I have introduced In the post [Sequential Consistency](http://modernescpp.com/index.php/sequential-consistency)the default memory model. This model, in which all operations in all threads takes place in a global time clock, has a big advantage but also a big disadvantage.

Heavyweight synchronization

The big advantage of the sequential consistency is, that is matches our intuition of many threads running in parallel. The big disadvantages is, that the system has a lot of work to do to synchronise all the threads.

The following program synchronises the producer and the consumer thread with the help of the sequential consistency.

// producerConsumer.cpp

#include <atomic>

#include <iostream>

#include <string>

#include <thread>

std::string work;

std::atomic<bool> ready(false);

void consumer(){

while(!ready.load()){}

std::cout<< work << std::endl;

}

void producer(){

work= "done";

ready=true;

}

int main(){

std::thread prod(producer);

std::thread con(consumer);

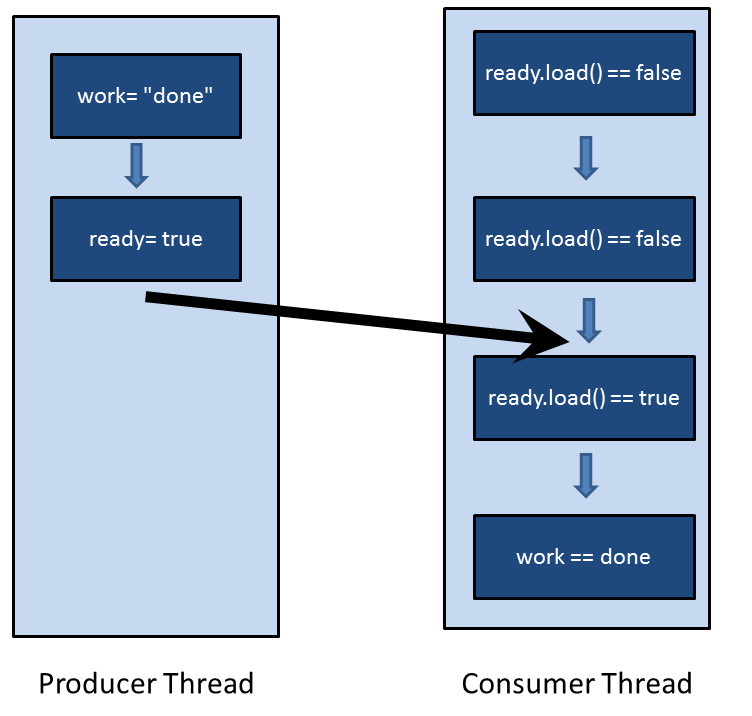
prod.join();

con.join();

}

Because of the sequential consistency, the program execution is totally deterministic. It always displays "done".

The graphic hit the spot. The consumer thread waits in the while-loop until the atomic variable ready is set to true. In case that happens, the consumer threads continue with its work.



It easy to reason, that the program will always return "done". I have only to use the two characteristics of the sequential consistency. At one hand, both threads execute their instructions in the source code order, at the other hand, each thread sees the operations of the other thread in the same order. So both threads are following the same global time clock. This time clock will also hold - with the help of the while(!ready.load()){}-loop - for the synchronisation of the producer and the consumer thread.

But I can do the reasoning a lot more formal by using the terminology of the memory model. So the formal version:

**=>** Means it follows in the next lines:

* 1. work= "done"  is sequenced-before ready=true**=>**work= "done" happens-before ready=true
  2. while(!ready.load()){} is sequenced-before std::cout<< work << std::endl **=>**while(!ready.load()){} happens-before std::cout<< work << std::endl
  3. ready= true synchronizes-with while(!ready.load()){} **=>** ready= true inter-thread happens-before while (!ready.load()){} **=>** ready= true happens-before while (!ready.load()){}

**=>**  Because the happens-before relation is transitive, it follows t: work= "done" happens-before ready= true happens-before while(!ready.load()){} happens-before std::cout<< work << std::endl

## From the sequential consistency to the acquire-release semantic

A thread sees the operations of another thread and therefore of all other threads in the same order. The key characteristic of the sequential consistency will not hold, if we use the acquire-release semantic for atomic operations. This is an area, in which C# or Java will not follow. But that's also an area, in which our intuition begins to wane.

There is no global synchronization between threads in the acquire-release semantic, there is only a synchronisation between atomic operations on the same atomic variable. So a write operation on one thread synchronises with a read operation on another thread on the same atomic variable. This synchronization relation on the same atomic variable helps to establish a happens-before relation between atomic variables and therefore between threads.