Synchronization

- Synchronization mechanisms are built with user-level software routines that rely on hardware-supplied synchronization instructions
  - For smaller multiprocessors or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value.
  - In high-contention situations, synchronization can become a performance bottleneck because contention introduces additional delays and latency is potentially greater in such multiprocessors.
- Lock and unlock operations form the foundation for creating mutual exclusion and more complex synchronization mechanisms.

Hardware Primitives for Synchronization

Atomic Exchange

Interchanges register value with memory value atomically. Used to implement locks where 0 indicates free and 1 indicates unavailable.

Test-and-Set

Tests a value and sets it if the test passes. Can be used similarly to atomic exchange for synchronization.

Fetch-and-Increment

Returns memory location value and atomically increments it. Using 0 to indicate unclaimed, it functions like exchange.

These primitives provide the ability to atomically read and modify a location, with a way to tell if the operations were performed atomically. They are building blocks for user-level synchronization operations like locks and barriers.

Load Linked and Store Conditional

- An alternative to single atomic operations uses a pair of instructions: load linked (or load locked) and store conditional.
- The store conditional returns 1 if successful and 0 if the memory location was changed between the load linked and store conditional, or if a context switch occurred.

```
try: MOV R3,R4      ;mov exchange value
     LL R2,0(R1)    ;load linked
     SC R3,0(R1)    ;store conditional
     BEQZ R3,try    ;branch store fails
     MOV R4,R2      ;put load value in R4
```

- This sequence implements an atomic exchange on the memory location specified by R1. If any processor modifies the value between LL and SC, the SC returns 0, causing the code to retry.

Atomic Fetch-and-Increment

- An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives
- Only register-register instructions should be inserted between LL and SC to avoid deadlock and the instruction count should be minimal to reduce SC failure probability.
- An example of the atomic fetch-and-increment implementation:

```
try: LL R2,0(R1)        ;load linked
     DADDUI R3,R2,#1    ;increment
     SC R3,0(R1)        ;store conditional
     BEQZ R3,try        ;branch store fails
```

Spin Lock with Coherence

```
lockit: LD     R2,0(R1)    ;load of lock
        BNEZ   R2,lockit   ;not available-spin
        DADDUI R2,R0,#1    ;load locked value
        EXCH   R2,0(R1)    ;swap
        BNEZ   R2,lockit   ;branch if lock wasn't
0
```

Step	P0	P1	P2	State	Acitivity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Un-lock	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Implementation with LL/SC

- This example shows another advantage of the load linked/store conditional primitives: The read and write operations are explicitly separated.
- The load linked need not cause any bus traffic.
- This fact allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock, the LL has replaced the LD, and the SC has replaced the EXCH):

```
lockit: LL      R2,0(R1)      ;load linked
        BNEZ    R2,lockit     ;not available-spin
        DADDUI  R2,R0,#1      ;locked value
        SC      R2,0(R1)      ;store
        BEQZ    R2,lockit     ;branch if store fails
```

- The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.