**3. Sharing data between threads**

If you’re sharing data between threads, you need to have rules for which thread can access which bit of data when, and how any updates are communicated to the other threads that care about that data.

Incorrect use of shared data is one of the biggest causes of concurrency-related bugs.

# 3.1 Problems with sharing data between threads

If all shared data is read-only, there’s no problem, because the data read by one thread is unaffected by whether or not another thread is reading the same data.

But if data is shared between threads, and one or more threads start modifying the data, there’s a lot of potential for trouble.

**Invariants**

statements that are always true about a particular data structure

These invariants are often broken during an update, especially if the data structure is of any complexity or the update requires modification of more than one value.

The simplest potential problem with modifying data that’s shared between threads is that of broken invariants.

**Example: Race condition**

Consider a doubly linked list

One of the invariants is that if you follow a “next” pointer from one node (A) to another (B), the “previous” pointer from that node (B) points back to the first node (A).

In order to remove a node from the list, the nodes on either side have to be updated to point to each other. Once one has been updated, the invariant is broken until the node on the other side has been updated too; after the update has completed, the invariant holds again.

If you don’t do anything special to ensure otherwise, if one thread is reading the doubly linked list while another is removing a node, it’s quite possible for the reading thread to see the list with a node only partially removed (because only one of the links has been changed), so the invariant is broken.

This is an example of one of the most common causes of bugs in concurrent code: a race condition.

## 3.1.1 Race conditions

**Race condition** is anything where the outcome depends on the relative ordering of execution of operations on two or more threads

It’s when the race condition leads to broken invariants that there’s a problem, such as with the doubly linked list example mentioned

Race conditions are generally timing-sensitive, **they can often disappear entirely when the application is run under the debugger**, because the debugger affects the timing of the program, even if only slightly.

**Data race** to mean the specific type of race condition that arises because of concurrent modification to a single object; data races cause the dreaded undefined behavior.

## 3.1.2 Avoiding problematic race conditions

There are several ways to deal with problematic race conditions.

**Use protection mechanism**

The simplest option is to wrap your data structure with a protection mechanism to ensure that only the thread performing a modification can see the intermediate states where the invariants are broken.

**Use lock-free programming**

Another option is to modify the design of your data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants. This is generally referred to as lock-free programming and is difficult to get right.

**Software Transactional Memory (STM)**

Another way of dealing with race conditions is to handle the updates to the data structure as a transaction. If the commit can’t proceed because the data structure has been modified by another thread, the transaction is restarted. This is termed software transactional memory (STM).

# References

Anthony Williams - C++ Concurrency in Action