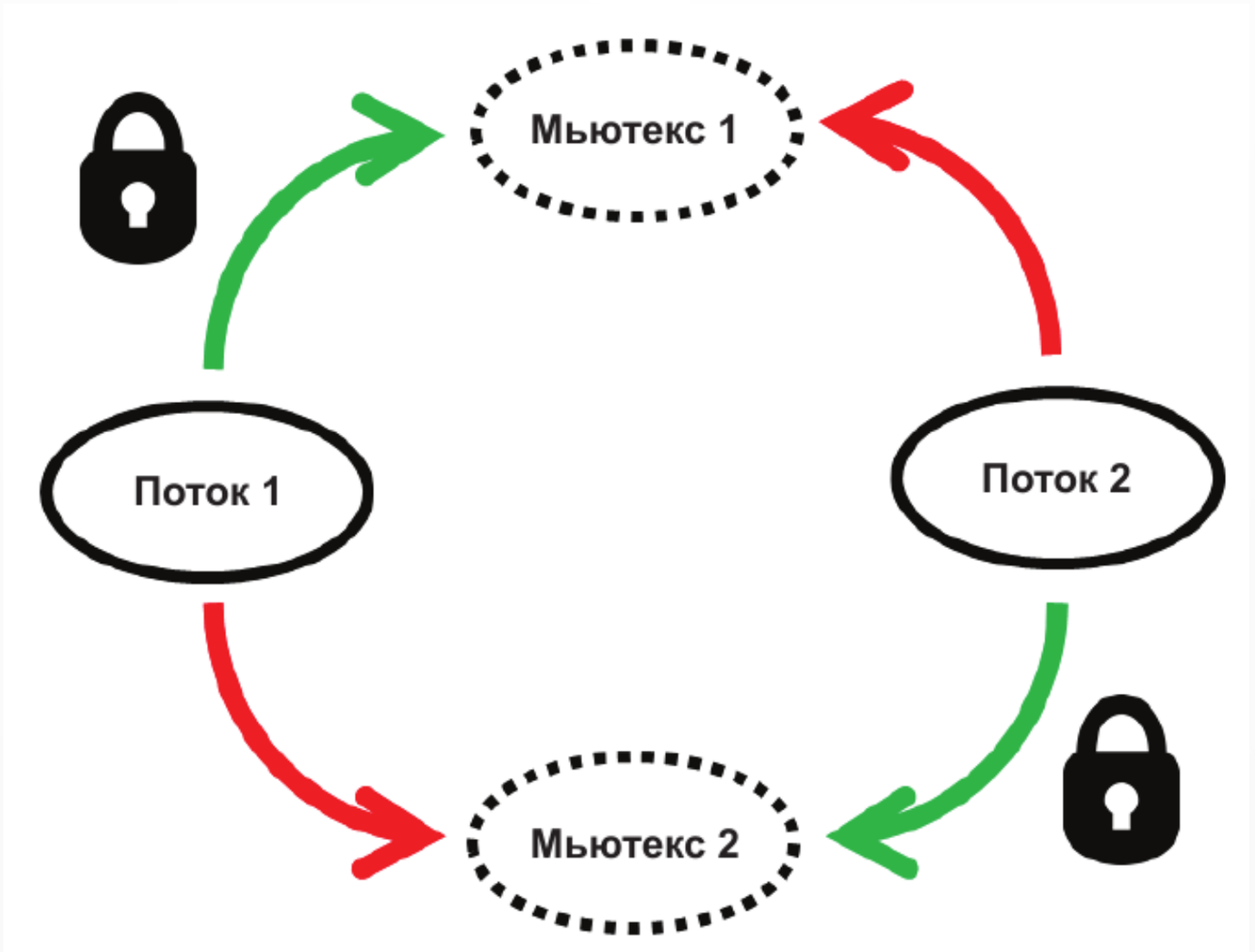
Пример возможного возникновения взаимоблокировки

```
#include <iostream>
#include <mutex>
int sharedVariable = 0;
std::mutex m;
int getVar() {
    return 10/sharedVariable;
}
int main() {
    m.lock();
    sharedVariable = getVar();
    m.unlock();
}
```

Правила работы с мьютексами

- Если функция getVar выбросит исключение, мьютекс m никогда не будет освобождён.
- Никогда, ни в коем случае нельзя вызывать из-под мьютекса функцию, внутреннее устройство которой неизвестно. Если функция getVar пытается захватить мьютекс m, поведение программы не определено, так как этот мьютекс — не рекурсивный. В большинстве подобных случаев неопределённое поведение выражается в мёртвой блокировке.
- Вызывать функции из-под блокировки опасно ещё по одной причине. Если функция определена в сторонней библиотеке, в новой её версии реализация и поведение функции могут измениться. Даже если первоначально опасности мёртвой блокировки не было, она может появиться в будущем.

Взаимоблокировка из-за захвата мьютексов в различном порядке



Блокировщики

- Блокировщики управляют захватом и освобождением ресурса посредством идиомы RAII.
- В стандарте имеется четыре вида блокировщиков:
  - **std::lock_guard** предназначен для однократного захвата мьютекса при создании блокировщика и однократного освобождения при уничтожении;
  - **std::unique_lock –** при создании блокировщика можно с помощью специального параметра отложить захват мьютекса; за время жизни блокировщика мьютекс можно многократно открывать и запирать; освобождение мьютекса в деструкторе гарантируется;
  - **std::shared_lock** (C++14) можно использовать для блокировки читателей и писателей;
  - **std::scoped_lock** (C++17) умеет запирать несколько мьютексов за одну атомарную операцию;

Потокобезопасная инициализация

- Если значение переменной никогда не изменяется, нет нужны синхронизировать доступ к ней с помощью дорогостоящих механизмов блокировки или даже атомарных переменных. Нужно лишь присвоить ей начальное значение потокобезопасным способом.
- В языке C++ есть три способа потокобезопасной инициализации:
  - константные выражения (constexpr);
  - функция **std::call_once** вместе с флагом **std::once_flag**;
  - локальная статическая переменная.

Функция std::call_once и флаг std::once_flag

```cpp
#include <iostream>
#include <thread>
#include <mutex>
std::once_flag onceFlag;
void do_once() {
    std::call_once(onceFlag, [](){ std::cout << "Only once." << std::endl; });
}
void do_once2() {
    std::call_once(onceFlag, [](){ std::cout << "Only once2." << std::endl; });
}
int main() {
    std::thread t1(do_once);
    std::thread t2(do_once);
    std::thread t3(do_once2);
    std::thread t4(do_once2);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}
```

Функция std::call_once и флаг std::once_flag
Как это работает

- Переменная типа **std::once_flag** может находиться в двух состояниях, показывающих, произошло ли уже некоторое однократное действие.
- Если в момент вызова функции **std::call_once** значение флага указывает, что действие уже состоялось, вызов немедленно завершается, ничего не делая; такой вызов функции **std::call_once** называется *пассивным*.
- В противном случае функция **std::call_once** вызывает переданный ей в качестве аргумента вызываемый объект (в частности, функцию) – такой вызов функции **std::call_once** называется *активным*.
- Если выполнение вызываемого объекта приводит к исключению, оно передаётся наружу из вызова функции **std::call_once**. Такой её вызов называется *исключительным*.
- Если же выполнение вызываемого объекта завершается нормально (вызов в этом случае называется *возвращающим*), флаг меняет своё состояние, и все последующие вызовы функции **std::call_once** с этим флагом гарантированно будут пассивными.
- Все вызовы функции **std::call_once** с одним и тем же флагом **std::once_flag** образуют вполне упорядоченную последовательность, в начале которой находится ноль или более исключительных вызовов, затем ровно один возвращающий, после которого следуют только пассивные вызовы.

Потокобезопасная инициализация Локальная статическая переменная

- Статические переменные, локальные для блока, инициализируются один раз ленивым образом.
- В стандарте C++ 11 для локальных статических переменных появилась новая гарантия: их инициализация потокобезопасна.
  - (нужно убедиться, что компилятор реализует требование стандарта C++ 11 о потокобезопасной инициализации статических переменных)

Потокобезопасная инициализация Мейерсовский одиночка

```
class MySingleton
{
public:
    static MySingleton& getInstance() {
        static MySingleton instance;
        return instance;
    }
private:
    MySingleton() = default;
    ~MySingleton() = default;
    MySingleton(const MySingleton&) = delete;
    MySingleton& operator=(const MySingleton&) = delete;
};

int main() {
    MySingleton::getInstance();
}
```

Вопросы?

Фетисов Михаил Вячеславович
fetisov.michael@bmstu.ru