LECTURE 12

Synchronization

Miquel Pericàs

EDA284/DIT361 - VT 2020

What's cooking

1. **Lectures**
   - Today (8h-10h) **Synchronization**
   - Tuesday (3/3) next week (9h-12h)**: Coherence**
   - Friday (6/3) next week (10:30h-12h)**: Consistency**
2. **Lab session**
   - Friday (8h-12h @ ED3507), GEM5 + Vector
   - Lab Intro has been posted in Canvas
3. **Project Status**
   - Deadline for first round of peer feedback is today

Overview of Lectures 12-14

Goal: Study the correct and reliable communication of values in shared-memory multiprocessors

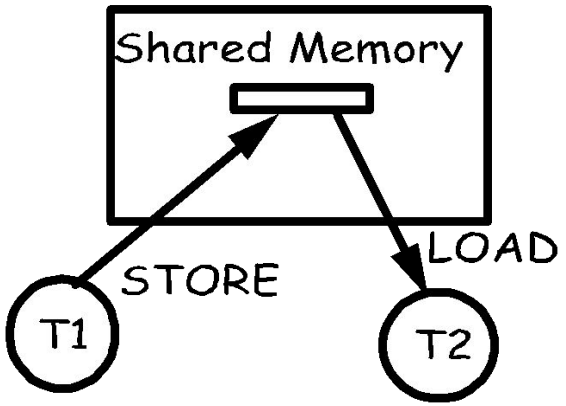
- **Synchronization (L12) 26/2**
- **Strict coherence (L13) 3/3**
- **Plain coherence (L13) 3/3**
- **Sequential consistency (L14) 3/3 or 6/3**
- **Memory consistency models (L14) 6/3**

AGENDA

- Synchronization: Why?
- Components of synchronization events
- Hardware-based Synchronization
- Software-based Synchronization
- Support for synchronization in High Level Languages

Shared-memory Communication

- **Implicitly via memory**



  - Processors share some memory
  - Communication is implicit through loads and stores
    - need to synchronize
    - need to know how the hardware interleaves accesses from different processors

No assumption on the relative speed of processors.
Programs must be correct independently of speed

Why "Mutual Exclusion"

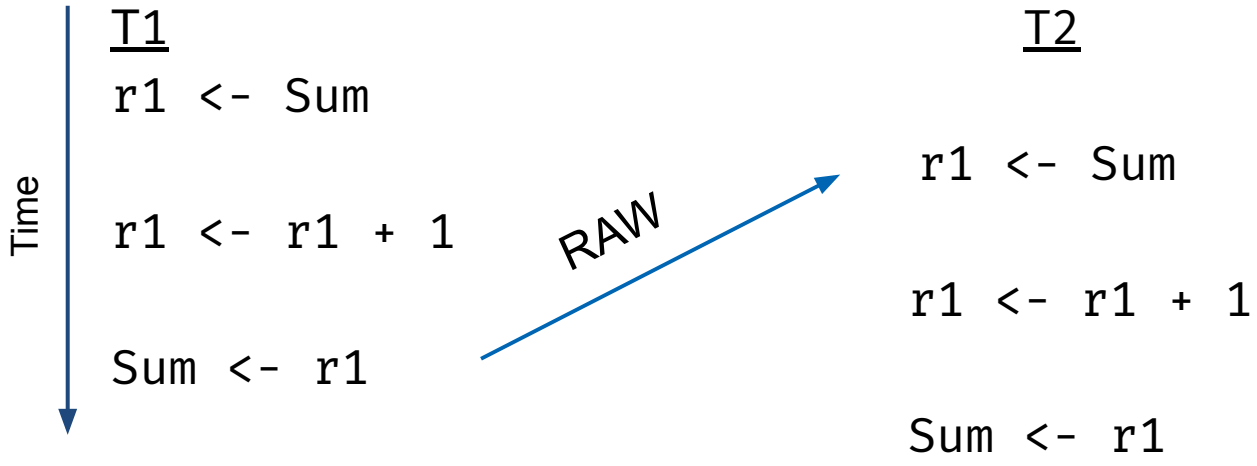
- assume the following statements are executed by 2 threads, T1 and T2, on sum

T1	T2
`Sum <- Sum+1`	`Sum <- Sum+1`

- **PROGRAMMER'S EXPECTATION:** final result independent of the order of execution of the statements (Sum ← Sum + 2)

- **BUT:** program statements are not executed atomically

- compiled code on a RISC ISA will result in several instructions
- a possible interleaving of execution is:



- at the end the result is that sum is incremented by 1 (NOT 2)

Mutual Exclusion

- **We must have a way to make program statements appear atomic**

- **Critical sections**
  - provided by lock and unlock primitives framing the statement(s)
  - modifications are “released” atomically at the end of the critical section

- **So the code should be:**

T1	**T2**	
`lock(La)`	`lock(La)`	`/acquire lock`
`A ← A+1`	`A ← A+1`	
`unlock(La)`	`unlock(La)`	`/release lock`

- The question then becomes: how do we implement correct/reliable a lock?

Dekker's Algorithm for Locking

- **Note: Simplified version (without "turn")**
  - Full version: https://en.wikipedia.org/wiki/Dekker%27s_algorithm

- Two variables 'A' and 'B' to indicate intention to enter critical section
- Assume A and B are both 0 initially

T1	**T2**	
`A:=1`	`B:=1`	`/acquire`
`while(B==1);`	`while(A==1);`	
`<critical section>`	`<critical section>`	
`A:=0`	`B:=0`	`/release`

- **At most one process can be in the critical section at any one time.**
- Deadlock if both threads enter the while( ) at same time.
  - We will consider that this is ok for the time being...
- Complex (to solve deadlock and synchronize more than 2 threads)

Not a general solution to the synchronization problem

Barrier Synchronization

- Global synchronization among all threads
- ALL threads must reach the barrier before ANY thread is allowed to execute beyond the barrier

P1

```
...
BAR := BAR+1;
while (BAR < 2);
```

P2

```
...
BAR := BAR +1;
while (BAR < 2);
```

- **Note: need a critical section to increment BAR!**
  - no need of a critical section to read BAR in the while statement
- In practice more complex because barrier count must be reset for the next iteration...

- **Exercise**: can you come up with a barrier that does not require locks?

Point-to-point Synchronization

- One thread (producer) signals another thread (consumer) that it has reached a certain point in execution

```
T1                              T2
                                A = 1;
                              FLAG = 1;      /release
while (FLAG==0);                             /acquire
  print A
```

- Note: no need for critical sections to update and read FLAG
- Signal sent by T2 to T1 through FLAG (Producer/Consumer synchronization)

Components of a Synchronization Event

- **Acquire Method**
  - Acquire accesses rights to the synchronization variable (to enter critical section, go past synchronization event).
  - Needs waiting method if a different thread already holds the rights
- **Release Method**
  - Enable other processors to acquire the right to the synchronization
- **Waiting Algorithm**
  - Blocking
  - Busy waiting

Waiting Algorithms

- **Blocking** (*)
  - Waiting processes are descheduled by O/S
  - High overhead
  - Allows processor to work on something else
  - Common method with O/S semaphores

- **Busy Waiting** (e.g. spinlock)
  - Waiting processes repeatedly test a location until it changes value
  - Low overhead but holds the processor
  - May generate high memory/network traffic

(*) This notion of blocking is unrelated to blocking/non-blocking communication. It is more closely related to the concept of synchronous communication

Busy-Waiting

- **Busy-waiting** keeps the processor resources occupied
- In hardware multithreaded cores it can be treated as a long latency event.
- **Options:**
  - switch to other thread
  - lower the thread's priority
  - use special instructions to reduce resource waste (e.g. PAUSE instruction in x86)
- **Busy-waiting is better when**
  - Scheduling overhead is larger than expected wait time
  - No other task to run
  - synchronization within the OS kernel
- **HYBRID** method: busy-wait for X attempts, then switch to blocking

Intel64 PAUSE instruction

13.5.3 Spin-Wait Loops

Use the PAUSE instruction in all spin wait loops. The PAUSE instruction de-pipelines the spin-wait loop to prevent it from consuming execution resources excessively and consuming power needlessly.

When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline.

The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation and prevent the pipeline flush. However, you should try to keep spin-wait loops with PAUSE short.

Example 3-4. Use of PAUSE Instruction

```
lock:   cmp eax, a
        jne loop
        ; Code in critical section:
loop:   pause
        cmp eax, a
        jne loop
        jmp lock
```

Point-to-point synchronization:

```
while(eax ≠ a) { }
```

Intel® 64 and IA-32 Architectures Optimization Reference Manual. June 2016

How can we implement scalable locking schemes?

Option 1: Hardware-based Synchronization

- **Hardware Locks**
  - separate lock lines on the bus: holder of a lock assert the line
  - lock registers
    - set of shared registers

- **Hardware Barrier**
  - dedicated bus line using open-collector connection
    - each thread tries to pull open collector down
    - succeed only if all threads have reached barrier
  - shared counting register

- **Inflexible**
  - not good for general purpose use
  - hardwired waiting algorithm
  - fixed number of synchronization resources

- **Example: Hardware Barriers to support Message Passing Systems**
  - Fujitsu SPARC64 Xifx & A64FX have HW barrier (“inter-core barrier”) (2018)
  - IBM Blue Gene/P Barrier Network (2007)

(Incorrect) Option 2: Simple Software Locks

```
Lock:       LW R2, lock
            BNEZ R2, Lock
            SW R1, lock      /R1 = 1
            RET

Unlock:     SW R0, lock      /R0 = 0
            RET
```

how can we guarantee that these operations execute atomically?

- **PROBLEM:** lock is not atomic--two threads can gain the lock at the same time

- SOLUTION: **need new atomic read/write instructions**
  - atomically read the value of the location and set it to another value
  - return success or failure

Option 3: Software-based Synchronization with Atomic operations

- **ISA support:** most modern machines provide some form of **atomic read-modify-write (R/M/W)**
  - **IBM 370**: atomic compare-and-swap, test-and-set (1973)
  - **x86**: any memory read-modify-write instruction can be prefixed with a lock (note that some instructions are implicitly prefixed)
  - **SPARC**: atomic swap
  - **MIPS**, **PowerPC**, **RISC-V**, **ARM**: support from pairs of instructions
    - Load-locked, Store-conditional (also called: Load-linked)

these basic mechanisms are used to build software locks

Atomic Software Locks

SIMPLEST ONE: TEST_AND_SET (T&S)

```
T&S R1, var
```

Operation:

- read `var` in R1
- write 1 in `var`
- Assume var is used to store a lock:
  - success if value read in R1 is 0 (lock=0 means lock was not taken)
  - failure if it is 1 (lock=1 means lock was already taken)

MUST BE ATOMIC

Using T&S to implement a lock:

```
Lock:       T&S R1, lock_var
            BNEZ R1, Lock
            RET

Unlock:     SW R0, lock_var
            RET
```

Atomic Software Locks

- other common R/M/W atomic operations:
  - **SWAP R1, MEM_LOC**
    - exchange the content of R1 and MEM_LOC
  - **FETCH&OP**
    - example: F&A (R1, MEM_LOC, const), where const is a small value.
    - fetch MEM_LOC in R1, then add const to mem_loc
  - **COMPARE&SWAP**
    - CAS (R1, R2, MEM_LOC)
    - compare MEM_LOC to R1. If they are equal swap R2 and MEM_LOC

T&S Implementation

In MEMORY (non-cacheable lock):

- Memory controller enforces atomicity of RMW cycle
- Execute load followed by store of 1 and return the value of the load

In CACHE (cacheable lock, most common)

- protocol should be invalidate (eg MSI-invalidate).
- T&S treated as store
  - cache must acquire modified copy (M) before attempting T&S
- T&S executed in cache by cache controller before the block can be flushed.

Problem: both memory and cache variants generate large amount of memory traffic when implementing locks

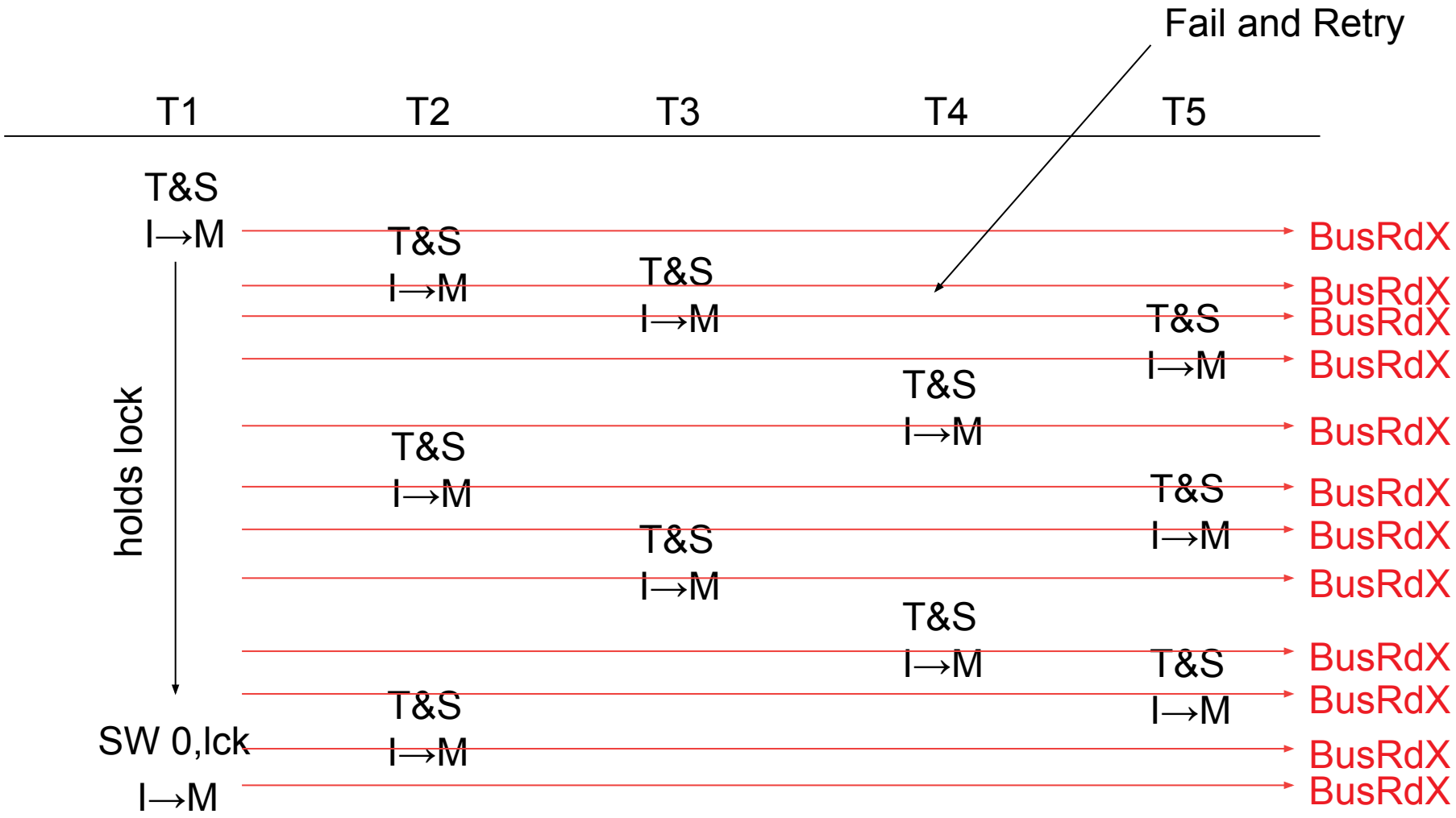
MEMORY → every T&S must reach main memory

CACHE → block containing the lock bounces back and forth between caches while threads in different cores are busy-waiting on the lock

T&S Memory Traffic Problem

- **T&S**

```
Lock:   T&S R1, lock    // PROBLEM: each thread writes '1' each iteration.
        BNEZ R1, Lock   // Generates stream of invalidations! (MSI-invalidate)
        RET
```



Reduce Frequency of Issuing T&S

- **T&S**

```
Lock:   T&S R1, lock    // PROBLEM: each thread writes '1' each iteration.
        BNEZ R1, Lock   // Generates stream of invalidations! (MSI-invalidate)
        RET
```

- **T&S WITH BACKOFF**
  - increase the delay until the next trial after every failure
  - e.g., exponential backoff
    - backoff by **k x c^{i}** at the *i*th trial

- **TEST AND TEST&SET LOCK**
  - test with ordinary loads
  - when value changes to 0, try to obtain lock with T&S
  - works well with cache. **<u>Think</u>: T&S vs T&T&S with MSI-invalidate**

```
Lock:    LW R1,lock
         BNEZ R1,Lock
         T&S R1,lock
         BNEZ R1,Lock
       RET

Unlock: SW R0,lock
        RET
```

Load-Linked and Store Conditional

Atomic RMW instructions are complex and do not fit well in a RISC pipeline:

- Require atomic execution of two memory accesses (load + store)
- Solution: introduce pair of instructions: **load-linked (LL)** and **store-conditional (SC):**
  - `LL Rx,lock`
  - `SC R1,lock`

- **LOAD-LINKED or LOAD-LOCKED (LL)**
  - LL reads `lock` into register Rx
- **STORE CONDITIONAL (SC)**
  - tries to store R1 in `lock`:
  - succeeds if no other thread has written into `lock` since LL
  - if SC succeeds the sequence LL-SC was atomic
  - if SC fails, it does not write to memory; rather it sets R1 to 0
- **SC CAN FAIL IF**
  - it detects intervening writes to lock since LL
  - it tries to get the bus, but another SC succeeds first

Advantages of Load-linked / Store-conditional

Two advantages:

- easier to implement in a pipeline (particularly for RISC ISAs)
- Flexibility

Eg: Implementation of T&S with LL + SC

```
T&S(Rx,lock): ADDI R1,R0,1
              LL Rx,lock
              SC R1,lock
              BEQZ R1, T&S
              RET
```

Fancier atomic ops can be implemented by adding code between LL and SC:

- keep it simple so that SC is likely to succeed
- avoid instructions that cannot be undone (eg, store, instructions causing exceptions)

Load-locked and Store Conditional

- **EXAMPLE: CAS**

```
CAS(Rx,Ry,X)    ADD R2,Ry,R0            /save Ry
                LL R1,X
                BNE Rx,R1,return
                SC R2,X                 /attempt to store Ry
                BEQZ R2,CAS
                ADD Ry,R1,R0            /return X in Ry
                RET
```

- **Implementation**
  - LL-bit is set when LL is executed
  - bus interface snoops update or invalidate signals and resets LL-bit
  - SC tests LL-bit, and fails if reset
  - LL-bits can track individual cache lines, larger address blocks (e.g. ARM 8-2048 bytes), or one LL-bit for the whole memory!

How to implement a T&S lock in high level languages (eg. C++)?

- **Inline assembly is not portable, how can we write portable locks?**
- **Option:** Write the lock as a regular C++ code?

```
bool lock = false;
#define LOCK_ACQUIRE() { while(lock){}; lock=true; }
#define LOCK_RELEASE() { lock=false; }
```

PROBLEM: acquire not atomic. Same as simple software locks

- **Solution 1:** Implement synchronization via library calls or assembly

  `pthread_mutex_lock() / pthread_mutex_unlock()  // Blocking`

- **Solution 2:** Extend the High Level Language with explicit synchronization variables and methods

Support for atomics in C++

C++ | **Atomic operations library**

Atomic operations library

The atomic library provides components for fine-grained atomic operations allowing for lockless concurrent programming. Each atomic operation is indivisible with regards to any other atomic operation that involves the same object. Atomic objects are free of data races.

Defined in header <atomic>

Atomic types

atomic (C++11)	atomic class template and specializations for bool, integral, and pointer types (class template)
atomic_ref (C++20)	provides atomic operations on non-atomic objects (class template)

Operations on atomic types

atomic_is_lock_free (C++11)	checks if the atomic type's operations are lock-free (function template)
atomic_store (C++11) **atomic_store_explicit** (C++11)	atomically replaces the value of the atomic object with a non-atomic argument (function template)
atomic_load (C++11) **atomic_load_explicit** (C++11)	atomically obtains the value stored in an atomic object (function template)
atomic_exchange (C++11) **atomic_exchange_explicit** (C++11)	atomically replaces the value of the atomic object with non-atomic argument and returns the old value of the atomic (function template)
atomic_compare_exchange_weak (C++11) **atomic_compare_exchange_weak_explicit** (C++11) **atomic_compare_exchange_strong** (C++11) **atomic_compare_exchange_strong_explicit** (C++11)	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (function template)

USING C++11 ATOMICS (x86_64)

```
// C++ allows to declare basic atomic data types via atomic<type>
#include <atomic>
int atomic_exchange(std::atomic<bool> *l) {
  return l->exchange(true);
}
#define LOCK_ACQUIRE(l)  while(atomic_exchange(lock))

std::atomic<bool>::exchange(bool, std::memory_order):
push rbp
mov rbp, rsp
mov QWORD PTR [rbp-24], rdi
mov eax, esi
mov DWORD PTR [rbp-32], edx
mov BYTE PTR [rbp-28], al
mov rdx, QWORD PTR [rbp-24]
movzx eax, BYTE PTR [rbp-28]
mov QWORD PTR [rbp-8], rdx
mov BYTE PTR [rbp-9], al
and BYTE PTR [rbp-9], 1
mov eax, DWORD PTR [rbp-32]
mov DWORD PTR [rbp-16], eax
movzx eax, BYTE PTR [rbp-9]
mov rdx, QWORD PTR [rbp-8]
xchg al, BYTE PTR [rdx]  <- Atomic exchange of 'al' and [rdx] memory value
test al, al
setne al
nop
pop rbp
ret
```

Compiler generates code to use the X86 atomic xchg instruction. note that **xchg** with a memory operand has an implicit **lock**

// Assembler Dump (using compiler explorer: https://godbolt.org/)

USING C++11 ATOMICS (ARM64)

```
// C++ allows to declare basic atomic data types via atomic<type>
#include <atomic>
int atomic_exchange(std::atomic<bool> *l) {
  return l->exchange(true);
}
#define LOCK_ACQUIRE(l)  while(atomic_exchange(lock))

std::atomic<bool>::exchange(bool, std::memory_order):
sub sp, sp, #32
str x0, [sp, 8]
strb w1, [sp, 7]
str w2, [sp]
ldr x0, [sp, 8]
str x0, [sp, 24]
ldrb w0, [sp, 7]
strb w0, [sp, 23]
ldr w0, [sp]
str w0, [sp, 16]
ldr x0, [sp, 24]
ldrb w1, [sp, 23]
.L4:
ldaxrb w2, [x0]        <- Load-acquire exclusive register byte.
stlxrb w3, w1, [x0]    <- Store-release exclusive register byte, returning status.
cbnz w3, .L4
and w0, w2, 255
cmp w0, 0
cset w0, ne
and w0, w0, 255
add sp, sp, 32
ret
```

Since ARM64 has no atomic exchange instruction, the compiler generates an equivalent sequence using load-linked store-conditional!

C++ hence provides portable atomics that can be used to construct locks and lock-free data structures

// Assembler Dump (using compiler explorer: https://godbolt.org/)

SUMMARY

- Synchronization: Why
- Components Of Synchronization Event
- Hardware Based Synchronization
- Software Based Synchronization
  - Atomic Read-modify-write
  - Load-linked / Store-conditional
- Synchronization In High Level Languages