**Synchronizing Threads**

Rarely are multiple threads run independently of one another. Threads might provide data processed by other threads or prepare preconditions necessary to start other processes. Techniques to synchronize threads:

1.   Mutexes and locks including call\_once()

2.   Condition variables

3.   Atomics

# Beware of Concurrency

The only safe way to concurrently access the same data by multiple threads without synchronization is when ALL threads only READ the data.

**Data race in C++.**

In the C++11 standard a data race is defined as “two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other.” A data race always results in undefined behavior.

# The Reason for the Problem of Concurrent Data Access

A standard such as C++ specifies the effect of statements and operations and not the corresponding generated assembler code. The standard describes the what, not the how.

In general, the behavior is not defined so precisely that there is only one way to implement it. In fact, behavior might even explicitly be undefined.

**Which guarantees does a language give?**

According to the so-called as-if rule, each compiler can optimize code as long as the behavior of the program visible from the outside behaves the same. Thus, the generated code is a black box and can vary as long as the observable behavior remains stable.

Any undefined behavior is provided to give both compiler and hardware vendors the freedom and ability to generate the best code possible, whatever their criteria for “best” are. Yes, it applies to both: Compilers might unroll loops, reorder statements, eliminate dead code, prefetch data, and in modern architectures, for example, a hardware buffer might reorder loads or stores.

Reordering can be useful to improve the speed of the program, but they might break the behavior. To be able to benefit from fast speed where useful, safety is not the default.

# The Extent of the Problem

To give compilers and hardware enough freedom to optimize code, C++ does not in general give a couple of guarantees you might expect because it would cost too much in performance.

In C++, we might have the following problems:

1.   **Unsynchronized data access:** When two threads running in parallel read and write the same data, it is open which statement comes first.

2.   **Half-written data:** When one thread reads data, which another thread modifies, the reading thread might even read the data in the middle of the write of the other thread, thus reading neither the old nor the new value.

3.   **Reordered statements:** Statements and operations might be reordered so that the behavior of each single thread is correct, but in combination of all threads, expected behavior is broken.

## Unsynchronized Data Access

A simple code such as:

std::vector<int> v;

...

if (!v.empty()) {

          std::cout << v.front() << std::endl;

}

can be a problem if v is shared between multiple threads, because between the call of empty() and the call of front(), v might become empty resulting in undefined behavior.

This problem also applies to code implementing a function provided by the C++ standard library. For example, the guarantee that

v.at(5)         // yield value of element with index 5

throws an exception if v does not have enough elements, no longer applies if another thread might modify v while at() is called.

Unless otherwise stated, C++ standard library functions usually don’t support writes or reads concurrently performed with writes to the same data structure.

That is, unless otherwise stated, multiple calls on the same object from multiple threads will result in undefined behavior.

C++ standard library provides some **guarantees regarding thread safety**:

·         Concurrent access to different elements of the same container is possible (except for class vector<bool>)

·         Concurrent access to a string stream, file stream, or stream buffer results in undefined behavior.

## Half-Written Data

Consider that we have a variable1

long long x = 0;

and one thread writing the data:

x = -1;

and one thread reading the data:

std::cout << x;

which value does the second thread read when it outputs x? Following answers are possible:

1.   0 (the old value of x), if the first thread has not assigned -1

2.   -1 (the new value of x), if the first thread assigned -1 already

3.   Any other value, if the second thread reads x during the assignment of -1 by the first thread

The last option — any other value — can, for example, easily happen if, on a 32-bit machine, the assignment results in two stores and the read by the second thread happens when the first store was done but the second store was not yet done.

Even for a fundamental data type, such as int or bool, the standard does not guarantee that a read or a write is atomic.

The same applies to more complicated data structures, even if they are provided by the C++ standard library.

## Reordered Statements

Suppose we have two shared objects, an int to pass data from one thread to another and a Boolean readyFlag, which signals when the first thread has provided the data:

long data;

bool readyFlag = false;

A naive approach is to synchronize the setting of the data in one thread with the consumption of the data in another thread. Thus, the providing thread calls:

data = 42;

readyFlag = true;

and the consuming thread calls:

while (!readyFlag) {        // loop until data is ready

          ;

}

foo(data);

Without knowing any details, almost every programmer at first would suppose that the second thread calls foo() when data has the value 42, assuming that the call of foo() can be reached only if the readyFlag is true, which itself can be the case only after the first thread assigned 42 to data, because this happens before the readyFlag becomes true.

But this is not necessarily the case. In fact, the output of the second thread might be the value data had before the first thread assigned 42 (or even any other value, because the assignment of 42 might be half-done). That is, the compiler and/or the hardware might reorder the statements so that effectively the following gets called:

readyFlag = true;

data = 42;

In general, such a reordering is allowed due to the rules of C++, which requires only that the observable behavior inside a thread of the generated code be correct.

Thus, reorderings of statements are allowed as long as the visible effect to the outside of a single thread is the same. For the same reason, even the second thread might reorder the statements, provided that the behavior of this thread is not affected:

foo(data);

while (!readyFlag) {        // loop until data is ready

          ;

}

Note that the observable behavior might be affected by such a reordering if foo() throws. Thus, it depends on details whether such reorderings are allowed, but in principle, the problem applies.

# Features to Solve the Problems

To solve the three major problems of concurrent data access, we need the following concepts:

## Atomicity and order

1.   **Atomicity:** This means that read or write access to a variable or to a sequence of statements happens exclusively and without any interruption, so that one thread can’t read intermediate states caused by another thread.

2.   **Order:** We need some ways to guarantee the order of specific statements or of a group of specific statements.

C++ standard library provides very different ways to deal with these concepts, so that programs benefit from additional guarantees regarding concurrent access:

1.   You can use futures and promises, which guarantee both atomicity and order.

2.   You can use mutexes and locks to deal with critical sections, or protected zones, whereby you can grant exclusive access.

3.   You can use condition variables to efficiently allow one thread to wait for some predicate controlled by another thread to become true.

4.   You can use atomic data types to ensure that each access to a variable or object is atomic while the order of operations on the atomic types remains stable.

5.   You can use the low-level interface of atomic data types, which allow experts to relax the order of atomic statements or to use manual barriers for memory access (so-called fences).

High-level features, such as futures and promises or mutexes and locks, are easy to use and provide little risk. Low-level fea-tures, such as atomics and especially their low-level interface, might provide better performance because they have lower latency and therefore higher scalability, but the risk of misuse grows significantly. Nevertheless, low-level features sometimes provide simple solutions for specific high-level problems.

## volatile and Concurrency

Note that I didn’t mention volatile here as a feature for concurrent data access, although you might have expected that for the following reasons:

1.   volatile is known as a C++ keyword to prevent too much optimization.

2.   In Java, volatile provides some guarantees about atomicity and order.

In C++, volatile “only” specifies that access to external resources, such as shared memory, should not be optimized away. For example, without volatile, a compiler might eliminate redundant loads of the same shared memory segment because it can’t see any modification of the segment throughout the whole program. But in C++, volatile provides neither atomicity nor a specific order. Thus, the semantics of volatile between C++ and Java now differs.

# END