

# Современные средства разработки ПО

## Модель памяти C++

Фетисов Михаил Вячеславович  
[fetisov.michael@bmstu.ru](mailto:fetisov.michael@bmstu.ru)

# Суровая реальность современного состояния ИТ

- Компьютер исполняет не тот код, который мы ему написали, и даже не тот, который передали на исполнение.

# Рассмотрим пример

## В каком случае может быть нарушен assert?

```
// FooBar.cpp
#include <iostream>
#include <thread>
#include <cassert>

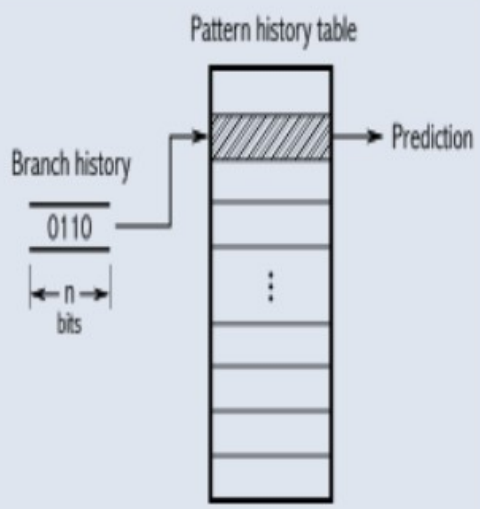
bool x = false;
bool y = false;

void foo() {
    x = true;
    y = true;
}

void bar() {
    while(!y);
    assert(x);
}
```

- foo и bar выполняются в разных потоках

# Как выполняется код

Code	Compiler	CPU	Cache																
<pre>void foo(int n, int m) {     int x=1;     for (         int i=0;         i&lt;n;         ++i     ){         if (m&lt;0) {             x -= x*m;         }     }     return x; }</pre>	<pre>mov     esi,1 test    ecx,ecx jle     011F1301 lea     ebx,[ebx] mov     eax,1 sub     eax,edx imul    esi,eax dec     ecx jne     011F12F0 ...</pre>		<table><tr><td>000</td><td>004</td></tr><tr><td>001</td><td>005</td></tr><tr><td>222</td><td>006</td></tr><tr><td>003</td><td>007</td></tr><tr><td>008</td><td>00C</td></tr><tr><td>999</td><td>00D</td></tr><tr><td>AAA</td><td>00E</td></tr><tr><td>00B</td><td>00F</td></tr></table>	000	004	001	005	222	006	003	007	008	00C	999	00D	AAA	00E	00B	00F
000	004																		
001	005																		
222	006																		
003	007																		
008	00C																		
999	00D																		
AAA	00E																		
00B	00F																		
	<ul style="list-style-type: none"><li>• Register allocation</li><li>• Loop unswitching</li></ul>	<ul style="list-style-type: none"><li>• Branch prediction</li><li>• Out of order exec.</li></ul>	<ul style="list-style-type: none"><li>• Prefetching</li><li>• Buffering</li></ul>																

Optimization

# Как исполняется код

## Пояснение

- Оптимизация (как со стороны компилятора, так и ЦПУ) может нарушать «наивную» модель последовательности выполнения.
- Необходимо договориться о правилах управления выполнением многопоточного кода, которые позволяли бы писать *простые, надёжные*, но также *оптимизируемые* программы.
- Со стандарта C++11 появилась *модель памяти C++*, которая решает эту задачу.

# Модель памяти

## Определение

- **Модель памяти** — это договорённость о том, что:
  - разработчик программы предотвращает гонку данных, а
  - компилятор гарантирует *строгость последовательности выполнения*.



# Строгость последовательности выполнения

## Определение

- **Строгость последовательности выполнения** (sequentially consistent) — это результат любого выполнения, как если бы:
  - операции всех потоков выполняются в некотором последовательном порядке;
  - операции каждого потока проявляются в этой последовательности в порядке, указанном программой.
- Например, если данные одного потока нужны в другом потоке, и используется какая-либо синхронизация потоков, то компилятор гарантирует *строгость последовательности выполнения*.

# Механизмы синхронизации потоков

## Мьютекс (см. соотв. лекцию)

```
std::list<unsigned long> li;  
std::mutex                li_mutex;  
void add_range(unsigned long from, unsigned long to) {  
    std::lock_guard<std::mutex> guard(li_mutex);  
    for(unsigned long i = from; i < to; ++i)  
        li.push_back(i);  
}
```



# Механизмы синхронизации потоков

## Переменные условия (см. соотв. лекцию)

```
void waitingForWork() {
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> locker(cond_var_mutex);
    cond_var.wait(locker, []{ return is_data_ready; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}

void setDataReady() {
    {
        std::lock_guard<std::mutex> locker(cond_var_mutex);
        is_data_ready = true;
    }
    std::cout << "Sender: Data is ready." << std::endl;
    cond_var.notify_one();
}
```

# Механизмы синхронизации потоков

## `std::atomic<>`

- Наиболее тонкий, сложный и низкоуровневый механизм синхронизации и реализации *атомарных операций*.
- Переменные, защищённые от гонки данных.
- Обеспечивает синхронизацию потоков:
  - метод **store** присваивает значение атомарной переменной и *управляемым образом* синхронизируется с
  - методом **load**, который обеспечивает чтение сохранённого значения.
- Обеспечивает строгость последовательности выполнения.
- Требуется поддержка процессора — не все обеспечивают lock-free операции.

# Атомарная операция

## Определение

- **Атомарная операция** — это операция, которую невозможно наблюдать в промежуточном состоянии, она либо выполнена либо нет.
- Атомарные операции могут состоять из нескольких операций.

# std::atomic<>

## Атомарные операции

- Атомарные операции:
  - load,
  - store,
  - fetch\_add,
  - compare\_exchange\_\* и другие.
- Последние две операции — это read-modify-write операции, атомарность которых обеспечивается специальными инструкциями процессора.
- *При записи атомарных операций можно управлять условиями (порядком) синхронизации.*

# Условие (порядок) синхронизации

## `std::memory_order_relaxed`

- Он гарантирует только свойство атомарности операций, при этом не может участвовать в процессе синхронизации данных между потоками.
- Свойства:
  - модификация переменной "появится" в другом потоке не сразу;
  - поток `thread2` "увидит" значения одной и той же переменной в том же порядке, в котором происходили её модификации в потоке `thread1`;
  - порядок модификаций разных переменных в потоке `thread1` не сохранится в потоке `thread2`.

# std::memory\_order\_relaxed

## Использование в качестве счетчика

```
#include <atomic>
#include <iostream>

std::atomic<size_t> counter{ 0 };

// process can be called from different threads
void process(/*Request req*/) {
    counter.fetch_add(1, std::memory_order_relaxed);
    // ...
}

void print_metrics() {
    std::cout << "Number of requests = " << counter.load() << std::endl;
    // ...
}
```



# std::memory\_order\_relaxed

## Использование в качестве флага остановки

```
// memory_order_relaxed_2.cpp
#include <atomic>
#include <iostream>

std::atomic<bool> stopped{ false };

void thread1() {
    while (!stopped.load(std::memory_order_relaxed)) {
        // ...
    }
}

void stop_thread1() {
    stopped.store(true, std::memory_order_relaxed);
}
```

В данном примере не важен порядок в котором thread1 увидит изменения из потока, вызывающего stop\_thread1.

Также не важно то, чтобы thread1 мгновенно (синхронно) увидел выставление флага stopped в true.

# Условие (порядок) синхронизации `std::memory_order_seq_cst`

- Наиболее строгий порядок синхронизации.
- Свойства:
  - порядок модификаций разных атомарных переменных в потоке `thread1` сохранится в потоке `thread2`;
  - все потоки будут видеть один и тот же порядок модификации всех атомарных переменных (сами модификации могут происходить в разных потоках);
  - все модификации памяти (не только модификации над атомиками) в потоке `thread1`, выполняющей `store` на атомарной переменной, будут видны после выполнения `load` этой же переменной в потоке `thread2`.
- Таким образом можно представить `seq_cst` операции, как барьеры памяти, в которых состояние памяти синхронизируется между всеми потоками программы.
- Этот флаг синхронизации памяти в C++ используется по умолчанию, т.к. с ним меньше всего проблем с точки зрения корректности выполнения программы.

# Механизмы синхронизации потоков

## std::atomic<>

```
// SpinLock.cpp
#include <atomic>
#include <thread>

class Spinlock {
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
public:
    void lock() {
        while( flag.test_and_set() );
    }

    void unlock() {
        flag.clear();
    }
};
```

```
Spinlock spin;

void workOnResource() {
    spin.lock();
    // shared resource
    spin.unlock();
}

int main() {
    std::thread t1(workOnResource);
    std::thread t2(workOnResource);

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

# std::atomic\_flag без спинлока

```
// ThreadSynchronisationAtomicFlag.cpp
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> my_vec{};
std::atomic_flag flag{};

void prepareWork() {
    my_vec.insert(my_vec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared." << std::endl;
    flag.test_and_set();
    flag.notify_one();
}

void completeWork() {
    std::cout << "Waiter: Waiting for data." << std::endl;
    flag.wait(false);
    my_vec[2] = 2;
    std::cout << "Waiter: Complete the work." << std::endl;
    for (auto i: my_vec) std::cout << i << " ";
}

int main() {
    std::thread t1(prepareWork);
    std::thread t2(completeWork);

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

# std::memory\_order\_release

## std::memory\_order\_acquire

- Флаг синхронизации памяти acquire/release является более тонким способом синхронизировать данные между парой потоков (работают только в паре над одним атомарным объектом).
- Свойства:
  - модификация атомарной переменной с release будет видна в другом потоке, выполняющем чтение этой же атомарной переменной с acquire;
  - все модификации памяти в потоке thread1, выполняющей запись атомарной переменной с release, будут видны после выполнения чтения той же переменной с acquire в потоке thread2;
  - процессор и компилятор не могут перенести операции записи в память раньше release операции в потоке thread1, и нельзя перемещать выше операции чтения из памяти позже acquire операции в потоке thread2.



# std::memory\_order\_release

## std::memory\_order\_acquire

- Используя `release`, мы даем инструкцию, что данные в этом потоке готовы для чтения из другого потока.
- Используя `acquire`, мы даем инструкцию "подгрузить" все данные, которые подготовил для нас первый поток.
- Но если мы делаем `release` и `acquire` на разных атомарных переменных, то получим UB вместо синхронизации памяти.



# std::memory\_order\_release

# std::memory\_order\_acquire

```
#include <atomic>

class mutex {
public:
    void lock() {
        bool expected = false;
        while(!_locked.compare_exchange_weak(expected, true, std::memory_order_acquire)) {
            expected = false;
        }
    }

    void unlock() {
        _locked.store(false, std::memory_order_release);
    }

private:
    std::atomic<bool> _locked;
};
```

## Дополнительные источники по условиям (порядку) синхронизации с использованием атомарных переменных

- `std::atomic`. Модель памяти C++ в примерах
  - <https://habr.com/ru/post/517918/>
- C++ Memory Model
  - [https://www.think-cell.com/en/career/talks/pdf/think-cell\\_talk\\_memorymodel.pdf](https://www.think-cell.com/en/career/talks/pdf/think-cell_talk_memorymodel.pdf)

# Временные характеристики задержек и пропускной способности систем

■ 1 ns

■ L1 cache reference: 0.5 ns

■ Branch mispredict: 5 ns

■ L2 cache reference: 7 ns

■ Mutex lock/unlock: 25 ns

■ = 100 ns

■ Main memory reference: 100 ns

■ = 1  $\mu$ s

■ Compress 1 KB with Zippy: 3  $\mu$ s

■ = 10  $\mu$ s

■ Send 1 KB over a Gbps network: 10  $\mu$ s

■ SSD random read (1 Gb/s SSD): 150  $\mu$ s

■ Read 1 MB sequentially from memory: 250  $\mu$ s

■ Round trip in same datacenter: 500  $\mu$ s

■ = 1 ms

■ Read 1 MB sequentially from SSD: 1 ms

■ Disk seek: 10 ms

■ Read 1 MB sequentially from disk: 20 ms

■ Packet roundtrip CA to Netherlands: 150 ms



# Параллельные алгоритмы в стандартной библиотеке

- С выходом стандарта C++ 17 появились 69 новых перегрузок для имеющихся алгоритмов и 8 полностью новых алгоритмов, способных работать параллельно.
- Новые перегрузки и новые алгоритмы принимают в качестве аргумента так называемую политику выполнения.
- С помощью политики выполнения программист может сообщить реализации, должен ли алгоритм выполняться последовательно, параллельно или параллельно с векторизацией.

# Параллельные алгоритмы

## Пример

- L04/ParallelSort.cpp

# Вопросы?

Фетисов Михаил Вячеславович  
[fetisov.michael@bmstu.ru](mailto:fetisov.michael@bmstu.ru)