Sequential Consistency

Familiarize yourself with sequential consistency in C++.

WE'LL COVER THE FOLLOWING

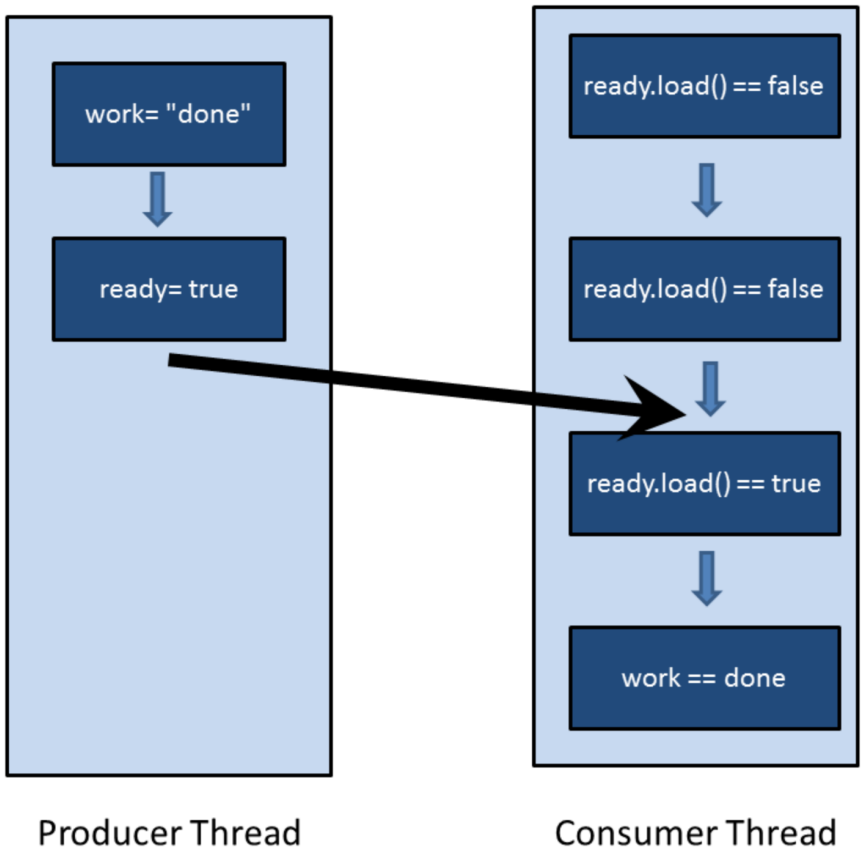
- Explanation:

Let us dive deeper into sequential consistency. The key for sequential consistency is that all operations on all threads obey a universal clock. This universal clock makes it quite intuitive to think about it.

The intuitiveness of the sequential consistency comes with a price. The downside is that the system has to do a lot of work to keep the threads in sync. The following program synchronizes the producer and the consumer thread with the help of 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();
}
```

The output of the program is not very exciting. Because of sequential consistency, the program execution is totally deterministic; its output is always “done”. The graphic depicts the sequence of operations. The consumer thread waits in the while-loop until the atomic variable `ready` is set to `true`. When this happens, the consumer thread will continue its work.



It is quite easy to understand that the program will always return “done”, as we only have to use the two characteristics of sequential consistency. On one hand, both threads execute their instructions in source code order; On the other hand, each thread sees the operations of the other thread in the same order. Both threads follow the same global timing. This timing will also hold - with the help of the `while(!ready.load()){}` loop - for the synchronization of the producer and the consumer thread.

Explanation:

Explanation:

I can explain the reasoning a lot more formally by using the terminology of the memory model. Here is the formal version:

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()){}`

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

In sequential consistency, a thread sees the operations of another thread and, therefore, of all other threads in the same order. The key characteristic of sequential consistency will not hold if we use the acquire-release semantic for atomic operations. This is an area where C# and Java will not follow. That's also an area where our intuition begins to wane. Let's look at acquire-release semantic in the next lesson.