Topics:
- Mutex / Locks
- Implementing Lock()/Unlock()
  - Atomic Exchange
  - Test + Write
  - LL/SC
- Test & Atomic Op
- Barriers
- Reusable Barriers

Cache Coherence → Synchronization → Memory Consistency

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. in other words, they're not a special processor construct or special memory location. They're literally just any plain location in shared memory. The think that makes them a lock is the instruction that the processor uses to access/modify them.

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. Because if two cores check the lock at the exact same time and both see that the lock is free, then they'll BOTH proceed thinking that they hold the lock. The lock checking & acquiring needs to be atomic.

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.

This capability relies on hardware support in order implement the locking mechanism. We need the processor to provide us with an ATOMIC instruction that we can use to read & write at the same time.

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

Atomic Instructions

3 Main Types of Atomic Instructions:

```
// if lockVar == 0, then unlocked.
// if lockVar == 1, then locked.
R1 = 1;
while (R1 == 1) {
   EXCH R1, lockVar;
}
```

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. (Look at the code to the left. While the lockVar is 1 [from being set by another core], then this core will keep doing the EXCH instruction.)

Atomic Exchange is like a Load AND a Store instruction in a single instruction.

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.*

For example, this single atomic instruction...

```
TSTSW R1, 78(R2)
```

... is equivalent to the following:

```
if ( Mem[78 + R2] == 0 ) {
   Mem[78 + R2] = R1;
   R1 = 1;
} else {
   R1 = 0;
}
```

... which can be used as a lock like so:

```
do {
   R1 = 1;
   TSTSW R1, lockvar;
while (R1 == 0);
```

In this case, when the lock is not free (when Mem[78 + R2] != 0), we DON'T write to the memory location because it's unnecessary. Compare this to the Atomic Exchange instruction, which ALWAYS writes to the memory location, regardless of whether its actually free or not.

We'd prefer the Test & Write (which avoids unnecessary writes to a locked variable) bc of COHERENCE. If you keep writing to a memory location, then you keep INVALIDATING all the other caches' copies. So everybody that is waiting on the lock keeps generating bus traffic because they're all repeatedly writing to the same memory location.

In contrast, if we use the Test & Write method, all the cores that are waiting on the lock var to become free simply gets to SHARE the lock var (keep it in SHARED state within their caches) and simply iterate their checks on their cached copies. Once the lock is actually freed, then the owning core will WRITE 0 to the lock, therefore INVALIDATING all the other cores' cached values. So there's only contention/communication regarding the lock when the lock becomes free.

Consider a typical 5 stage pipeline: Fetch -> Decode/Read Regs -> ALU -> Memory Access -> Write Results to Registers.
An atomic Read/Write instruction needs to access MEMORY, but it wouldn't be possible to complete everything (BOTH a LOAD AND a STORE) in a SINGLE "Memory Access" Pipeline stage within a single cycle. Instead, we'd have to have additional "Memory Access" stages, but this would add extra pipeline overhead for ALL other instructions. This is not worth it. So the alternative is to keep our pipeline the way it is but have a LL/SC that still provides atomicity.

LL/SC uses Cache COHERENCE to ensure atomicity!

1) LL R1, lockAddr; // this loads R1 with the VALUE at lockAddr AND it stores lockAddr in the LINK REG.

2) // SNOOP/listen on the Mem Bus. If ANY other core issues a WRITE to the same address as the one stored in the LINK REGISTER, then we will automatically set our LINK REGISTER = 0.

3) SC R2, lockAddr; // If lockAddr == the address stored in our LINK REGISTER, then complete the store/write and set R2 = 1. Otherwise if they don't match (bc the Link Register is 0), then do not complete the store/write and set R2 = 0.

4) While R2 == 0, keep doing this!

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

Even though these are two separate instructions, they behave as an atomic instruction because of the LINK REGISTER. In order to behave like a single atomic instruction, writes by other cores cannot come between them without us knowing. We use the Link Register for this.

The LL/SC splits the "Atomic Read/Write" back into two separate instructions, but its behavior is still atomic. The Linked Load is the "Read" part and the Store Conditional is the "Write" part.

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. This is a hidden register that can only be accessed through the LL & SC instructions.
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.

If the computed address == the address stored in the Link Register:
then the STORE/WRITE is completed to the address and a 1 is returned.
ELSE the STORE/WRITE is NOT completed and a 0 is returned.

How is LL and SC Atomic?

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

"If the code is simple" = if the critical section is just a single memory location, then we can use LL/SC directly on that mem loc. Otherwise, we'd use LL/SC to protect access to a lock, which we'd then use that lock to protect the larger critical section.

```
void lock(mutex_type &lockVar) {
  trylock:  MOV R1, 1
            LL R2, lockVar
            SC R1, lockVar
            BNEZ R2, tryLock
            BEQZ R1, tryLock
}
```

There are 2 conditions we need to check:
1) From the last read we've done (LL into R2), was the lock busy (!= 0)?
2) Were we successfully able to store R1 into tryLock? This depends on if the lock value has CHANGED since we last read it?

Locks and Performance (of Atomic Exchange)

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. Locks based on Atomic Exchange leads to a lot of Cache Thrashing.

Test and Atomic Op Lock

Atomic Exchange

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.

Note that this can also be done to LL/SC by checking the result of LL first before even trying SC.

Original (thrash-y) Atomic Exchange method:

```
R1 = 1;
while (R1 == 1)
   EXCH R1, lockVar;
```

Less thrash-y "Test & Atomic Op Lock" method:

```
R1 = 1;
while (R1 = 1) {
   while (lockvar == 1) {};
   EXCH R1, lockVar;
}
```

The "while (lockvar == 1)" allows us to just keep checking our Cache until it is INVALIDATED. This prevents us from unnecessarily thrashing the other Caches and adding bus contention.

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

Simple (but not correct) barrier implementation:

```
lock (counterLock);
   if (count == 0) release = 0;
   count++;
unlock (counterLock);
if (count == total) {
   count = 0;
   release = 1;
} else {
   // spin while release != 1
}
```

This simple barrier implementation doesn't work.

Simple Barrier Implementation Doesn't Work

Basically, in this edge case, Core 0 is sitting at the spin waiting for release == 1. Meanwhile, Core 1 gets to increment the count and sees that it is okay to set the release to 1. It does so and exits the barrier. Normally, Core 0 should be able to see that release was set to 1 and be able to exit the Spin, but let's say it gets interrupted by an interrupt. Then let's also say that Core 1 returns BACK to the beginning of the barrier synchronization block, which re-initializes the release BACK TO 0. Then Core 1 spins, thinking that Core 0 will come along and set the release to 1, however Core 0 is still stuck in the FIRST BARRIER. So both of them see release == 0 and will spin forever.

So the issue with the simple barrier implementation is that is works fine for 1 time use, but then it becomes a RACE CONDITION with a potential DEADLOCK outcome if we try to REUSE the barrier again.

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~~.

The threads instead keep a thread-local variable of what they want the release to be. Every time they reach the barrier, they flip that thread-local variable. This ensures that even if the different threads are on different "laps" of the barrier, they will be expecting the release to be different things, allowing one of them to proceed. They'll only have the same values for their thread-local variable if they're on the same "barrier lap".

```
0) localSense != localSense;
1) lock (counterLock);
2)     count++;
3)     if (count == total) {
4)         count = 0;
5)         release = localSense;
6)     }
7) unlock(counterLock);
8) // spin while release != localSense
```

So let's say that we're using a barrier for two threads. Initially, they BOTH see localSense to be 0. Count is also 0.
- At (0), they both set their own localSense to 1. Release is also initialized to 0.
- From (1 - 7) only one thread is allowed to proceed at a time. Let's say Core 0 proceeds and increments count to 1. Then it unlocks and spins until Release == its localSense, which is 1.
- Core 1 then proceeds into the critical section and increases count to 2, which is also equal to the total, so it's allowed within the if block. It resets count to 0 and sets the release equal to its localSense, which is 1 as well. It unlocks and then skips the spin, since release == its localSense (1 == 1).
- Let's say Core 0 is interrupted and doesn't get to check the release yet. Meanwhile, Core 1 is back at the beginning of the barrier.
- Core 1 flips its localSense from 1 to 0, then it enters the critical section. It increments count to 1, then unlocks, and SPINS UNTIL RELEASE == its localSense (which is 0). At this moment, release is still set to 1, so it stays spinning.
- This prevents the deadlock described in the simple barrier implementation!
- Eventually, Core 0 gets to go back and check if release == its localSense (which is 1). It is! So it continues on as well. Eventually, it will get back to the beginning of the barrier as well, flip its own localSense (1 -> 0), enter the critical section, increment the count, see that count == total, reset count and set release = its localSense (0), then exit the critical section, and eventually proceed past the SPIN.
- Core 1 will then see the effects of Core 0, and see that release is now equal to its own localSense as well (0) and exit out of its spin.