Лекция 3
Многопоточное программирование
C++ 11/14 Threading

Михаил Георгиевич Курносов

Email: mkurnosov@gmail.com

WWW: http://www.mkurnosov.net

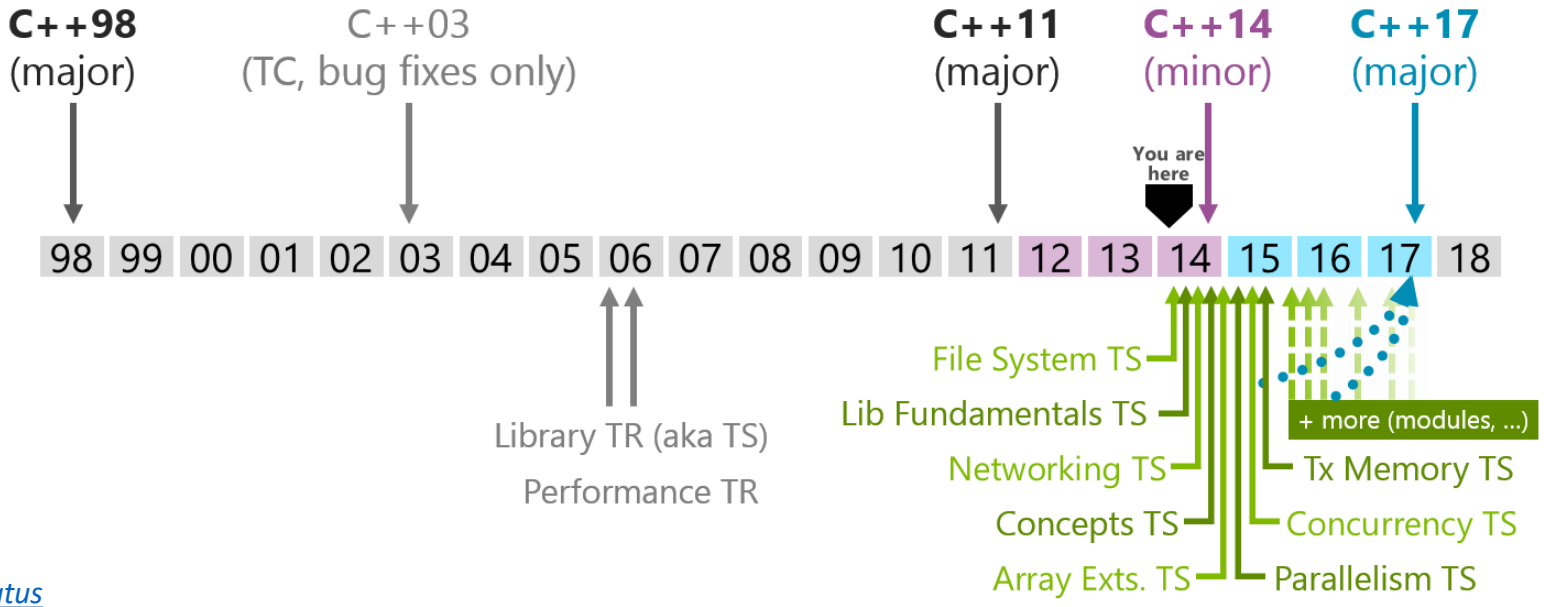
Курс «Параллельные вычислительные технологии»

Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Весенний семестр, 2017

C++11 & C++14

- **Стандарт C++11:** принят в 2011 году, ISO/IEC 14882:2014 (major): многопоточность
- **Стандарт C++14:** принят в 2014 году, ISO/IEC 14882:2014 (minor) http://isocpp.org/std/the-standard
- **Working Draft C++14 (N4567):** http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4567.pdf



[*] *https://isocpp.org/std/status*

Поддержка C++14 компиляторами

- **GCC** (default: gnu++14)
  https://gcc.gnu.org/projects/cxx0x.html (-std=c++11)
  https://gcc.gnu.org/projects/cxx1y.html (-std=c++14, experimental)
  https://gcc.gnu.org/onlinedocs/gcc-5.3.0/libstdc++/manual/manual/status.html#status.iso.2011

- **Clang** (default: c++98)
  http://clang.llvm.org/cxx_status.html
  Options: -std=c++11, -std=c++14

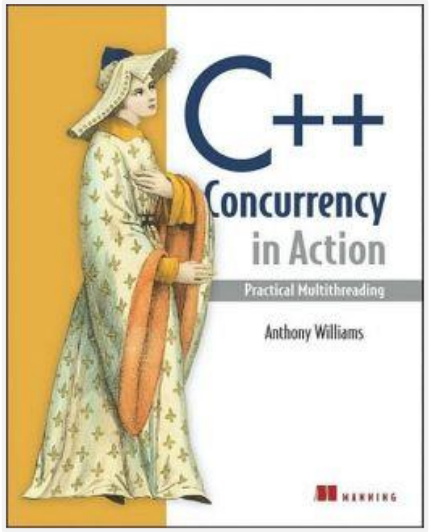
- **Intel C++**
  https://software.intel.com/en-us/articles/c14-features-supported-by-intel-c-compiler

- **Oracle Solaris Studio**
  http://www.oracle.com/technetwork/server-storage/solarisstudio/features/compilers-2332272.html

- **Visual C++**
  http://blogs.msdn.com/b/vcblog/archive/2014/08/21/c-11-14-features-in-visual-studio-14-ctp3.aspx

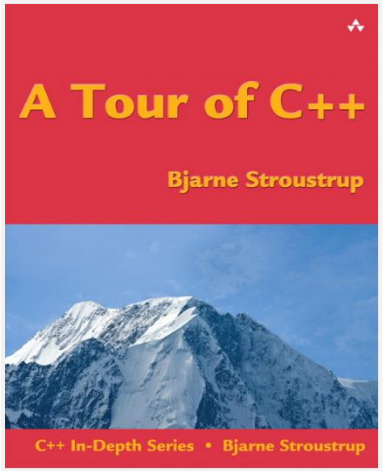
Литература

- Уильямс Э. **Параллельное программирование на C++ в действии. Практика разработки многопоточных программ**. - М.: ДМК Пресс, 2012.



- Anthony Williams. **C++ Concurrency in Action. Practical Multithreading**, Manning, 2012

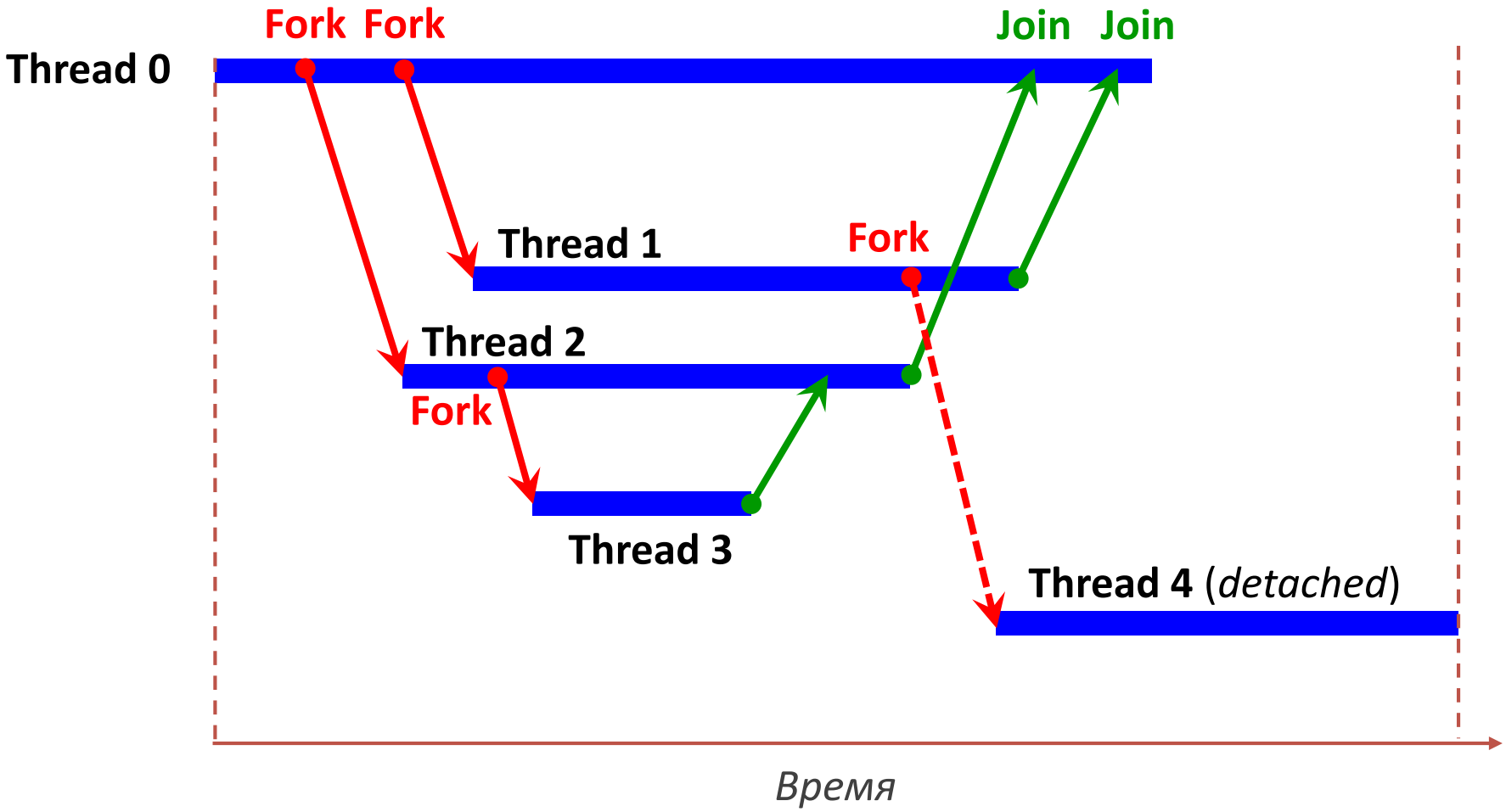
- Bjarne Stroustrup**. A Tour of C++.** The C++ In-Depth Series, Pearson Education, 2013



C++11 Thread support library + Atomic operations library

- **Threads** (class std::thread, namespace std::this_thread)
- **Mutual exclusion** (mutex, lock_guard, lock, call_once, …)
- **Condition variables** (condition_variable)
- **Futures** (future, promise, async, launch, packaged_task)
- **Atomic operations** (std::atomic)
- …

C++11 Threads – Fork-Join Model

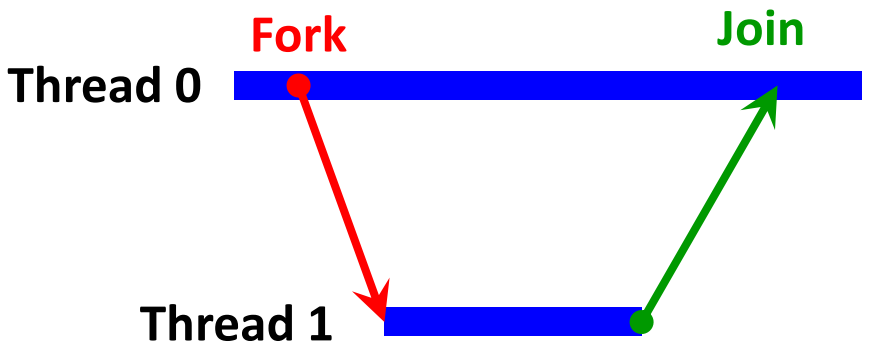


- **Fork –** создание нового потока
- **Join** – ожидание одним потоком завершения другого (объединение потоков управления)

Управление потоками

Hello, Multithreaded World!

```cpp
#include <iostream>
#include <thread>

void hello()
{
    std::cout << "Hello, Multithreaded World!\n";
}

int main()
{
    // Создаем и запускаем новый поток выполнения
    std::thread mythread(hello);

    // Продолжаем вычисления в главном потоке

    // Ожидаем завершения потока mythread
    mythread.join();
    return 0;
}
```



Компиляция многопоточных программ на C++11

```
# GNU/Linux GCC compiler
$ g++ -Wall -std=c++11 -pthread -ohello ./hello.cpp


# GNU/Linux Clang (LLVM)
$ clang++ -Wall -std=c++11 -pthread -ohello ./hello.cpp


# GNU/Linux Intel C++ Compiler
$ icpc -Wall -std=c++11 -pthread -ohello ./hello.cpp
```

- **Oracle Solaris Studio** (GNU/Linux, Oracle Solaris)
- **Microsoft Visual C++ Express**
- **Online C++ Compilers:** liveworkspace.org, coliru.stacked-crooked.com, gcc.godbolt.org, rise4fun.com/vcpp, www.compileonline.com, comeaucomputing.com/tryitout

Класс std::thread

- **Методы класса std::thread**
- `std::thread::id get_id() const;` `// ==, !=, <, >, operator<<`
- `bool joinable() const;`
- `native_handle_type native_handle();`
- `static unsigned hardware_concurrency();`
- `void join();`
- `void detach();`
- `void swap(thread& other);`

http://en.cppreference.com/w/cpp/thread/thread

Запуск потока

- В конструктор класса **std::thread** можно передавать объект любого типа, допускающие вызов (Callable):
  - функцию возвращающую значение типа void
  - объект-функцию (function object)
  - лямбда-выражение (lambda expression)

Использование объекта-функции

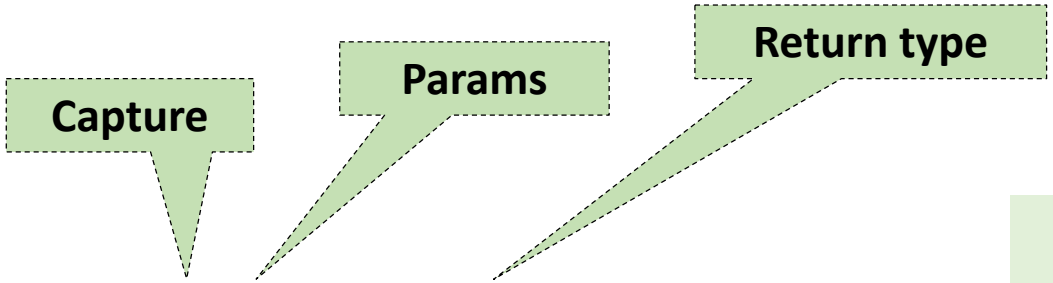
```
#include <iostream>
#include <thread>

class background_task {
public:
    void operator()() const
    {
        std::cout << "Hello, Multithreaded World!\n";
    }
};

int main()
{
    background_task bgtask;
    std::thread mythread(bgtask);   // Инициализировали поток объектом-функцией

    // Продолжаем вычисления в главном потоке

    mythread.join();
    return 0;
}
```

Использование лямбда-выражения

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

int main()
{
    std::thread mythread([]() -> void {
        std::cout << "Hello, Multithreaded, World\n";
    });

    // Продолжаем вычисления в главном потоке

    mythread.join();
    return 0;
}
```

Использование лямбда-выражения

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

int main()
{
    std::thread mythread([]() -> void {
        std::cout << "Hello, Multithreaded, World\n";
    });

    // Продолжаем вычисления в главном потоке

    mythread.join();
    return 0;
}
```



Capture – определяет какие символы (объекты) будут видны в теле лямбда-функции

- **[a, &b]** – *a* захвачена по значению, *b* захвачена по ссылке
- **[this]** захватывает указатель *this* по значению
- **[&]** захват всех символов по ссылке
- **[=]** захват всех символов по значению
- **[]** ничего не захватывает

Передача данных потоку

- Дополнительные аргументы конструктора thread::thread() **копируются** в память потока, где они становятся доступными новому потоку

- **Сценарии передачи аргументов потоку:**

  - ❑ Передача по значению

  - ❑ Передача по ссылке – поток модифицирует переданный объект

  - ❑ Передача в поток только перемещаемых объектов (movable only)

Передача данных потоку

```
void fun(int i, std::string const& s)
{
    std::cout << "1: i = " << i << "; s = " << s << "\n";
}

int main()
{
    const char *s = "Hello";
    int i = 3;

    std::thread t(fun, i, s);  // Поток вызывает fun(i, s)

    t.join();
    return 0;
}
```

- s преобразуется в std::string const& уже в контексте нового потока

Передача данных потоку по ссылке

```
void load_htmldoc(htmldoc& doc) {
    doc.setContent("Page1");
}

void process_htmldoc(htmldoc& doc) {
    std::cout << "DOC: " << doc.getContent() << "\n";
}

int main()
{
    htmldoc doc("DefaultPage");

    std::thread t(load_htmldoc, std::ref(doc));

    t.join();
    process_htmldoc(doc);
    return 0;
}
```

Передача данных потоку

```
void fun(vector<double>& v) {
    cout << "v[0] = " << v[0] << std::endl;
}

struct F {
    vector<double>& v;
    F(vector<double>& vv): v{vv} {};
    void operator()() {
        cout << "v[0] = " << v[0] << std::endl;
    }
};

int main()
{
    vector<double> v1{1, 2, 3, 4};
    vector<double> v2{10, 20, 30, 40};

    thread t1{fun, ref(v1)}; // fun(v1) выполняется в отдельном потоке
    thread t2{F{v2}}; // F(v2)() выполняется в отдельном потоке

    t1.join(); t2.join();
    return 0;
}
```

Передача потоку только перемещаемых объектов (move only)

```
void fun(std::unique_ptr<std::string> sptr)
{
    std::cout << "1: *sptr = " << *sptr << "\n";
    std::cout << "1: sptr.get() = " << sptr.get() << "\n";
}

int main()
{
    // Перемещение s
    std::unique_ptr<std::string> sptr(new std::string("Big object"));
    std::cout << "0: sptr.get() = " << sptr.get() << "\n";
    std::thread t(fun, std::move(sptr));  // Передали владение sptr потоку
    std::cout << "0: sptr.get() = " << sptr.get() << "\n";
    t.join();
    return 0;
}
```

```
0: sptr.get() = 0x1a25010
0: sptr.get() = 0
1: *sptr = Big object
1: sptr.get() = 0x1a25010
```

Возврат значения из потока

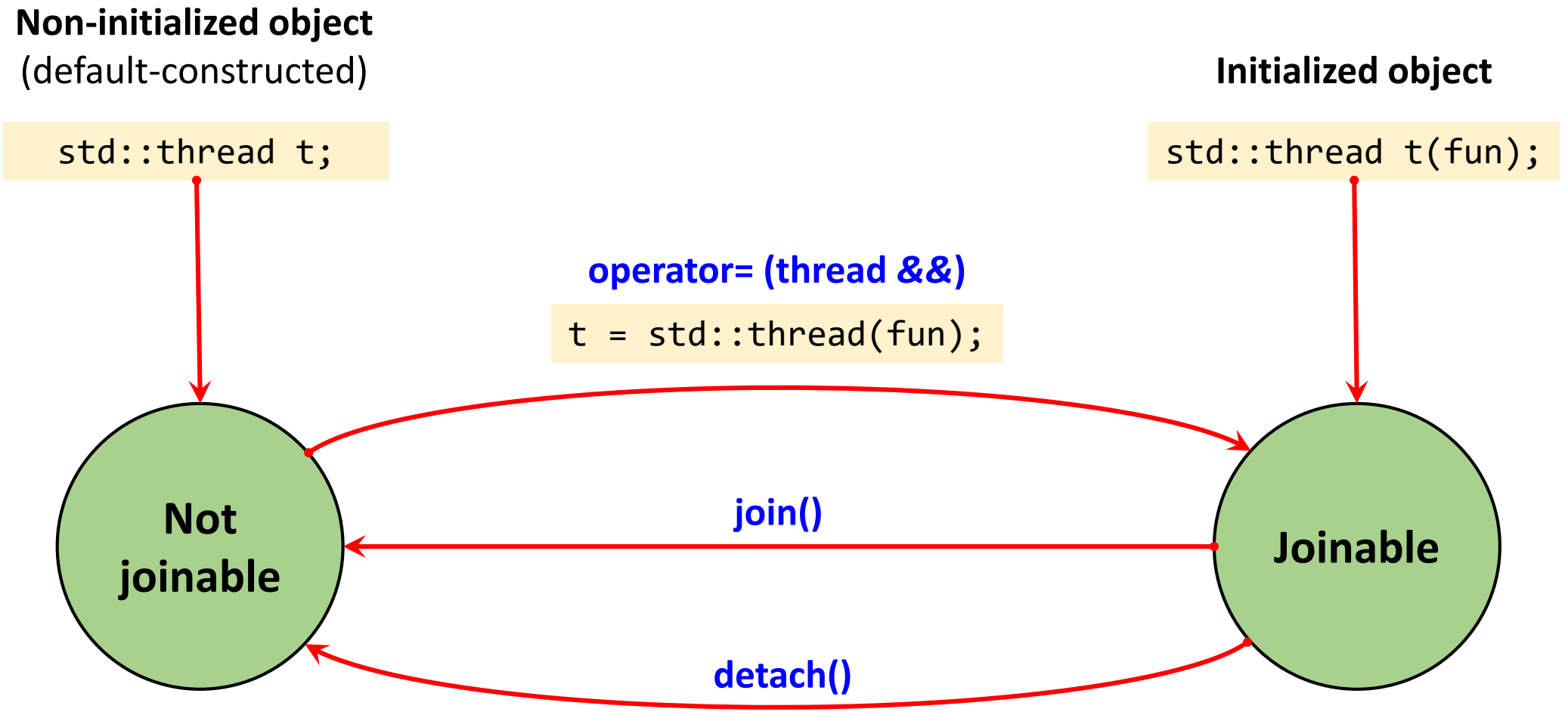
```
void fun(const vector<double>& v, double* res) {
    // ...
    *res = sum;
}

class F {
public:
    F(const vector<double>& vv, double* res): v{vv}, result{res} {};
    void operator()() {
        // ...
        *result = sum;
    }

private:
    const vector<double>& v;
    double* result;
};

int main()
{
    vector<double> v1{1, 2, 3, 4};
    vector<double> v2{10, 20, 30, 40};
    double res1, res2;

    thread t1{fun, cref(v1), &res1}; // fun(v1) executes in a separate thread
    thread t2{F{v2, &res2}}; // F(v2, &res2)() executes in a separate thread

    t1.join();  t2.join();
    return 0;
}
```

Подсоединяемые (joinable) и отсоединённые (detached) потоки

- **Подсоединяемым поток (joinable thread)** – это поток, завершение которого можно дождаться вызвав метод thread::join()
- **Join** – объединить потоки управления (control flows)
- При обращении к методу thread::join() выполнение вызывающего потока блокируется
- При запуске нового потока он по умолчанию является подсоединяемым

- После запуска потока можно изменить его тип на отсоединенный (detached)
- **Отсоединенный поток (detached)** – поток, у которого разорвана связь с исходным объектом std::thread
- Дождаться завершения отсоединено потока невозможно (“живет своей жизнью”)

Подсоединяемые (joinable) и отсоединённые (detached) потоки



Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread()
{
    std::thread mythread(handler);  // Подсоединяемый поток (joinable)
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Вызывается деструктор объекта mythread
    // Объект mythread подсоединяемый (joinable) => деструктор вызывает std::terminate
}

int main()
{
    spawn_thread();
    return 0;
}
```

Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread()
{
    std::thread mythread(handler);  // Подсоединяемый поток (joinable)
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Вызывается деструктор объекта mythread
    // Объект mythread подсоединяемый (joinable) => деструктор вызывает std::terminate
}

int main()
{
    spawn_thread();
    return 0;
}
```

```
thread::~thread()
{
    if (joinable())
        std::terminate();
}
```

Программа завершается с ошибкой
“terminate called without an active exception”

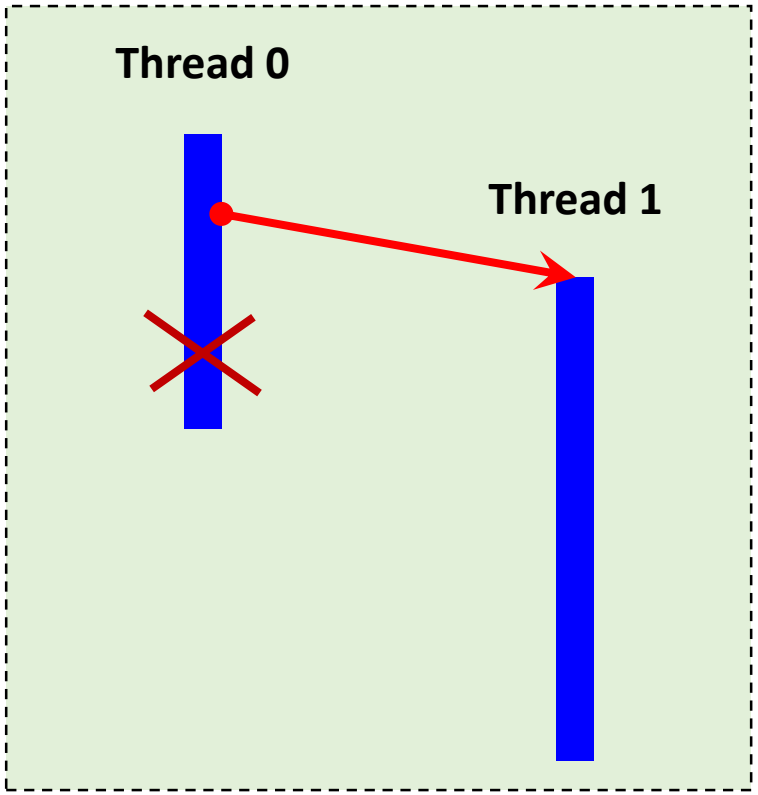
Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread_detached()
{
    std::thread mythread(handler);  // Запустили подсоединяемый поток (joinable)
    mythread.detach();  // Отсоединили поток
    std::this_thread::sleep_for(std::chrono::seconds(2));
    // Вызывается деструктор объекта mythread
}

int main()
{
    spawn_thread_detached();

    return 0; // Вызывается std::exit(0), завершаются все потоки процесса
}
```

Выполнение отсоединенного потока mythread будет прервано при завершении главного потока (при вызове std::exit())

Подсоединяемые (joinable) и отсоединённые (detached) потоки

- Как сделать так, чтобы отсоединенный поток (detached) продолжил свое выполнение после завершения главного потока?



Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread_detached()
{
    std::thread mythread(handler);  // Подсоединяемый поток (joinable)
    mythread.detach(); // Отсоеденили поток
    std::this_thread::sleep_for(std::chrono::seconds(2));
    // Вызывается деструктор объекта mythread
}

int main()
{
    spawn_thread_detached();

    pthread_exit(NULL); // Завершается только главный поток (библиотека POSIX Threads)
}
```

Висячие ссылки

```
class handler {
    int& state_;   // Объект класса handler хранит ссылку на данные
public:
    handler(int& state): state_(state) {}
    void operator()() const  {
        for (int i = 0; i < 5; ++i) {
            std::cout << "State = " << state_ << "\n";
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
    }
};

void run() {
    int i = 33;
    handler h(i);     // Инициализируем объект h ссылкой на переменную i  (размещена в стеке потока 0)
    std::thread t(h);
    t.detach();
} // Объект i разрушается, поток t может все еще использовать i (некорректные значения)

int main() {
    run();
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 0;
}
```

Ожидание в случае исключения

```
void handler() {
    for (int i = 0; i < 3; i++)
        std::this_thread::sleep_for(std::chrono::seconds(1));
}

void do_something() {
    throw "Exception";
}

void spawn_thread() {
    std::thread t(handler);
    do_something();
    t.join();
}

int main()
{
    try { spawn_thread(); }
    catch (...) { std::cout << "Exception\n"; }
    return 0;
}
```

- Функция do_something() генерирует исключение
- Происходит раскрутка стека
- При выходе из функции spawn_thread() вызывается деструктор объекта t
- Деструктор thread::~thread() вызывает std::terminate()
- Метод thread::join() не вызван!

Ожидание в случае исключения

```
// ...

void do_something()
{
    throw "Exception";
}

void spawn_thread()
{
    std::thread t(handler);
    try { do_something(); }
    catch (...) {
        t.join();
        throw;
    }
    t.join();
}

int main()
{
    try { spawn_thread(); }
    catch (...) { std::cout << "Exception\n"; }
    return 0;
}
```

- В случае исключения, дожидаемся завершения потока и повторно генерируем исключение
- Можно использовать идиому RAII

Запуск многих потоков

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

void handler(unsigned int id)
{
    std::cout << id << ": Hello, Multithreaded World!\n";
}

int main()
{
    unsigned int nthreads = std::thread::hardware_concurrency();
    std::cout << "Logical processors: " << nthreads << "\n";

    std::vector<std::thread> threads;
    for (size_t i = 0; i < nthreads; ++i) {
        threads.push_back(std::thread(handler, i));
    }
    std::for_each(threads.begin(), threads.end(),
                  std::mem_fn(&std::thread::join));
    return 0;
}
```

Запуск многих потоков

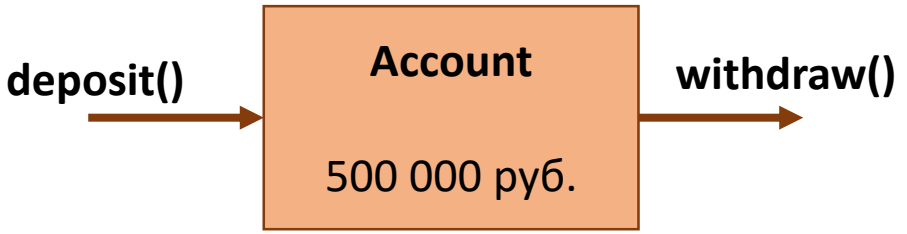
```
$ ./thread_vector
Logical processors: 4
0: Hello, Multithreaded World!
2: Hello, Multithreaded World!
1: Hello, Multithreaded World!
3: Hello, Multithreaded World!

$ ./thread_vector
Logical processors: 4
01: Hello, Multithreaded World!
3: Hello, Multithreaded World!
: Hello, Multithreaded World!
2: Hello, Multithreaded World!
```

- std::cout не гарантирует потокобезопасного поведения

Синхронизация потоков

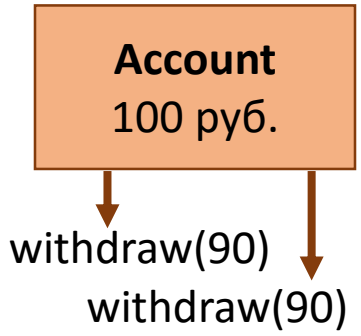
Банковский счет (Account)

```
class Account {
public:
    Account(int balance): balance(balance) { }

    int getBalance() const { return balance; }

    void deposit(int amount) { balance += amount; }

    bool withdraw(int amount)
    {
        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

private:
    int balance;
};
```



Клиент банка – снимает со счета определенную сумму

```
void client(int clientid, Account &account, int amount)
{
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
    bool result = account.withdraw(amount); // Снимаем со счета сумму amount
    if (result)
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
    else
        std::printf("Client %d withdraw %d FAILED\n", clientid, amount);
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
}

int main(int argc, char *argv[])
{
    Account account(100);
    std::thread t1(client, 1, std::ref(account), 90);
    std::thread t2(client, 2, std::ref(account), 90);
    t1.join();  t2.join();
    std::cout << "Account balance " << account.getBalance()
              << "\n";
    return 0; }
```



Ожидаемый результат

```
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

Результаты запусков (Fedora 20, Intel Core i5-3320M – 2 cores + HT)

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

```
$ ./bank_account
Client 1 balance: 100
Client 2 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 withdraw 90 OK
Client 2 balance: -80
Account balance -80
```

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 100
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

```
$ ./bank_account
Client 1 balance: 100
Client 2 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: -80
Client 2 withdraw 90 OK
Client 2 balance: -80
Account balance -80
```

Результаты

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
```

```
$ ./bank_account
Client 1 balance: 100
Client 2 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
```

В чем причина?

```
Client 1 balance: 10
Client 2 balance: 100
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

```
Client 1 withdraw 90 OK
Client 1 balance: -80
Client 2 withdraw 90 OK
Client 2 balance: -80
Account balance -80
```

Отрицательный баланс

```
bool withdraw(int amount)
{
    if (balance >= amount) {
        // Приостановим поток на 1 мс, второй успеет “проскочить” условие
        std::this_thread::sleep_for(std::chrono::milliseconds(1));

        balance -= amount;
        return true;
    }
    return false;
}
```

Отрицательный баланс

```
bool withdraw(int amount)
{
    if (balance >= amount) {
        balance -= amount;
        return true;
    }
    return false;
}
```

Два потока осуществляют конкурентный доступ к полю balance – одновременно читают его и записывают

```
// balance -= amount
Load balance -> %reg0
Load amount -> %reg1
Sub %reg0 %reg1 -> %reg0
Store reg0 -> balance
```

Состояние гонки (Race condition, data race)

- **Состояние гонки (race condition, data race)** – это состояние программы, в которой несколько потоков одновременно конкурируют за доступ к общей структуре данных (для чтения/записи)
- Порядок выполнения потоков заранее не известен – носит случайный характер
- Планировщик ОС динамически распределяет процессорное время учитывая текущую загруженность процессорных ядер, нагрузку (потоки, процессы) создают пользователи, поведение которых носит случайных характер
- Состояние гонки данных (race condition, data race) трудно обнаруживается в программах и воспроизводится в тестах (Гейзенбаг – heisenbug)

Обнаружение состояния гонки (Race condition, data race)

Динамические анализаторы кода

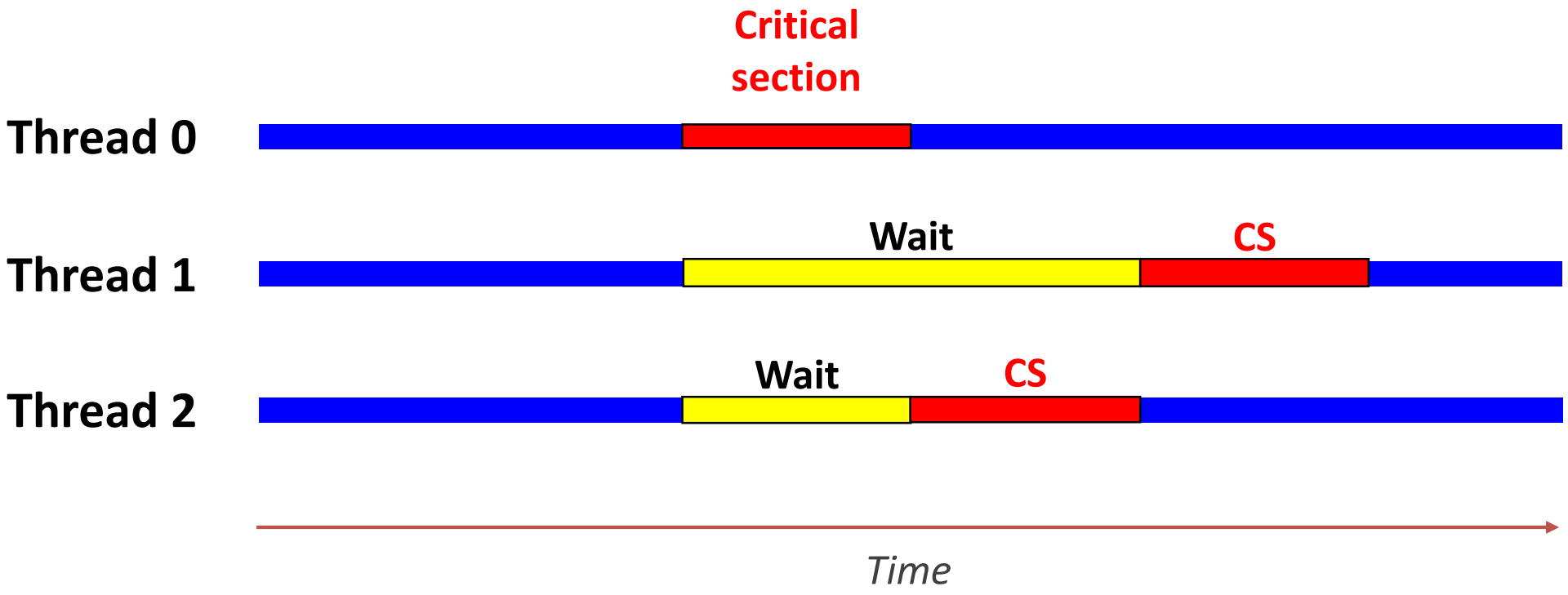
- Valgrind Helgrind, DRD
- ThreadSanitizer – a data race detector for C/C++ and Go (gcc 4.8, clang)
- Intel Thread Checker
- Oracle Studio Thread Analyzer
- Java ThreadSanitizer
- Java Chord

Статические анализаторы кода

- PVS-Studio (Viva64)

Понятие критической секции (Critical section)

- **Критическая секция** (Critical section) — это участок исполняемого кода, который в любой момент времени выполняется только одним потоком



Банковский счет (версия 1 – исходная)

```
class Account {
public:
    Account(int balance): balance(balance) { }
    int getBalance() const { return balance; }
    void deposit(int amount) { balance += amount; }

    bool withdraw(int amount)
    {
        if (balance >= amount) {
            balance -= amount;    // Data race!
            return true;
        }
        return false;
    }

private:
    int balance;
};
```

Банковский счет (версия 2 – mutex)

```
class Account {
public:
    Account(int balance): balance(balance) { }
    int getBalance() const {
        m.lock();
        int val = balance;  // Критическая секция
        m.unlock();
        return val;
    }
    bool withdraw(int amount) {
        m.lock();
        if (balance >= amount) {
            balance -= amount; // Критическая секция
            m.unlock();
            return true;
        }
        m.unlock();
        return false;
    }
private:
    int balance;
    mutable std::mutex m;
};
```

Банковский счет (версия 2 – mutex + lock_guard)

```
class Account {
    // ...
    int getBalance() const {
        std::lock_guard<std::mutex> lock(m);
        return balance;
    }

    bool withdraw(int amount) {
        std::lock_guard<std::mutex> lock(m);
        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

private:
    int balance;
    mutable std::mutex m;
};
```

Результаты

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 100
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

- В функции client() потоки одновременно обращаются к методу Account::getBalance(), затем вызывают метод Account::withdraw()
- Код функции client() должен выполняться как неделимая операция – между вызовами Account::getBalance() и Account::withdraw() другие потоки не должны изменять состояния счета

Банковский счет (версия 3 – recursive_mutex)

```
void client(int clientid, Account &account, int amount)
{
    std::lock_guard<std::recursive_mutex> lock(account.getMutex());
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
    bool result = account.withdraw(amount);
    if (result) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
    } else {
        std::printf("Client %d withdraw %d FAILED\n", clientid, amount);
    }
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
}
```

Критическая секция

- Функция client – критическая секция

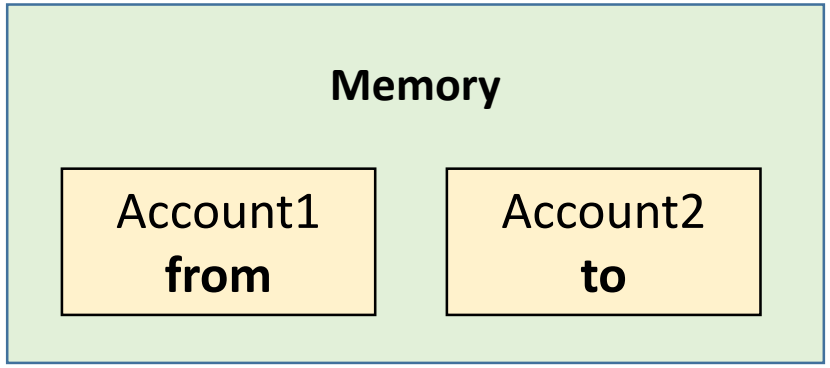
Банковский счет (версия 3 – recursive_mutex)

```
class Account {
    // ...
    int getBalance() const {
        std::lock_guard<std::recursive_mutex> lock(m);
        return balance;
    }

    bool withdraw(int amount) {
        std::lock_guard<std::recursive_mutex> lock(m);
        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    std::recursive_mutex& getMutex() { return m; }

private:
    int balance;
    mutable std::recursive_mutex m;
};
```

Результаты

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

Многопоточный перевод денег между счетами



Thread 0

```
void transfer(
  int clientid,
  Account& from,
  Account& to,
  int amount
)
```

Thread 1

```
void transfer(
  int clientid,
  Account& from,
  Account& to,
  int amount
)
```

Перевод денег между счетами

```
void transfer(int clientid, Account& from, Account& to, int amount)
{
    std::unique_lock<std::mutex> lock_from(from.getLock());
    // 1. Снимаем
    if (from.withdraw(amount)) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);

        std::unique_lock<std::mutex> lock_to(to.getLock());
        to.deposit(amount);   // 2. Зачисляем
        lock_to.unlock();
        std::printf("Client %d deposit %d OK\n", clientid, amount);

        lock_from.unlock();
    } else {
        lock_from.unlock();
        std::printf("Client %d withdraw %d ERROR\n", clientid, amount);
    }
}
```

Захват счета from

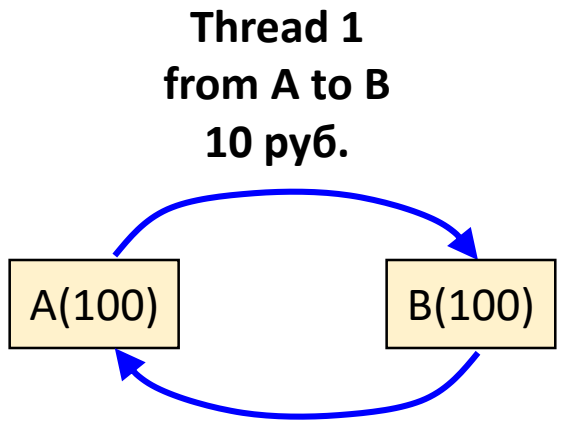
Захват счета to

Перевод денег между счетами

```
class Account {
    // ...
    int getBalance() const { return balance; }
    void deposit(int amount) { balance += amount; }
    bool withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    std::unique_lock<std::mutex> getLock()
    {
        std::unique_lock<std::mutex> lock(m);
        return lock;   // Вызывается перемещающий конструктор unique_lock
    }

private:
    int balance;
    std::mutex m;
};
```

Перевод денег между счетами

```
int main(int argc, char *argv[])
{
    Account a(100);
    Account b(100);
    std::thread t1(transfer, 1, std::ref(a), std::ref(b), 10);
    std::thread t2(transfer, 2, std::ref(b), std::ref(a), 20);
    t1.join();
    t2.join();

    // Assert: a=110, b=90
    std::cout << "A balance: " << a.getBalance() << "\n";
    std::cout << "B balance: " << b.getBalance() << "\n";
    return EXIT_SUCCESS;
}
```




```
./bank_account
Client 2 withdraw 20 OK
Client 1 withdraw 10 OK
... Program hangs
```

Взаимная блокировка (Deadlock)

- **Взаимная блокировка** (Deadlock) – несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими потоками

Шаг	Thread 1	Thread 2
1	Захватил ресурс A (mutex)	Захватил ресурс B (mutex)
2	Пытается захватить ресурс B – ***ожидает*** его освобождение потоком 1	Пытается захватить ресурс A – ***ожидает*** его освобождение потоком 0

```
a.lock()
b.lock()  // ∞ ожидание
// Do something
b.unlock()
a.unlock()
```

```
b.lock()
a.lock()  // ∞ ожидание
// Do something
a.unlock()
b.unlock()
```

Взаимная блокировка (Deadlock)

Перевод денег (версия 2)

```
void transfer(int clientid, Account& from, Account& to, int amount)
{
    if (from.withdraw(amount)) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
        to.deposit(amount);
        std::printf("Client %d deposit %d OK\n", clientid, amount);
    } else {
        std::printf("Client %d withdraw %d ERROR\n", clientid, amount);
    }
}

void client(int clientid, Account& from, Account& to, int amount)
{
    if (&from > &to) {
        std::unique_lock<std::mutex> lock_from(from.getLock());
        std::unique_lock<std::mutex> lock_to(to.getLock());
    } else {
        std::unique_lock<std::mutex> lock_to(to.getLock());
        std::unique_lock<std::mutex> lock_from(from.getLock());
    }
    transfer(clientid, from, to, amount);
}
```

Вводим нумерацию ресурсов (иерархия)

```
./bank_account
Client 1 withdraw 10 OK
Client 1 deposit 10 OK
Client 2 withdraw 20 OK
Client 2 deposit 20 OK
A balance: 110
B balance: 90
```

Перевод денег (версия 3): активное ожидание + sleep

```
class Account {
public:
    // ...
    std::unique_lock<std::mutex> getDeferLock()
    {
        // Строим lock, но не захватываем мьютекс
        std::unique_lock<std::mutex> lock(m, std::defer_lock);
        return lock;
    }
    // ...
};
```

Перевод денег (версия 3): активное ожидание + sleep

```
void client(int clientid, Account& from, Account& to, int amount)
{
    bool success = false;
    std::unique_lock<std::mutex> lock_from(from.getDeferLock());
    std::unique_lock<std::mutex> lock_to(to.getDeferLock());
    while (true) {
        if (lock_from.try_lock()) {
            if (lock_to.try_lock()) {
                transfer(clientid, from, to, amount);
                lock_to.unlock();
                success = true;
            }
            lock_from.unlock();
        }
        if (success)
            break;
        std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 100));
    }
}
```

Перевод денег (версия 4): std::lock

```
void transfer(int clientid, Account& from, Account& to, int amount)
{
    std::unique_lock<std::mutex> lock_from(from.getDeferLock());
    std::unique_lock<std::mutex> lock_to(to.getDeferLock());
    std::lock(lock_from, lock_to);

    if (from.withdraw(amount)) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
        to.deposit(amount);
        std::printf("Client %d deposit %d OK\n", clientid, amount);
    } else {
        std::printf("Client %d withdraw %d ERROR\n", clientid, amount);
    }
}
```

Домашнее чтение

- Bjarne Stroustrup**. A Tour of C++.** The C++ In-Depth Series, Pearson Education, 2013
  Chapter 13. Concurrency

- Уильямс Э. **Параллельное программирование на C++ в действии. Практика разработки многопоточных программ**. - М.: ДМК Пресс, 2012
  Глава 2. Управление потоками

Конкурентный доступ к разделяемой структуре данных

```
void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {
        counter++;
    }
}

int main()
{
    int counter = 0;

    std::thread t1(handler, std::ref(counter));
    std::thread t2(handler, std::ref(counter));
    t1.join();
    t2.join();

    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

Конкурентный доступ к разделяемой структуре данных

```
void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {
        counter++;                          // Data race
    }
}

int main()
{
    int counter = 0;

    std::thread t1(handler, std::ref(counter));
    std::thread t2(handler, std::ref(counter));
    t1.join();
    t2.join();

    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

- **Ожидаемое значение 20 000**
- **Результат запусков**
  - Counter = 19747
  - Counter = 19873
  - Counter = 19813
  - …

Организация критической секции

```
int lock = 0;

void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {

        do {                        // Ожидаем освобождение блокировки
            if (lock == 0) {
                lock = 1;           // Захватываем блокировку
                break;
            }
        } while (1)

         counter++;

         lock = 0;                  // Освобождаем блокировку
    }
}
```

Intel 64 atomic operations (Intel ASDM, Vol.3, Ch.8)

Guaranteed atomic operations

- Reading/writing a byte
- Reading/writing a word aligned on a 16-bit boundary
- Reading/writing a doubleword aligned on a 32-bit boundary
- Reading/writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus
- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

```
int flag;
// ...
flag = 1
```

ISO C/C++ scalar types alignment (int, char, …)
Linux ABI // http://www.x86-64.org/documentation/abi.pdf

Locked atomic operations (locking system bus, prefix #LOCK)

- The bit test and modify instructions: BTS, BTR, and BTC
- The exchange instructions: XADD, CMPXCHG, and CMPXCHG8B
- The LOCK prefix is automatically assumed for XCHG instruction
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR

```
// int counter; counter += val
   lock xaddl counter, val
```

Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A: Chapter 8. Multiple-processor management
http://download.intel.com/products/processor/manual/325384.pdf

Cache locking (using #LOCK prefix)

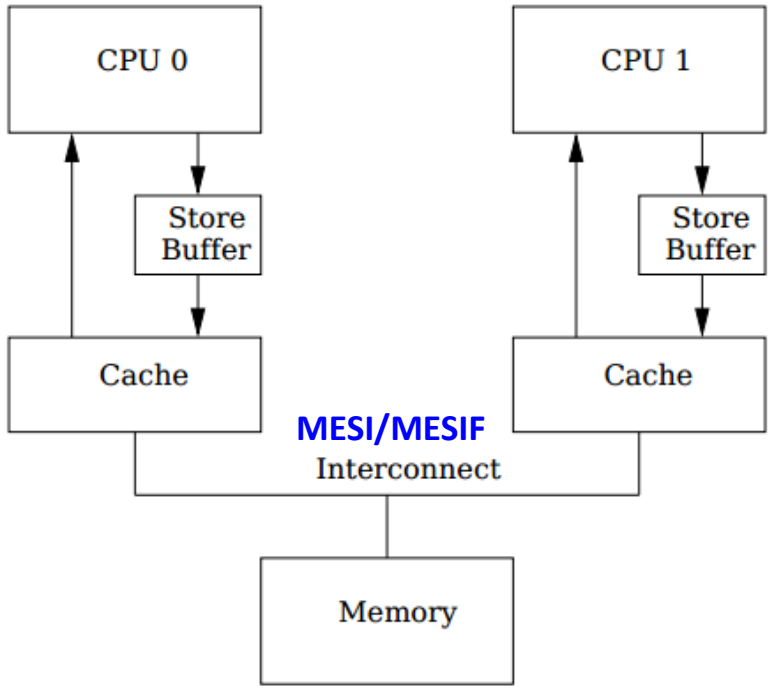
- If <area of memory is cached> then
  // No bus locking!
  Modify data in cache line
- Cache coherency mechanism ensures atomicity (MESI, MESIF)

Atomic test_and_set

```
int lock = 0;

void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {

        while (test_and_set(&lock) == 1) // Ожидаем освобождение блокировки
            ;

        counter++;

        lock = 0;                        // Освобождаем блокировку
    }
}
```

Модель согласованности памяти

- **Операции чтения и записи данных в память могут выполняться в порядке отличном от исходного**
  - ❑ Иерархия кеш-памяти + протоколы поддержания когерентностей кешей (MESI/MESIF)
  - ❑ Store buffers – очереди записи (сокрытие латентности MESI)
  - ❑ Внеочередное выполнение команд



❑ Paul E. McKenney. **Memory Barriers: a Hardware View for Software Hackers** // http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf

Модель согласованности памяти

Type	Alpha	ARMv7	POWER	SPARC PSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y			Y		Y	
Loads reordered after stores	Y	Y	Y			Y		Y	
Stores reordered after stores	Y	Y	Y	Y		Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y	Y					Y	
Atomic reordered with stores	Y	Y	Y	Y				Y	
Dependent loads reordered	Y								
Incoherent Instruction cache pipeline	Y	Y	Y	Y	Y	Y		Y	Y

- Paul E. McKenney. **Memory Barriers: a Hardware View for Software Hackers** // http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf

Memory ordering Intel64 (Core 2 Duo)

Chapter 8.2 [1]: In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles:

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions: writes executed with the CLFLUSH instruction; streaming stores (writes) executed with the non temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes.
- MFENCE instructions cannot pass earlier reads or writes.

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide // http://download.intel.com/products/processor/manual/325384.pdf

Последовательная согласованность (Sequential consistency)

- Операции с памятью (load, store) выполняются в программном порядке (исходном) – их относительный порядок не должен меняться
- Модель системы: нет кеш-памяти, нет буферов записи (store buffers)

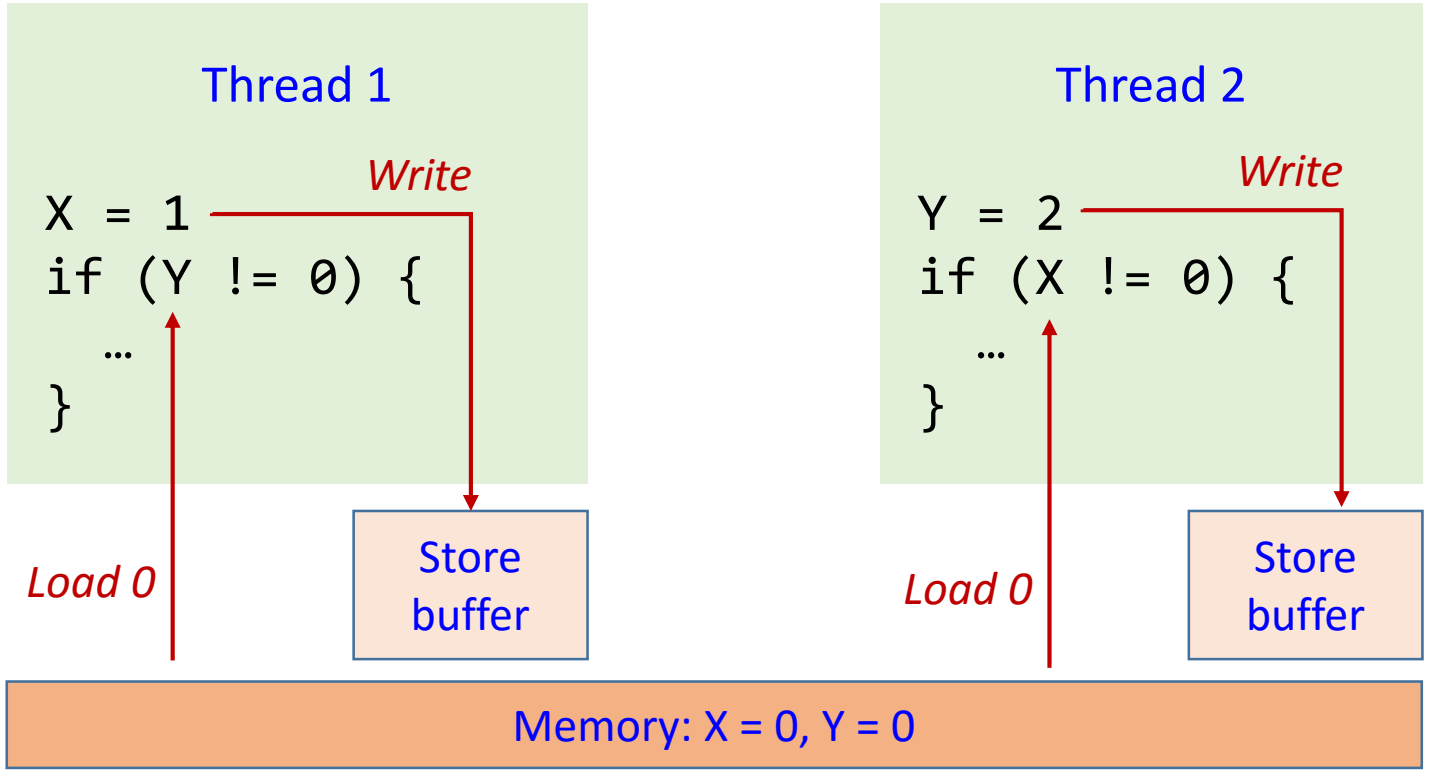
Thread 1	Thread 2
X = 5 Z = Y	C = A B = 2

Разрешены любые сценарии выполнения потоков, но недопустимо менять относительный порядок выполнения X = 5 и Z = Y, а также C = A и B = 2

Проста для понимания, ограничивает потенциальные возможности процессора

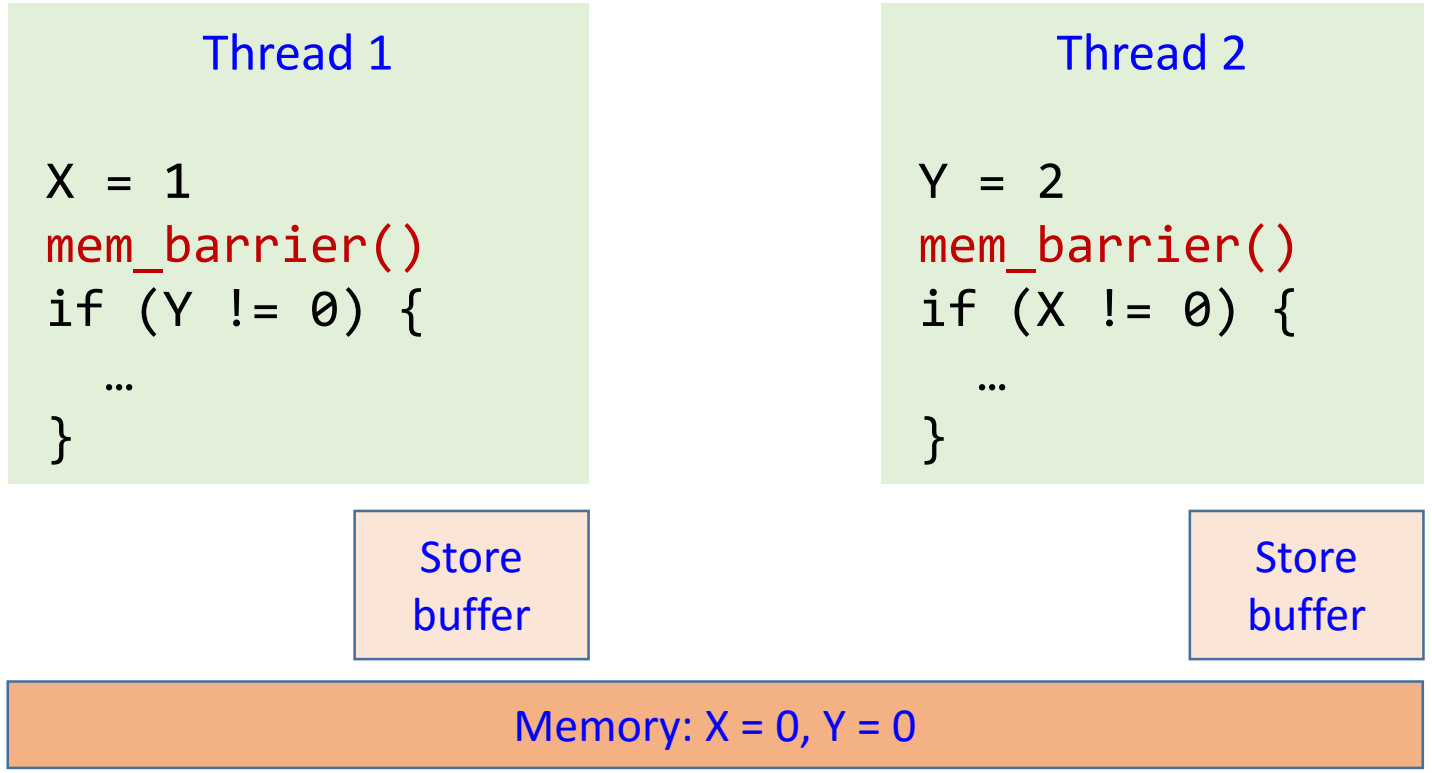
Ослабленные модели согласованности (Relaxed consistency)

- Операции с памятью (load, store) могут выполняются в порядке отличном от исходного
- Модель системы: кеш-память + поддержка когерентности кешей



Барьер памяти (memory barrier)

- **Барьер памяти** – инструкция, которая сбрасывает буфер записи/чтения
- Следующие операции работы с памятью не будут выполнены, пока не завершаться все находящиеся в очереди



Memory barrier

- **Compiler memory barrier** – предотвращает перестановку инструкций компилятором (в ходе оптимизации)

```
/* GNU inline assembler */
asm volatile("" ::: "memory");

/* Intel C++ intrinsic */
__memory_barrier();

/* Microsoft Visual C++ */
_ReadWriteBarrier()
```

- **Hardware memory barrier** – предотвращает перестановку инструкций процессором

```
/* x86, x86_64 */
void _mm_lfence();   /* lfence */
void _mm_sfence();   /* sfence */
void _mm_mfence();   /* mfence */
```

- **GCC: Built-in functions for atomic memory access //** http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Atomic-Builtins.html
- **LLVM Atomic Instructions and Concurrency Guide** // http://llvm.org/docs/Atomics.html
- **Linux kernel memory barriers** // https://www.kernel.org/doc/Documentation/memory-barriers.txt

Memory barrier

```
volatile bool stopflag;
int a, b;

void run() {
    while (!stopflag);
    // Здесь нужен барьер, чтобы чтение stopflag всегда предшествовало обновлению b
    b = a;
}

int main() {
    stopflag = false;
    a = b = 0;
    std::thread mythread(run);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    a = 1;
    // Здесь нужен барьер, чтобы a была видна всем потокам, перед обновлением stopflag,
    stopflag = true;

    mythread.join();
    return 0;
}
```

Атомарные типы

- **Атомарный тип (atomic type)** – это тип данных, который поддерживает атомарное выполнение операций
- std::atomic<bool>, std::atomic<Integral>, std::atomic<T*>
- **Методы класса std::atomic**
  - **bool is_lock_free()** – true, если реализации операций не использует блокировок
  - **T load(memory_order = std::memory_order_seq_cst)** – атомарно загружает и возвращает значение атомарной переменной
  - **void store(T desired, memory_order = std::memory_order_seq_cst)** – атомарно записывает значение в переменную
  - **…**

C++11 Atomic operations library

```
#include <atomic>

std::atomic<bool> stopflag;
int a, b;

void run()  {
    // Атомарное чтение stopflag + гарантия сохранения порядка выполнения операций
    while (!stopflag.load(/* memory_order = std::memory_order_seq_cst) */));
    b = a;
}

int main() {
    stopflag.store(false);
    a = b = 0;
    std::thread mythread(run);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    a = 1;
    stopflag.store(true);
    mythread.join();
    return 0;
}
```

Барьерная синхронизация потоков

- **Барьерная синхронизация (Barrier)** – это примитив синхронизации, который заставляет каждый поток ожидать, пока остальные потоки не достигнут общей точки синхронизации

```
std::atomic<int> gsum;

void thread_routine(int id, int nthreads)
{
    int sum = 0;
    for (int i = 0; i < n; i += nthreads)
        sum += fx(i);
    gsum.fetch_add(sum);

    BARRIER();  // Синхронизируем все потоки, значение gsum дальше требуется всем потокам
    post_process(gsum);
}
```

Барьерная синхронизация потоков

```
void handler(int id, int n, barrier& b) {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 5000));
    print_time("Before barrier");
    b.wait();
    print_time("After barrier");
}

int main() {
    std::vector<std::thread> threads(10);
    barrier b(threads.size());

    for (size_t i = 0; i < threads.size(); ++i) {
        threads[i] = std::thread(handler, i, threads.size(), std::ref(b));
    }

    std::for_each(threads.begin(), threads.end(),
                  std::mem_fn(&std::thread::join));
    return 0;
}
```

Барьерная синхронизация потоков

```
class barrier {
    unsigned int const count;
    std::atomic<unsigned int> spaces;
    std::atomic<unsigned int> generation;

public:
    explicit barrier(unsigned nthreads): count(nthreads), spaces(nthreads),
                                         generation(0) { }

    void wait() {
        unsigned const my_generation = generation;
        if (!--spaces) {
            spaces = count;
            ++generation;
        } else {
            while (generation == my_generation)
                std::this_thread::yield();
        }
    }
};
```

- Maurice Herlihy, Nir Shavit. **The Art of Multiprocessor Programming, Morgan Kaufmann**, 2012, [C. 397] “17. Barriers”
- Эндрюс Г. **Основы многопоточного, параллельного и распределенного программирования**. - М.: Вильямс, 2003, [С. 103] “3.4 Барьерная синхронизация”

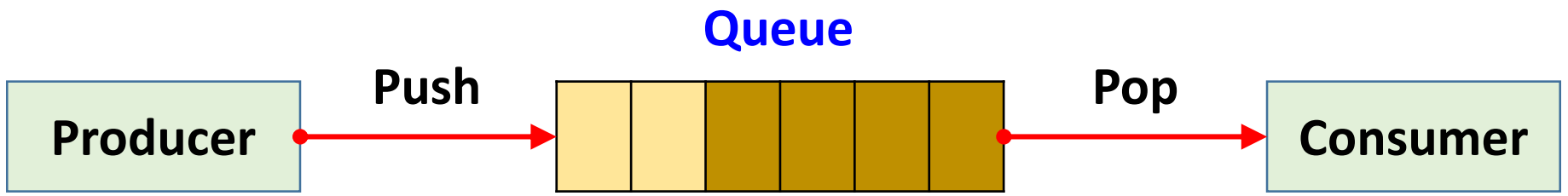
Барьерная синхронизация потоков

```
$./barrier
Thread 139857187747584: Before barrier: Wed Feb 19 11:53:51 2014
Thread 139857179354880: Before barrier: Wed Feb 19 11:53:51 2014
Thread 139857238103808: Before barrier: Wed Feb 19 11:53:51 2014
Thread 139857162569472: Before barrier: Wed Feb 19 11:53:52 2014
Thread 139857170962176: Before barrier: Wed Feb 19 11:53:52 2014
Thread 139857204532992: Before barrier: Wed Feb 19 11:53:52 2014
Thread 139857221318400: Before barrier: Wed Feb 19 11:53:53 2014
Thread 139857212925696: Before barrier: Wed Feb 19 11:53:53 2014
Thread 139857196140288: Before barrier: Wed Feb 19 11:53:54 2014
Thread 139857229711104: Before barrier: Wed Feb 19 11:53:55 2014
Thread 139857229711104: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857212925696: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857179354880: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857238103808: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857170962176: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857221318400: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857204532992: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857162569472: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857196140288: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857187747584: After barrier: Wed Feb 19 11:53:55 2014
```

Условная синхронизация (Condition synchronization)

Producer–Consumer Problem

- **Производитель-потребитель** (Producer–consumer, bounded-buffer problem) – классическая задача синхронизации многопоточных программ
- Производитель помещает данные в очередь фиксированного размера, а потребитель забирает данные из нее
- Производитель не может помещать данные в заполненную очередь
- Потребитель не может забирать данные из пустой очереди



Условная синхронизация

- **Решение 1** – использовать разделяемую переменную-флаг, значение которой периодически опрашивается (поток будет непрерывно нагружать процессор)
- **Решение 2** – это решение 1 + периодически отправлять поток “спать”, второй поток успеет сменить флаг

```
bool flag;
std::mutex m;

void wait_for_flag() {
    std::unique_lock<std::mutex> l(m);
    while (!flag) {
        l.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));  // Как выбирать 100? 200?
        l.lock();
    }
}
```

Условные переменные (Condition variables)

- **Условная переменная (Condition variable)** – это примитив синхронизации, позволяющий организовать ожидания наступления определенного события
- Условная переменная работает в паре с мьютексом
- #include <condition_variable>
- class condition_variable;
- **Notification**
  - void **notify_one**() – снимает блокировку с одного потока ожидающего на *this
  - void **notify_all**() – снимает блокировки со всех потоков ожидающих на *this
- **Wating**
  - void **wait**(std::unique_lock<**std::mutex**>& lock, Predicate pred) – ожидает пока предикат не примет значение истина

Решение 1 – очередь на базе std::queue<T>

```
#include <condition_variable>

std::queue<int> data_queue;
std::mutex data_mutex;
std::condition_variable data_condvar;

void producer()
{
    for (int i = 0; i < 10; ++i) {
        std::lock_guard<std::mutex> lg(data_mutex);  // Защищаем доступ к очереди
        int val = (i < 9) ? i + 1 : -1;
        data_queue.push(val);
        data_condvar.notify_one(); // Извещаем заблокированный поток о новых данных

    }  // unlock mutex
}
```

Решение 1 – очередь на базе std::queue<T>

```
void consumer()
{
    while (true) {
        std::unique_lock<std::mutex> lock(data_mutex); // Защищаем доступ к очереди
        // Wait:
        // Проверяем условие - если не выполнено, освобождаем lock и ожидаем извещения
        // Получили извещение - захватываем lock и проверяем условие
        // Условие выполнено - захватываем lock и выходим из wait
        data_condvar.wait(lock, []{ return !data_queue.empty(); });

        int val = data_queue.front();
        data_queue.pop();
        lock.unlock();

        std::cout << "Consumer " << val << "\n";
        if (val == -1)
            break;
    }
}
```

Решение 1 – очередь на базе std::queue<T>

```
int main()
{
    std::thread c(consumer);
    producer();
    //std::thread p(producer);
    c.join();
    //p.join();
    return 0;
}
```

Потокобезопасная очередь

```
template <typename T> class threadsafe_queue {
public:
    void put(T val) {
        std::lock_guard<std::mutex> lock(mutex);
        queue.push(val);
        cond.notify_one();   // в очереди появились данные
    }

    T take() {
        std::unique_lock<std::mutex> ulock(mutex);
        //while (queue.empty())
        //    cond.wait(ulock);
        cond.wait(ulock, [this]() { return !queue.empty(); });
        T val = queue.front();
        queue.pop();
        return val;
    }

private:
    std::queue<T> queue;
    std::mutex mutex;
    std::condition_variable cond;
};
```

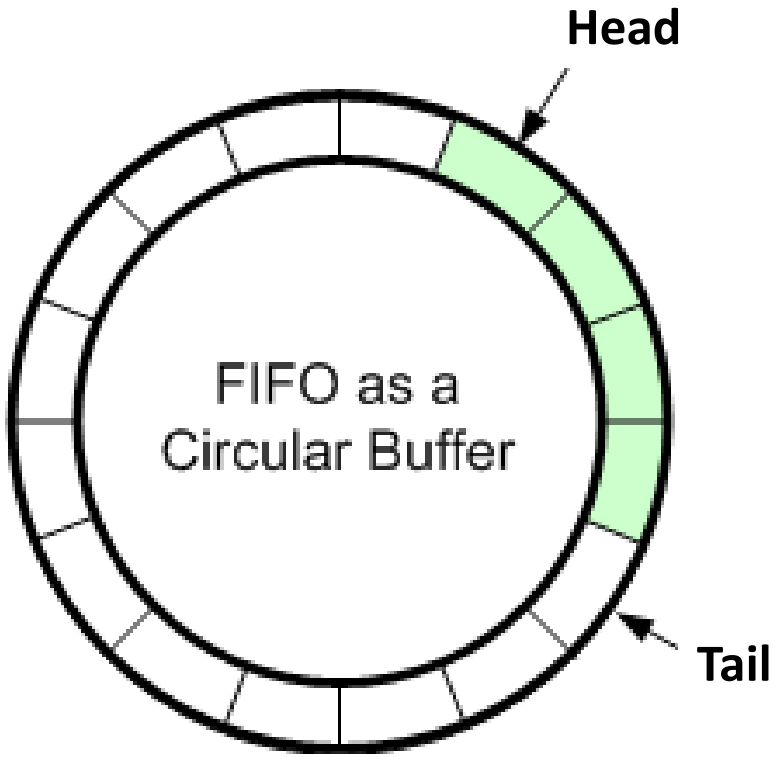
Потокобезопасная очередь

```
template <typename T>
void producer(int id, threadsafe_queue<T>& queue)
{
    for (size_t i = 0; i < 15; ++i) {
        queue.put(id + 100 + i);
        std::cout << "Producer " << id << " put " << id + 100 + i << "\n";
    }
}

template <typename T>
void consumer(int id, threadsafe_queue<T>& queue)
{
    for (size_t i = 0; i < 10; ++i) {
        T val = queue.take();
        std::cout << "Consumer " << id << " take " << val << "\n";
    }
}
```

Потокобезопасная очередь

```
int main()
{
    threadsafe_queue<int> queue;

    int nconsumers = 3;
    std::vector<std::thread> consumers;
    for (int i = 0; i < nconsumers; ++i)
        consumers.push_back(std::thread(consumer<int>, i, std::ref(queue)));

    int nproducers = 2;
    std::vector<std::thread> producers;
    for (int i = 0; i < nproducers; ++i)
        producers.push_back(std::thread(producer<int>, i, std::ref(queue)));

    for (std::thread& t : consumers)
        t.join();
    for (std::thread& t : producers)
        t.join();
    return 0;
}
```

Кольцевой буфер (Circular buffer, Bounded buffer)



Кольцевой буфер (Circular buffer, Bounded buffer)

```
template <typename T> class ringbuffer {
    T* buffer;
    int capacity;
    int head;
    int tail;
    int count;

    std::mutex mutex;
    std::condition_variable not_full;    // Сообщение - “в буфере есть свободная позиция”
    std::condition_variable not_empty;   // Сообщение - “буфер не пуст”

public:
    ringbuffer(int capacity): capacity(capacity), head(0), tail(0), count(0) {
        buffer = new T[capacity];
    }

    ~ringbuffer() {
        delete[] buffer;
    }
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
void put(T value)
{
    std::unique_lock<std::mutex> ulock(mutex);
    // Wait for free positions in the buffer
    not_full.wait(ulock, [this](){ return count != capacity; });

    buffer[tail] = value;
    tail = (tail + 1) % capacity;
    ++count;

    // Buffer has elems, notify waiting thread
    not_empty.notify_one();
}
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
    T take()
    {
        std::unique_lock<std::mutex> ulock(mutex);
        // Wait for elem in the buffer
        not_empty.wait(ulock, [this](){ return count != 0; });

        T value = buffer[head];
        head = (head + 1) % capacity;
        --count;
        // Buffer has free position now, notify waiting thread
        not_full.notify_one();
        return value;
    }

}; // class ringbuffer
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
template <typename T> void producer(int id, ringbuffer<T>& buf)
{
    for (size_t i = 0; i < 15; ++i) {
        buf.put(id + 100 + i);
        std::cout << "Producer " << id << " put " << id + 100 + i << "\n";
    }
}

template <typename T> void consumer(int id, ringbuffer<T>& buf)
{
    for (size_t i = 0; i < 10; ++i) {
        T val = buf.take();
        std::cout << "Consumer " << id << " take " << val << "\n";
    }
}
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
int main()
{
    ringbuffer<int> buffer(100);

    int nconsumers = 3;
    std::vector<std::thread> consumers;
    for (int i = 0; i < nconsumers; ++i)
        consumers.push_back(std::thread(consumer<int>, i, std::ref(buffer)));

    int nproducers = 2;
    std::vector<std::thread> producers;
    for (int i = 0; i < nproducers; ++i)
        producers.push_back(std::thread(producer<int>, i, std::ref(buffer)));

    for (std::thread& t : consumers)
        t.join();
    for (std::thread& t : producers)
        t.join();
    return 0;
}
```

Кольцевой буфер – активное ожидание (Busy wait)

```
void put(T value)
{
    std::unique_lock<std::mutex> ulock(mutex, std::defer_lock);
    // Wait for free positions in the buffer
    while (true) {
        ulock.lock();
        if (count != capacity) {
            buffer[tail] = value;
            tail = (tail + 1) % capacity;
            ++count;
            break;
        }
        ulock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
```