

# ЛЕКЦИЯ 14

# КОНКУРЕНТНОСТЬ

АЛГОРИТМИЗАЦИЯ И  
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.



# ПОТОКИ

Что такое **поток исполнения** (thread of execution)?

# ПОТОКИ

Что такое **поток исполнения** (thread of execution)?

По определению стандарта:

*A thread of execution (also known as a thread) is a single flow of control within a program* [intro.multithread.general]

При этом:

*The execution of the entire program consists of an execution of all of its threads* [C++17, 4.7]

Логическая многопоточность (concurrency) внутри программы не имеет отношения к аппаратной параллельности (parallelism)

# СОЗДАНИЕ ПОТОКА

- Поток создаётся, начиная с C++11, в конструкторе класса `std::thread`

```
int main() {  
    std::thread t([]{  
        std::cout << "Hello, world!" << std::endl;});  
    // тут что угодно пока поток выполняется  
    t.join();  
}
```

- Поток стартует с переданной в конструктор функции или функтора
- Обязательно надо сделать `join()` или `detach()`

# ПОТОК – ЭТО ВСЕГДА ПОТОК

- Создаваемый поток в принципе ничем не отличается от родителя

```
std::cout << "Main: " << std::this_thread::get_id()
<< std::endl;
std::thread t([]{
    std::cout << "Spawned: " <<
    std::this_thread::get_id() << std::endl;});
```

Теперь очевидно, что в функции два потока: на экране два разных id

На уровне языка нас мало волнует кто и как управляет потоками  
Кстати, видите ли вы некую опасность в данном коде?

# ОБЛАСТИ ПАМЯТИ

- Любая программа работает с объектами (objects). Объект – это что угодно, требующее места (storage) для хранения значения.

`int x = 42; // x – это объект, а 42 – нет`

`foo(&x); // теперь x требует сохранения в память`

- Область памяти (memory location) определена в стандарте так:  
*A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width* (C++17, 4.4)
- Основная часть этой лекции будет посвящена тому, как передача управления в программе сочетается с состоянием областей памяти в ней.

# ГОНКА ЗА ОБЛАСТЬ ПАМЯТИ

- Следующая программа весьма иллюстративна

```
int x;  
void race() {  
    for(int i = 0; i < 100; ++i) {x += 1; use(x); }  
    for(int i =0; i < 100; ++i) { x -= 1; use(x) }  
}
```

- Для одного потока всё хорошо

```
std::thread t{race}; t.join();  
assert(x == 0);
```

- Что здесь произойдет, если в эту функцию зайдут два потока?

# DATA RACE

- Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. [intro.races]
- The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior. [intro.races]



# ПОПЫТКА ИСПРАВИТЬ СИТУАЦИЮ

- Давайте отключим оптимизацию вокруг x

```
volatile int x; // область памяти
```

```
void race() {  
    for(int i = 0; i < 100; ++i) {x += 1; use(x); }  
    for(int i =0; i < 100; ++i) { x -= 1; use(x) }  
}
```

- Для двух потоков всё очень плохо

```
std::thread t1{race}, t2{race}; t1.join(), t2.join();  
std::cout << x << std::endl;
```

- volatile **не работает** для предупреждения data races

# ОБСУЖДЕНИЕ

- А для чего он тогда работает?
- Что вы, например, думаете про его использование в MMIO?

```
void waitForMMIO() {  
    volatile unsigned short* semPtr == MMAPED_ADDR;  
    while ((*semPtr) != ISOK);  
}
```

- К чему приведет попытка убрать `volatile` в этом коде?

# VOLATILE – ЭТО ОБЕЩАНИЕ

- В языке у него такой же статус, как у const: вы что-то обещаете

```
int foo(int *p);
```

```
int bar(const int *p);
```

```
int buz(volatile int *p);
```

```
int i; const int ci = 0; volatile int vi;
```

```
foo(&i); foo(&ci); foo(&vi);
```

```
bar(&i); bar(&ci); bar(&vi);
```

```
buz(&i); buz(&ci); buz(&vi);
```

Какие из этих выводов ошибочны?

# НИКАКИХ ГАЙДЛАЙНОВ

- Обратите внимание

*Two accesses to the same object of type `volatile std::sig_atomic_t` do not result in a data race [intro.races]*

- То есть ровно для одного типа `volatile` по стандарту работает

# БОЛЕЕ ТОНКАЯ ПОПЫТКА ИСПРАВЛЕНИЯ

- Взрослые люди умеют делать взрослые вещи

```
void race() {  
    stick_this_thread_to_core(0);  
    for (int i = 0; i < 100; ++i) { x += 1; use(x); }  
    for (int i = 0; i < 100; ++i) { x -= 1; use(x); }  
}
```

- Тут мы внешними средствами (например средствами POSIX) установили привязку всех входящих в функцию потоков к одному ядру.
- Это снимает проблемы когерентности и визуально чинит проблему. Все ли тут хорошо?

# ПРАВИЛА ГОНКИ

- Гонка происходит если сколько угодно потоков читают область памяти и хотя бы один одновременно пишет в неё же

```
unsigned x = 0, i = 0, j = 0; // области памяти
void readerf() { while(i++ < 'g') x += 0x1; }
void writerf() { while (j++ < 'g') x += 0x10000; }
std::thread t1{readerf}, t2{writerf};
t1.join(); t2.join();
```

- Тут все равно не определено что будет на экране , несмотря на то, что запись и чтение в разных потоках трогают разный байты объекта.

# УБИРАЕМ ГОНКУ

- Область памяти – это скалярный объект.

```
char x[2] = {0, 0}, i = 0, j = 0; // области памяти
void readerf() { while(i++ < 'g') x[0] += 0x1; }
void writerf() { while (j++ < 'g') x[1] += 0x1; }
std::thread t1{readerf}, t2{writerf};
t1.join(); t2.join();
```

- Теперь мы работаем с разными скалярными объектами и data race нет

# СОМНИТЕЛЬНЫЙ КОД

- Вернемся к примеру выше

```
std::cout << "Main: " << std::this_thread::get_id()
<< std::endl;
std::thread t([]{
    std::cout << "Spawned: " <<
    std::this_thread::get_id() << std::endl;});
```

- Основное сомнение – нет ли тут UB?
- Такое чувство, что мы пишем в одну область памяти



# ОБЪЕКТ НИКОГДА НЕ ОБЛАСТЬ ПАМЯТИ

- Мы можем подозревать здесь гонку

```
std::cout << "Main: " << std::this_thread::get_id()  
<< std::endl;  
std::thread t([]{  
    std::cout << "Spawned: " <<  
    std::this_thread::get_id() << std::endl;});
```

- Сложный объект с поведением (например `std::cout`) это никогда не область памяти. Тот же `std::cout` по стандарту (C++17, 30.4.2.5) безопасен
- Несмотря на то, что с объектом потока ничего не случится, порядок вывода на экран символов в этой программе не определен.

# ОБЪЕКТ НИКОГДА НЕ ОБЛАСТЬ ПАМЯТИ

- Такое чувство, что под `std::cout` всё-таки есть объекты скалярного типа
- Как же ему удалось добиться (сравнительной) безопасности?

# ПРОСТАЯ СИНХРОНИЗАЦИЯ

Для синхронизации доступов между потоками управления служат мьютексы

```
int x; // область памяти
std::mutex mforx; // мьютекс для синхронизации x
```

```
void race() {
    for (int i = 0; i < 100; ++i) {
        mforx.lock(); x += 1; mforx.unlock();
    }
    // тоже самое для второго цикла
}
```

Мьютексы вводят отношение happens-before между доступами

# ИНТЕРФЕЙС МЬЮТЕКСА

`std::mutex` поддерживает «очень простой» интерфейс с сюрпризами

метод	сюрприз
<code>lock()</code>	Попытка вызвать повторно в том же потоке это UB. Также может кинуть исключение <code>std::system_error</code>
<code>try_lock()</code>	То же, что и для <code>lock()</code>
<code>unlock()</code>	Попытка разблокировать незахваченный мьютекс – это UB

Какой главный сюрприз в этом интерфейсе?

# RAII СИНХРОНИЗАЦИЯ

Разумеется, в таких условиях делать `lock()` и `unlock()` руками никто никогда не будет

```
for (int i = 0; i < 100; ++i) {  
    lock_guard<mutex> lk{mforx};  
    x += 1;  
}
```

- Защёлка `std::lock_guard<T>` это RAII обёртка над любым классом, поддерживающим интерфейс из методов `lock()` и `unlock()`.
- По логике `scoped_lock` (сравнить со `scoped_ptr`) без копирования и перемещения.

# ИНТЕРМЕДИА: БЕЗОПАСНОСТЬ ИСКЛЮЧЕНИЙ

Когда мы говорим, что некий объект **безопасен относительно исключений**, что мы имеем ввиду?

# НАПОМИНАЛКА

- Код, в котором при исключении могут утечь ресурсы, оказаться в несогласованном состоянии объекты и прочее, называют небезопасным относительно исключений.
- Базовая гарантия: исключение при выполнении операции может изменить состояние программы, но не вызывает утечек и оставляет все объекты в согласованном (**но не обязательно предсказуемом**) состоянии
- Строгая гарантия: при исключении гарантируется **неизменность состояния** программы относительно задействованных в операции объектов (commit/rollback)
- Гарантия бессбойности: функция не генерирует исключений (noexcept)

# ПРИМЕР КАРГИЛЛА

- Все ли понимают, что тут плохо?

```
template <typename T> class MyVector {  
    T* buffer_ = nullptr;  
    size_t capacity_, size_ = 0;  
public:  
    MyVector(const MyVector &rhs) : buffer_(new T[rhs.size_]),  
                                     capacity_(rhs.size_),  
                                     size_(rhs.size_) {  
        std::copy(rhs.buffer_, rhs.buffer_ + rhs.size_, buffer_);  
    }  
}
```

- Например, где бы вы вставили `catch (...)`?



# ОБСУЖДЕНИЕ: БЕЗОПАСНОСТЬ ПОТОКОВ

- Когда мы говорим, что некий объект **безопасен относительно многопоточности**, что мы имеем ввиду?

# НУЛЕВОЙ УРОВЕНЬ БЕЗОПАСНОСТИ

- У нас есть некий нулевой уровень – это нейтральные объекты, например `int`.
- Если `int` защищён синхронизацией, он безопасен, если нет, то нет.
- Можем ли мы спуститься ниже нейтрального уровня?

# ХУЖЕ НЕЙТРАЛЬНОСТИ

- Можем ли мы спуститься ниже нейтрального уровня?
- Да и очень просто. Например, указатель на `int`

```
{  
    lock_guard<mutex> lk{mforx};  
    *px += 1;  
}
```

- Понимаете ли вы в чем проблема?

# ЛУЧШЕ НЕЙТРАЛЬНОСТИ

- Безопасным относительно многопоточного окружения является объект, никакие операции с которым в этом окружении не приводят к data race.
- Шире говоря: не приводят к неконсистентному состоянию
- В этом смысле безопасность относительно потоков похожа на строгую безопасность относительно исключений.
- Не ждет ли нас на этом пути больше сюрпризов?

# ПОТОКОБЕЗОПАСНЫЙ БУФЕР

Вспомним буфер, спроектированный для безопасности исключений

```
template <typename T> struct MyBuffer : public BufImpl {  
    void pop() {  
        size_ -= 1;  
        destroy(buffer_ + size_);  
    }  
  
    T top() const { return buffer_[size_ - 1]; }  
    bool empty() const {return (size_ == 0); }  
    // что-то ещё  
}
```

Как сделать его безопасным для использования несколькими потоками?

# ПОТОКОБЕЗОПАСНЫЙ БУФЕР

Похоже все эти методы необходимо защищать мьютексом

```
mutex bufmut_;  
void pop() {  
    lock_guard<mutex>{bufmut_};  
    size_ -= 1;  
    destroy(buffer_ + size_);  
}  
T top() const; // аналогично  
bool empty() const; // аналогично
```

Сначала простой вопрос. Все ли хорошо в этом коде?

# ПОТОКОБЕЗОПАСНЫЙ БУФЕР



Нельзя забывать про имена объектов, в том числе объектов синхронизации.

```
mutex bufmut_;  
void pop() {  
    lock_guard<mutex> lk{bufmut_};  
    size_ -= 1;  
    destroy(buffer_ + size_);  
}
```

А теперь всё в порядке?

# API RACES

Представим следующий кусок кода, выполняемый сразу двумя потоками на одном глобальном объекте `s`.

```
if (!s.empty()) {    
    auto elem = s.top();  
    s.pop();  
    use(elem);  
}
```

В буфере остался **один** объект, но оба потока прошли проверку на `empty`



# API RACES

Представим следующий кусок кода, выполняемый сразу двумя потоками на одном глобальном объекте `s`.

```
if (!s.empty()) {  
    auto elem = s.top();  
    s.pop();  
    use(elem);  
}
```



- **Первый** поток считал и снял элемент
- **Второй** поток пытается считать несуществующий более элемент

# API RACES



Представим следующий кусок кода, выполняемый сразу двумя потоками на одном глобальном объекте `s`.

```
if (!s.empty()) {  
    auto elem = s.top();  
    s.pop();  
    use(elem);  
}
```

- Это конфликт между `pop()` и `empty()`
- Строго говоря: это не `data race` и не `UB`. Но явно что-то идёт не так. Такие случаи называются `API race`

# API RACES

Представим следующий кусок кода, выполняемый сразу двумя потоками на одном глобальном объекте `s`.


```
if (!s.empty()) {    
    auto elem = s.top();  
    s.pop();  
    use(elem);  
}
```

- В буфере осталось **два** объекта, оба потока прошли проверку на `empty()`

# API RACES

Представим следующий кусок кода, выполняемый сразу двумя потоками на одном глобальном объекте `s`.

```
if (!s.empty()) {  
    auto elem = s.top();  
    s.pop();  
    use(elem);  
}
```



- Оба потока удалили один первый объект, второй оставшийся. В итоге буфер пуст но у обоих объектов копия одного объекта, а второй навсегда потерян.
- Это гонка `pop()` и `top()`.

# ОБСУЖДЕНИЕ

Кажется, попытка сделать буфер потокобезопасным провалилась

```
mutex bufmut_;  
void pop() {  
    lock_guard<mutex> lk{bufmut_};  
    size_ -= 1;  
    destroy(buffer_ + size_);  
}
```

- Что делать? Что пошло не так?

# ПОПЫТКИ ИСПРАВЛЕНИЯ СИТУАЦИИ

Заметим, что никаких API races не будет, если объединить pop и top

```
mutex bufmut_;  
bool try_pop(T &loc) {  
    lock_guard<mutex> lk{bufmut_};  
    if (empty()) return false;  
    loc = buffer_[size_ - 1];  
    size_ -= 1;  
    destroy(buffer_ + size_);  
    return true;  
}
```

- Хороша ли эта идея?

# ПОПЫТКИ ИСПРАВЛЕНИЯ СИТУАЦИИ

Правда, что теперь делать с безопасностью исключений?

```
mutex bufmut_;  
bool try_pop(T &loc) {  
    lock_guard<mutex> lk{bufmut_};  
    if (empty()) return false; // что произойдет здесь?  
    loc = buffer_[size_ - 1]; // сколько времени это займет?  
    size_ -= 1;  
    destroy(buffer_ + size_);  
    return true;  
}
```

- Слишком широкие критические секции – это плохо. Есть ли еще идеи?

# ПОПЫТКИ ИСПРАВЛЕНИЯ СИТУАЦИИ

Как вам такое?

```
mutex bufmut_;  
shared_ptr<T> try_pop();
```

- В целом, выходов много и все они спорные
- Важно отметить, что исторически все контейнеры стандартной библиотеки были спроектированы так, будто никаких потоков нет. Это частично избавляет от API races, но гарантирует в лучшем случае слабую нейтральность.
- Интересно то, что API races – **не самое плохое**



# ПОКРИТИКУЙТЕ SWAP

Следующий метод, который часто хочется реализовывать безопасным – это обмен значениями.

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
    if (this == &rhs) return;
    std::lock_guard<mutex> lk1{bufmut_};
    std::lock_guard<mutex> lk2{rhs.bufmut_};
    std::swap(buffer_, rhs.buffer_);
    std::swap(size_, rhs.size_);
    std::swap(capacity_, rhs.capacity_);
}
```

- Что тут может пойти не так?

# ПОКРИТИКУЙТЕ SWAP

Следующий метод, который часто хочется реализовывать безопасным – это обмен значениями.

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
    if (this == &rhs) return;
    std::lock_guard<mutex> lk1{bufmut_};
    std::lock_guard<mutex> lk2{rhs.bufmut_};
    std::swap(buffer_, rhs.buffer_);
    std::swap(size_, rhs.size_);
    std::swap(capacity_, rhs.capacity_);
}
```

- Что тут может пойти не так?

# DEADLOCKS

Сценарий ошибки: **первый** поток зовёт `x.swap(y)`, а **второй** – `y.swap(x)` для глобальных `x` и `y`

```
if (this == &rhs) return;  
    std::lock_guard<mutex> lk1{bufmut_};  
    std::lock_guard<mutex> lk2{rhs.bufmut_};  
    std::swap(buffer_, rhs.buffer_);  
    std::swap(size_, rhs.size_);  
    std::swap(capacity_, rhs.capacity_);
```



- **Первый** поток взял `x.bufmut_`.
- **Второй** поток взял `y.bufmut_`.

# DEADLOCKS

Сценарий ошибки: **первый** поток зовёт `x.swap(y)`, а **второй** – `y.swap(x)` для глобальных `x` и `y`

```
if (this == &rhs) return;  
std::lock_guard<mutex> lk1{bufmut_};  
std::lock_guard<mutex> lk2{rhs.bufmut_};  
std::swap(buffer_, rhs.buffer_);  
std::swap(size_, rhs.size_);  
std::swap(capacity_, rhs.capacity_);
```



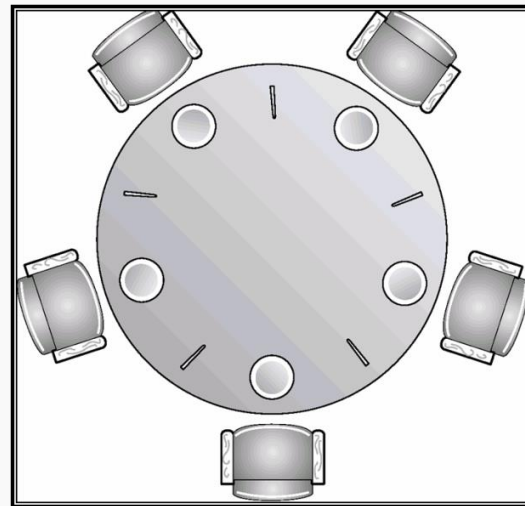
- **Первый** поток ждёт `y.bufmut_`.
- **Второй** поток ждёт `x.bufmut_`.
- Они будут ждать **вечно**. Это deadlock

# ОБЕДАЮЩИЕ ФИЛОСОФЫ

- Задача была сформулирована еще Дейкстрой и Хоаром
- За столом едят несколько философов
- Каждый может есть или думать
- Есть каждый может только двумя вилками
- Задача написать функцию

```
void take(mutex &left, mutex &right) {  
    // ???  
}
```

которая корректно берет левую и правую вилки

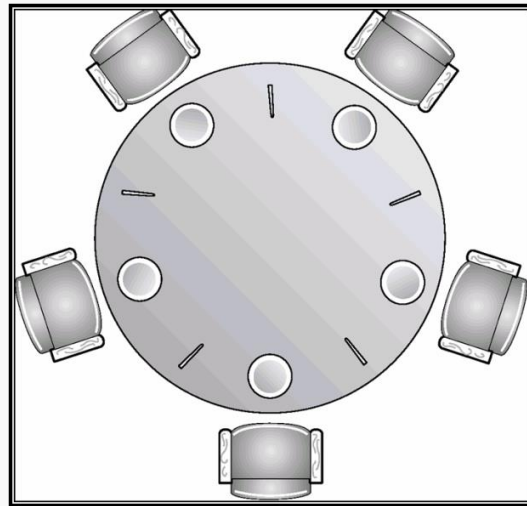


# ОБЕДАЮЩИЕ ФИЛОСОФЫ

- Суть простейшего решения на C++

```
void take(mutex &left, mutex &right) {  
    for (;;) {  
        left.lock();  
        if (right.try_lock())  
            break;  
        left.unlock();  
        std::this_thread_yield();  
    }  
}
```

- Есть стандартный алгоритм



# ОБЕДАЮЩИЕ ФИЛОСОФЫ И C++

- Стандартная функция `std::lock()` позволяет безопасно захватить произвольное количество мьютексов

```
template <class lockable1, class lockable2, class...  
lockableN>
```

```
void lock(lockable1 &lock1, lockable2 &lock2, lockableN&...  
lockn);
```

- Для решения проблемы swap ее можно использовать напрямую

```
template <typename T>  
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {  
    if (this == &rhs) return;  
    std::lock(bufmut_, rhs.bufmut_);
```

- Но тогда придется делать ручной unlock в конце

# РЕШЕНИЕ ПРОБЛЕМЫ SWAP

Используем `adopt_lock`, чтобы захватить взятый `mutex` в оболочку `lock_guard`

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
    if (this == &rhs) return;
    std::lock_guard<mutex> lk1{bufmut_, std::adopt_lock};
    std::lock_guard<mutex> lk2{rhs.bufmut_, std::adopt_lock};
    std::swap(buffer_, rhs.buffer_);
    std::swap(size_, rhs.size_);
    std::swap(capacity_, rhs.capacity_);
}
```

- Что тут может пойти не так?



# РЕШЕНИЕ ПРОБЛЕМЫ SWAP В C++17

Начиная с 2017 года можно использовать оболочку RAII `std::scoped_lock`

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
    if (this == &rhs) return;
    std::scoped_lock sl{bufmut_, rhs.bufmut_};
    std::swap(buffer_, rhs.buffer_);
    std::swap(size_, rhs.size_);
    std::swap(capacity_, rhs.capacity_);
}
```

- Она внутри использует `std::lock` и хранит несколько взятых защелок.

# ОБСУЖДЕНИЕ

- Проблемы проектирования выглядят гораздо более серьезными, чем в случае с исключениями
- Это и понятно: исключения рисуют произвольное количество выходных дуг, а потоки – еще и произвольное количество входных дуг
- На этом месте следует задать один вопрос: а хотим ли мы безопасные относительно потоков контейнеры?
- И если да, то в каком виде?
- Ответ на этот вопрос мы пока отложим

# РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М.: «Вильямс», 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы - The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М.: «Вильямс», 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.