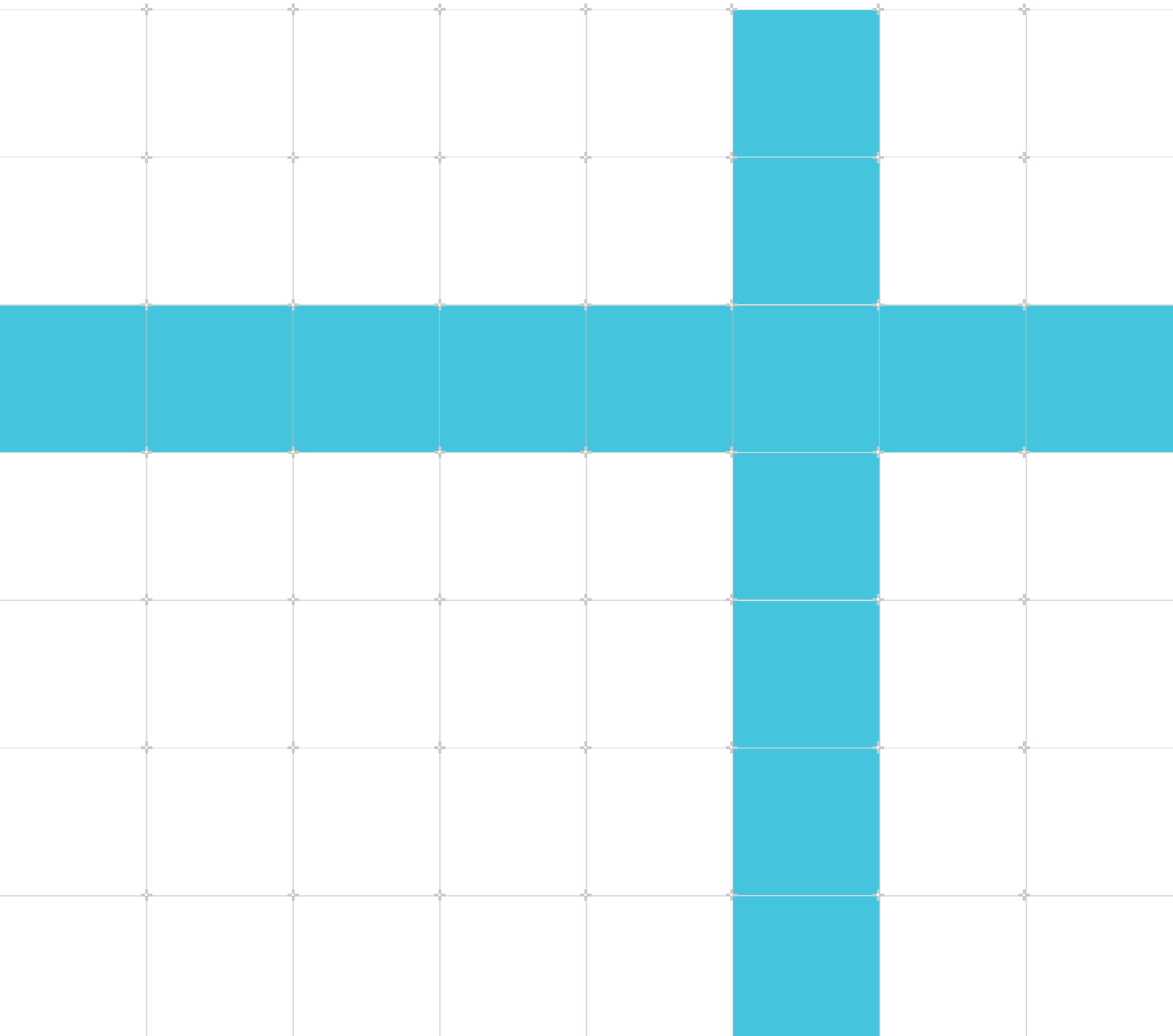
arm

Learn the architecture - Implementation Software Synchronization Primitives in A64

Version 1.0

Issue 01
110478_0100_01_en



Learn the architecture - Implementation Software Synchronization Primitives in A64

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	10 September 2025	Non-Confidential	First release

Proprietary Notice

makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

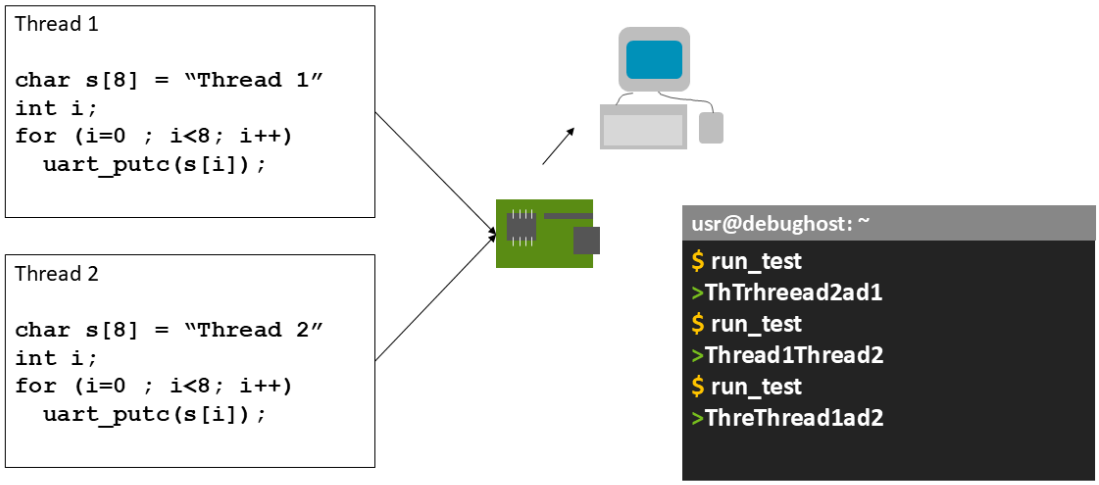
Contents

1. Background of Implementation Synchronization Primitives

A system contains resources such as peripherals or memory buffers, that are shared between multiple threads, processes, or other software agents. Software needs to carefully manage concurrent access to the shared resources to prevent race conditions.

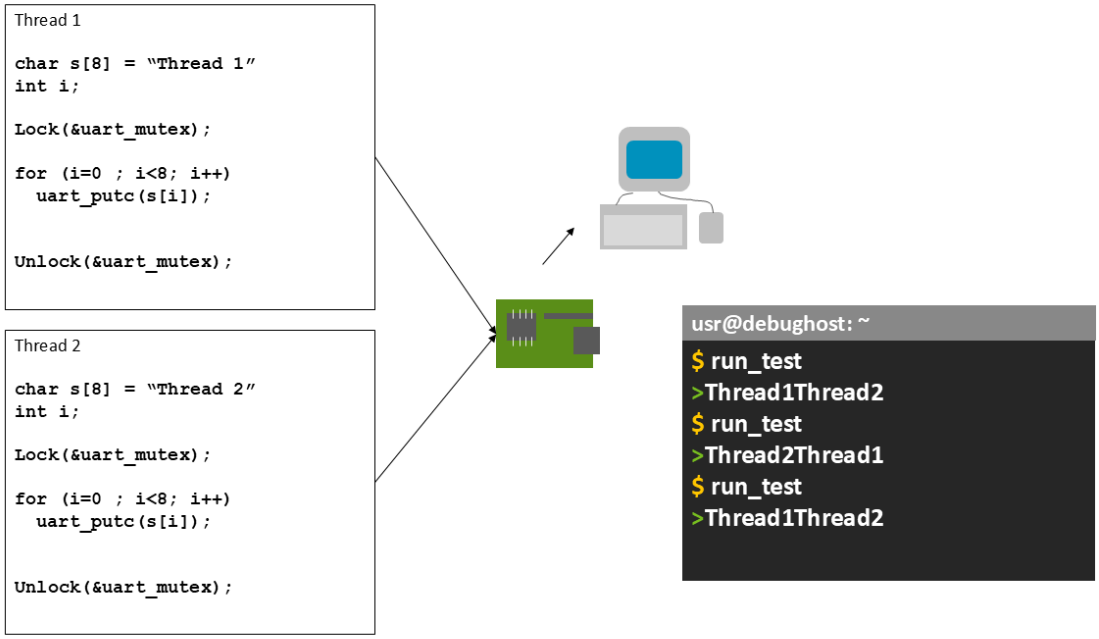
For example, if two threads try to write to the console at the same time, and the console shows results with submixed characters or other garbled combinations as the Figure 1-1 shows.

Figure 1-1: Concurrent issue between Multiple threads for console



You can prevent inappropriate concurrent execution through many approaches such as software synchronization and managing shared resources. The Figure 1-2 shows the example with a mutex access to console.

Figure 1-2: Fix concurrent issue between Multiple threads for console



This guide does not try to introduce and explain all the different options. Instead, this guide describes the Arm AArch64 synchronization primitives, simple examples showing the use of architecture features for synchronization, and some real-world examples taken from Linux kernel version 6.6.10.

2. AArch64 Atomic Instructions

This section describes the atomic instructions defined in the Arm architecture, and their use in real software implementations.

2.1 The different groups of A64 atomic instructions

The Armv8.1-A architecture introduced atomic instructions as part of the Large System Extensions (LSE). It improves the performance and scalability of atomic operations in systems with many processors. With LSE, atomic instructions provide a non-interruptible read-modify-write sequence in a single instruction. The atomic instructions can perform simple arithmetic or logical operations on the specified memory location, and return the original value to the processor.

The instructions include :

- Compare and swap instructions, `CAS` and `CASP`
- Atomic memory operation instructions, `LD<OP>` and `ST<OP>`, where `OP` is one of arithmetic and bitwise operations
- Swap instruction, `SWP`

Introduced in Armv9.4-A, FEAT_LSE128 extends FEAT_LSE to support atomics on 128-bit data. FEAT_LSE128 enables efficient synchronization on large data types, such as management of 128-bit translation table descriptors.

FEAT_LSE128 includes the following instructions:

- `LDCLRP`, atomic bit clear on quadword in memory, atomically loads a 128-bit quadword from memory
- `LDSETP`, atomic bit set on quadword in memory, atomically loads a 128-bit quadword from memory
- `SWPP`, swap quadword in memory, atomically loads a 128-bit quadword from memory, and stores a new value to memory

These instructions can optionally have Acquire and Release semantics:

- `A` Acquire: Ensures all younger - later in program order loads and stores do not observe before the acquire-load
- `L` Release: Ensures all older - program order prior loads and stores have completed and become observable before this store-release becomes observable
- `AL` Acquire-Release: Ensures memory operations with both acquire and release semantics

Example: `LDADDAL X0, X1, [X2]`, atomic add with Acquire-Release semantics.

Note

- ST<op> operations do not have acquire semantics
- For more information on memory ordering, and acquire and release semantics, see the Memory Systems, Ordering, and Barriers guide.

The following table describes each instruction.

Feature extension	Instruction	Syntax	Description	Key use case
FEAT_LSE	CAS	CAS <Xs>, <Xt>, [<Xn>]	Atomically compare the value at memory address [Xn] with Xs (32/64-bit), and return original value to Xs. If the compared value are equal, replace value in [Xn] with Xt	Lock-free data structures
	CASP	CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn>]	Compare two consecutive 32/64-bit values at [Xn] with register Xs, X(s+1), and return original value to Xs, X(s+1). If the compared value are equal, swap the pair in [Xn] with Xt, X(t+1).	Atomic updates to paired data (e.g., pointers with metadata).
	LD<OP>	LDADD/LDEOR/ LDCLR <Xs>, <Xt>, [<Xn>]	Atomically load from [Xn], perform OP (ADD/EOR/CLR/ SET/ SMAX/SMIN/UMAX/UMIN) with Xt, store result back, and return the original value to Xs.	Atomic arithmetic and bitwise operations (counters, flags).
	ST<OP>	STADD/STEOR/ STCLR <Xs>, [<Xn>]	Atomically perform OP (ADD/EOR/CLR/SET/SMAX/SMIN/ UMAX/UMIN) on the value at [Xn] using Xs, then store the result. No return value.	Constrained atomic updates without needing a load result.
	SWP	SWP <Xs>, <Xt>, [<Xn>]	Atomically swap the value at [Xn] with Xt, then return the original value to Xs.	Simple mutual exclusion or value exchange.
FEAT_LSE128	LDCLRP	LDCLRP <Xd>, <Xs>, [<Xn>]	Atomically load the value at [Xn], clears (AND-NOT) the bits specified by Xs, store the result back, and return the original value to Xd.	Atomic operation of 128-bit page table descriptors. Clearing flag bits atomically in multi-core systems (e.g., lock release).
	LDSETP	LDSETP <Xd>, <Xs>, [<Xn>]	Atomically load the value at [Xn], set (OR) the bits specified by Xs, store the result back, and return the original value to Xd.	Setting status flags atomically (e.g., task scheduling).
	SWPP	SWPP <Xt1>, <Xt2>, [<Xn>]	Atomically swap the value at [Xn] with Xt1 and Xt2, store the original value into Xt1 and Xt2 .	Swapping pointers with metadata in lock-free data structures. Pair-atomicity for aligned 128-bit memory.

The CAS, LD<OP>, ST<OP>, and SWP instructions also support the 8/16-bit data type variant, this is denoted by 'B' or 'H' suffix to the instruction mnemonic.

2.2 Examples with atomic instructions

This section shows some examples with atomic instructions.

2.2.1 A simple lock()/unlock() example

Spinlocks offer a simple mutual exclusion mechanism with low storage overhead. Concurrent accesses by multiple threads of execution often slow down both acquisition of the lock as well as the release. The following is a simple code example implementing a spinlock. The critical section is abstracted out, whereas real code has a return from the acquire function. After executing the critical section, the real code calls the release function.

```
; lock() - lock acquire portion

    MOV X1, #0x1

Spin:
    LDR   X0, [<lock>]      ;  load lock
    CBNZ  X0, Spin          ;  branch back to Spin if lock is taken
    CASA  X0, X1, [<lock>]  ;  attempt to lock
    CBNZ  X0, Spin          ;  branch back to Spin if lock failed

; critical section
    ...

; unlock() - lock release portion
    STLR  XZR, [<lock>]
```

2.2.2 A lock-free stack example

Software typically uses lock-free updates to shared locations to avoid the overhead of using locks. Lock-free code usually starts with a load to the shared location, then performs some dependent computation, and completes the atomic update with a Compare-And-Swap (CAS) atomic operation. The success of the final CAS operation depends on the value observed by the load at the location not being changed by another thread. The following is an example for updating a lock-free linked list:

```
Loop:
    LDAR   X2, [X0]       ; load the head element
    STR    X2, [X1]       ; store the head into old location
    MOV    X3, X2         ; save original expected value in X2
    CASAL  X3, X1, [X0]   ; atomically point the head to the next element
    CMP    X3, X2
    B.ne   Loop           ; branch back to Loop if head is not updated
```

2.2.3 A thread barrier example

Thread barriers provide a synchronization point that ensures all threads of execution have reached that point before any participating threads continue execution. Thread barriers are common between parallel sections of code in various applications. A typical thread barrier implementation

might include a counting variable to detect when all threads have reached the synchronization point.

Below is an example of a code section that operates on this count value while detecting the last synchronizing thread:

```
    MOV       W1, #-1        ; Set the increment value (negative value for
 subtraction)
    LDADDAL   W1, W1, [X0]   ; Perform LDADD with ACQ_REL semantics, value in [X0] =
 old value, the new value in [X0] = old value - 1

wait_loop:                   ; Spin-wait for other threads
    LDAR      W1, [X0]       ; Load-Acquire current counter value
    CBNZ      W1, wait_loop  ; Continue waiting if the current value is not zero

continue:
                             ; All threads resume here after synchronization
```

2.2.4 Atomic bit manipulation example

Management of thread-safe status flags should be synchronized due to access by multiple threads and preemptive scheduling. Before accessing a shared resource such as a hardware peripheral like GPU cores or DMA channels, a thread sets a status bit atomically to indicate the shared resource is already allocated.

Below is the assembler code.

```
; Set bit 2 of a 64-bit status register atomically
; Assume X2 points to the status register

    MOV      X1, #0x4          ; Bitmask: 0b100 (bit 2)

1:
    LDSETAL  X1, X0, [X2]      ; Atomically set bit 2, store old value in x0
    TBNZ     X0, #2, 1b        ; Retry if bit 2 was already set (optional)
```

The `LDSETAL` instruction finishes atomically loading the 64-bit value from [X2], performing a bitwise OR with the bitmask 0x4 (setting bit 2), storing the result back to [X2] and returning the original value to X0.

The AL suffix ensures acquire-release semantics and guarantees the ordering of memory accesses to different memory locations.

This example also suits task scheduling, where a task is marked as “running” without locks in a real-time OS.

2.3 The memory attribute of address which is atomic accessed

The memory types for which it is architecturally guaranteed that the atomic instructions are atomic are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient

In some implementations, and for some memory types, the properties of atomicity can be met only by functionality outside the PE. Some system implementations might not support atomic instructions for all regions of the memory. In particular, this can apply to:

- Any type of memory in the system that does not support hardware cache coherency
- Device, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency

Software programmers care whether atomicity is enforced inside or outside the CPU, because it directly affects the performance, portability, and reliability of concurrent issues. When atomic operations rely on external components like interconnects or memory controllers, behavior can vary across systems, especially for memory types like device or non-cacheable regions that might not support hardware cache coherency. This variability can lead to race conditions, inconsistent data, or issues that are difficult to debug. More, known the functionality of inside or outside the CPU is useful to get the memory characteristics. Understanding of the memory characteristics is required to support atomic or exclusive operations.

Atomic instructions implemented inside the CPU are typically faster and more predictable, making them preferable for performance-critical tasks. Understanding where atomicity is guaranteed helps developers make informed decisions about memory mapping, synchronization strategies, and system compatibility.

Note

- Some implementations might treat some cacheable type as non-cacheable.
- If the atomic operations are not atomic in regard to other agents that access memory, then performing an atomic operation to such a location can have side effects. See at section “Possible implementaion restrictions on using atomic instructions” of Arm ARM.

Atomic operations can be executed in different places of the memory system, depending on cache attributes and cache coherence status of the targeted memory location:

- Near atomic: executed locally. If the instruction hits in the L1 data cache in a unique state then it is performed as a near atomic in the L1 memory system.
- Far atomic : instruction forwarded externally to the CPU core. For load-atomics except the `ST<op>` instructions, the processor attempts to fetch the targeted cache line into L1D so that a

near operation can be performed. While, Store atomics are by default done far so the operation is sent into the memory system by the processor.

If the operation misses everywhere within the cluster, and the master interface is configured as CHI, and the interconnect supports far atomics, then the atomic is passed on to the interconnect to perform the operation. If the operation hits anywhere inside the cluster, or the interconnect does not support atomics, the L3 memory system performs the atomic operation and allocates the line into the L3 cache if it is not already there.


Note

- In some CPU implementations, such as Arm Cortex-A55, the CPUECTLR register can modify the behavior of the atomic instructions.
- The conditions for deciding near and far atomic implementations are more complicated, vary between systems, and even can be configured by privileged software.

2.4 Usage in the Linux Kernel

This section gives examples of using the atomic instructions in real-world software.

2.4.1 Typical atomic operation

Definitions of atomic instructions in Linux kernel locates in the `/arch/arm64/include/asm/atomic_lse.h`. For example, the following code is a typical implementation of atomic instructions about FEAT_LSE. It generates the various function calls for atomic operation with different memory ordering.

```
#define ATOMIC_FETCH_OP(name, mb, op, asm_op, cl...)                    \
static __always_inline int                                              \
__lse_atomic_fetch_##op##name(int i, atomic_t *v)                       \
{                                                                       \
    int old;                                                            \
                                                                        \
    asm volatile(                                                       \
     __LSE_PREAMBLE                                                     \  // #define
 __LSE_PREAMBLE ".arch_extension lse \n"                                \
    "    " #asm_op #mb " %w[i], %w[old], %[v]"                          \  // Template for
 inline assembler
    : [v] "+Q" (v->counter),                                            \  // Output:
      [old] "=r" (old)                                                  \  // Output: old value
    : [i] "r" (i)                                                       \
    : cl);                                                              \  // Input
                                                                        \  // Clobber list
    return old;                                                         \  // The address of old
 value
}

#define ATOMIC_FETCH_OPS(op, asm_op)                                    \
    ATOMIC_FETCH_OP(_relaxed,   , op, asm_op)                           \  // Relaxed semantic
    ATOMIC_FETCH_OP(_acquire,  a, op, asm_op, "memory")                 \  // Acquire semantic
    ATOMIC_FETCH_OP(_release,  l, op, asm_op, "memory")                 \  // Release semantic
    ATOMIC_FETCH_OP(        , al, op, asm_op, "memory")                    // Acquire-Release
 semantic
```

```
ATOMIC_FETCH_OPS(andnot, ldclr)    // Generate the function atomic_fetch_andnot_*()
ATOMIC_FETCH_OPS(or, ldset)        // Generate the function atomic_fetch_or_*()
ATOMIC_FETCH_OPS(xor, ldeor)       // Generate the function atomic_fetch_xor_*()
ATOMIC_FETCH_OPS(add, ldadd)       // Generate the function atomic_fetch_add_*()

#undef ATOMIC_FETCH_OP
#undef ATOMIC_FETCH_OPS
```

With expanding the macro `ATOMIC_FETCH_OPS(add, ldadd)`, 4 functions will be generated and one of them is like below.

```
static __always_inline int
__lse_atomic_fetch_add_acquire(int i, atomic_t *v) {
    int old;
    asm volatile(
        __LSE_PREAMBLE
        "ldadda %w[i], %w[old], %[v]"
        : [v] "+Q" (v->counter), [old] "=r" (old)
        : [i] "r" (i)
        : "memory"
    );
    return old;
}
```

These functions can be used for spinlock/mutex, reference counting, lock-free lists, and read-copy-update.

2.4.2 An implementation of SpinLock in Linux kernel

The `hyp_spin_lock` function below locates in `arch/arm64/kvm/hyp/include/nvhe/spinlock.h`. It implements a ticket-based spinlock optimized based on the `LDADD` atomic instruction. The code is designed for low-level synchronization, such as in a hypervisor or kernel context, when multiple CPUs or Virtual Machines access the shared data structure.

When the use of ticket locks is requested, the ticket locks determine the order of the users for the critical sections. A ticket lock allocates each thread a ticket number when it first requests the lock, and then compares that number with the current number for the lock. If they are the same, then the critical section can be entered. Otherwise, the thread waits until the current number is equal to the ticket number for that thread.

The reading of the current number of the lock needs acquire semantics for the lock to be acquired.

```
static inline void hyp_spin_lock(hyp_spinlock_t *lock)
{
    u32 tmp;
    hyp_spinlock_t lockval, newval;

    asm volatile(

    /* Atomically increment the next ticket. */

    ARM64_LSE_ATOMIC_INSN(
    /* LL/SC */
    ...

    /* LSE atomics */
```

```
"   mov %w2, #(1 << 16)          \n"   // Prepare the increment value
"   ldadda  %w2, %w0, %3         \n"   // Atomic add to the counter with acquire
 semantics
    __nops(3))

    /* Did we get the lock? */
"   eor %w1, %w0, %w0, ror #16   \n"
"   cbz %w1, 3f                  \n"

    /*
     * No: spin on the owner. Send a local event to avoid missing an
     * unlock before the exclusive load.
     */

"   sevl                         \n"
"   2:  wfe                      \n"
"   ldaxrh  %w2, %4              \n"
"   eor %w1, %w2, %w0, lsr #16   \n"
"   cbnz    %w1, 2b              \n"

    /* We got the lock. Critical section starts here. */
"   3:"
    : "=&r" (lockval), "=&r" (newval), "=&r" (tmp), "+Q" (*lock)
    : "Q" (lock->owner)
    : "memory");
}
```

Releasing the ticket lock simply involves incrementing the current ticket number and using a store release.

```
static inline void hyp_spin_unlock(hyp_spinlock_t *lock)
{
    u64 tmp;

    asm volatile(
    ARM64_LSE_ATOMIC_INSN(
    /* LL/SC */
    ...

    /* LSE atomics */
    "   mov      %w1, #1\n"   // Prepare the increment value (1)
    "   staddlh  %w1, %0\n"   // Atomic add-release to owner (16-bit)

    __nops(1))                // Padding for alignment
    : "=Q" (lock->owner), "=&r" (tmp)
    :
    : "memory");
}
```

This implementation of spinlock is a combination of ticket fairness, atomic operation, and low-power waiting. It is suitable for medium contention, low-latency scenarios such as modifying the page table, protecting per-CPU data in a multi-core system and updating VM state in a hypervisor.

3. Exclusives

The Load-Exclusive and Store-Exclusive, exclusive access instructions are implemented for the Load Link and Store Conditional (LL-SC) mechanism from Armv6 architecture. They split the operation of atomically updating memory into two separate steps. Together, they provide atomic updates with exclusive monitors that track exclusive memory accesses.

This section describes the A64 exclusive memory access instructions, the use of them to perform atomic memory accesses with exclusive monitor, and practical use cases of this mechanism.

3.1 Overview of Load and Store exclusive access instructions

The A64 instruction set has two instructions for generating an exclusive access to a memory location:

- Load Exclusive (LDXR)
- Store Exclusive (STXR)

Both LDXR and STXR support 8/16/32/64-bit data sizes. LDXP and STXP support 128-bit atomic operations.

Instruction Syntax	Description	Key Use Case
`LDXR <Xd>, [<Xn>]`	Loads a 32/64-bit value from memory address `[Xn]` into register `Xd`, and marks the memory location as exclusively accessed by the core.	Starts an exclusive operation. Reads a value before attempting to modify it atomically.
`STXR <Ws>, <Xt>, [Xn]`	Attempts to exclusively store a 32/64-bit value from register `Xt` to memory address `[Xn]`. Returns success/failure in `Ws`: • Ws = 0: Success(no other core modified `[Xn]` since LDXR) • Ws != 0: Failure.	Completes an exclusive operation. Must follow LDXR to ensure atomicity. Requires retry loops on failure.

An `LDXR` returns the current value of the location and marks the location as exclusive for this processor. A subsequent `STXR` to the same location only succeed if the marker (exclusive) is still present.

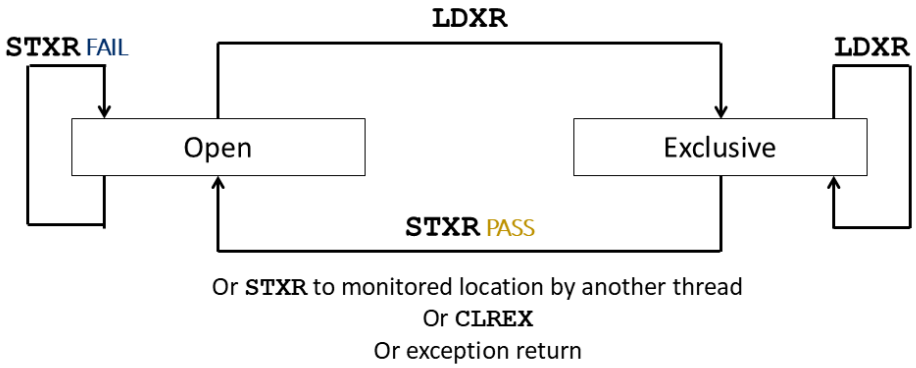
Consider the following simple example of a function that performs an atomic increment:

```
atomic_inc:
    LDXR W1, [W0]        // Read current value
    ADD  W1, W1, #1      // Add 1 to read value
    STXR W2, W1, [X0]    // Attempt to update value in memory
    CBNZ W2, atomic_inc  // If store does not succeed, try again
```

```
    RET
```

The LDXR marks the location of the value being incremented as exclusive for this thread. A write by any other thread to the monitored region clears the marker and causes STXR to fail. Therefore, if the STXR succeeds, software knows that no other thread has updated the location between the exclusive read and exclusive write, providing the atomic guarantee.

Figure 3-1: The state change process of Exclusive access



The exclusive state of an address is maintained by a piece of hardware known as an Exclusive Monitor, which can monitor only one address per processor.

The exclusive access instructions also support acquire and release semantics.

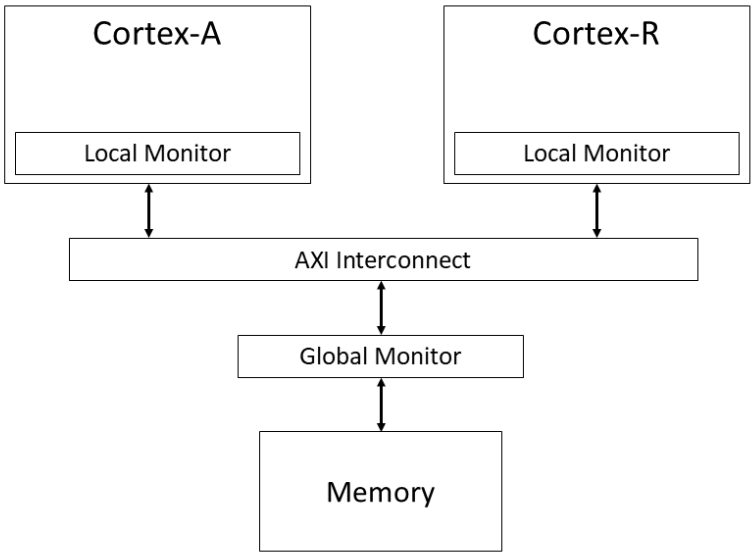
- Load-Acquire Exclusive (LDAXR): Ensures that all memory accesses after the load are not reordered before it. This guarantees visibility of shared data after acquiring a lock.
- Store-Release Exclusive (STLXR): Ensures that all memory accesses before the store are not reordered after it. This ensures changes made in critical section are visible to other threads when releasing a lock.

3.2 Exclusive monitors

An exclusive monitor is a simple state machine, with the possible states Open and Exclusive. To support synchronization between processors, a system must implement two sets of monitors: Local Monitor and Global Monitor.

The figure 3-2 shows an example system consisting of one Cortex-A series processor, and one Cortex-R series processor, and a memory device shared between the two.

Figure 3-2: Local monitors and global monitors in a multi-core system



3.2.1 Local monitor

Each core has its own local monitor. All Exclusive accesses are checked against the core's local monitor. Accesses to regions marked as Non-shareable are only required to check the local monitor, accesses to regions marked as Shared additionally check the global monitor.

Note

If the location is cacheable, the synchronization might take place without any external bus transactions. The result of an atomic operation is visible to other observers and processors later. But other observers might not know about the operation until they read the targeted memory location.

3.2.2 Global monitor

A global monitor tracks exclusive accesses to memory regions marked as Shareable. Any Store-Exclusive operation that targets Shareable memory must check its local monitor and the global monitor to determine whether it can update memory.

For example, if software executing on the Cortex-A CPU in figure 3-2 must synchronize its operation with software executing on the Cortex-R CPU, it can do this using a mutex placed in Shareable memory. The resulting Load-Exclusive and Store-Exclusive instructions access both the local monitor and the global monitor.

Although we describe the local and global monitors as being different things, they might actually share logic. For example, one implementation approach is to use a combination of the per-core local monitors and the cache coherency logic to provide global monitor functionality for cacheable locations.

It is an architectural requirement that the following memory types are able to function with a global monitor:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.


Note

- Other memory types are not guaranteed to be covered by a global monitor, although it is also possible to implement a global monitor as part of the system-on-chip level memory system. If this type of monitor is present, it is often built into the RAM controller of memory interconnect. Refer to the documentation of the related SoC to see what is available.
- If software performs an exclusive access to a location where they are not supported, the exclusive stores do not function correctly. The processor might generate an abort in these scenarios.

3.2.3 Exclusive reservation granule

Exclusive monitors typically do not record precise addresses, instead recording only the upper address bits. The Exclusive Reservation Granule (ERG), reported in the CTR_EL0 register, tells software what the granularity of address monitoring is. Two locations in the same ERG appear to be the same to the monitors.

The granularity of monitoring can be important for efficient software. For example, consider the atomic increment example from earlier.

Figure 3-3: Two threads access the same cacheline

Thread 0

```
LDXR w1, [Addr A]
ADD  w1, w1, #1
STXR w2, w1, [Addr A]
```

Thread 1

```
LDXR w1, [Addr B]
ADD  w1, w1, #1
STXR w2, w1, [Addr B]
```

The two threads should be able to complete in parallel because they are accessing different locations. However, if the Addr A and Addr B are in the same ERG, they appear to be the same location to the monitors. This can lead to a false negative, where the `STXR` by Thread 0 clears the monitor of Thread 1.

Typically, the ERG is one cache line.

3.2.4 Clearing local monitors on ERET and CLREX

When an operating system performs a context switch, it must reset the local monitor to open state, to prevent false positives occurring. For example, if a context switch schedules out a process after the process has performed a Load-Exclusive but before it performs the Store-Exclusive, the Store-Exclusive returns a failure when the process resumes, and memory is not updated.

Starting from the Armv8 architecture, the clearing of the monitors for context switching is automatic in most cases. For example, local monitor is automatically cleared with the execution of `ERET` instruction.

In the Armv7 and earlier architecture, software executes manually `CLREX` on any kind of context switch to avoid one thread picking up the exclusive state of the previous thread.

3.3 Examples of simple lock()/unlock() using exclusives

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region, as the following example shows. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

For a critical region, the requirement on taking a lock is usually for acquire semantic, while the clearing of a lock requires release semantic. The Store-Exclusive fails and is retried if there is a store operation to the location being monitored between the Load-Exclusive and Store-Exclusive.

```
; void lock (lock_t * ptr)

    PRFM PSTL1KEEP, [X0]    ; Preload into cache in unique state

Loop:
    MOV     W2, #1          ;
    LDAXR   W1, [X0]        ; Read lock with acquire
    CBNZ    W1, Loop        ; Check if 0
    STXR    W1, W2, [X0]    ; Attemp to store new value
    CBNZ    W1, Loop        ; Test if store succeeded and retry if not
```

Releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single thread is allowed to write to the lock. However, any thread must observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, the unlock call needs release semantics.

```
; void unlock ( lock_t * ptr )

    STLR     WZR, [X0]      ; Clear the lock with release semantics
```

3.3.1 Practical uses

In the Linux kernel, the LDAXR and STLXR instructions are critical for implementing spinlock on AArch64 architecture. They are one implementation of Load-Linked/Store-Conditional (LL/SC) mechanism, ensuring safe concurrent access to shared memory.

Like the atomic operation to `hyp_spin_lock`, the exclusive access instruction can also implement this function. At `arch/arm64/kvm/hyp/include/nvhe/spinlock.h`,

```
static inline void hyp_spin_lock(hyp_spinlock_t *lock)
{
    u32 tmp;
    hyp_spinlock_t lockval, newval;

    asm volatile(
    /* Atomically increment the next ticket. */
    ARM64_LSE_ATOMIC_INSN(
    /* LL/SC */
"   prfm    pstl1strm, %3       \n"    // Prefetch the lock for write
"   1:  ldaxr   %w0, %3         \n"    // Load-Acquire Exclusive the lock value (32-
bit)
"   add %w1, %w0, #(1 << 16)    \n"    // Increment the lock value
"   stxr    %w2, %w1, %3        \n"    // Store-Exclusive the new value
"   cbnz    %w2, 1b             \n",   // Retry if store failed

    /* LSE atomics */
    ...

    __nops(3))

    ...

    /* We got the lock. Critical section starts here. */
"   3:"
    : "=&r" (lockval), "=&r" (newval), "=&r" (tmp), "+Q" (*lock)
    : "Q" (lock->owner)
    : "memory");
}
```

```
static inline void hyp_spin_unlock(hyp_spinlock_t *lock)
{
    u64 tmp;

    asm volatile(
    ARM64_LSE_ATOMIC_INSN(
    /* LL/SC */
"   ldrh    %w1, %0  \n"       // Load the 16-bit old value into register %w1
"   add %w1, %w1, #1  \n"      // Increment the value
"   stlrh   %w1, %0    ",      // Store-Release the new value

    /* LSE atomics */
    ...

    __nops(1))
    : "=Q" (lock->owner), "=&r" (tmp)
    :
    : "memory");
}
```

4. Use of WFE

If a piece of code fails to lock a mutex, or to decrement a semaphore, it can either:

- Retry until successful
- Return an error code, indicating that the operation cannot succeed at this time.

In many situations, the expected result is to return only when the thread has acquired the lock. However, looping and retrying consumes power without performing any useful work. Because it is not possible to acquire the resource until an external agent modifies the synchronization variable, a better solution is to either:

- Put the processor into a low-power state
- Request that the operating system schedules in a new process, and retry at a later point

Software can use the Wait For Event mechanism, `WFE`, to minimize the energy cost of polling variables by putting the PE into a low-power state with clocks disabled, until a wakeup event occurs.

The wake-up events for WFE include:

- The execution of an SEV instruction on any processor in a multi-core system
- An unmasked interrupt
- Writing to a location in the monitored region from another PE which clears the global monitor
- Event stream from generic timer

The Arm architecture provides a user space accessible counter-timer, which is incremented at a fixed but machine-specific rate. Software can wait until the counter-timer reaches the desired value. To make the waiting more power efficient, Armv8.7-A includes the WFET instruction, which enables the processor to enter a low-power state for a set time, or until an event is received.

4.1 Examples of lock() with WFE

`WFE` can be used with simple locks or with ticket locks.

```
; void lock(*ptr)

acquire_lock:
    LDAXR  W1, [X0]                ; Read lock with acquire to set the monitor for
 WFE
    CBNZ   W1, wait_loop           ; Check if 0, if not, enter into the wait_loop
    MOV    W2, #1                  ; Set the locked value
    CASA   W1, W2, [X0]            ; Attempt to store new value
    CBNZ   W1, wait_loop           ; Test if store succeeded and retry if not

    RET

wait_loop:
    WFE                            ; Wait for event
```

```
    B     acquire_lock              ; Retry acquiring the lock
```

In the before `hyp_spin_lock()` function, a core/thread which obtains the ticket lock checks first if the held ticket is the current ticket. If the current ticket is not the same as held, the core enters a low-power wait state by executing the below code sequence.

```
...
2:
    WFE                           ; Wait For Event (saves power while spinning)

    LDAXRH  W2, [X0]              ; Read the current ticket
    EOR     W1, W2, W3, LSR #16   ; Compare loaded the current ticket with the
 thread's ticket
    CBNZ    W1, 2b                ; Loop back if not equal
...
```

Here, WFE puts the CPU in a low-power state until an event occurs. The Load-Exclusive instruction moves the monitor into the exclusive state to create an event when the monitor changes state, which wakes up the low-powered CPU.

4.2 Event on losing exclusive

WFE does have advantage of decreasing the power consumption when a core/thread is waiting for a lock. However, using the WFE might increase memory contention.

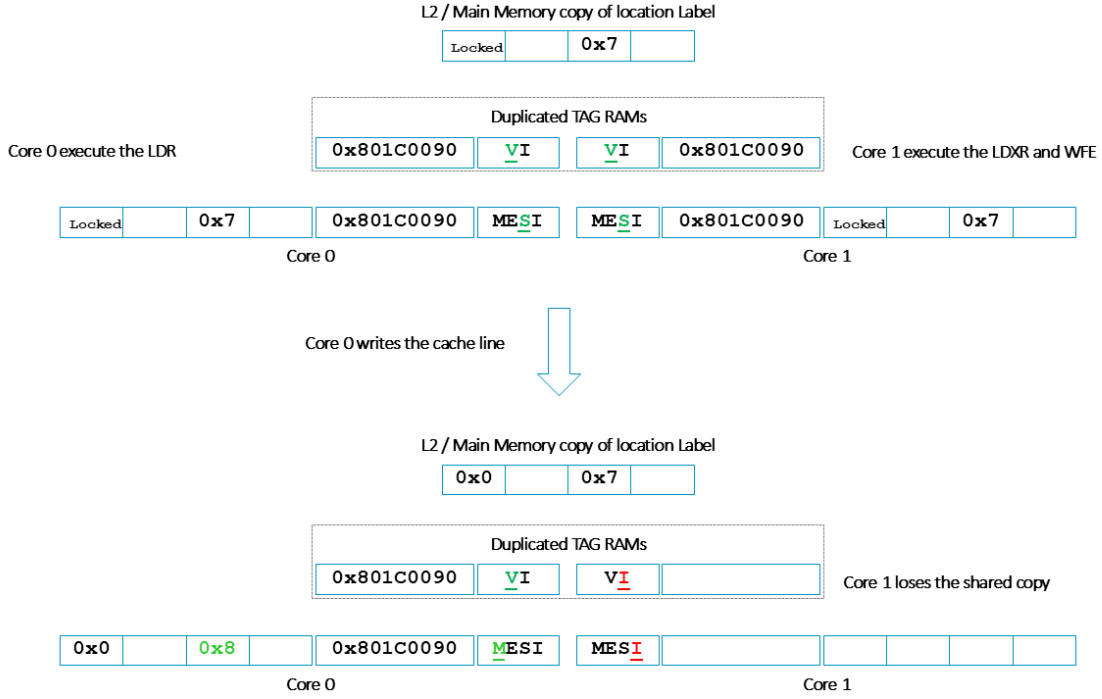
Memory contention is when multiple threads or PEs access the same cache line, and at least one access is a write operation. Memory contention might change the monitored exclusive status, due to the MESI-based cache coherency protocols.

For example, in figure 4-1, both the local caches of Core 0 and Core 1 share the same cache line, while `LDXR` of Core 1 causes the processor to obtain additional exclusive monitor status. Because the value of the lock located in this cache line is held by some other threads, Core 1 is halted through WFE instruction and waits for an event to be woken up.

Any update to the monitored cache line generates a local signal which can be a wakeup event. Therefore, with MESI, a write by Core 0 requires the processor to get the corresponding cache line in unique state, which causes the invalidation of that line from Core 1 and wakes up Core 1. The monitor in Core 1 loses the unique status. Therefore, Core 1 performs another load exclusive to check the locked value. This load exclusive operation cannot be avoided even if a false wake up occurs.

The event stream wake ups also contribute to the memory contention. Because all Cores in WFE, in addition to Core 1, are woken up at the same time and then load the monitored memory location at the same time, that causes extra memory contention.

Figure 4-1: Memory contention with MESI



Writes do not have to be explicit. Like above, cache maintenance operation and the software or hardware prefetching, can have the same memory contention effect.

Note

More details about usage of WFE, are referred from the blog - "The when, why and how of waiting and backoff in multi-threaded applications on Arm".

---

5. Practical use of synchronization

This section describes how you can use the software synchronization primitives.

5.1 How C/C++ atomic operations map to Arm architectural support

Atomic operation is an indivisible operation on a memory location. This can be load, store, exchange, compare, or arithmetic operation. You can use atomics to define higher level primitives including locks and concurrent queues. ISO C/C++ defines a range of supported atomic types and operations.

This section shows a mapping from an atomic operation to a sequence of AArch64 instructions in Arm architecture.

A stable Application Binary Interface makes fully uses the new features provided by the Arm architecture. An atomic ABI is a specification of the mappings.

The following table shows some of the mappings that are interoperable, with `val` as 64-bit types.

Atomic operation	AArch64	

`exchange(loc, val, relaxed)`	Arm8-A	`loop:`
		`LDXR  X0, [X1]`
		`STXR  X3, X2, [X1]`
		`CBNZ  X3, loop`

	FEAT_LSE	`SWP  X2, X0, [X1]`

`fetch_add(loc, val, acquire)`	Armv8-A	`loop:`
		`LDAXR  X0, [X1]`
		`ADD  X2, X2, X0`
		`STXR  X3, X2, [X1]`
		`CBNZ  X3, loop`

	FEAT_LSE	`LDADDA X0, X2, [X1]`
`store(loc, val, relaxed)`	Arm8-A	`STR  X2, [X1]`

Atomic operation	AArch64	

`compare_exchange_strong( loc, exp, val, release, release)`	Armv8-A	`MOV X4, X0`
		`loop:`
		`LDXR X0, [X1]`
		`CMP X0, X4`
		`B.NE fail`
		`STLXR X3, X2, [X1]`
		`CBNZ X3, loop`
		`fail:`

	FEAT_LSE	`CASL X0, X2, [X1]`

5.1.1 How to implement the atomic operations

Instead of using platform-specific inline assembly and kernel helpers, the builtin functions can implement the atomic operations. According to the GCC documentation, there are two alternatives:

- `__sync` builtin functions for atomic memory access, which are as of GCC-4.1. The `__sync` builtin functions are more widely supported in the older versions of GCC, but are now considered legacy and is deprecated.
- `__atomic` builtin functions for memory model aware atomic operations, which are as of GCC-4.7. Both of these are clang-compatible. The `__atomic` builtin functions are the replacement for `__sync`, and do provide support for ARMv5 through fall-back library functions on libatomic, but only support the C++11 memory model. As per the GCC manual, atomic functions have both a generic version and a typed version. This duality is reflected in the libatomic codebase.

With the introduction of LSE and real atomic operations, if the programmers do not want to compile twice: once with LSE and once without, the solution is to emit calls to helper functions - `__aarch64`, which check at runtime whether the system supports LSE.

The following example shows how to implement a lock by using the atomic operation in C language.

```
#include <stdatomic.h>
#include <stdint.h>


// Ticket Lock Structure
typedef struct {
    atomic_uint_least64_t next_ticket;             // Next available ticket number
    atomic_uint_least64_t current_ticket;          // Currently serving ticket
 number
} ArmTicketLock;

// Initialize the ticket lock
```

```
void arm_ticket_init(ArmTicketLock* lock) {
    atomic_init(&lock->next_ticket, 0);           // Start with ticket 0
    atomic_init(&lock->current_ticket, 0);        // Start serving ticket 0
}

// Acquire the lock
void arm_ticket_lock(ArmTicketLock* lock) {
    // 1. Get ticket number and increment next_ticket atomically:
    // - Uses relaxed ordering because ticket order doesn't affect critical section
 access
    // - fetch_add returns the pre-increment value (our ticket number)

    uint_least64_t my_ticket = atomic_fetch_add_explicit(
        &lock->next_ticket, 1, memory_order_relaxed
    );

    // 2. Wait until our ticket is served:

    while (1) {

        // Check current serving ticket with acquire semantics:
        // - Ensures all previous writes from the previous lock holder are visible
        // - Synchronizes with the release operation in unlock()

        uint_least64_t curr = atomic_load_explicit(
            &lock->current_ticket, memory_order_acquire
        );

        // Exit loop when our ticket is called

        if (curr == my_ticket) break;

        __DSB();

        // WFE puts CPU in low-power state, and will be woken by SEV from unlock()

        __builtin_arm_wfe();
    }
}

// Release the lock
void arm_ticket_unlock(ArmTicketLock* lock) {

    // Release the lock by incrementing current_ticket:
    // - Uses release remantics: ensures all our critical section writes are visible
 to the next lock holder
    // - Atomic increment ensures safe transition to next ticket

    atomic_fetch_add_explicit(
        &lock->current_ticket, 1, memory_order_release
    );

    // SEV wakes all cores sleeping via WFE

    __builtin_arm_sev();
}
```

5.2 Differences to other common architecture

This section compares how atomic operations are implemented and used on AArch64 versus x86, through memory models, instruction sequences, barriers, and compiler mappings.

5.2.1 Memory model differences between Arm and X86

From below table, at x86, writes from one CPU become visible to all other CPUs in program order. Loads might be reordered with earlier stores to different addresses, with the reordering tightly constrained. This means that most atomic instructions on x86 already imply a full memory fence. Actually, the extra memory-barrier instructions are rarely needed.

Arm has a relaxed memory model so multiple accesses may be observed in any order. Programmers can use acquire/release instructions or explicit barriers to constrain the potential ordering. Using acquire and release instructions like LDAR, you can get a total order of all acquire-loads and release-stores, which is called RCsc (Release Consistency sequential consistency). Barriers do not necessarily create the expected memory ordering.

Aspect	Arm	x86
Default memory order	Weakly ordered without guarantee of implicit ordering	Total Store Order, providing strong ordering
Barriers needed	Basically never needed because of acquire and release semantic	Fewer explicit barriers; most ordering is implicit
Acquire/release	Requires LDAR and STLR	No special instructions needed; plain loads/stores suffice

5.2.2 Atomic instruction sequences

You can translate the following C code into different assembler instructions:

```
atomic_fetch_add_explicit(&val, 1, memory_order_relaxed)
```

x86 asm is `LOCK ADD DWORD [val], 1`, while Arm with FEAT_LSE asm is `LDADD  W1, W0, [X0]`, with `W1` as 1 initially.

For x86, scalar loads behave like load-acquire and scalar stores behave like store-release. All atomic operations, such as `ADD`, `SUB`, `XCHG` are natively supported via the `LOCK` prefix, such as `LOCK ADD`, `LOCK CMPXCHG`, which provide sequential consistency like `AL` versions of Arm atomic instructions. There is no need for load-linked/store-conditional loops, and atomic updates are single-instruction. `__ATOMIC_RELAXED` often compiles to the same code as `__ATOMIC_SEQ_CST` due to x86's strong ordering.

5.2.3 Memory barriers and ordering primitives

When porting assembler code from x86 to Arm, the programmers must consider some differences in the x86 and Arm memory models and instruction semantics.

Memory order	Arm	x86
`__ATOMIC_RELAXED`	No barriers, only atomicity guaranteed	Still uses `LOCK` prefix without barrier
`__ATOMIC_ACQUIRE`	`LDAR` or `DMB ISHLD`	Scalar loads
`__ATOMIC_RELEASE`	`STLR` or `DMB ISH`	Scalar stores

Memory order	Arm	x86
__ATOMIC_SEQ_CST	STLR	XCHG

On x86, every scalar load has acquire semantics and every scalar store has release semantics. Synchronization between threads is default. On Arm, the necessary synchronize-with relations are needed to be identified and use acquire/release semantics where necessary.

5.2.4 Compiler mapping: C11/C++11 atomics

Modern GCC/Clang on both platforms translate standard atomics into the appropriate instruction sequences:

C11/C ++11 atomic operation	AArch64 instructions		x86 instructions
atomic_load_explicit(ptr, memory_order_relaxed)	LDR xN, [ptr]		MOV reg, [ptr]
atomic_store_explicit(ptr, value, memory_order_relaxed)	STR xN, [ptr]		MOV [ptr], reg
atomic_load_explicit(ptr, memory_order_acquire)	LDAPR xN, [ptr]		MOV reg, [ptr]
atomic_store_explicit(ptr, value, memory_order_release)	STLR xN, [ptr]		MOV [ptr], reg
atomic_exchange_explicit(ptr, value, memory_order_seq_cst)	FEAT_LSE	SWPAL W2, W0, [X1]	LOCK XCHG [ptr], re
	Armv8-A	Loop: LDAXR X0, [ptr] STLXR W1, xN, [ptr] CBNZ W1, Loop	
atomic_compare_exchange_strong(ptr, &exp, desired, mo_acq_rel, mo_scquire)	FEAT_LSE	CASAL W0, W2, [X1]	LOCK CMPXCHG [ptr], reg
	Armv8-A	Loop: LDAXR X0, [ptr] CMP X0, x_expected B.NE fail STLXR W1, x_desired, [ptr] CBNZ W1, Loop	

5.3 Usage in Linux or from applications

This section describes some examples about the atomic operation usage in real-world software.

5.3.1 The generic spinlock in latest Linux kernel

The generic spinlock implementation in Linux kernel uses the Queued Spinlocks as the default mechanism. This design significantly improves scalability under contention by eliminating the concurrent contention problem.

The main code is divided into two paths: fast path and slow path. At the fast path, the locked byte is atomically set by `atomic_try_cmpxchg_acquire()`. If the lock is uncontended, the setting succeeds. The slow path has extra pending bit setting if the lock is held. All waiting threads that set the pending bit are in a queue waiting for the “locked” flag to be cleared.

```
static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
    int val = 0;

    if (likely(atomic_try_cmpxchg_acquire(&lock->val, &val, _Q_LOCKED_VAL)))
        return;

    queued_spin_lock_slowpath(lock, val);
}
```

The unlock function performs an atomic store with release semantics, that makes sure all memory operations inside the critical section complete before the lock is released. Then, unlock function atomically clears the “locked” state and marks the lock as available.

```
static __always_inline void queued_spin_unlock(struct qspinlock *lock)
{
    /*
     * unlock() needs release semantics:
     */
    smp_store_release(&lock->locked, 0);
}
```

5.3.2 The lockless ring buffer

A lockless ring buffer is a fixed-size data structure that enables concurrent access between a producer (writer) and a consumer (reader) without locks. It ensures thread safety through atomic operations and memory ordering constraints instead of mutual exclusion like mutexes or spinlocks.

Single-producer/single-consumer: only one thread writes (producer), and one thread reads (consumer).

```
struct ring_buffer {
    uint32_t *buffer;       // Data storage
    size_t capacity;        // Max number of elements
    atomic_size_t head;     // Write position (producer-only)
    atomic_size_t tail;     // Read position (consumer-only)
```

```
};
```

Producer (Writer): Writes data to the tail index and advances the tail. It never interferes with the consumer's head.

```
bool enqueue(struct ring_buffer *rb, uint32_t data) {
    size_t tail = atomic_load_explicit(&rb->tail, memory_order_relaxed);
    size_t head = atomic_load_explicit(&rb->head, memory_order_acquire);

    if (tail - head == rb->capacity) {
        return false;                                  // Buffer full
    }

    rb->buffer[tail % rb->capacity] = data;
    atomic_store_explicit(&rb->tail, tail + 1, memory_order_release);
    return true;
}
```

Consumer (Reader): Reads data from the head index and advances the head. It never interferes with the producer's tail.

```
bool dequeue(struct ring_buffer *rb, uint32_t *data) {
    size_t head = atomic_load_explicit(&rb->head, memory_order_relaxed);
    size_t tail = atomic_load_explicit(&rb->tail, memory_order_acquire);

    if (head == tail) {
        return false;  // Buffer empty
    }

    *data = rb->buffer[head % rb->capacity];
    atomic_store_explicit(&rb->head, head + 1, memory_order_release);
    return true;
}
```