# Synchronization

Synchronization requires the use of locks and barriers. This lesson discusses synchronization as well as various locks and barriers.

**Synchronization**

Thread A and Thread B are executing programs. If the threads both modify the same memory location, the order of the modifications is important. Thus the need for synchronization.

Atomic Code Sections - these are sections of code that must be executed completely, one at a time.

Mutual Exclusion - a type of synchronization used for atomic sections of code. Also called a lock.

Before entering a critical section, a lock must be implemented. If the lock is held by a core, other cores that want to use the lock have to wait (or spin) until the lock is free. The lock is freed at the end of the critical section.

Locks force mutual exclusion.
Locks are just locations in shared memory.

**Lock Synchronization**

Mutex = mutual exclusive variable

Checking and acquiring a lock must be atomic, otherwise two or cores could hold the lock at the same time, defeating synchronization.

**Implementing Lock()**

1. The lock is checked.
2. If the lock is free, acquire it.
3. If the lock is not free, spin until the lock is free.
4. If the lock is held by the core, unlock it.

An instruction that both reads and writes to memory is required to implement lock().

**Atomic Instructions**

3 Main Types of Atomic Instructions:

1. Atomic Exchange - the contents of a register and memory location are swapped.The drawback of this is it continues to swap while waiting to acquire the lock.

2. Test and Write - test the lock before trying to acquire it. If the lock is free, it can be acquired. If the lock is not free, don't try to write to it. This method corrects the drawback of the Atomic Exchange.  But this is a strange instruction, it is neither a store or load.

Atomic read/write in the same instruction is bad for pipelining.

3. Load Linked / Store Conditional (LL/SC) -Linked Load - behaves like normal load but also saves the address from which it loaded to a link register.
Store Conditional- checks first if the computed address == the one in link register. If the two addresses are the same, the instruction is completed. If the two addresses are not the same, the store is not completed.

**How is LL and SC Atomic?**
The store conditional fails if another core has already done LL. If the code is simple, locks are not needed, the LL/SC can be used directly.

**Locks and Performance**
If cores are waiting on a lock, the energy spent spinning is quite high and it overloads the bus. When the bus is overloaded with cores checking lock availability even the core that has acquired the lock is slowed down.

**Test and Atomic Op Lock**
The drawback of LL/SC can be mitigated by using a Test-and-Atomic Operation Lock.
Test - wait for lockvar to become free using a normal read, when it becomes free, do an Exchange. This will reduce the bus traffic, only the cache is checked.

**Barrier Synchronization**
A barrier makes sure all the threads wait for the completion of a task before allowing any to go past the barrier.

Two variables are required for a barrier:
1. A counter to count the number of threads as they arrive at the barrier.
2. A flag that is set when the number of threads at the barrier equals the number of threads required.

**Simple Barrier Implementation**
-The first thread will set the release to 0
-Count each thread
-When the last thread arrives, unlock the barrier.
-Set the release to 1
-Spin while waiting for the release to be 1

This simple barrier implementation doesn't work.

**Simple Barrier Implementation Doesn't Work**

If core 0 arrives first, it spins and waits for core 1 to set the release to 1. If core 0 is delayed from checking the release, core 0 will wait at the barrier. If core 1 returns to the barrier, it will now also wait. The two threads will wait indefinitely for the other to set the release, this is called Deadlock.

**Reusable Barrier**

To solve the deadlock problem, a reusable barrier must be used.
Instead of setting the release to 0 and waiting for the release to equal 1, the threads look for the release to be flipped.