Программирование без блокировок

Герб Саттер:

«Если Вы думаете, что программирование без блокировок это просто, значит или Вы – один из тех 50, которые умеют это делать, или же используете атомарные инструкции недостаточно аккуратно»

Цели многопоточного программирования **без блокировок**:

- Исключение накладных расходов на блокировку потока (переключение контекста) => (в большинстве случаях) повышение эффективности
- Без использования мьютексов (т.е. без явной организации критической секции) исключить неопределенное поведение при работе с разделяемыми данными
- Решение проблемы выбора размера критической секции (простота или масштабируемость)
- В некоторых случаях решение проблем, связанными с взаимными блокировками (dead locks) в случае необдуманного использования мьютексов (порядок блокировки разных мьютексов в разных потоках)

Виды алгоритмов, свободных от блокировок:

- без ожиданий (wait-free) – “**никто никогда не ждет**”. Каждая операция завершается за N шагов без каких-либо условий. Гарантии:
  - максимум пропускной способности системы
  - отсутствие голодания

- без блокировок (lock-free) – “**всегда какой-то из потоков работает**”. Гарантии:
  - максимум пропускной способности системы
  - один из потоков может постоянно ожидать

- без остановок (obstruction-free) – “**поток работает, если нет конфликтов**”. При возникновении конфликта за ограниченное число шагов один поток достигает результата при условии, что конфликтующие потоки остановлены
  - все потоки не блокируются из-за проблем с другими потоками
  - не гарантируется прогресс, если одновременно работают два и больше потоков

MEMORY MODEL
C++11

М.Полубенцева

Б. Страуструп – multithreading in C++11

«С точки зрения **параллельного** программирования ключевые новшества C++11 состоят в:

- **организации/модели памяти**,
- портируемости многопоточных программ»

До стандарта C++11

Порядок вычисления выражения компилятором? Точки следования?

Примеры точек следования:

```
int f(int, const char*);
const char* fStr();

int main()
{
        int n = <выражение>, m = <выражение>;
        int res = f(n+m, fStr()); //???
}
```

Зачем нужна модель памяти

- модель памяти регламентирует **разрешенное** поведение многопоточных программ при обращении потоков к **разделяемым данным**
- программист (хороший), управляя моделью памяти, получает возможность создавать не только **безопасные**, но и **эффективные** параллельные программы

Модель памяти

Программист должен представлять:

- размещение составляющих программы в памяти (код, локальные данные, статические данные, динамические данные, thread local данные) – **структурный аспект модели памяти**
- для создания параллельных программ добавляется обеспечение синхронизации доступа к разделяемым данным - **параллельный аспект модели памяти,**
  ! иначе **неопределенное поведение**!

При обращении к разделяемым данным в многопоточной программе

Модель памяти == контракт:

- программист обеспечивает корректную синхронизацию,
- система в целом (компилятор + ОС + процессор + кэш) обеспечивает иллюзию того, что наш код исполняется так, как мы задумали

Правила формируются относительно понятия memory location

memory location – структурный аспект memory model (размещение данных в памяти)

- в C++11 введено понятие **memory location** (до многопоточных приложений было неактуально)
- смысл memory location (то, для чего можно получить адрес):
  - все данные в C++ программе строятся из объектов
  - struct, class – объект, который является контейнером для других подобъектов
  - в стандарте C++ программный объект определяется как «область памяти»
  - программный объект может занимать одну и более memory locations
  - переменные базовых типов (int, char…), указатель занимают **одну memory location** независимо от размера
  - смежные битовые поля (adjacent bit fields) принадлежат одной и той же memory location

Примеры memory location

```
struct Sample{
    int n:
    double d;
    std::string str;
    int x:4:
    unsigned int y:8;
};
```

одна memory location

одна memory location

несколько memory locations

одна memory location

Цель введения в C++11 понятия memory locations?

для формирования правил упорядочения обращения к данных в **многопоточном** приложении

memory location – аспект параллелизма memory model

- Модификация **различными** потоками **различных** memory locations → OK
- Чтение **различными** потоками **одной** memory location → OK
- Чтение/запись одним потоком memory location, модифицируемой другим потоком
  → Data race → Undefined Behavior!!!

Модель памяти C++11:

- - учитывает специфику многопоточных приложений
- - предоставляет программисту **средства** для **принудительного** упорядочения параллельного выполнения кода

Актуальны только при многопоточной организации программы

ПРОБЛЕМЫ КОНКУРЕНТНОГО ПРОГРАММИРОВАНИЯ

М.Полубенцева

Актуальность проблем:

- До поддержки многопоточности (C++11) – НЕ актуальны
- При создании многопоточных приложений:
  – несинхронизированный доступ к данным
  – неатомарные операции чтения/записи данных
  – переупорядочение выполнения кода

Несинхронизированный доступ к данным. Пример:

```
if( value <0 )
{ value = -value;}
```

//используем **только** положительное значение value

Проблемы:

- в однопоточном приложении???
- в многопоточном приложении???

Несинхронизированный доступ к данным. Еще один пример:

std::vector<int> v;	

//поток 1
if(!v.empty())
{
 	
std::cout << v.front() ; //???
…
}
	//поток 2
	…
	
	v.pop();

Несинхронизированный доступ к данным. Замечание:

При использовании средств стандартной библиотеки следует учитывать:

функции стандартной библиотеки не поддерживают (пока) **конкурентные операции чтения и записи разными потоками одних и тех же данных!**

http://libcds.sourceforge.net/

libCDS –**Concurrent Data Structure** - open source C++ библиотека **lock-free** контейнеров и алгоритмов безопасного освобождения памяти (safe memory reclamation).

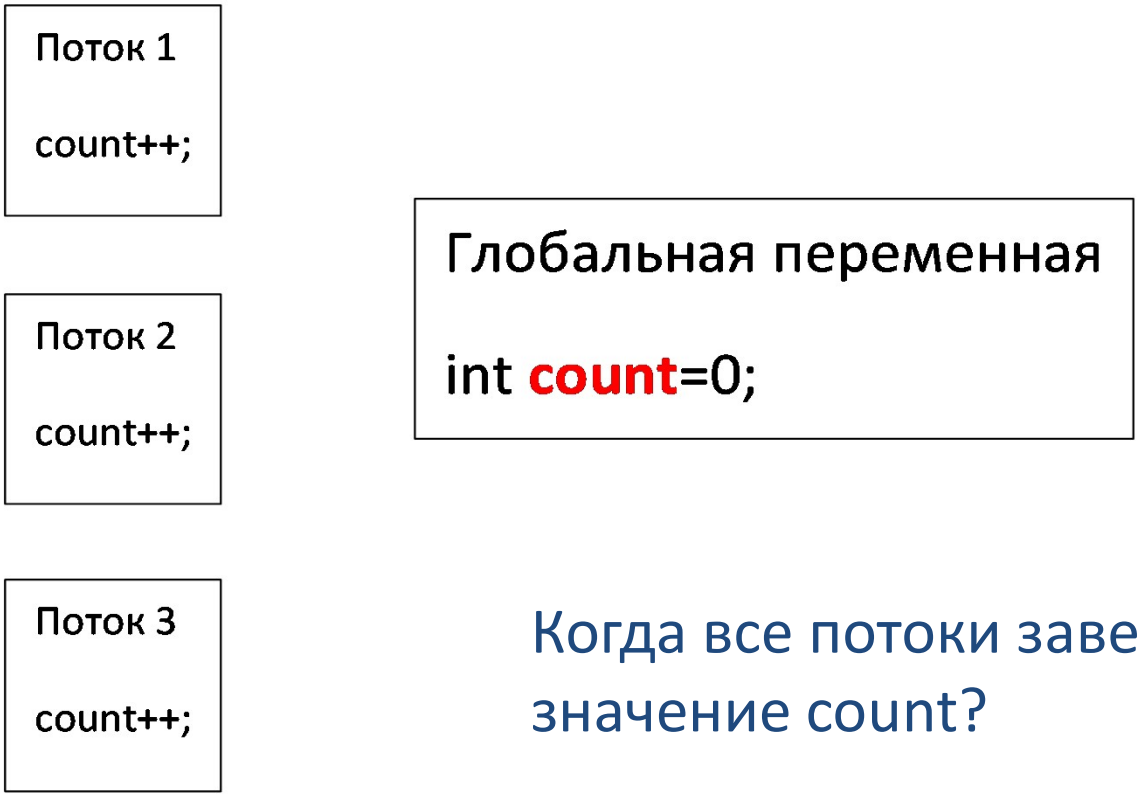
Параллельная STL (Parallel Studio XE)

например,

for_each(exec_policy, …); //**seq, par**, vect

Замечание: некоторые новшества вошли в стандарт C++17

Неатомарные операции модификации данных – «частично» модифицированные данные



Когда все потоки завершатся, значение count?

Детализация неопределенного поведения: count = ???

Поток 1

count++;

Поток 2

count++;

Поток 3

count++;

Глобальная переменная

int **count**=0;

```
count++;
mov eax,dword ptr ds:[смещ]
add eax,1
mov dword ptr ds:[смещ],eax
```

volatile тоже не помогает!

Поток 1:
count++;

Поток 2:
count++;

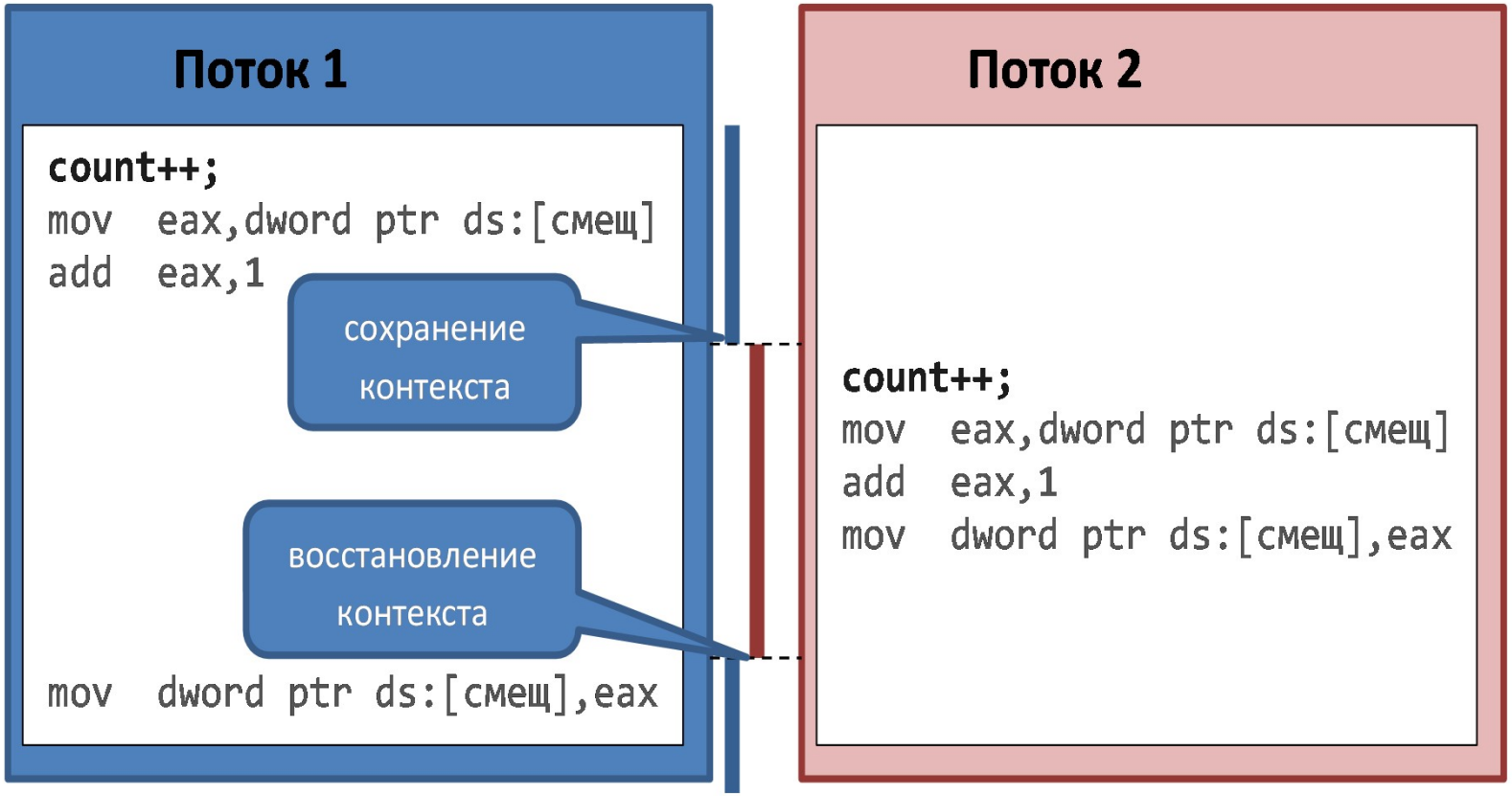
Поток 3:
count++;

Глобальная переменная:
volatile int count=0;

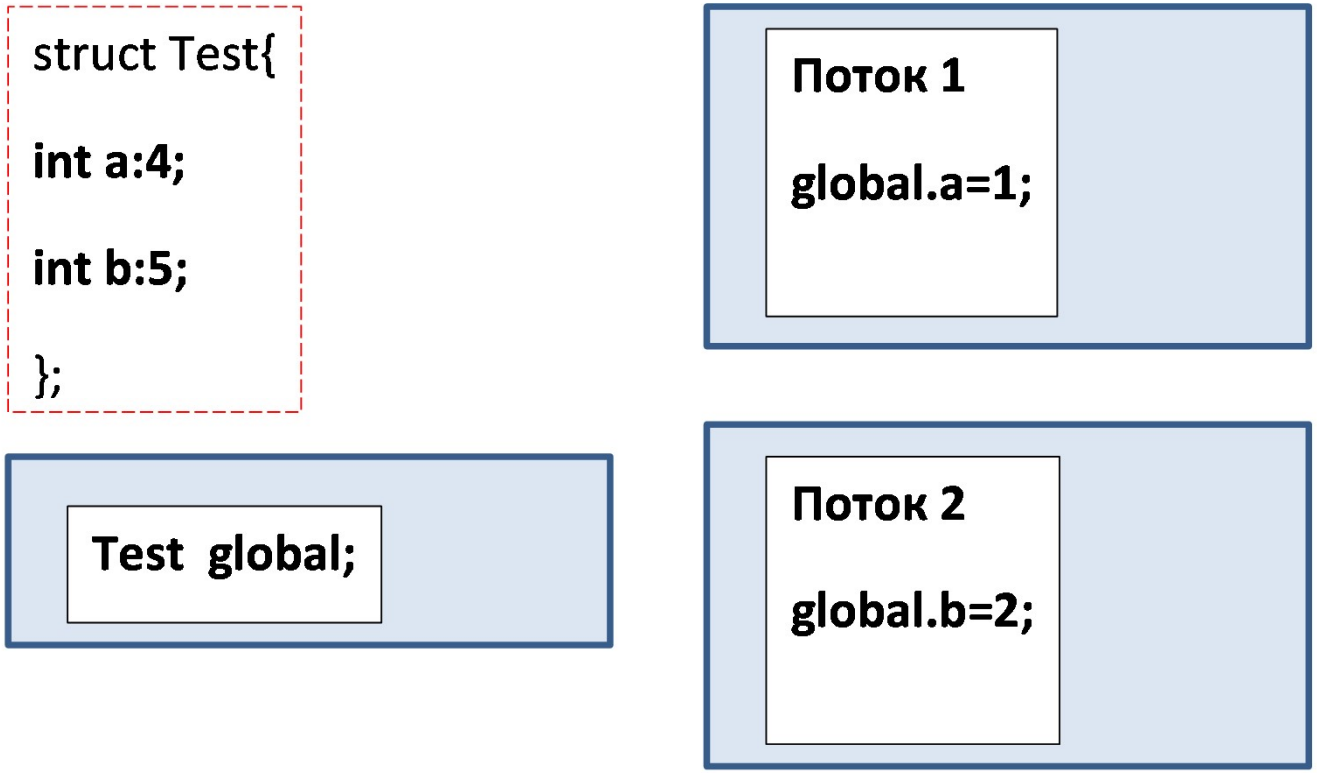
count++;

```
mov eax,dwordptr ds:[смещ]
add eax,1
mov dwordptr ds:[смещ],eax
```

Демонстрация проблемы



Разные данные, принадлежащие одному memory location ???



В общем случае стандарт не дает гарантий даже относительно разных memory locations!

global.**a** = ? global.**b** = ?

```
struct Test{
int a;
int b;
};
```

```
Test  global;
```

Поток 1

```
global.a=1;
```

Поток 2

```
global.b=2;
```

Согласно стандарту компилятор имеет полное право преобразовать код:

```
Test    global; //Глобальная
...
//Поток 1
Test tmp = global;
tmp.a = 5;
global = tmp;
...
//Поток 2
Test tmp = global;
tmp.b = 4;
global = tmp;
```

Переупорядочение кода (reodering)

Эпиграф:

Если Вы думаете, что программа всегда выполняется так и в таком порядке, как Вы ее написали,

Вы ошибаетесь!

Стандарт C++11:

«Реализация может свободно **игнорировать** любое требование международного стандарта, если результирующее поведение программы выглядит так, «**как будто**» это требование было выполнено.

Например, фактическая реализация не обязана вычислять часть выражения,

- которая в дальнейшем не используется,
- и при этом не возникают побочные эффекты»

Следовательно

сгенерированный (для одного и того же высокоуровневого текста на C++) код – это

«черный ящик»,

- содержимое которого может быть разным
- в то время, как **наблюдаемое поведение** должно быть одинаковым (независимо от реализации)

Надежды и реальность:

- Программист надеется на то, что порядок действий будет таким, как он запланировал
- Но! в современных сложно организованных архитектурах **в целях повышения производительности** этот порядок может **не** соблюдаться!

=> в общем случае (без использования средств C++11) соответствия

«расположено перед» == «происходит раньше»

нет!

Кто и зачем меняет наш код:

- **Компилятор** – оптимизация, переупорядочивание кода (reordering), линеаризация циклов, исключает то, что с его точки зрения бесполезно (“мертвый код”). . .
- **Процессор** – предвыборка кода (prefetch), спекулятивное выполнение (speculative execution), вычислительный конвейер . . .
- **Кэши** – могут содержать разные значения для одной и той же ячейки памяти (memory location) => синхронизация кэшей

Предвыборка кода

- — это выдача запросов со стороны процессора в оперативную память для считывания инструкций **заблаговременно**, до того момента как эти инструкции потребуется исполнять.
- В результате этих запросов, инструкции загружаются из памяти в **кэш** => когда инструкции потребуется исполнять, доступ к ним будет осуществляться значительно быстрее, так как задержка при обращении в кэш на порядки меньше, чем при обращении в оперативную память.

Параллелизм на уровне команд (ILP)

- **Спекулятивное выполнение** - опережающее выполнение команд прежде, чем становится известно, что их выполнение необходимо (speculative execution)
- **Вычислительный конвейер** - выполнение нескольких инструкций может частично перекрываться

Пример переупорядоченных компилятором инструкций – взгляд программиста

```
int data;
bool readyFlag = false;
```

Поток, формирующий данные	**Поток, обрабатывающий данные**
data = <значение>; readyFlag = true;	while(!readyFlag){;} //Считаем, что данные готовы => обрабатываем

Пример переупорядоченных компилятором инструкций – взгляд компилятора

- Компилятор может сгенерировать код в том порядке, который задал программист:

  data = <значение>;

  readyFlag = true;

- А может и в другой (оптимизируя код):

  readyFlag = true;

  data = <значение>;

так как с точки зрения компилятора на конечный результат порядок не влияет (компилятор ничего не знает про второй поток)

Пример переупорядочения компилятором низкоуровневых инструкций при включении оптимизаций

```
int A, B=33;
void f()
{
   A = B + 1;
   B = 0;
}
```

//Без оптимизаций (псевдокод)	//с включенными оптимизациями
mov eax, DWORD PTR [B] add eax, 1 mov DWORD PTR [A], eax **mov DWORD PTR [B], 0**	mov eax, DWORD PTR [B] **mov DWORD PTR [B], 0** add eax, 1 mov DWORD PTR [A], eax

Модель памяти C++11:

предоставляет различные **ограничения** на переупорядочение компилятором операций при выполнении действий над **атомарными типами**

Важно! эти ограничения формируются вокруг атомарных операций, но воздействуют как на атомарные операции, так и опосредованно на обычные (неатомарные)

Итог – проблемы, возникающие при конкурентном программировании:

- несинхронизированный доступ к данным (порядок чтения и записи в разных потоках)
- частично записанные данные (чтение в одном потоке началось раньше, чем закончилась запись в другом)
- переупорядоченные операции (компилятор может изменять порядок выполнения инструкций, если при перестановке поведение **данного** конкретного потока остается корректным)

Проблемы, порождающие неопределенное поведение:

- несинхронизированный доступ к данным
- неатомарные операции модификации данных
- переупорядочение выполнения кода

СПОСОБЫ РЕШЕНИЯ ПРОБЛЕМ

При конкурентном программировании

необходимо **принудительно** упорядочить обращения к данным из разных потоков!

Способы решения проблем:

- **атомарность** (в широком смысле – операция или последовательность операций в одном потоке не может быть прервана другим потоком)
- **порядок** – гарантия того, что порядок выполнения операций будет упорядочен

Средства, предоставляемые **C++11** для решения проблем конкурентного доступа:

- **атомарные типы данных**
- **атомарные операции**
- условные переменные (conditional variables)
- мьютексы и блокировки
- futures и promises

Недостатки использования примитивов синхронизации, предоставляемых ОС:

- в некоторых реализациях являются объектами исполняющей системы
- => обращение к такому объекту может быть очень дорогим: может потребоваться переключение контекста, переход на уровень ядра ОС, поддержка очередей ожидания доступа к защищаемым примитивом синхронизации данным и пр.

И если это нужно только для того, чтобы выполнить **одну-две ассемблерных инструкци**и, то это:

- неэффективно
- + объект ядра ОС – это ограниченный по количеству ресурс.

Важно!

- **Высокоуровневые средства** (futures, promises, mutexes, conditional variables)
  - относительно просты в использовании
  - и безопасны
  - но! менее эффективны
- **Низкоуровневые средства** (атомарные переменные и атомарные операции)
  - обеспечивают бОльшую производительность
  - являются более универсальными
  - но! не могут работать с данными любого типа!
  - кроме того! риск их неправильного использования гораздо выше!

Неблокирующая синхронизация

Википедия:

«Неблокирующая синхронизация - подход в параллельном программировании на симметрично-многопроцессорных системах, проповедующий **отказ от традиционных примитивов блокировки, таких, как семафоры, мьютексы и события**. Разделение доступа между потоками идёт за счёт **атомарных операций и специальных, разработанных под конкретную задачу, механизмов блокировки**»

Важно!

- Писать lock-free код не просто.
- Писать **правильный** lock-free код невероятно сложно.

Цели lock-free параллельных программ:

- Повышение масштабируемости путем сокращения блокировок и ожиданий
- При этом нужно гарантировать отсутствие неопределенного поведения
- Решение проблем, связанных с блокировками, например, dead-lock-ов

#include <atomic>

АТОМАРНЫЕ ОПЕРАЦИИ И АТОМАРНЫЕ ТИПЫ

М.Полубенцева

Атомарные объекты и атомарные операции -

это самый низкий уровень синхронизации в многопоточных приложениях, доступный в C++

Атомарность может быть реализована:

- на аппаратном уровне (когда непрерывность обеспечивается аппаратурой) – действительная атомарность
- или эмулироваться программно.

Замечание:
volatile и параллелизм в C++

volatile в C++ :

- только предотвращает агрессивную оптимизацию
- не гарантирует
  - ни атомарности
  - ни порядка

$\Rightarrow$ для C++11 указание volatile для атомарной переменной - только для предотвращения оптимизаций компилятора

$\Rightarrow$ не обязательно

Напоминание - пример неопределенного поведения

Поток 1

count++;

Поток 2

count++;

Поток 3

count++;

Глобальная переменная

int **count**=0;

```
count++;
mov eax,dword ptr ds:[смещ]
add eax,1
mov dword ptr ds:[смещ],eax
```

Модифицируем пример, используя атомарные типы и операции

Поток 1

```
++atomicCount;
```

Поток 2

```
++atomicCount;
```

Поток 3

```
++atomicCount;
```

Глобальная переменная

```
std::atomic<int> atomicCount(0);
```

```
++atomicCount;
mov ecx,<&atomicCount> //this
call std::_Atomic_int::operator++
```

Важно!

Использование атомарных операций

- **не** предотвращает гонку (какая атомарная операция выполнится раньше, не гарантируется)
- но! позволяет избежать **неопределенного поведения**

Атомарные операции – один из способов избежать неопределенного поведения

- -операции, которые гарантированно будут выполнены **целиком** даже в том случае, когда в процессе выполнения такой операции квант времени потока истекает или данный поток должен быть вытеснен другим более приоритетным потоком

  Замечание: - это «облегченная» альтернатива мьютексу
- согласно стандарту C++11 в атомарных операциях используются специальные атомарные типы

Атомарные операции – альтернатива мьютексам

Почему введены (помимо мьютексов) атомарные операции:

- ‘+’
  - эффективность (выигрыш по производительности, так как не требуется переключение контекста, переход на уровень ядра…)
  - программисту не нужно **явно** обеспечивать неделимость (реализуется стандартной библиотекой посредством команд процессора или эмулируется)
- ‘-’
  - относительная сложность корректного использования

Замечание:

- мьютексы **могут быть** (**необязательно**) медленнее операций над атомарными объектами
- => не нужно воспринимать это как мьютексы плохие, а атомарные объекты – «наше всё».

Мьютексы и атомарные объекты созданы для разных ситуаций => прежде чем использовать атомарные объекты, нужно оценить **cost/benefit**

Когда выигрыша от использования атомарных операций можно не получить:

- атомарный доступ может понадобится к данным любой сложности и размера
- настоящей атомарной операцией над данными можно считать лишь ту, которую процессор может выполнить **одной командой**
- процессор не имеет таких команд для структур данных любой сложности
- => чтобы оставить возможность атомарного доступа, и не скатиться лишь к базовому набору типов, **атомарность должна эмулироваться для всех типов**, которые процессор не может обрабатывать атомарно! Возможно, за счёт тех же самых мьютексов.

Возможные способы реализации/эмуляции атомарных операций:

- **использование «атомарных» инструкций процессора**
- запрет/разрешение прерываний (для очень коротких действий?)
- синхронизирующие примитивы (для длинных?)
- запрещение переключения контекста потока
- блокировка шины
- повышение приоритета

Атомарные операции

- делятся на простые
  - чтение
  - запись
- и операции атомарного изменения (read-modify-write, RMW)

Пример: атомарные инструкции x86

- **CMPXCHG/CMPXCHG8B/CMPXCHG16B** — основная атомарная команда x86 (сравнение и обмен). При использовании с префиксом **LOCK** атомарно выполняет сравнение переменной с указанным значением и пересылку в зависимости от данного сравнения. Является основой реализации всех безблокировочных алгоритмов. Часто используется в реализации спинлоков и RWLock'ов, а также практически всех высокоуровневых синхронизирующих элементов, таких как семафоры, мьютексы, события и пр. в качестве внутренней реализации
- **XCHG** — Операция обмена между памятью и регистром. Выполняется атомарно на x86-процессорах. Часто используется в реализации спинлоков.

Продолжение

Кроме того, многие команды вида Чтение-Модификация-Запись могут быть сделаны искусственно атомарными с помощью префикса **LOCK**:

- Команды сложения и вычитания **ADD/ADC/SUB/SBB**, где операнд-приемник является ячейкой памяти
- Команды инкремента/декремента **INC/DEC**
- Логические команды **AND/OR/XOR**
- Однооперандные команды **NEG/NOT**
- Битовые операции **BTS/BTR/BTC**
- Операция сложение и обмен **XADD**

Префикс **LOCK** вызывает блокировку доступа к памяти на время выполнения инструкции. **Блокировка может распространяться на область памяти шире, чем длина операнда, например, на длину кэш-линии**.

Замечание:

- В современных процессорах **гарантируется настоящая атомарность** чтения/записи/обмена только **выровненных** простых (integral) типов – целых чисел и указателей.

Атомарные инструкции и компилятор

Компиляторы языков высокого уровня сами никогда не используют при генерации кода атомарные инструкции, так как:

- атомарные операции во много раз более ресурсоёмкие, чем обычные
- у компилятора нет информации, когда доступ должен осуществляться атомарными инструкциями

=> программист использует один из следующих подходов:

- ассемблерная вставка соответствующей атомарной инструкции
- использование расширения компилятора (функции семейства __builtin_ или __sync_)
- использование «высокоуровневой» обертки посредством специальной библиотеки
- вызов системной функции (Windows – семейство interlock-функций)
- **Использование C++11, поддерживающего типы atomic и функции семейства atomic_**

Ассемблерная вставка:

```
 int nGlobal=0;

int main()
{
    __asm lock inc dword ptr [nGlobal]
}
```

класс **atomic_flag**
<atomic>

- не шаблон!
- единственная **гарантированная** (независимо от реализации) lock-free структура данных
- предоставляет минимум функциональности
- => используется в очень простых задачах

Инициализация **atomic_flag:**

- обязательно (независимо от времени жизни) должен быть проинициализирован значением **ATOMIC_FLAG_INIT** (создается в сброшенном состоянии)
- atomic_flag(const atomic_flag&) = delete;
- atomic_flag(); //unspecified state

Функциональность **atomic_flag:**

- atomic_flag& operator=(const atomic_flag&) = delete;
- void atomic_flag::**clear**(<порядок>);
  глобальной функции вида –
  void **atomic_flag_clear**(std::atomic_flag* p);
  void **atomic_flag_clear_explicit**(std::atomic_flag* p, <порядок>);
- bool atomic_flag::**test_and_set**(<порядок>);
  bool **atomic_flag_test_and_set**(std::atomic_flag* p);
  bool **atomic_flag_test_and_set_explici**t (std::atomic_flag* p, <порядок>);

Замечание: единственный и обязательный способ инициализации

Замечание: реализация ATOMIC_FLAG_INIT зависит от реализации:

```
#define ATOMIC_FLAG_INIT /* implementation-defined */

//std::atomic_flag f(ATOMIC_FLAG_INIT);
        //ошибка – attempting to reference a deleted function
        //даже, если компилятор ошибки не выдает - unspecified

std::atomic_flag f1{ ATOMIC_FLAG_INIT }; //ОК
std::atomic_flag f2 = ATOMIC_FLAG_INIT; //ОК
```

Замечание:

```
std::atomic_flag f3; //unspecified state
```

bool atomic_flag::**test_and_set()**

- устанавливает флаг в true и
- возвращает текущее значение:
  - true, если флаг был на момент вызова установлен (true)
  - иначе false

Замечание: такой проверкой можно «отследить» **сброс** флага в другом потоке!!!

Пример: реализация spin-lock “мьютекса” посредством **atomic_flag**

```
class handmade_mutex{
   std::atomic_flag flag;
public:
  handmade_mutex():flag{ATOMIC_FLAG_INIT}{}
  void lock() { while(flag.test_and_set()){;}  }
  void unlock() { flag.clear(); }
};
```

Реализация try_lock()

???

Реализация try_lock()

```
bool try_lock()
{
   return !flag.test_and_set();
}
```

Продолжение примера. Использование hand made мьютекса

```
handmade_mutex m; //глобальная переменная

//поток, выполняющий работу над разделяемыми данными
void myThread()
{
        m.lock();
        //работа с разделяемыми данными
        m.unlock();
}
```

Продолжение примера. Использование lock_guard

```
handmade_mutex m; //глобальная переменная

//поток, выполняющий работу над разделяемыми данными
void thread()
{
        std::lock_guard<handmade_mutex> lk(m);
        //работа с разделяемыми данными
} //???
```

Специфика atomic_flag:

- нет операций load() и store()
- и не допускает до C++20 проверки без изменения значения
  C++20 – добавлены:
  - test() – проверка состояния
  - wait(bool b) – сравнивает текущее состояние с “b”. Если равны, поток блокируется и ожидает notify
  - notify_one – разблокирует один (ожидающий на wait()) поток
  - notify_all - все

Шаблон структуры std::**atomic**<T>

- <atomic>
  template< class T > struct atomic; //генеральный шаблон
  template< class U > struct atomic<**U***>; //частичная специализация для указателей

- <memory>
  C++20
  частичная специализация для smart pointer-ов
  template< class U > struct atomic<**std::shared_ptr**<U>>;
  template< class U > struct atomic<**std::weak_ptr**<U>>;

Шаблон структуры std::**atomic**<T>

- является **оберткой** для атомарного значения
- предоставляет основные операции для выполнения атомарных действий над хранящимся значением:
  конструкторы,
  load(),
  store(),
  is_lock_free(),
  exchange(), …
- запрещает копирование и присваивание объектов типа atomic
  (так как любые операции над двумя объектами НЕ могут быть атомарными)
- большинство параметров принимаются **по значению**
- результат возвращается **по значению**
- для большинства методов есть перегруженные
  () volatile

Замечание:

Специализации шаблона предоставляют дополнительную функциональность. Например:

- специализации для целых типов:
  fetch_add(), fetch_sub(), fetch_and(), fetch_or(), fetch_xor()
- специализации для плавающих типов (**C++20**!):
  fetch_add(), fetch_sub()

Псевдонимы:

typedef	тип
std::atomic_bool	std::atomic<bool>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
…	…

Специализации шаблона atomic<T>:

- **bool**
- **любых целых типов**
- **любых указателей**

Замечание: класс **atomic_flag** – это **не** специализация atomic<T>:

- гарантированно lock-free
- и не эквивалентен atomic<bool>

Замечание:

большинство методов класса atomic, например:

T std::atomic<T>::**load();**

дублируются шаблонами глобальных функций вида:

template< class T > T **atomic_load**
(const std::atomic<T>* obj);

Инициализация:

- **atomic()**=default ; //until C++20!
  для завершения инициализации можно использовать !!!!! deprecated in C++20
  template< class T > void **atomic_init**(std::atomic<T>* obj, T desired);
  ,которая не является атомарной!

- **constexpr atomic();**// C++20

- **atomic(T desired)**; // выполняется не атомарно!

- **atomic(const atomic&) = delete**;

store() и load()

- методы класса вида:
  T **load**(<порядок>) const;
  void **store**(T desired, <порядок>);

- шаблоны глобальных функций вида:

template< class T > T **atomic_load**(const std::atomic<T>* obj);

template< class T > void **atomic_store**(std::atomic<T>* obj, T desr);

template< class T > T **atomic_load_explicit**(const std::atomic<T>* obj, <порядок>);

template< class T > void **atomic_store_explicit** (std::atomic<T>* obj, T desr , <порядок>);

is_lock_free()

метод bool **is_lock_free**() const;

шаблон глобальной функции - template< class T > bool
atomic_is_lock_free(const std::atomic<T>* obj);

Проверка:

- true – если операции для данного типа действительно реализуются атомарно (посредством одной команды процессора)
- false – если операции эмулируют атомарность

Пример:

```
class A1 { bool b; };
class A2 {
        bool b;
        int n;
};
class A3 { double d; };
class A4 {
        double d;
        int n;
};
```

```
bool b1 = std::atomic<A1>{}.is_lock_free(); //???
bool b2 = std::atomic<A2>{}.is_lock_free(); //???
bool b3 = std::atomic<A3>{}.is_lock_free(); //???
bool b4 = std::atomic<A4>{}.is_lock_free(); //???
```

C++17 - is_always_lock_free()

```
int x = 1;
double y = 2.2;
struct Ar {
        char ar[100] = "qwerty";
};
constexpr bool b1 = std::atomic<decltype(x+y)>::is_always_lock_free;
constexpr bool b2 = std::atomic<decltype(&y)>::is_always_lock_free;
constexpr bool b3 = std::atomic<Ar>::is_always_lock_free;
```

operator=

atomic& **operator=**(**const atomic&**) **= delete**;

T **operator=**(**T** desired); //возвращает копию desired
//эквивалентно store(desired)

std::atomic<int> n(1);
int m=33, z=0;
z=n=m; //???

operator T ()

- эквивалентен load()

```
std::atomic<int> n(1);

int m1 = n;
int m2 = static_cast<int>(n);
```

exchange()

метод:

T **exchange**(T desired, <порядок>);

шаблон глобальной функции:

template< class T > T
atomic_exchange(std::atomic<T>* obj, T desr);

template< class T > T
atomic_exchange_explicit(std::atomic<T>* obj,
T desr, <порядок>);

???

```
std::atomic<int> n(1);
int m = n.exchange(2);
// n==? m==?
```

compare_exchange_weak(), compare_exchange_strong()

bool **compare_exchange_strong**
(T& expected, T desired, <порядок>);

- сравнивают expected (ожидаемое значение) с хранящимся атомарным значением
- и, если они совпадают, заменяет хранящееся значение на desired, возвращает true
- если не совпадают, загружает текущее значение по адресу expected, возвращает false (текущее при этом не меняется!)

Замечание: соответствующие шаблоны глобальных функций – **atomic_compare_exchange_strong**()

Как работает (псевдокод)

```
bool cmp_exch(T* cur, T* expected, T desired)
{
        if(*cur==*expected){
                * cur = desired;
                return true;
        }else{
                *expected = *cur;
                return false;
        }
}
```

Пример:

```
std::atomic<int> current(1);
int expected = 0;

bool b1 = atomic_compare_exchange_strong(&current,
                                        &expected, 3); //false, expected = 1, current не изм.
//или
//bool b1 = current.compare_exchange_strong( expected, 3);

expected = 1;
bool b2 = std::atomic_compare_exchange_strong(&current,
                                        &expected, 3);
//true, expected не изм., current =3
```

Пример обычного использования

```
std::atomic<bool> b(false); //глобальная используемая несколькими потоками переменная

//Поток:
{
   bool expected = false;
    while(//цикл продолжается, пока b==true
     b. compare_exchange_strong(expected, true) ==false)  {expected=false;}
          //работаем «под защитой»
     b.store(false); //позволяем другому потоку продолжить выполнение
//или b=false;
}
```

Или так:

```
extern std::atomic<bool> b; //внешняя

//Поток
{
  bool expected;
  do{expected=false;}
   while(//цикл продолжается, пока b==true
     !b. compare_exchange_strong(expected, true) ) ;

}
```

Замечание:

версия **weak** – эффективнее, но отличается тем, что сохранение desired может **не** произойти даже в том случае, когда текущее значение совпадает с expected (текущее значение не изменится, а функция вернет false). Такое возможно, если в наборе команд процессора нет аппаратной команды сравнить-и-обменять => поток может быть вытеснен в середине требуемой последовательности => процессор не может гарантировать атомарность => “ложный” отказ => обычно используется в цикле (с маленьким телом) -> возможна дополнительная проверка

Пример использования compare_exchange_weak() - защита от ложного отказа

extern std::atomic<bool> b; //внешняя – если true, то «занята» другим потоком => нужно дождаться, пока другой поток освободит (сбросит в false)

```
//Поток
{
  bool expected = false;
  while( //ждем, пока кто-нибудь сбросит b в false
b. compare_exchange_weak(expected, true)==false && !expected
        ) {expected=false;}
        //работаем «под защитой»
        b.store(false);
}
```

Замечание 1: только для специализаций шаблона atomic<Integral> :

- добавлены специализированные методы вида:
  T **fetch_add**(T , <порядок>);
  …
  Важно! атомарно модифицируют хранящееся значение, а возвращают то значение, которое было до выполнения операции
- перегружены операторы:
  operator++ //Важно! **T** operator++(); экв. fetch_add(1)+1
  operator--
  совмещенные операторы присваивания:
  operator+=
  operator&=
  …
- методам соответствуют шаблоны глобальных функций вида:
  template< class Integral > Integral
  atomic_fetch_add(std::atomic<Integral>* obj, Integral arg);

Пример: ???

```
void f(std::atomic<int>& x){ x.fetch_add(5); }

int main(){
    std::atomic<int> data(33);
    std::thread t1(f, data);
    std::thread t2(f, data);
    t1.join(); t2.join();
}
```

Исправляем!

```
void f(std::atomic<int>& x){ x.fetch_add(5); }


int main(){
    std::atomic<int> data(33);
    std::thread t1(f, std::ref(data));
    std::thread t2(f, std::ref(data));
    t1.join(); t2.join();
}
```

Замечание 2: только для специализаций шаблона atomic<T*> :

- добавлены специализированные методы вида:
  T* **fetch_add**(std::ptrdiff_t , <порядок>);
  …
- перегружены операторы:
  operator++
  operator--
  operator+=
  operator-=

Отличие fetch_add(1) – можно указать упорядочение, а в operator++ - по умолчанию

Пример:

```
void fPtr(std::atomic<int*>& x) { … x++;  …}
int main(){
        int ar[] = {1,2,3,4};
        std::atomic<int*> data = ar;
        std::thread t1(fPtr, std::ref(data));
        std::thread t2(fPtr, std::ref(data));
        t1.join(); t2.join();
}
```

Важно!

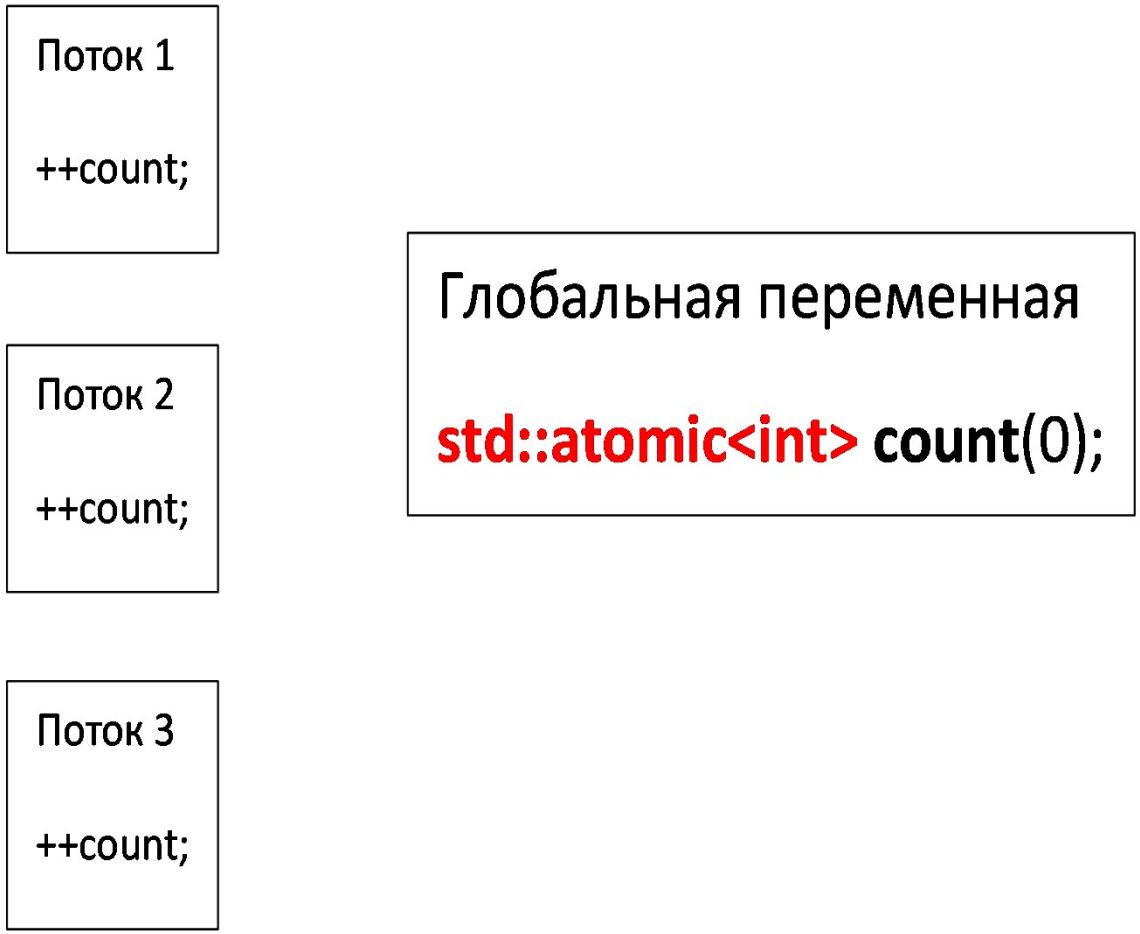
запрещено использовать с нетривиальными типами, например:

```
std::atomic<std::vector<int>>
```

так как:

- у вектора нетривиальный конструктор копирования
- нетривиальный operator=

Переписываем пример в атомарных терминах



Использование атомарных операций

```
#include <atomic>
std::atomic<int> count (0);

поток_1
{
  ++count; // например, count==1

//или
   int previous = count.fetch_add(1);     //например, previous==1, count==2…
}
```

Важно!

- каждая из операций может быть атомарной, но их комбинация атомарной НЕ ЯВЛЯЕТСЯ. Например:

```
std::atomic<int> integer(0);
std::atomic<int> otherInteger(0);
integer++;    //Атомарно
otherInteger += ++integer;    //Не атомарно!
```

Атомарные пользовательские типы

- Если размер пользовательского типа <= <размер_регистра>, то в большинстве случаев компилятор может сгенерировать код, состоящий из атомарных операций
- если размер > <размер_регистра>, то в большинстве случаев эмуляция атомарности (=> выигрыша по сравнению с использованием мьютекса можно не получить)
- ограничения на пользовательские типы:
  - тривиальные конструктор копирования и operator= (которые реализуются компилятором автоматически посредством побитового копирования - memcpy),
  - сравнение должно осуществляться посредством побитового сравнения – memcmp()
  - нет виртуальных функций и виртуальных базовых классов

Пример атомарного пользовательского типа

```
class Test {
   int a, b;
public:
   Test(int _a=0, int _b=0)noexcept :a(_a), b(_b) {}
};

std::atomic<Test> t(Test(1, 2));
```

Специфика:

```
class Test {
   int a, b;
public:
   Test(int _a=0, int _b=0)noexcept :a(_a), b(_b) {}
   Test(const Test& other) { a = other.a; b = other.b; };
};

int main(){
   Test   t1(1, 2),  t2(3, 4);
   std::atomic<Test> a1(Test(1, 2));  //ошибка - atomic<T> requires T to be
                                      trivially copyable
}
```

Но!

```
class Test {
  int a, b;
public:
  Test(int _a=0, int _b=0)noexcept :a(_a), b(_b) {}
  bool operator==(const Test& right) const
                  { return a == right.a && b == right.b; }
};

int main(){
  Test   expected(1, 2),  desired(3, 4);
  std::atomic<Test> a1(Test(1, 2));  //OK
  bool b = a1.compare_exchange_strong(expected, desired);
                                    //operator==  для сравнения не  вызывается
}
```

std::atomic<float>
std::atomic<double>

- формально разрешены, так как удовлетворяют тем же ограничениям, которым должны следовать пользовательские типы (побитовое копирование и сравнение)
- но! могут иметь разное внутреннее представление ??? => при сравнении равных значений можно получить false
- не определены атомарные арифметические операции

Замечание: C++20 – включены в стандарт!

C++20 - std::atomic<>
добавлена функциональность:

- wait() - blocks the thread until notified and the atomic value changes
- notify_one() - notifies a thread blocked in atomic_wait
- notify_all() - notifies all threads blocked in atomic_wait

C++20 - std::atomic_ref<>
<atomic>

template< class T > struct **atomic_ref**;

template< class T > struct **atomic_ref<T*>**;

применяют атомарные операции к значениям, хранящимся по адресам (compare_exchange_strong(), operator++(), fetch_add(),…)

Ограничение – типы должны быть тривиально копируемыми

Отличия:

```
int main(){
        int n=1; //не атомарная переменная, но тип тривиальный!
        std::atomic<int> a1(n);  //OK
        a1++; // n==???   a1 = ???
        std::atomic_ref<int> a2(n);
        a2++; // n==???   a2 = ???
}
```

C++20 - std::atomic_ref<>
добавлена функциональность:

- wait() - blocks the thread until notified and the atomic value changes
- notify_one() - notifies a thread blocked in atomic_wait
- notify_all() - notifies all threads blocked in atomic_wait

C++20 - std::atomic<std::shared_ptr<T>> std::atomic<std::weak_ptr<T>>

- приспособлен для использования smart-pointer-ов в многопоточной программе
- гарантирует:
  - атомарную работу с управляющим блоком (в частности инкремент и декремент счетчиков ссылок)
  - для целевого объекта вызов delete-функции только один раз!
  - чтение целевого объекта может производиться параллельно в разных потоках

Memory order

ПОРЯДОК ИСПОЛНЕНИЯ

М.Полубенцева

Порядок исполнения

- До C++11 – правила для однопоточного приложения:
  - зависимые друг от друга вычисления выполняются в предусмотренном программистом порядке,
  - остальные вычисления могут выполняться в том порядке, который компилятор + процессор считает оптимальным
- В многопоточном приложении может возникать необходимость упорядочить в одном потоке исполнение вычислений
  => *порядок изменения(**ПИ**)* учитывает наличие потоков, и является глобальным по отношению к ним, т.е. *ПИ* выстраивает порядок не относительно какого-либо потока а является общим для всех потоков, в которых участвуют **атомарные объекты данного *ПИ***. *ПИ* имеет отношение лишь к атомарным объектам, следовательно он влияет только на порядок вычисления выражений, в которых вовлечены атомарные объекты

Пример неопределенного порядка исполнения

```
void f(int a, int b){std::cout<<a<<‘ ‘<<b;}
int get(){
   static int val=0;
   return ++val;
}
int main(){
   f( get(), get() ); //порядок вызова не определен => ???
}
```

До C++11 - точка следования (sequence point)

- точка программы, в которой все побочные эффекты от предыдущих вычислений должны завершиться, а от последующих еще не начаться
- C++11 - sequence point заменено на:
  - **sequenced before**
  - sequenced after
  - indeterminately sequenced (один из before/after – неизвестно, какой конкретно)
  - unsequenced

- C++11 - появилось отношение «Synchronized with»

Специфика:

- Отношение – «**synchronized with**» возможно только между операциями над **атомарными** типами!
- Отношения – «**sequenced before/ sequenced after**» характеризуют, каким образом неатомарные операции группируются вокруг атомарных в одном потоке

Без упорядочения:

(только два потока)

<table>
<tr><td colspan="2">std::vector<int> data;
bool ready_flag{false};</td></tr>
<tr><td>void writer(){
//запись
data.push_back(33);
ready_flag= true;
}</td><td>void reader(){
while(!ready_flag){…}
//данные готовы =>чтение
std::cout<<data.back();
}</td></tr>
<tr><td colspan="2">Без упорядочения – неопределенное поведение!</td></tr>
</table>

Демонстрация отношений

(только два потока)

std::vector<int> data;
std::atomic<bool> ready_flag(false);

```
void writer(){
//запись
  data.push_back(33);
  ready_flag.store( true);
}
```

```
void reader(){
 while(!ready_flag.load()){…}
 //данные готовы =>чтение
 std::cout<<data.back();
}
```

В результате чтение происходит гарантированно после записи!
=> принудительное упорядочение (пока по умолчанию) !

Глубина упорядочения ???

- определяется точками следования?

БАРЬЕРЫ

Средства принудительного упорядочения выполнения:

- **compiler barrier** - запрет компилятору на переупорядочение (это просто указание компилятору не переставлять генерируемые инструкции «за барьер» или наоборот)
- **memory barrier** - запретить переупорядочение инструкций процессором, то есть на момент «прохождения» барьера:
  - заставляет выполнится весь конвейер до барьера => все предшествующие барьеру операции должны быть завершены
  - сброс предвыборки (PrefetchFlush) - благодаря чему следующие за барьером инструкции будут заново выбраны и декодированы из памяти (или кэша???).

Барьеры компилятору

Для запрета переупорядочения кода существует универсальный метод — установка барьера компилятору

- Барьеры приводят к частичному упорядочиванию операций доступа к памяти по обе (или по указанную) стороны барьера.
- Переупорядочение инструкций возможно только до барьера или только после барьера, но не через барьер!
- Барьер ничего не блокирует – просто препятствует оптимизации
- Существует несколько видов барьеров памяти: полный, release fence и acquire fence.

Характеристики барьеров (это относится как к процессору, так и к компилятору):

- **Полный барьер** гарантирует, что все чтения и записи расположенные до/после барьера будут выполнены также до/после барьера, то есть никакая инструкция обращения к памяти не может «перепрыгнуть» барьер.
- **Acqire fence (полубарьер)** не упорядочивает предыдущие store-операции с последующими load/store.
- **Release fence (полубарьер)** не упорядочивает предыдущие load с последующими load, равно как и предыдущие store с последующими load.

Иллюстрация работы барьеров. Полный барьер:

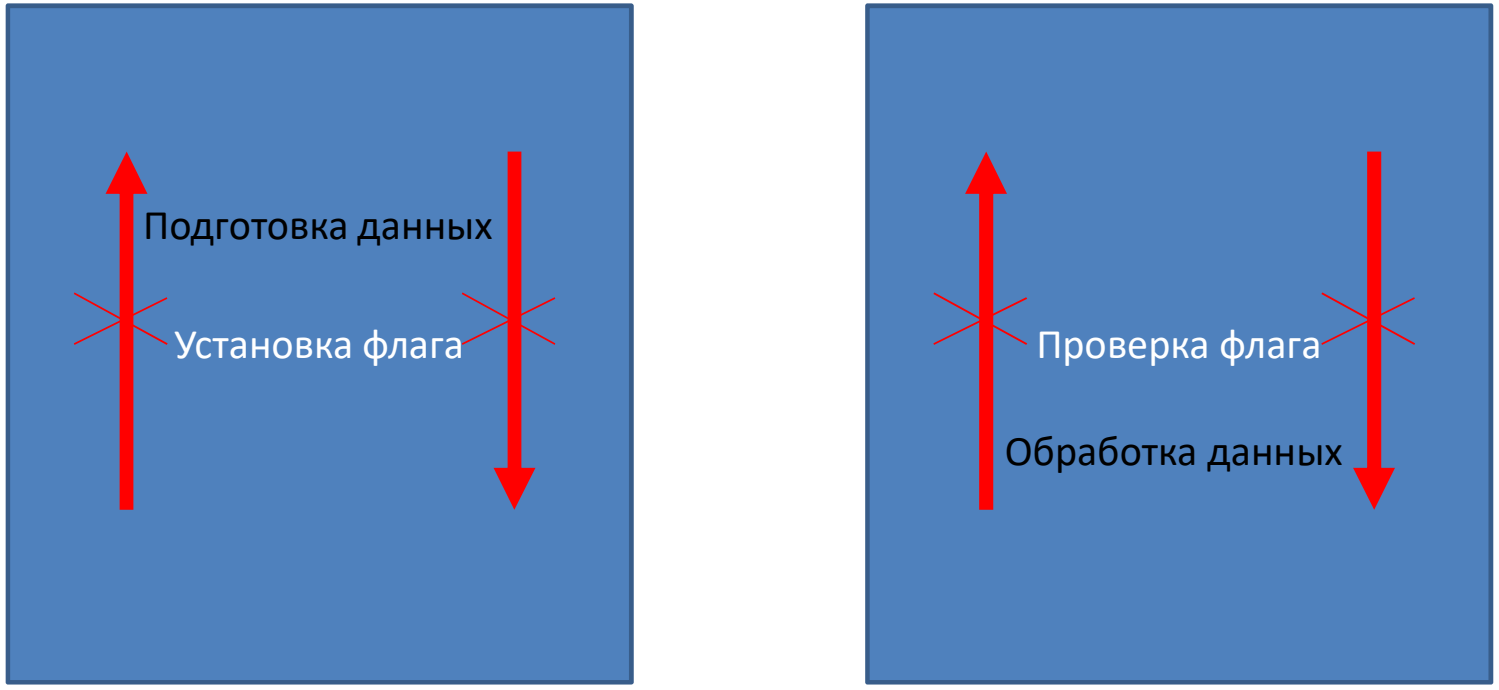
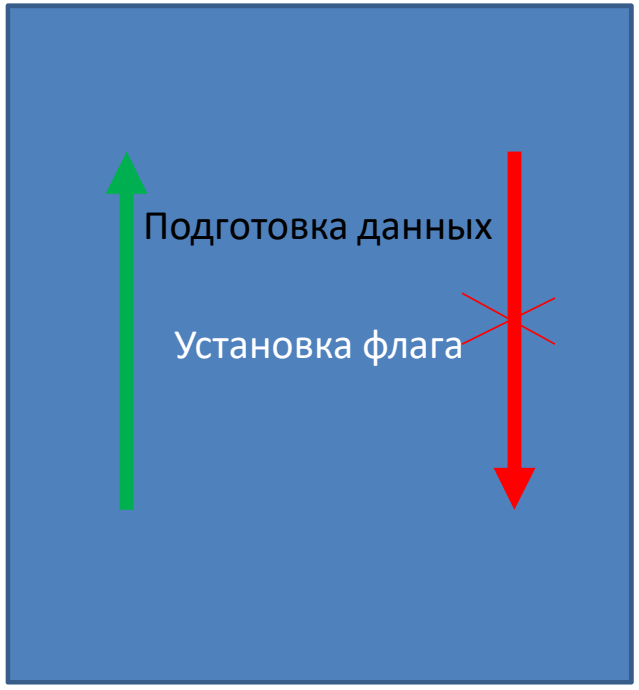
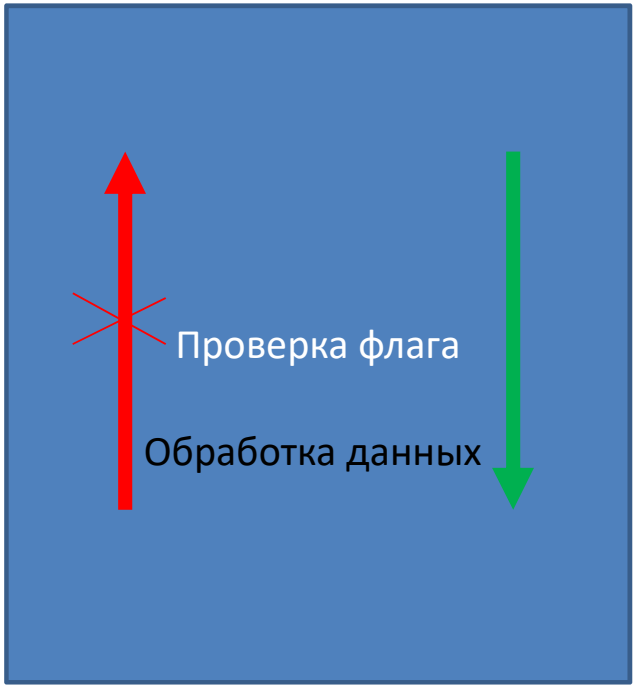


Иллюстрация работы барьеров. acquire-release барьер:

release барьер



acquire барьер



Барьеры памяти

- для корректного выполнения параллельного кода **процессору** необходимо «подсказывать», до каких пределов ему разрешено проводить свои внутренние оптимизации чтения/записи.
- Эти подсказки – барьеры памяти. Барьеры памяти позволяют в той или иной мере упорядочить обращения к памяти (точнее, кэшу, — процессор взаимодействует с внешним миром только через кэш).
- Степень упорядочения может быть разной, — каждая архитектура может предоставлять целый набор барьеров “на выбор”. Используя те или иные барьеры памяти, мы можем построить разные модели памяти — набор гарантий, которые будут выполняться для наших программ.

Барьер памяти

Процессоры используют множество приёмов для повышения производительности:

переупорядочивание операций, предвыборка (prefetch - раннее чтение данных, предсказание переходов) и совмещение операций (спекулятивное выполнение) доступа к памяти, и различные типы кеширования.

Барьеры доступа к памяти служат для подавления этих механизмов.

Барьеры для процессора

- Store Memory Barrier (также ST, SMB, smp_wmb) — инструкция, заставляющая процессор выполнить все **store**, уже находящиеся в буфере, прежде чем выполнять те, что последуют после этой инструкции

- Load Memory Barrier (также LD, RMB, smp_rmb) — инструкция, заставляющая процессор применить все **invalidate**, уже находящиеся в очереди, прежде чем выполнять какие-либо инструкции load

Демонстрация использования барьеров. Псевдокод:

```
void executedOnCpu0() {
    value = 10;
    storeMemoryBarrier();
    finished = true;
}

void executedOnCpu1() {
    while(!finished);
    loadMemoryBarrier();
    assert (value == 10);
}
```

УПОРЯДОЧЕНИЕ ДОСТУПА К ПАМЯТИ ДЛЯ АТОМАРНЫХ ОПЕРАЦИЙ

М.Полубенцева

Напоминание. Важно!

Основное время при выполнении программы уходит на **обращения к памяти =>**

- на вычисления – мало!
- на чтение/запись – много!

=> компилятор оптимизирует низкоуровневый код,
а схемотехники оптимизируют все действия, связанные с чтением/записью

=> порядок выполнения программы не определен (гарантируется только корректное получения результата в пределах одного потока)

Средства C++11 управления упорядочением

- Для задания компилятору правил упорядочения вокруг действий с атомарными переменными в большинстве методов и соответствующих глобальных функций-аналогов вводится дополнительный параметр **std::memory_order**
  Замечание: при этом барьер устанавливается внутри соответствующих функции (switch)
- + функции для «ручной» установки барьеров:
  std::atomic_thread_fence(<порядок>**) –** барьер памяти
  std::atomic_signal_fence(<порядок>**) –** барьер компилятора

enum std::memory_order

предписание компилятору (опосредованно процессору):

- разрешается ему или нет переупорядочивать (reoder)
  - атомарные операции
  - и другие (неатомарные) обращения к памяти до и/или после атомарной операции

Упорядочение доступа к памяти для атомарных операций

Варианты задаются константами memory_order:

- memory_order_**relaxed**
- memory_order_**consume**
- memory_order_**acquire**
- memory_order_**release**
- memory_order_**acq_rel**
- **memory_order_seq_cst** - умолчание

Шесть вариантов представляют три модели (разная эффективность, зависят от архитектуры):

- **sequentially consistent** - последовательно согласованное упорядочение (глобальное) - memory_order_seq_cst
- **acquire-release** - упорядочение захват/освобождение (между парами потоков) - memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel
- ослабленное упорядочение - memory_order_**relaxed**

Замечание:

- три модели упорядочения обычно влекут за собой различные издержки для процессоров с разной архитектурой => разная эффективность => в общем случае
  - захват/освобождение «дешевле», чем последовательно согласованное
  - а ослабленное «дешевле», чем захват/освобождение
- последовательно согласованное упорядочение
  - интуитивно понятнее
  - проще в использовании
- Важно! наличие разных моделей упорядочения доступа к памяти позволяет **эксперту** добиться повышения производительности за счет более точного управления отношениями упорядочения (по сравнению с использованием последовательно согласованного упорядочения)

memory_order_seq_cst – принудительное последовательно согласованное упорядочение (по умолчанию)

- если все операции над экземпляром **атомарного типа** последовательно (принудительно) согласованы, то поведение многопоточной программы в целом такое же, как если бы эти операции выполнялись в определенной последовательности в одном потоке
- упорядочение глобальное, то есть ВСЕ потоки «видят» один и тот же порядок операций изменения разделяемых данных
- гарантирует: на момент чтения атомарных данных в одном потоке
  - все действия, предшествующие записи в другом потоке, должны быть выполнены!
  - все действия в данном потоке, следующие за чтением, еще не выполнены

Важно!

- если в одном потоке задано последовательно согласованное сохранение данных (запись)
- то чтение в другом потоке должно быть тоже последовательно согласовано (если задан ослабленный порядок, то никаких гарантий нет)

=> Для «**глобального**» упорядочения при доступе к одним и тем же разделяемым данным (memory location) в разных потоках требуется использовать последовательно согласованное упорядочение

=> При этом при модификации данных одним потоком своей копии, находящиеся во всех кэшах –> invalidate => синхронизация кэшей (процессоров)

В противном случае

- В отсутствие явных ограничений на упорядочение кэши различных процессоров и внутренние буферы могут содержать **различные значения** для одной и той же memory location

Производительность

Последовательно согласованное упорядочение в **некоторых** системах реализуется посредством дополнительных команд синхронизации => может привести к потерям производительности системы в целом

Замечание:

некоторые архитектуры (x86, x86-64) обеспечивают последовательную согласованность с относительно низкими издержками => прежде, чем отказываться от этого средства, смотри документацию по конкретному процессору!

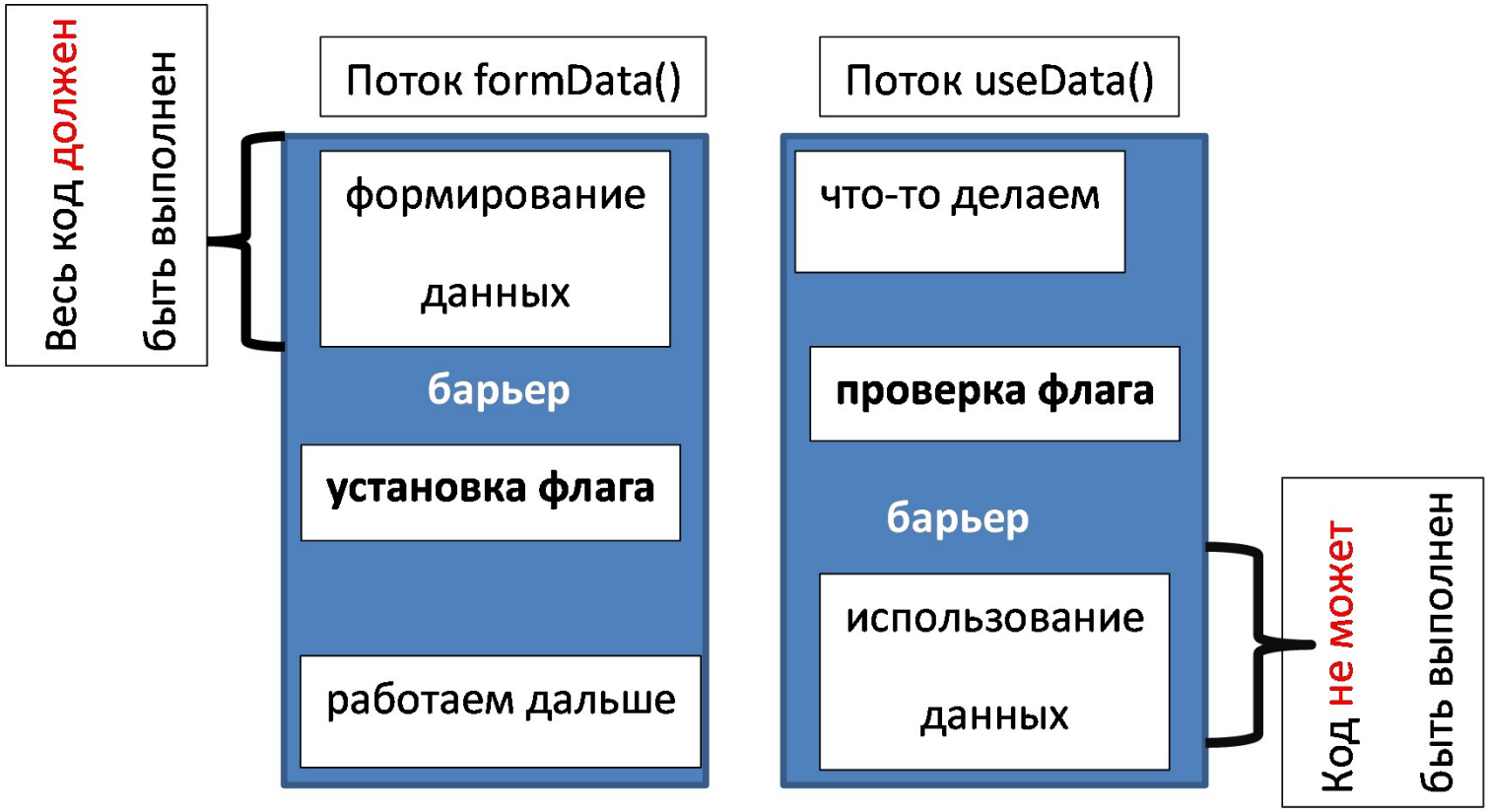
Проблемы?

int valGlobal;
bool readyFlag = false;

```
//поток, формирующий данные
void formData()
{
   //…
    valGlobal = 33;
    readyFlag=true;
  //…
}
```

```
//поток, обрабатывающий данные
void useData()
{
    while (!readyFlag)
    {  /*что-то делаем*/   }
    assert(valGlobal==33);
   //…
}
```

Требуется:


Весь код должен быть выполнен
Поток formData()
формирование данных
барьер
установка флага
работаем дальше
Поток useData()
что-то делаем
проверка флага
барьер
использование данных
Код не может быть выполнен

Пример – синхронизация и принудительное упорядочение

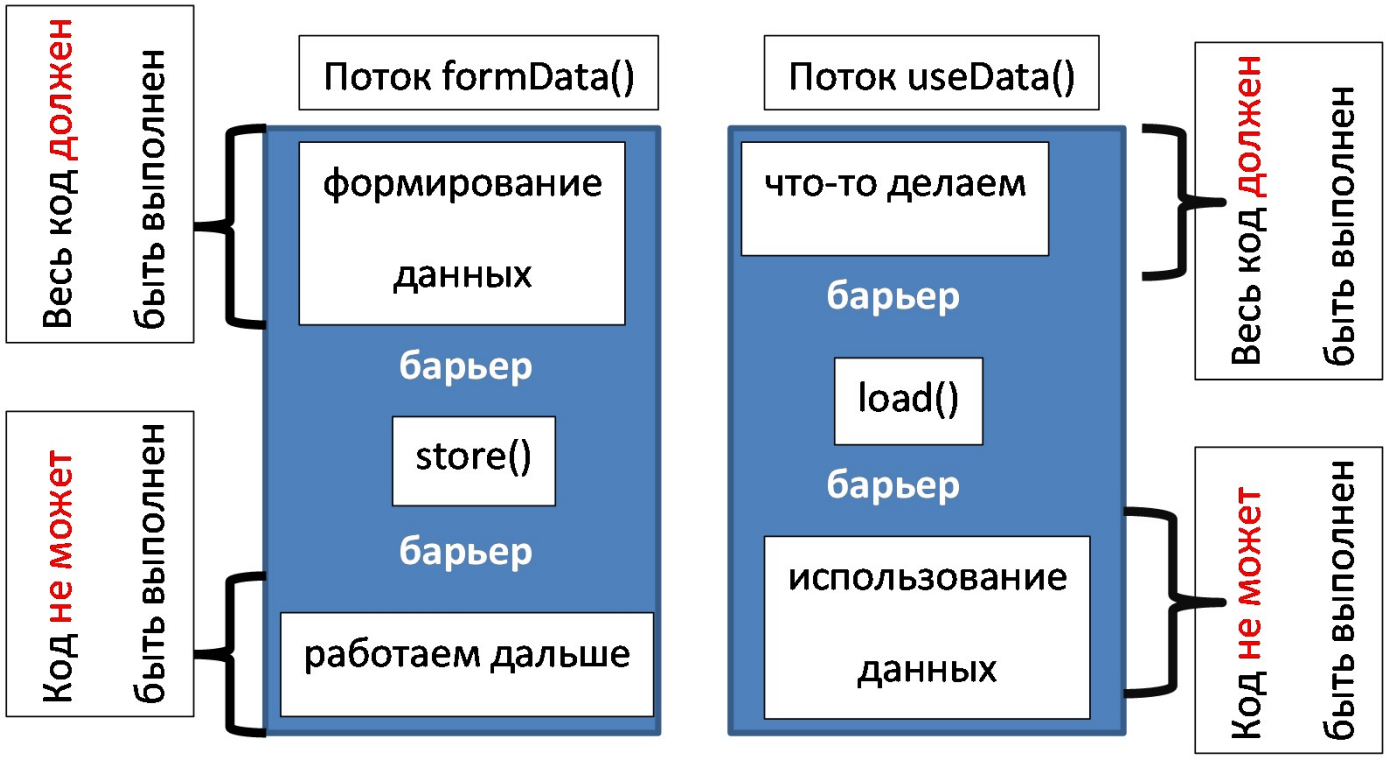
int valGlobal; //неатомарные данные

std::atomic<bool> readyFlag(false);

```
//поток, формирующий данные
void formData()
{
   //…
    valGlobal = 33;
    readyFlag.store(true,
          std::memory_order_seq_cst);
  //…
}
```

```
//поток, обрабатывающий данные
void useData()
{
    while (!readyFlag.load(
std::memory_order_seq_cst))
    {  /*что-то делаем*/   }
    assert(valGlobal==33);
   //…
}
```

А на самом деле ограничения гораздо сильнее



Пример (Вилльямс):

```
std::atomic<bool> x(false),y(false);
std::atomic<int> z = {0};

void write_x() {x.store(true, std::memory_order_seq_cst);}
void write_y(){y.store(true, std::memory_order_seq_cst);}

void read_x_then_y(){
  while (!x.load(std::memory_order_seq_cst)){std::cout << 'x';}
  if (y.load(std::memory_order_seq_cst)) {++z;}
}

void read_y_then_x(){
  while (!y.load(std::memory_order_seq_cst)){ std::cout << 'y'; }
  if (x.load(std::memory_order_seq_cst)) { ++z; }
}
```

Продолжение

```
int main(){
        std::thread a(write_x);
        std::thread b(write_y);
        std::thread c(read_x_then_y);
        std::thread d(read_y_then_x);

        a.join(); b.join(); c.join();  d.join();

        // z = ???
}
```

memory_order_relaxed

- нет ограничений на переупорядочение неатомарных операций вокруг атомарной в одном потоке =>
  - эффективность высокая
  - но используется только тогда, когда
    упорядочение действительно не требуется
    или упорядочение программист обеспечивает «вручную» -
    std::atomic_thread_fence()
- единственное ограничение – операции доступа **к одной и той же атомарной переменной** внутри потока переупорядочивать нельзя!

Специфика memory_order_relaxed

- обеспечивается только атомарность выполнения
- операции в этом режиме не поддерживают отношение «синхронизируется с»
- для атомарной relaxed-записи стандартом запрещена спекулятивная запись
- ограничение: атомарные операции доступа над **одним и тем же объектом** в данном потоке нельзя переупорядочить

Хороший пример слабого упорядочения

std::atomic<int> atomicCount(0);	
//Поток 1 … atomicCount. fetch_add(1, std::memory_order_**relaxed**); …	//Поток 2 … atomicCount. fetch_add(1, std::memory_order_**relaxed**); …

Пример:

```
std::atomic<bool> x(false),y(false);
std::atomic<int> z = {0};


void write_x_then_y(){
   x.store(true,std::memory_order_relaxed);
   y.store(true,std::memory_order_relaxed);
}

void read_y_then_x(){
  while (!y.load(std::memory_order_relaxed)){std::cout << 'y'; }
   if (x.load(std::memory_order_relaxed)) { ++z; }
}
```

Продолжение

```
int main()
{
        std::thread a(write_x_then_y);
        std::thread b(read_y_then_x);
        a.join(); b.join();

        // z = ???
}
```

Упорядочение захват/освобождение acquire/release

- атомарные операции load() – захват (acquire)
- атомарные операции store() - освобождение (release)
- атомарные операции exchange() – и захват, и освобождение

Модель захват-освобождение acquire - release

- в отличие от слабого упорядочения предоставляет **некоторую** попарную синхронизацию между потоком, захватившим ресурс, и потоком, освободившим ресурс
- операции load() – захват (memory_order_acquire) – чтение из памяти, операции store() – освобождение (memory_order_release) – запись в память
- атомарные операции чтение-модификация-запись (memory_order_acq_rel) – exchange(), fetch_add()… -

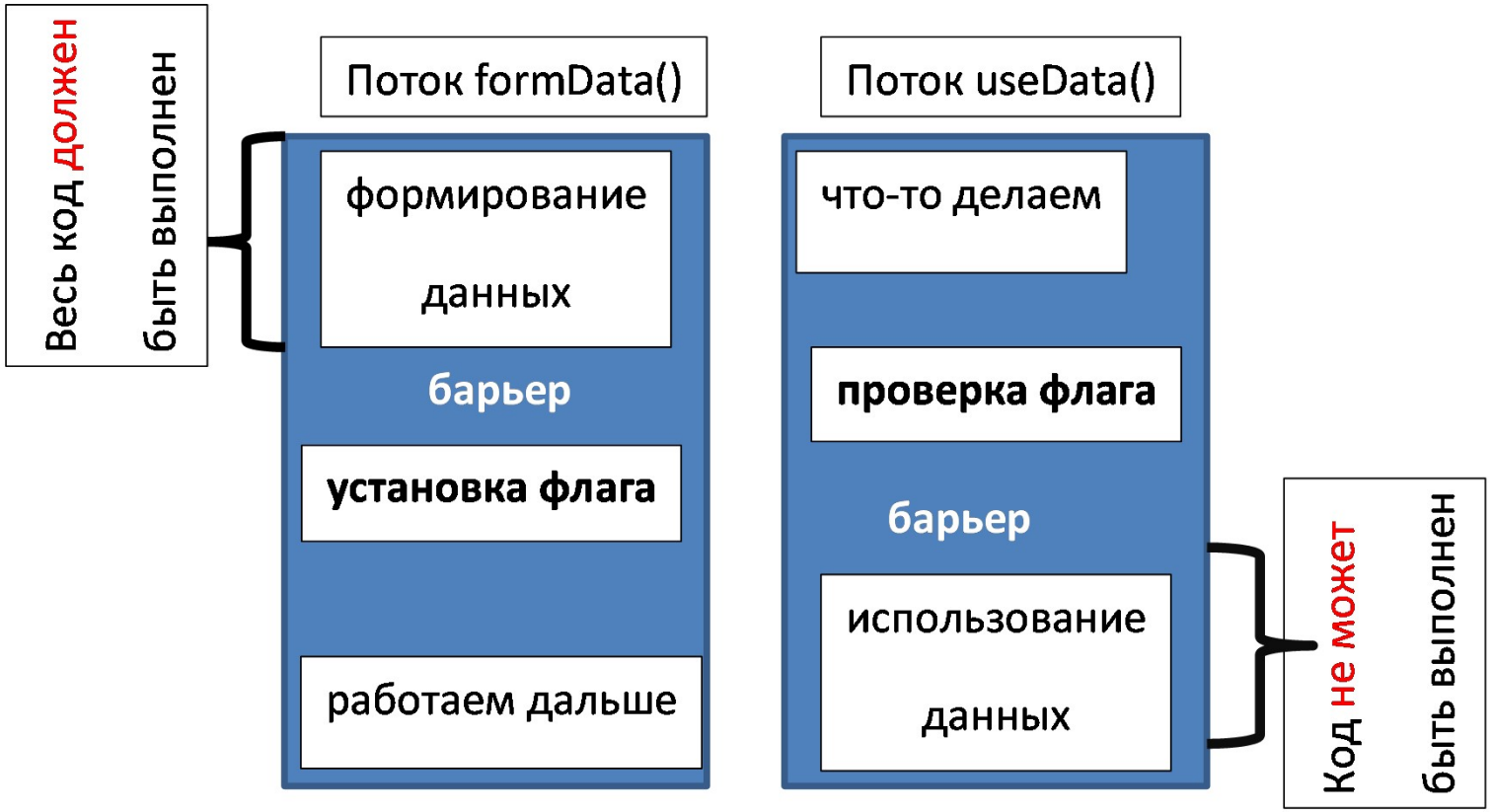
Отличия от memory_order__seq_cst

- memory_order__seq_cst – глобальное упорядочение
- memory_order_acquire, memory_order_release, memory_order_acq_rel
  попарное упорядочение между двумя потоками

Правила *acquire| release*:

- предшествующие операции записи не могут выполняться после *store() – установка барьера записи (release)!*
- следующие операции как чтения, так и записи не могут выполниться до *load() – установка барьера чтения (acquire)!*

Модифицируем пример:



Пример – *acquire| release*

int valGlobal; //неатомарные данные
std::atomic<bool> readyFlag(false);

```
//поток, формирующий данные
void formData()
{
   //…
    valGlobal = 33;
    readyFlag.store(true,
          std::memory_order_release);
  //…
}
```

```
//поток, обрабатывающий данные
void useData()
{
    while (!readyFlag.load(
std::memory_order_acquire))
    {  /*что-то делаем*/   }
    assert(valGlobal==33);
   //…
}
```

Пример посложнее

```
std::atomic<bool> x(false),y(false);
std::atomic<int> z = {0};

void write_x() {…x.store(true, std::memory_order_release);…}
void write_y(){…y.store(true, std::memory_order_release);…}

void read_x_then_y(){
  while (!x.load(std::memory_order_aquire)){std::cout << 'x';}
  if (y.load(std::memory_order_aquire)) {++z;}
}

void read_y_then_x(){
  while (!y.load(std::memory_order_aquire)){ std::cout << 'y'; }
  if (x.load(std::memory_order_aquire)) { ++z; }
}
```

Продолжение

```
int main()
{
        std::thread a(write_x);
        std::thread b(write_y);
        std::thread c(read_x_then_y);
        std::thread d(read_y_then_x);
        a.join(); b.join(); c.join();    d.join();

        // z = ???
}
```

std::memory_order_consume ??? – C17 deprecated?

- специализированная разновидность **acquire**-release семантики
- вводит новые отношения упорядочения – «по данным»:
  - предшествует-по-зависимости (dependency-ordered-before)
  - переносит-зависимость-в (carries-a-dependency-to)
- самое важное использование – атомарная загрузка (load()) указателя

Использование std::memory_order_consume с указателями

std::atomic<int*> pVal; int val;	

void write_val(){
val = 33;
pVal.store(&val,
  std::**memory_order_release**);
}
	void read_val(){
	int* p;
	while (!(p=pVal.load(
	  std::**memory_order_consume**))) {}
	int n = *p; //гарантирует: read/write операции с «указываемыми» данными не будут переупорядочены перед load. Для всех остальных данных переупорядочение возможно
	}

Пример std::memory_order_consume поинтереснее

```
struct X {
int i;
std::string s;
…
};
//глобальные данные
std::atomic<X*> p;
std::atomic<int> a;
```

Продолжение примера std::memory_order_consume

```
void create_x(){
  X* x = new X(33, "abc“);
  a.store(1, std::memory_order_relaxed);
  p.store(x, std::memory_order_release);
}
void use_x(){
  X* x=nullptr;
  while(!(x=p.load(std::memory_order_consume))){}
  bool b1 = (x->i == 33);   //гарантировано true
  bool b2 = (x->s == "abc");   //гарантировано true
  int n = a.load(std::memory_order_relaxed); // a.load() может быть
                                              переупорядочено до p.load()
  bool b3 = n == 1; //никаких гарантий
}
```

STD::ATOMIC_THREAD_FENCE()

Барьеры можно ставить «вручную» std::atomic_thread_fence()

- C++11
  atomic_thread_fence() – запрет переупорядочения
  atomic_signal_fence() - запрет переупорядочения + синхр. кеша

- GCC
  __asm volatile("mfence" ::: "memory"); //аппаратный барьер памяти - процессору
  __asm volatile("" ::: "memory"); //программный барьер памяти - компилятору

Пример:

```
data.store(3, std::memory_order_relaxed);
std::atomic_thread_fence(std::memory_order_release);
flag.store(1, std::memory_order_relaxed);
flag2.store(2, std::memory_order_relaxed);
```

не эквивалентно:

```
data.store(3, std::memory_order_relaxed);
flag.store(1, std::memory_order_release);
flag2.store(2, std::memory_order_relaxed);
```

так как flag2.store() может быть перемещен перед data.store()

C++20 - Latches and Barriers

М.Полубенцева