Synchronization and Ordering

In this lesson, we will learn about different memory models in C++ that relate to synchronization and ordering.

WE'LL COVER THE FOLLOWING

You cannot configure the atomicity of an atomic data type, but you can accurately adjust the synchronization and ordering constraints of atomic operations. This is a possibility unique to C++, and it is not possible in C#'s or Java's memory model.

There are six variations of the memory model in C++. So, what are the characteristics of these model variations?

The six variants of the C++ memory model

We already know that C++ has six variants of the memory models. The default for atomic operations is `std::memory_order_seq_cst`. This expression stands for ***sequential consistency***. Additionally, you can explicitly specify one of the other five variants. So, what does this technique of C++ have to offer?

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

To classify these six memory models, it helps to answer the following two

questions:

1. Which type of memory model should each atomic operation use?
2. Which synchronization and ordering constraints are defined by the six variants?

Let's examine these questions further in the following section.

Kinds of Atomic Operations

There are three different kinds of operations:

- Read operation: `memory_order_acquire` and `memory_order_consume`
- Write operation: `memory_order_release`
- Read-modify-write operation: `memory_order_acq_rel` and `memory_order_seq_cst`

`memory_order_relaxed` defines no synchronization and ordering constraints. It does not fit in this taxonomy.

The table in the previous lesson ordered the atomic operations based on their reading and/or writing characteristics.

Different Synchronization and Ordering Constraints

There are approximately three different types of synchronization and ordering constraints in C++:

- **Sequential consistency**: `memory_order_seq_cst`
- **Acquire-release**: `memory_order_consume`, `memory_order_acquire`, `memory_order_release` and `memory_order_acq_rel`
- **Relaxed**: `memory_order_relaxed`

While the **sequential consistency** establishes a global order between threads, the **acquire-release** semantic establishes an ordering between read and write operations on the same atomic variable on different threads. The **relaxed semantic** guarantees that operations on one specific data type in the same thread cannot be reordered. This guarantee is called modification order consistency, but other threads can see this operation in a different order.

The different memory models and their effects on atomic and non-atomic operations make the C++ memory model an interesting but challenging technique. In the next lessons, we will discuss the synchronization and ordering constraints of the sequential consistency.

Atomics are the base of the C++ memory model. By default, the strong version of the memory model is applied to atomics, meaning it is important to understand the features of the strong memory model.

As you may already know from the subsection on The Contract, we refer to **sequential consistency** with the strong memory model, and we refer to **relaxed semantic** and with the weak memory model.

Strong Memory Model

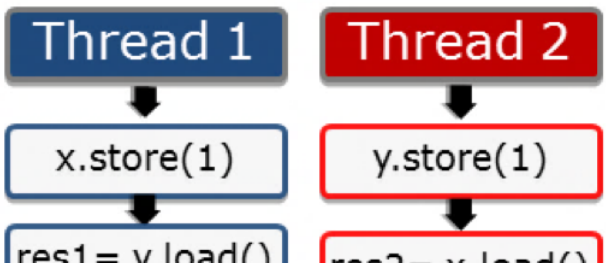
Java 5.0 got its current memory model in 2004, and C++ in 2011. Before that time, Java had an erroneous memory model, while C++ had no memory model. The foundations of multithreaded programming are 40 to 50 years old. Leslie Lamport defined the concept of sequential consistency in 1979, as we will read below.

Sequential consistency provides two guarantees:

- The instructions of a program are executed in source code order.
- There is a global order of all operations on all threads.

Before we dive deeper into these two guarantees, it must be emphasized that these statements only hold for atomics, but they do influence non-atomics.

This graphic shows two threads. Each thread stores its variable *x* or *y* respectively loads the other variable *y* or *x*, and stores them in the variable *res1* or *res2*.

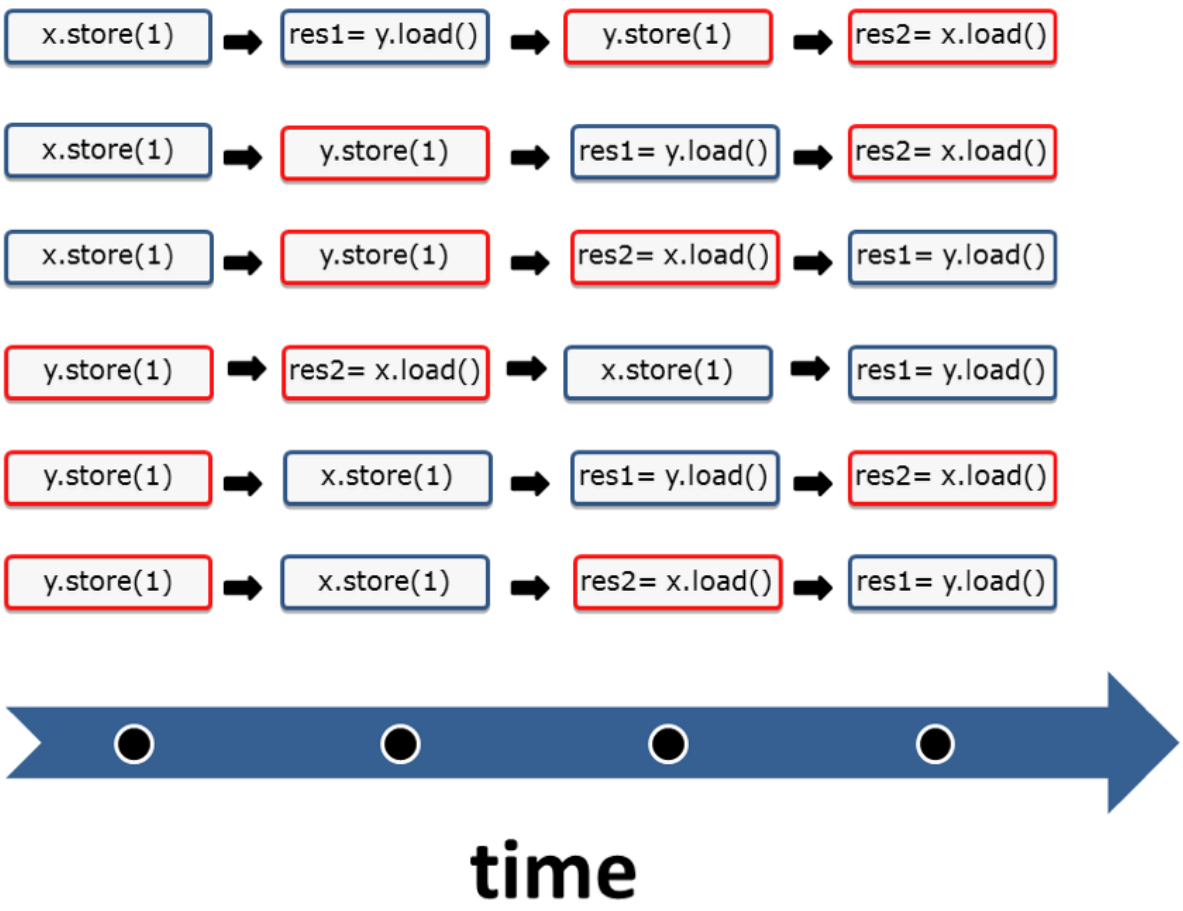


Because the variables are atomics, the operations are executed atomically. By default, sequential consistency applies. The question is: in which order can the statements be executed?

Firstly, the sequential consistency guarantees that the instructions are executed in the order defined in the source code, meaning that no store operation can overtake a load operation.

Secondly, the sequential consistency guarantees that the instructions of all the threads must follow a global order. In the case listed above, thread 2 implements the operations of thread 1 in the same order in which thread 1 executes the operations. This is important to observe: thread 2 sees all operations of thread 1 in the source code order of thread 1. The same holds for the perspective of thread 1. Thinks about the characteristics of number 2 as a global clock that all threads must obey. The global clock is, therefore, the global order of the operation as a whole. Each time the clock ticks, one atomic operation occurs, but it is hard to know which one.

This problem is not yet solved. We still must examine the different interleaving executions of the two threads, making the following six interleavings of the two threads possible.



That was easy, right? We have now implemented sequential consistency, also known as the strong memory model.

Let’s look at an example of sequential consistency in the next lesson.