The C++ Memory Model

Rainer Grimm

Training, Coaching and Technology Consulting

www.grimm-jaud.de

Multithreading with C++

C++'s answers to the requirements of the multicore architectures.

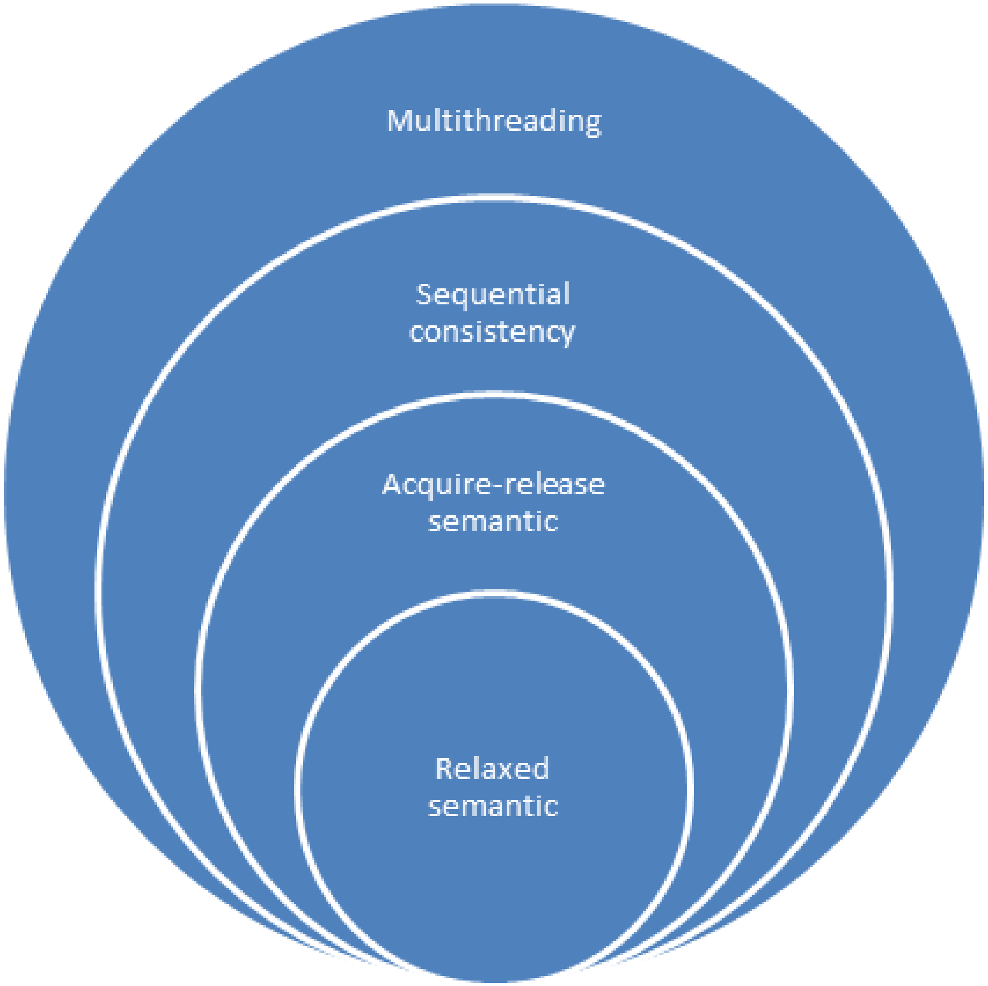
A well defined memory model

- Atomic Operations
- Partial ordering of operations
- Visible effects of operations

A standardized threading interface

- Threads and tasks
- Protection and safe initialization of shared data
- Thread local data
- Synchronization of threads

Expert Levels



The C++ Memory Model

The Contract

Atomics

Synchronization and Ordering Constraints

Singleton Pattern

The C++ Memory Model

The Contract

Atomics

Synchronization and Ordering Constraints

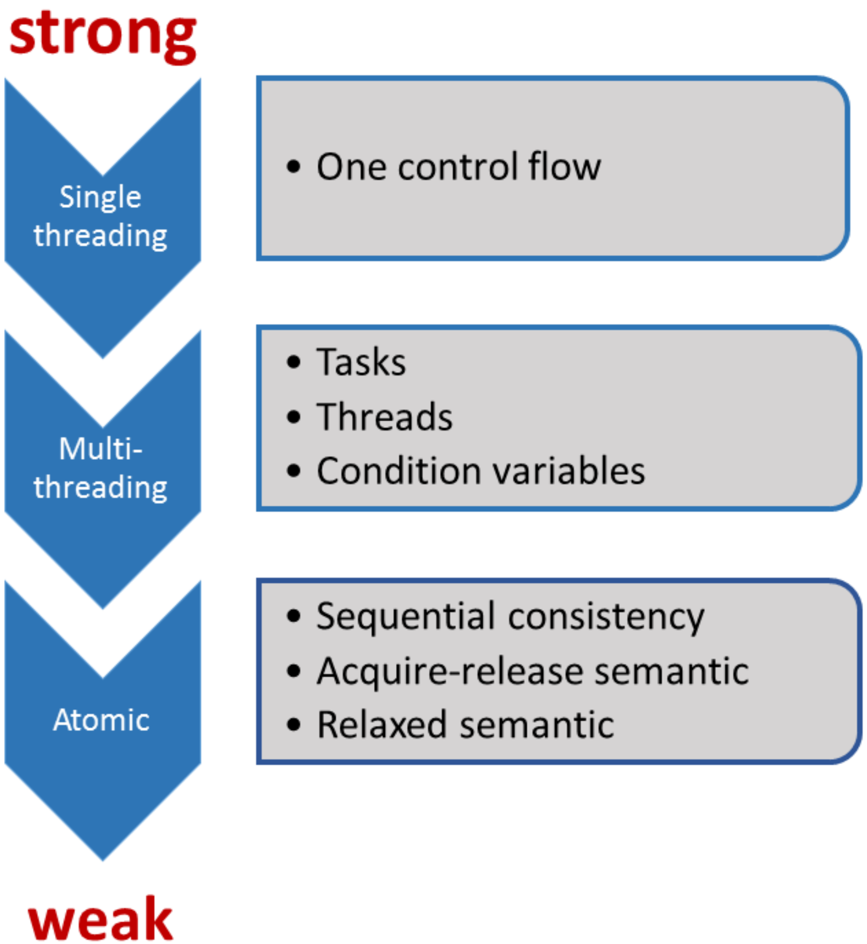
Singleton Pattern

The Contract

- Developer follow the rules
  - Atomic operations
  - Partial ordering of operations
  - Visible effects of operations
- System wants to optimize
  - Compiler
  - Processor
  - Memory system

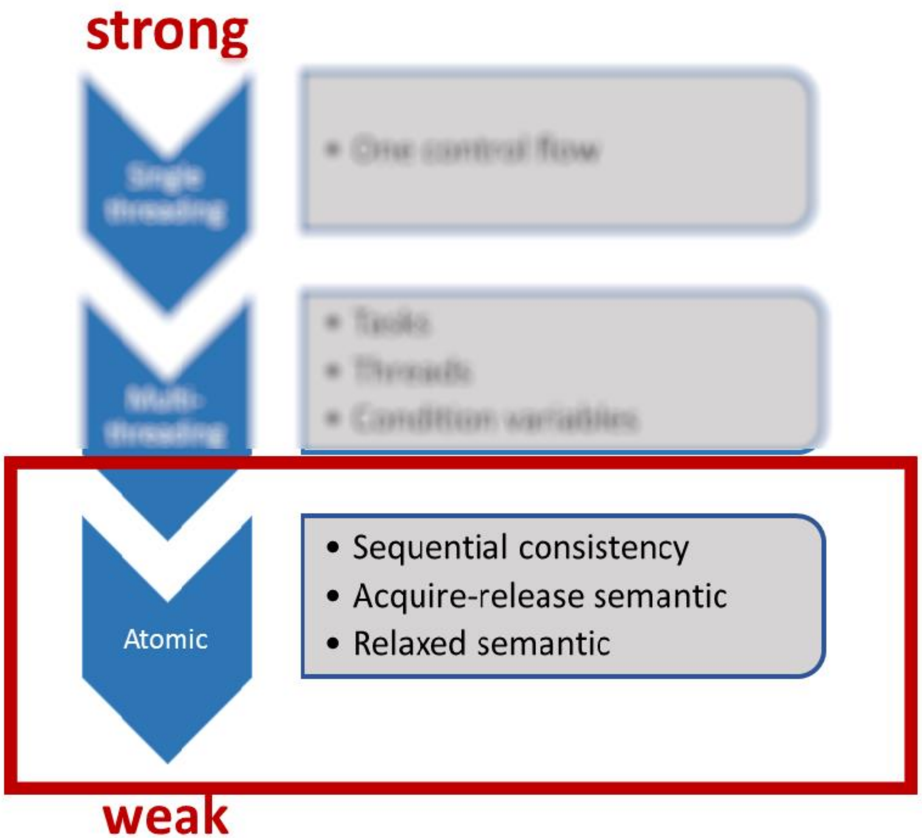
Highly optimized program.
Tailored for the architecture.

The Contract



- More optimization possibilities for the system
- Number of potential interleaving's grows exponentially
- More and more the domain of experts
- Break of the intuition
- Area of micro optimization

The Contract



- Sequential consistency
  - *Strong memory model*
  - Universal clock

Break of the sequential consistency

- Acquire-release semantic
  - Synchronization of atomics (between threads)
- Relaxed semantic
  - *Weak Memory Model*
  - Weak guarantees

The C++ Memory Model

The Contract

Atomics

Synchronization and Ordering Constraints

Singleton Pattern

Atomics

Atomics are the foundation of C++ memory model.

Atomic operations on atomics define the synchronization and ordering constraints.

- Synchronization and ordering constraints holds for atomics and non-atomics.
- Synchronization and ordering constraints are used by the high level threading interface.
  - Threads and tasks
  - Mutexe and locks
  - Condition variables
  - ...

Atomics: std::atomic_flag

The atomic flag `std::atomic_flag`

- has a very simple interface
  - `clear` and `test_and_set`

- is the only lock-free data structure.

All other atomics for integral types, pointer, and user defined atomics can internally use a lock.

- is the building block for higher abstractions.

Spinlock

Atomics: std::atomic_flag

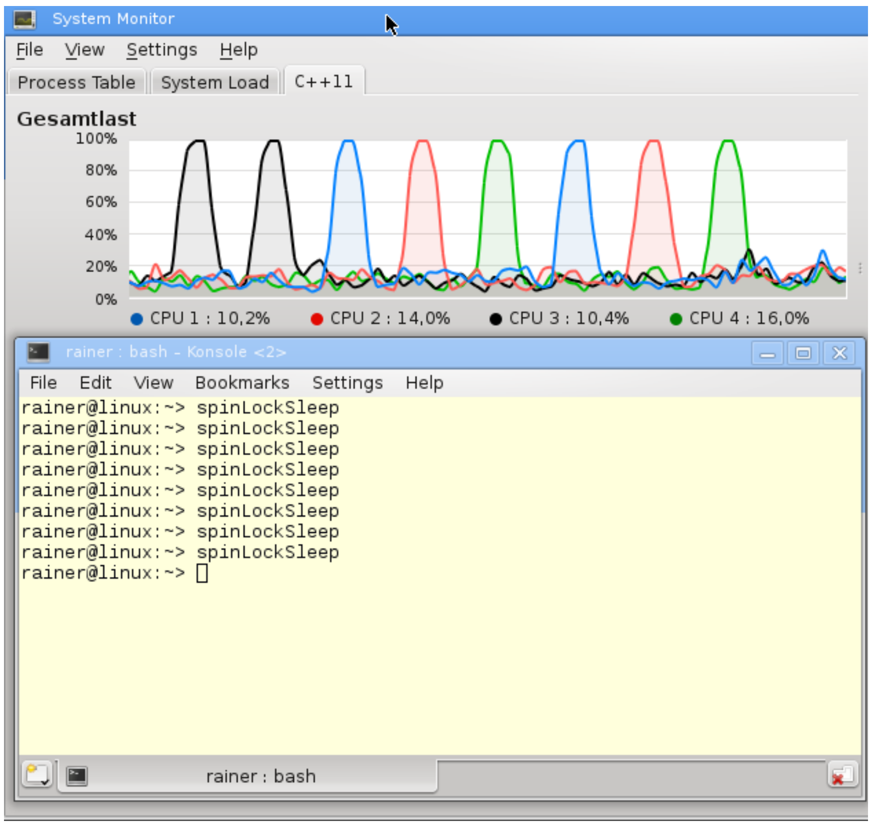
Spinlock

```
class Spinlock{
 std::atomic_flag  flag;
public:

Spinlock():flag(ATOMIC_FLAG_INIT){}

  void lock(){
    while(flag.test_and_set());
  }

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

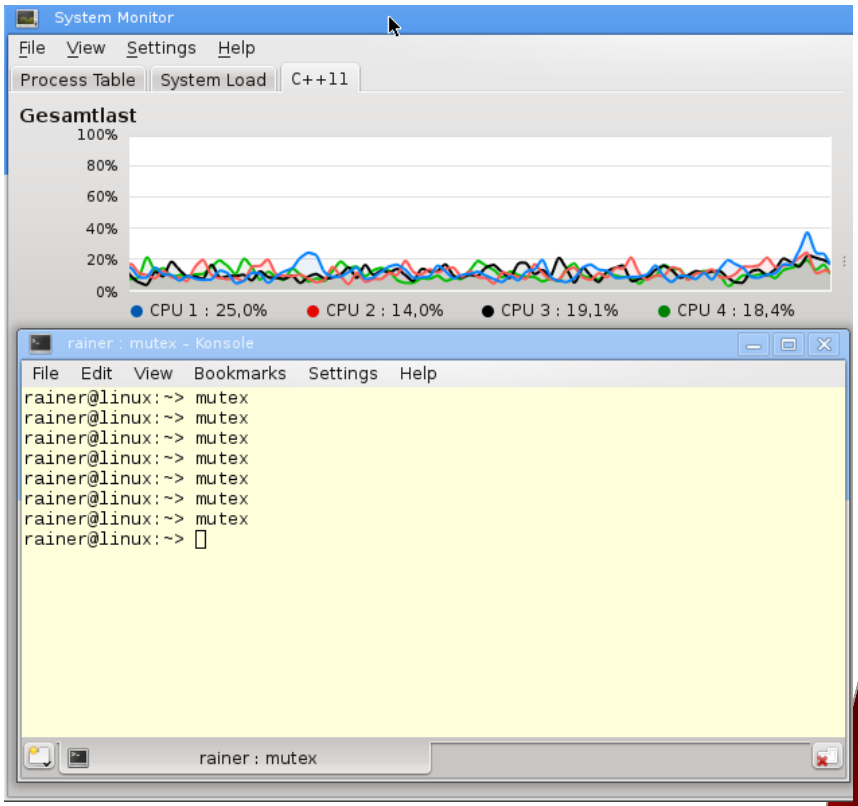
```
Spinlock spin;
// Mutex spin;
void workOnResource(){
  spin.lock();
  sleep_for(seconds(2));
  spin.unlock();
}
int main({
  thread t(workOnResource);
  thread t2(workOnResource);
  t.join();
  t2.join();
}
```

Atomics: std::atomic_flag

Spinlock



Mutex



Atomics: std::atomic<bool>

The atomic Boolean `std::atomic<bool>`

- can explicitly set to `true` or `false`.
- supports the function `compare_exchange_strong`.
  - Fundamental function for atomic operations.
  - Compares and sets a value in a atomic operation.
  - **Syntax**: `bool compare_exchange_strong(exp, des)`
  - **Strategy**: `atom.compare_exchange_strong(exp, des)`

    `*atom == exp` `*atom= des;` returns `true`

    `*atom != exp` `exp= *atom;` returns `false`
- can be used for implementing a condition variable.

Atomics: Condition Variable

```
std::vector<int> mySharedWork;
std::mutex mutex_;
std::condition_variable condVar;

bool dataReady;

void setDataReady(){
  mySharedWork={1 ,0, 3};
  {
    lock_guard<mutex> lck(mutex_);
    dataReady=true;
  }
  condVar.notify_one();
}
```

```
void waitingForWork(){
  unique_lock<mutex> lck(mutex_);
  condVar.wait(lck, []{ return dataReady; });
  mySharedWork[1]= 2;
}

int main(){
  thread t1(waitingForWork);
  thread t2(setDataReady);

  t1.join();
  t2.join();

   for (auto v: mySharedWork){
      std::cout << v << " ";
  } // 1 2 3
}
```

Atomics: std::atomic<bool>

```
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void setDataReady(){
  mySharedWork={1,0,3};
  dataReady= true;
}

void waitingForWork(){
  while (!dataReady.load()){
     sleep_for(milliseconds(5))
  }
  mySharedWork[1]= 2;
}
```

```
int main(){
  thread t1(waitingForWork);
  thread t2(setDataReady);
  t1.join();
  t2.join();
  for (auto v: mySharedWork){
    cout << v << " ";
  }
}  // 1 2 3
```

sequenced-before

synchronizes-with

Atomics: std::atomic

All further atomics are partially or fully specializations of `std::atomic`.

```
std::atomic<T*>
std::atomic<Integral type>
std::atomic<User-defined type>
```

- There are restrictions on user-defined types
  - Its copy-assignment operator and that of the base classes must be trivial.
  - They can not have virtual methods or virtual base classes.
  - They must be bitwise comparable.

Atomics: std::atomic

Operation	read operation	write operation	read-modify-write operation
`test_and_set`			✓
`clear`		✓	
`is_lock_free`	✓		
`load`	✓		
`store`		✓	
`exchange`			✓
`compare_exchange_weak` `compare_exchange_strong`			✓
`fetch_add, +=` `fetch_sub, -=`			✓

`fetch_or, |=`
`fetch_and, &=`
`fetch_xor, ^=`
			✓

`++` `--`			✓

! There is no multiplication or division.

Atomics: std::atomic

```
template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
  T oldValue= shared.load();
  while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
  return oldValue;
}
int main(){
  std::atomic<int> myInt{5};
  std::cout << myInt << std::endl;
  fetch_mult(myInt,5);
  std::cout << myInt << std::endl;
}
```

VS2013 x64 Native Tools Command Prompt

```
C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>fetch_mult.exe
5
25

C:\Users\Rainer\MicrosoftVisualStudio\fetch_mult>
```

The C++ Memory Model

The Contract

Atomics

Synchronization and Ordering Constraints

Singleton Pattern

Synchronization and Ordering

C++ has six different memory models.

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

- Sequential consistency is the default.
  - The memory model for C# and Java.
  - `memory_order_seq_cst`
  - Implicit argument for atomic operations.

```
std::atomic<int> shared;
shared.load() ≅ shared.load(std::memory_order_seq_cst);
```

Synchronization and Ordering

To get a systematic in the memory model you have to answer two questions.

1. For which kind of operations should you use which memory model?

2. Which synchronization and ordering constraints are defined by the various memory models?

Synchronization and Ordering

1. For which kind of operations should you use which memory model?

- **read** operations:

  `memory_order_acquire` and `memory_order_consume`
- **write** operations:

  `memory_order_release`
- **read-modify-write** operations:

  `memory_order_acq_rel` and `memory_order_seq_cst`

! `memory_order_relaxed` doesn't define synchronization and ordering constraints.

Synchronization and Ordering

Operation	read operation	write operation	read-modify-write operation
test_and_set			✓
clear		✓	
is_lock_free	✓		
load	✓		
store		✓	
exchange			✓
compare_exchange_weak compare_exchange_strong			✓
fetch_add, += fetch_sub, -=			✓
fetch_or, |= fetch_and, &= fetch_xor, ^=			✓
++ --			✓

Synchronization and Ordering

2. Which synchronization and ordering constraints are defined by the various memory models?

- **Sequential consistency**
  - Global ordering of all threads

    `memory_order_seq_cst`
- **Acquire-release semantic**
  - Ordering between read and write operations on the same atomic

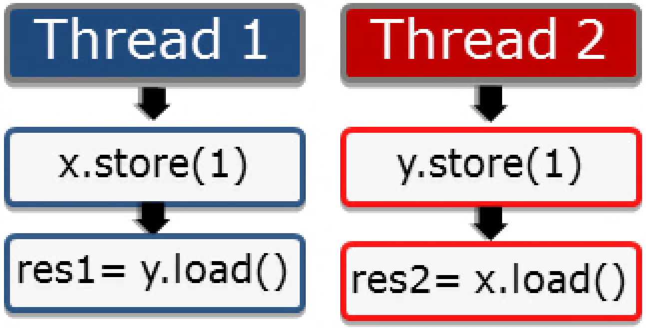
    `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel`
- **Relaxed semantic**
  - No synchronizations and ordering constraints

    `memory_order_relaxed`

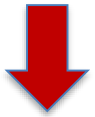
Synchronization and ordering

Sequential consistency(Leslie Lamport 1979)

1. The operations of a program will be executed in source code order.
2. There is a global order of all operations on all threads.

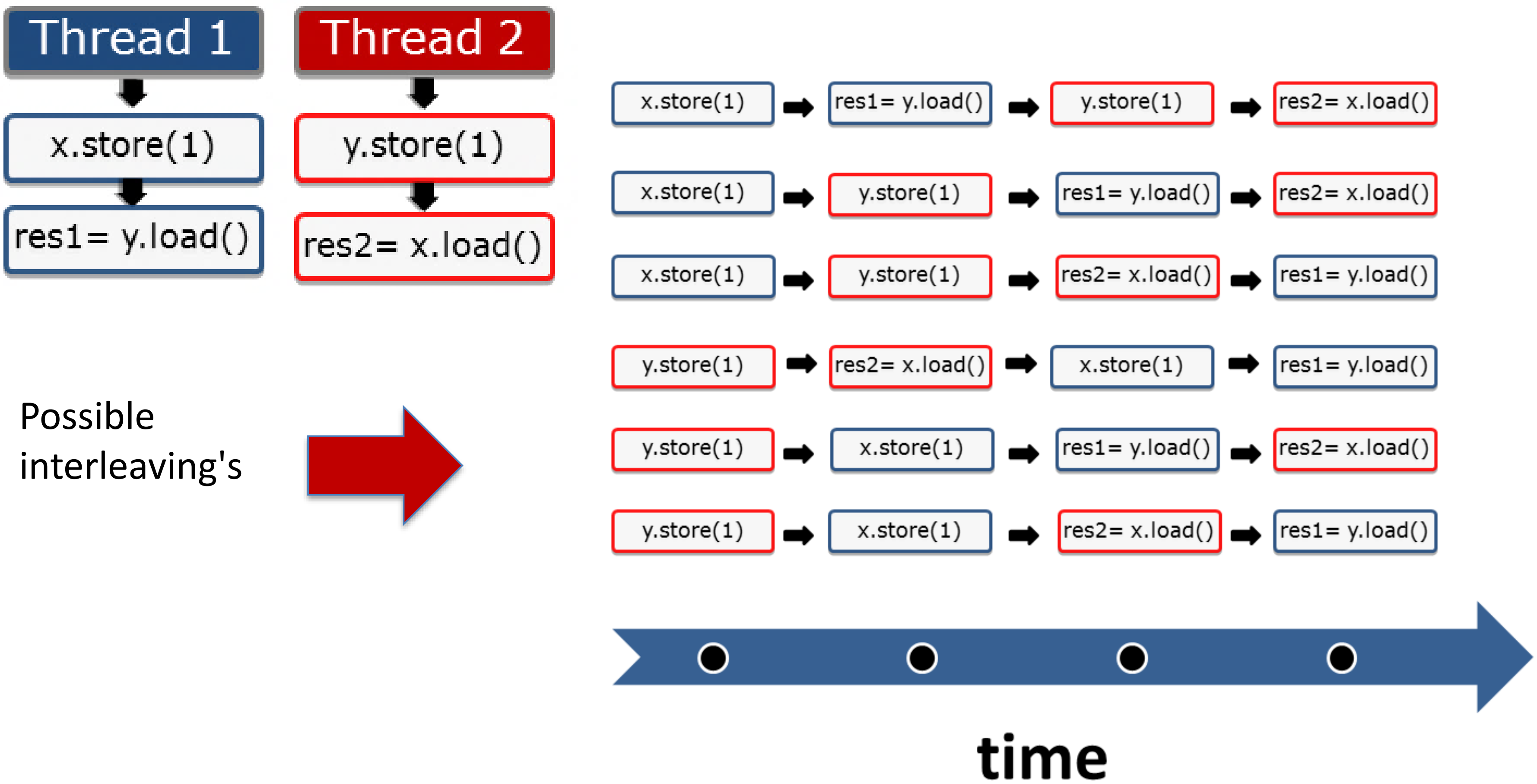


Sequential consistency causes



1. The statements are executed in the source code order.
2. Each thread sees operations of each other thread in the same order (unique clock).

Synchronization and Ordering



Synchronization and Ordering

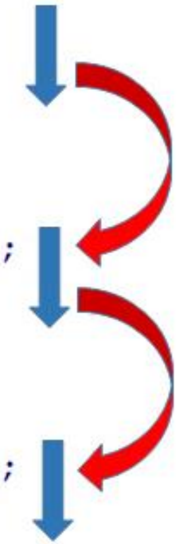
Acquire-release semantic

- A release-operation on a atomic synchronizes with a acquire-operation on the same atomic and establishes in addition an ordering constraint.

- Acquire-operation:
  - Read-operation (`load` or `test_and_set`)
- Release-operation:
  - Write-operation (`store` or `clear`)
- Ordering constraints:
  - Read- and write-operations can not be moved **before** an acquire-operation.
  - Read- and write-operations can not be moved **after** a release-operation.

Synchronization and Ordering

```
void dataProducer(){
    mySharedWork={1,0,3};
    dataProduced.store(true, std::memory_order_release);
}

void deliveryBoy(){
    while( !dataProduced.load(std::memory_order_acquire) );
    dataConsumed.store(true,std::memory_order_release);
}

void dataConsumer(){
    while( !dataConsumed.load(std::memory_order_acquire) );
    mySharedWork[1]= 2;
}
```



Thread 1

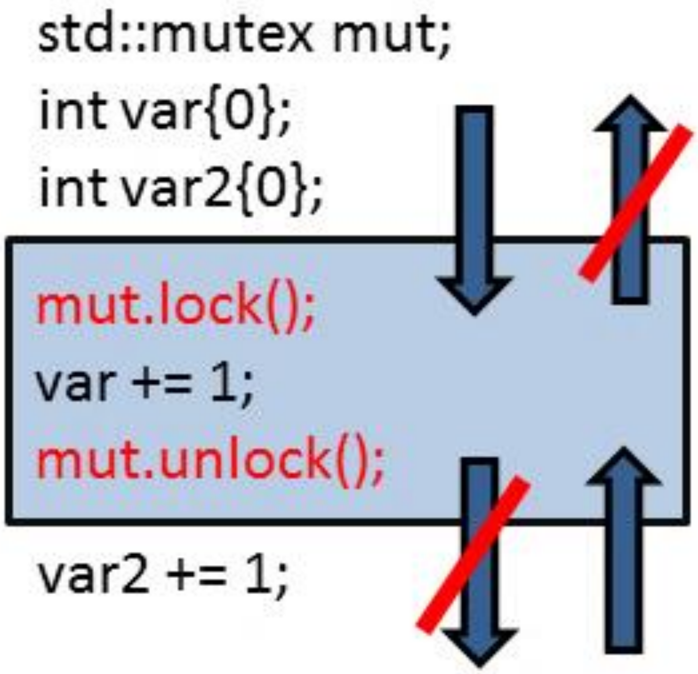
Thread 2

Thread 3

sequenced-before

synchronizes-with

Synchronization and Ordering



- Acquire-operations
  - Locking of a mutex
  - Waiting for a condition variable
  - Starting of a thread
- Release-operations
  - Unlocking of a mutex
  - Notification of a condition variable
  - join-call on a thread

Synchronization and Ordering

```
class Spinlock{
  std::atomic_flag flag;
public:
  Spinlock(): flag(ATOMIC_FLAG_INIT){}

  void lock(){
    while(flag.test_and_set(memory_order_acquire));
  }

  void unlock(){
    flag.clear(std::memory_order_release);
  }
};
```

Synchronization and Ordering

Consume-release semantic

- Consume-release semantic is the acquire-release semantic without ordering constraints.

- Has a legendary reputation
  - Extremely difficult to get
  - The compiler maps `std::memory_order_consume` to `std::memory_order_acquire`.
  - No compiler implements it (Temporary exception *GCC*)

- Deals with data dependencies
  - In a thread: *carries-a-dependency-to*
  - Between threads: *dependency-ordered-before*

Synchronization and Ordering

```
atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(14,memory_order_relaxed);
    ptr.store(p,memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_acquire)));
    cout << *p2 << " " << data;
    cout << atoData.load(memory_order_relaxed);
}
```

```
atomic<string*> ptr,
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(14,memory_order_relaxed);
    ptr.store(p, memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_consume)));
    cout << *p2 << " " << data;
    cout << atoData.load(memory_order_relaxed);
}
```

Synchronization and Ordering

```
std::atomic<std::string*> ptr;
int data;
std::atomic<int> atoData;

void producer(){
    std::string* p  = new std::string("C++11");
    data = 2011;
    atoData.store(2014,std::memory_order_relaxed);
    ptr.store(p, std::memory_order_release);
}

void consumer(){
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)));
    std::cout << "*p2: " << *p2 << std::endl;
    std::cout << "data: " << data << std::endl;
    std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

dependency-ordered-before

carries-a-dependency-to

Synchronization and Ordering

Last words from cppreference.com

The specification of release-consume ordering is being revised, and the use of memory_order_consume is temporarily discouraged. (since C++17)

Synchronization and Ordering

Relaxed semantic

- There are no synchronization and ordering constraints. The operations are only atomic.

- Rule
  - Atomic operations with stronger memory orderings are used to order atomic operations with relaxed semantic.

- Typical use case ⇨ Atomic counter (`shared_ptr`)

! Threads can see the operations in another thread in a different order.

Synchronization and Ordering

```
std::atomic<int> cnt = {0};
void f(){
    for (int n = 0; n < 1000; ++n) {
        cnt.fetch_add(1, std::memory_order_relaxed);
    }
}
int main(){
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n){
        v.emplace_back(f);
    }
    for (auto& t : v) {
        t.join();
    }
    std::cout << "Final counter value is " << cnt << '\n';
}
```

The C++ Memory Model

The Contract

Atomics

Synchronization and Ordering Constraints

Singleton Pattern

Singleton

```
mutex myMutex;

class MySingleton{
public:
  static MySingleton& getInstance(){
    lock_guard<mutex> myLock(myMutex);
    if( !instance ) instance= new MySingleton();
    return *instance;
  }
private:
  MySingleton();
  ~MySingleton();
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;
  static MySingleton* instance;
};
MySingleton::MySingleton()= default;
MySingleton::~MySingleton()= default;
MySingleton* MySingleton::instance= nullptr;
...
MySingleton::getInstance();
```

➡ Performance problem

Sequential Consistency

```
class MySingleton{
public:
  static MySingleton* getInstance(){
    MySingleton* sin= instance.load();
    if ( !sin ){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin= instance.load(std::memory_order_relaxed);
      if( !sin ){
        sin= new MySingleton();
        instance.store(sin);
      }
    }
    return sin;
  }
private:
  static std::atomic<MySingleton*> instance;
  static std::mutex myMutex;
  . . .
```

Acquire-Release Semantic

```
class MySingleton{
public:
  static MySingleton* getInstance(){
    MySingleton* sin= instance.load(std::memory_order_acquire);
    if ( !sin ){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin= instance.load(std::memory_order_relaxed);
      if( !sin ){
        sin= new MySingleton();
        instance.store(sin,std::memory_order_release);
      }
    }
    return sin;
  }
. . .
```

Meyers Singleton

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

! Will only work with Microsoft Visual Studio 2015.

Singleton: The Performance Test

Compiler	Optimization	Single Threaded	std::lock_guard (Mutex)	Sequential consistency	Acquire-release semantic	Meyers Singleton
GCC	yes	0.03	12.47	0.09	0.07	0.04
cl.exe	yes	0.02	15.48	0.07	0.07	0.03

You can find `std::call_once` and the rest of the details here: [Thread safe initialization of a singleton]().

Singleton: The Performance Test

- My conclusions
  - Singleton awakes many emotions.
  - The optimizer removes the calls of MySingleton::getInstance().
  - Meyers singleton is the simplest and the fastest implementation.

The C++ Memory Model

The Contract

Atomics

Synchronization and Ordering Constraints

Singleton Pattern

Further Information

- **Modernes C++:** Training, coaching, and technology consulting by Rainer Grimm
  - www.ModernesCpp.de

- Blog to modern C++
  - www.grimm-jaud.de (German)
  - www.ModernesCpp.com (English)

- Contact
  - @rainer_grimm (Twitter)
  - schulungen@grimm-jaud.de