TWLKH

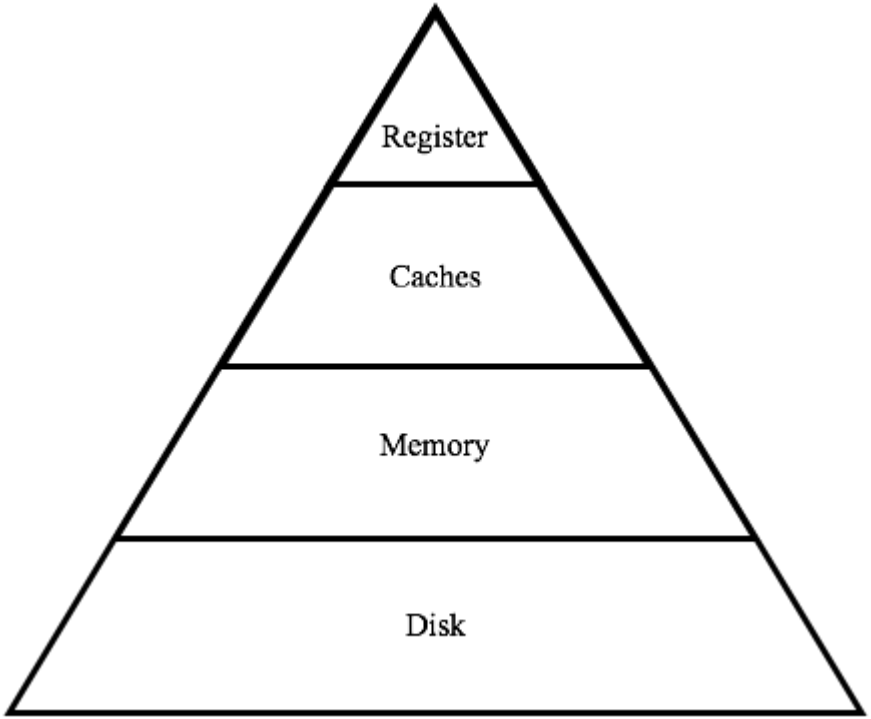
Prerequisite Knowledge for Shared Memory Concurrency

Viller Hsiao

Oct. 24, 2017

From the Perspective of Low Level Implementation and Examples

Memory Hierarchy

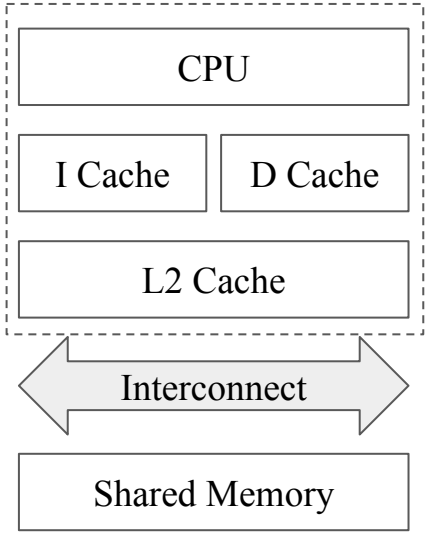


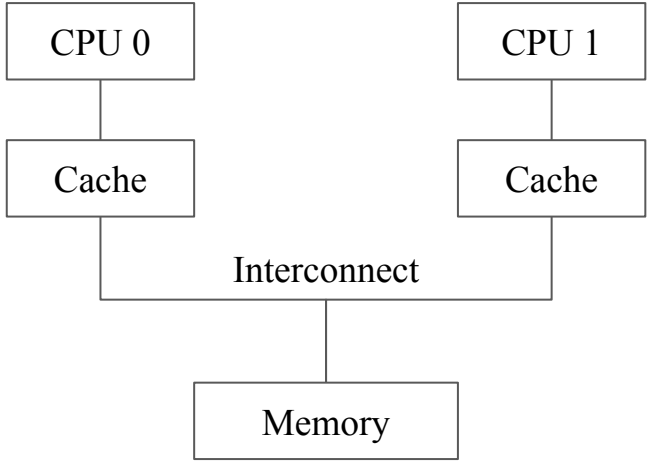
AMD Opteron A1170 from www.7-cpu.com (ARM Cortex-A57)

- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L2 Cache Latency = 18 cycles
- L3 Cache Latency = 60 cycles
- RAM Latency = 60 cycles + 124 ns

Data Consistency among Memory Hierarchy

- Among Local memories
  - I-Cache, D-Cache and TLB walk interface
- Among Caches of each core





Recall the spinlock() Example

AArch64 Assembly for Shared Memory Concurrency

First Trial

- spinlock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available.

```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

Issues 1

- Let’s see the issues in the example by our own eyes
  - Lock status checking will be moved out of loop after optimized
    - https://godbolt.org/g/MFy2Yy
    - Add *volatile* qualifier to avoid it
  - Better optimize *volatile* usage
    - https://godbolt.org/g/W199ef

Issues 1 Recap

- *volatile* qualifier
  - Linux kernel
    - ACCESS_ONCE()
    - READ_ONCE()
    - WRITE_ONCE()

Issues 2

```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

Two threads may come here concurrently

Issues 2

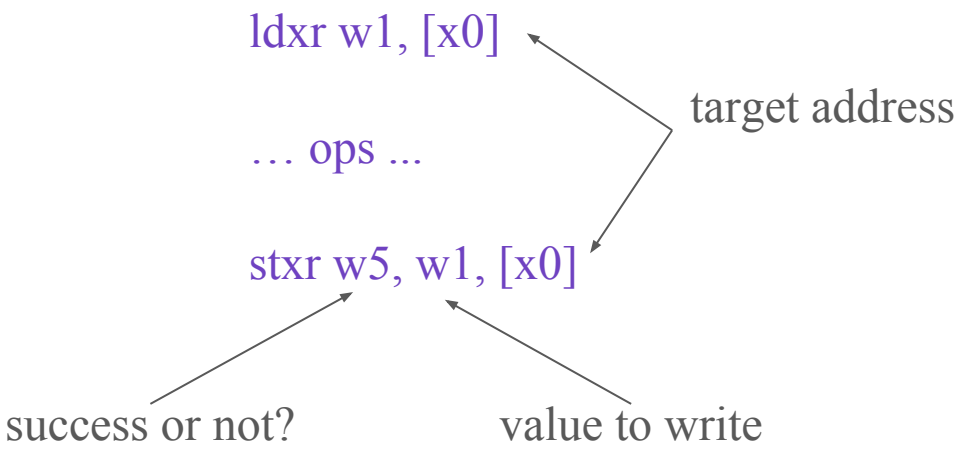
```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

We need to protect the atomicity of the read-modify-write operation.

We can disable interrupt in single core but still fail in multicore.

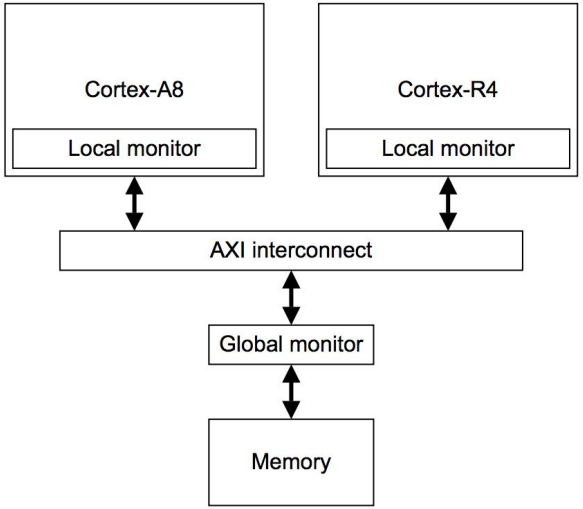
Atomic Instructions Supported by CPU

- compare-and-swap
- test-and-set
- load-exclusive / store-exclusive

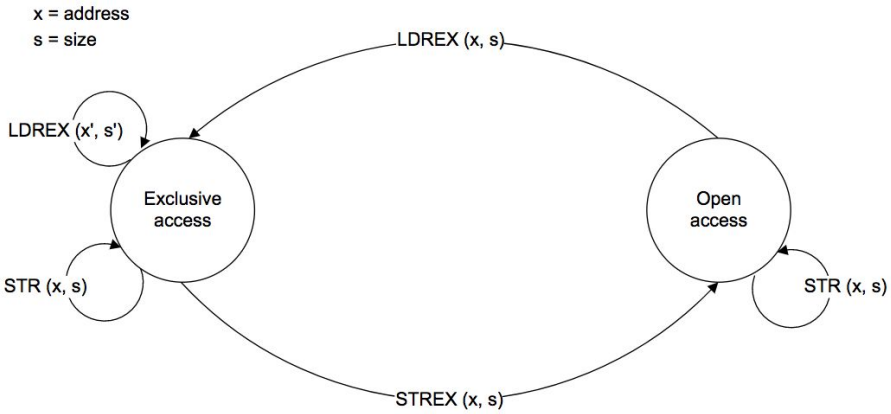


Implementation of ldrex/strex

Local/Global exclusive monitor



arm11 ldrex/strex state machine



STREX value	Result
STREX (Tagged_address, Tagged_size)	OKAY
STREX (Overlap)	XFAIL
STREX (!Overlap)	XFAIL
Cache Line = (Tagged_address)	The cache line is cleaned and invalidated

Example of ldrex/strex Execution

Linux-2.6.19 ARM spin_lock() implementation

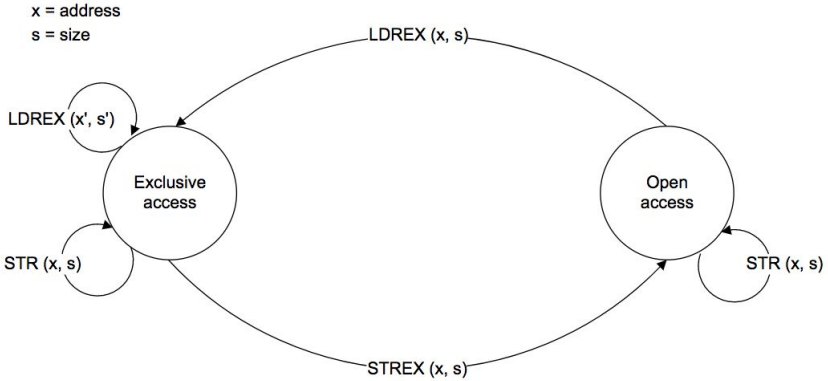
```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    unsigned long tmp;

    __asm__ __volatile__(
"1:  ldrex   %0, [%1]\n"
"    teq     %0, #0\n"
"    strexeq        %0, %2, [%1]\n"
"    teqeq   %0, #0\n"
"    bne     1b"
    : "=&r" (tmp)
    : "r" (&lock->lock), "r" (1)
    : "cc");

    smp_mb();
}
```

Take the executing sequence as example:

ldrex (T1) => ldrex (T2) => strex (T2) => strex (T1)
Result: failed retry



STREX value	Result
STREX (Tagged_address, Tagged_size)	OKAY
STREX (Overlap)	XFAIL
STREX (!Overlap)	XFAIL
Cache Line = (Tagged_address)	The cache line is cleaned and invalidated

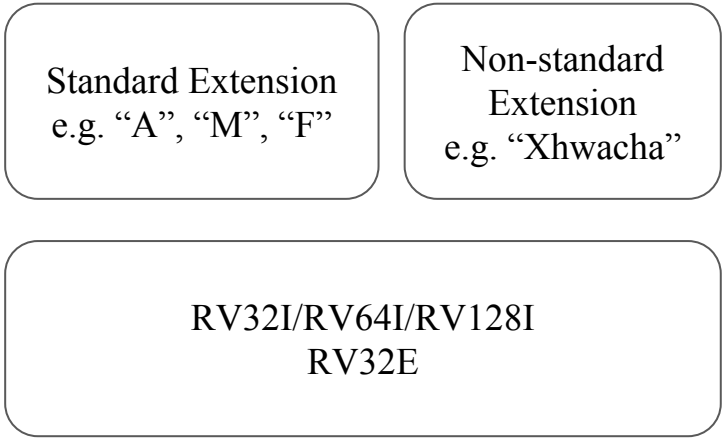
ARMv8.x-A

- ARMv8-A
  - LDXR / STXR
  - LDAXR / STLXR

- ARMv8.1-A
  - Read-Modify-Write instructions
    - CAS, LD{ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, UMIN}, SWP
  - Good discussion in https://www.facebook.com/scottt.tw/posts/1324089220996467

RISC-V: ISA

- “A” extension: Atomic Instruction v2.0
- “T” extension: Transactional Memory, v0.0



RISC-V: Atomic Extension [1.1]

- Load-Reserved/Store-Conditional Instructions

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR	ordering		0	addr	width	dest	AMO
SC	ordering		src	addr	width	dest	AMO

```
cas:
    lr.w t0, (a0)          # Load original value.
    bne t0, a1, fail       # Doesn't match, so fail.
    sc.w a0, a2, (a0)      # Try to update.
    jr ra                  # Return.
fail:
    li a0, 1               # Set return to failure.
    jr ra                  # Return.
```

compare-and-swap example

RISC-V: Atomic Extension

- Atomic Memory Operation(AMO)

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO
AMOADD.W/D	ordering		src	addr	width	dest	AMO
AMOAND.W/D	ordering		src	addr	width	dest	AMO
AMOOR.W/D	ordering		src	addr	width	dest	AMO
AMOXOR.W/D	ordering		src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO

```
    li            t0, 1        # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez          t0, again    # Retry if held.
```

spinlock example

spinlock(): 2nd Trial with ARMv8-A ldxr/stxr

```
1          .section .text
2          .global spin_lock
3  spin_lock:
4          ldxr w5, [x0]       /* read lock */
5          mov w1, #1
6          cbnz w5, spin_lock /* check if 0 */
7          stxr w5, w1, [x0]  /* attempt to store new value */
8          cbnz w5, spin_lock /* test if store suceeded
9                                retry if not */
10         dmb ish             /* ensures that all subsequent accesses are observed after the
11
12
13         ret
14
15         .global spin_unlock
16 spin_unlock:
17
18
19         str wzr, [x0]       /* clear the lock */
20         ret
```

Note: Courtesy of Scott Tsai. Some lines about memory barrier are masked in the [example](#) for demonstration.

Example 2

- x = x + 1;

	; x = x + 1
ldr r3, .L2	; load x to r3
ldr r3, [r3, #0]	
add r2, r3, #1	; r2 = r3 +1
ldr r3, .L2	
str r2, [r3, #0]	; store r2 to x

We need to protect the atomicity of the read-modify-write operation

Quiz: How to implement atomic_add() by ldxr/stxr?

Dekker's Algorithm

- First known correct solution to the mutual exclusion problem in concurrent programming.

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Memory Ordering
and
Memory Barrier

Compiler Reordering

- Instructions order might be re-ordered for efficiency
- https://godbolt.org/g/C45pBr

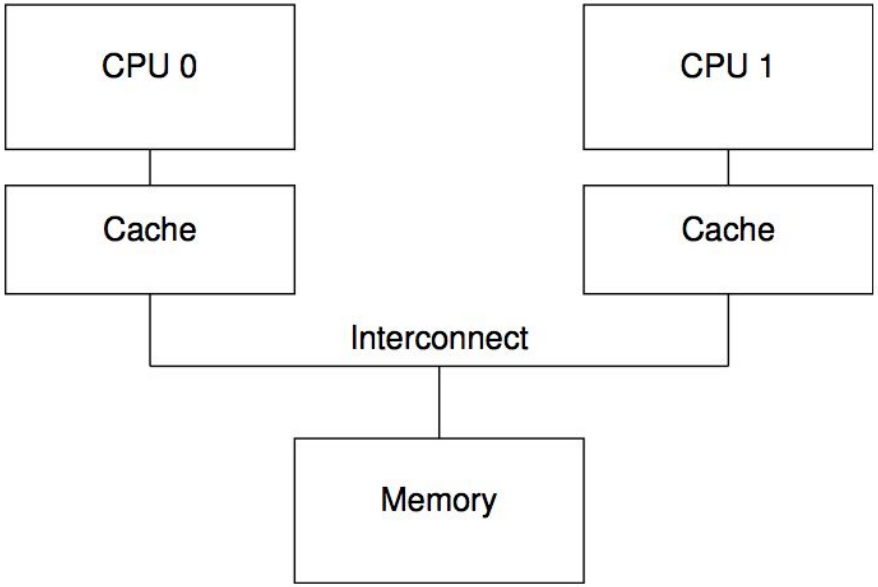
```
int A, B;
void foo (void)
{
    A = B + 1;
    B = 5;
}
```

Courtesy of Jeff Preshing

Compiler Reordering

- Compiler barrier
  - gcc
    - `asm volatile ("" ::: "memory");`
    - barrier() macro in Linux kernel

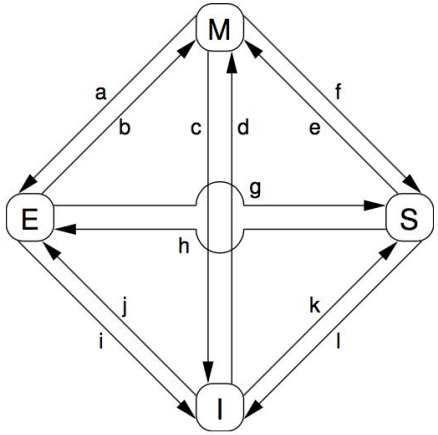
Modem Coumpter System Cache Structure

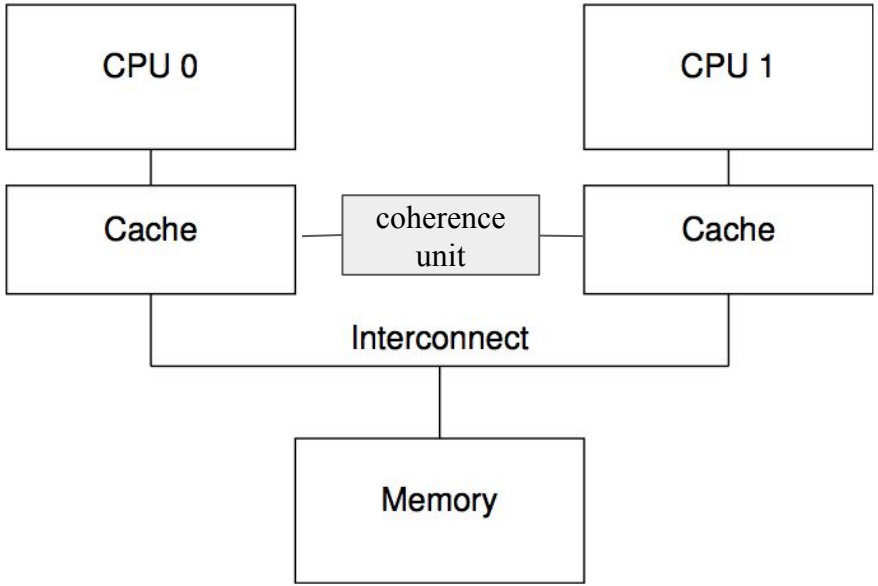


ref: [3.1] Appendix C

Cache Coherence [3.1]

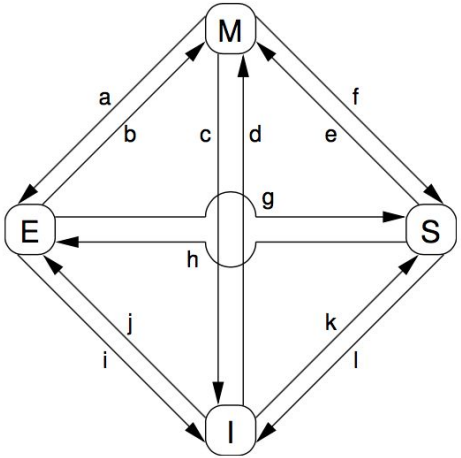
- MESI
  - Modified
  - Exclusive
  - Shared
  - Invalid





MESI: Example Transition Sequence [3.1]

- MESI
  - Modified
  - Exclusive
  - Shared
  - Invalid

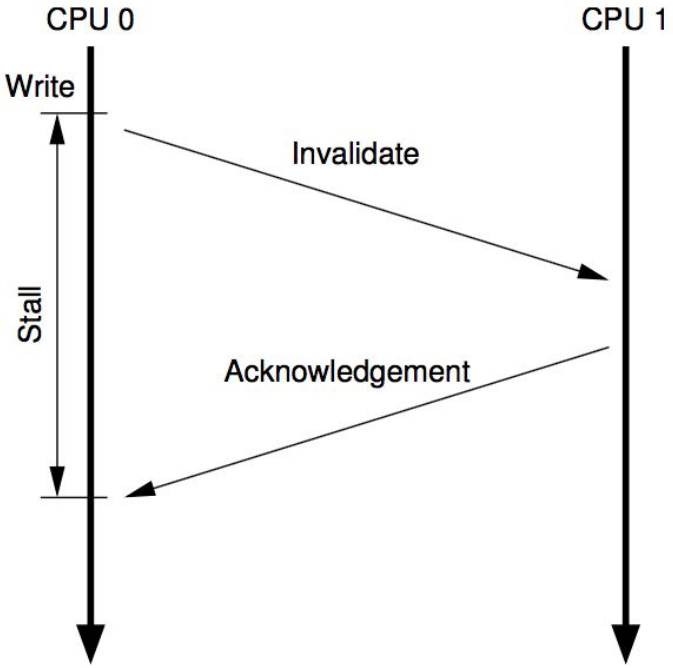


Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Invalidation	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

Quiz: How about the MOESI protocol?

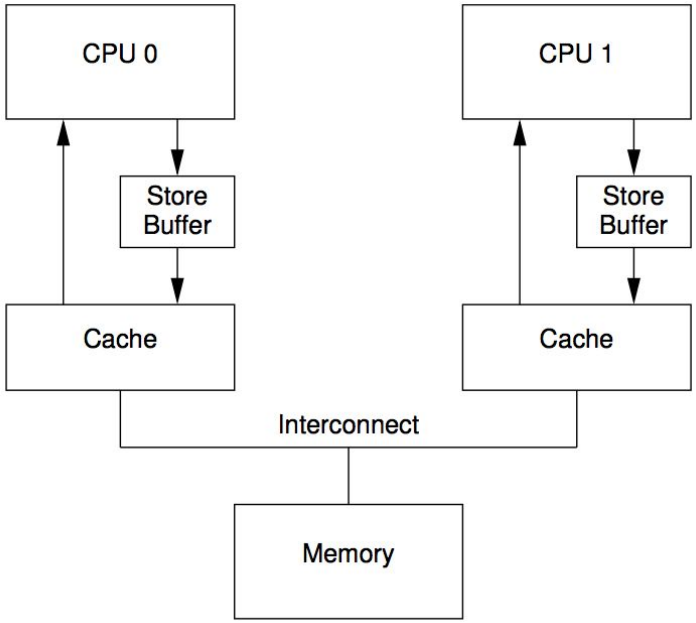
Hint: See chapter 14.3 "Multi-core cache coherency within a cluster" in "ARM Cortex-A Series Programmer's Guide for ARMv8-A"

Stall Cycles in Cache Coherence Protocol [3.1]



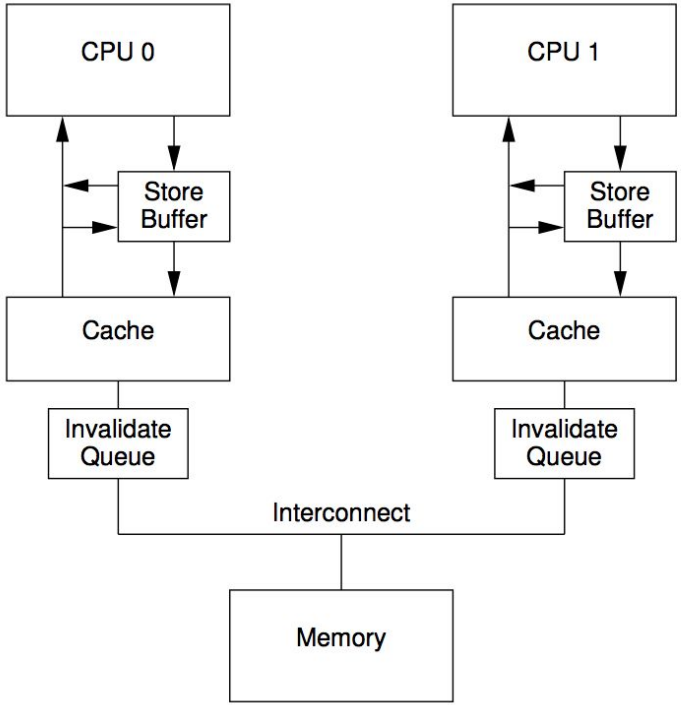
Write Buffer [3.1]

- Add a store buffer to hide the write stall cycles



Invalidate Queue [3.1]

- Write buffer is small
  - When Invalidated cache is quite busy, the time to reply acknowledgement might be delayed
- Add invalidate queue to reply acknowledgement before it's really done



Optimizations in CPU Design [4.1]

- Bypass write buffer
- Overlapped writes
- Non−blocking reads

Quiz: Any other reason that memory access may be re-ordered?

Hint: Chapter 13 “Memory Ordering” in “ARM Cortex-A Series Programmer’s Guide for ARMv8-A”

Memory Access Reordering

- Compile time
- Run time

Abstract Memory Access Reordering

- Example
  - X: LOAD A; STORE B
  - Y: LOAD C; LOAD D
  - Possible results
    - LOAD A, STORE B, LOAD C, LOAD D
    - STORE B, LOAD A, LOAD C, LOAD D
    - …. 16 possible results

Memory Reordering

- Operations

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

- Reorder scope
  - same/different variables
  - same/different threads
- Dependency Load Re-ordering
  - Alpha
  - Linux kernel barrier [smp_read_barrier_depends()]

```
g = Guard.load(memory_order_consume);
if (g != nullptr)
    p = *g;
```

```
    movw    r3, :lower16:(_Guard-(L0+4))
    movt    r3, :upper16:(_Guard-(L0+4))
L0:
    add     r3, pc
    ldr     r4, [r3]        load from Guard
    cmp     r4, #0
    it      ne
    ldrne   r2, [r4]        load from *g
```

Courtesy of [Jeff Preshing]

Architectures Memory Model [3.1]

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
ARMv7-A/R	Y	Y	Y	Y	Y	Y		Y
IA64	Y	Y	Y	Y	Y	Y		Y
MIPS	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

Memory Access re-ordering

- Consider a simple message passing example

Initial state: data=0 ∧ flag=0	
Thread 0	Thread 1
data = 42	while (flag == 0)
flag = 1	r2 = data
Forbidden: Thread 1 register r2 = 0	

- Memory consistency model
  - We need some consesus to make sure correctness under concurrency (multi-core environment)

Memory Barrier

- A type of barrier instruction that causes CPU to enforce an ordering constraint on memory operations issued before and after the barrier instruction. ~ wikipedia
  - ARMv7-A
    - dmb / dsb / isb
  - ARMv8-A
    - dmb / dsb / isb
    - ldar / stlr
    - ldaxr / stlxr
  - RISC-V
    - fence / fence.i
    - aq / rl bits in AMO, LR/SC instruction

Litmus Tests [2.4]

- Recall the MP example
  - Memory consistency model
    - We need some consesus to make sure correctness under concurrency (multi-core environment)
    - Only few words in architecture programming guide

Initial state: data=0 ∧ flag=0	
Thread 0	Thread 1
data = 42	while (flag == 0)
flag = 1	r2 = data
Forbidden: Thread 1 register r2 = 0	

MP Pseudocode

Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: x=0 ∧ y=0	
Allowed: 1:r1=1 ∧ 1:r2=0	

MP+dmb/syncs Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y dmb/sync r2=x
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=1 ∧ 1:r2=0	

spinlock(): 3nd Trial with ARMv8-A ldxr/stxr/dmb

```
        .section .text
        .global spin_lock
spin_lock:
        ldxr w5, [x0]      /* read lock */
        mov w1, #1
        cbnz w5, spin_lock /* check if 0 */
        stxr w5, w1, [x0]  /* attempt to store new value */
        cbnz w5, spin_lock /* test if store suceeded
                              retry if not */
        dmb ish            /* ensures that all susequent accesses are observed after the
                             gaining of the lock is observed */
        /* loads and stores in the critical region can now be performed */
        ret

        .global spin_unlock
spin_unlock:
        dmb ish            /* ensure all previous accesses are observed before lock is
                              cleard */
        str wzr, [x0]      /* clear the lock */
        ret
```

Note: The example is courtesy of Scott Tsai.

Some Common Rules [3.1]

- All accesses by a given CPU will appear to that CPU to have occurred in program order.
- All CPUs' accesses to a single variable will be consistent with some global ordering of stores to that variable.
- Memory barriers will operate in a pair-wise fashion.
- Operations will be provided from which exclusive locking primitives may be constructed.

Memory Consistency Model

Sequencial Consistency

Defined by Lamport as follows,

"A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Relaxed Model [4.1]

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Processor Consistency [4.1]

- Relaxed operation

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

- The order in which other processors see the writes from any individual processor is the same as the order they were issued.
  - Does not require writes from all processors to be seen in the same order

Total Store Order

- SPARCv8 TSO
- x86-TSO
  - Reads or writes cannot pass (be carried out ahead of) I/O instructions, locked instructions, or serializing instructions.
- Relaxed operation

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

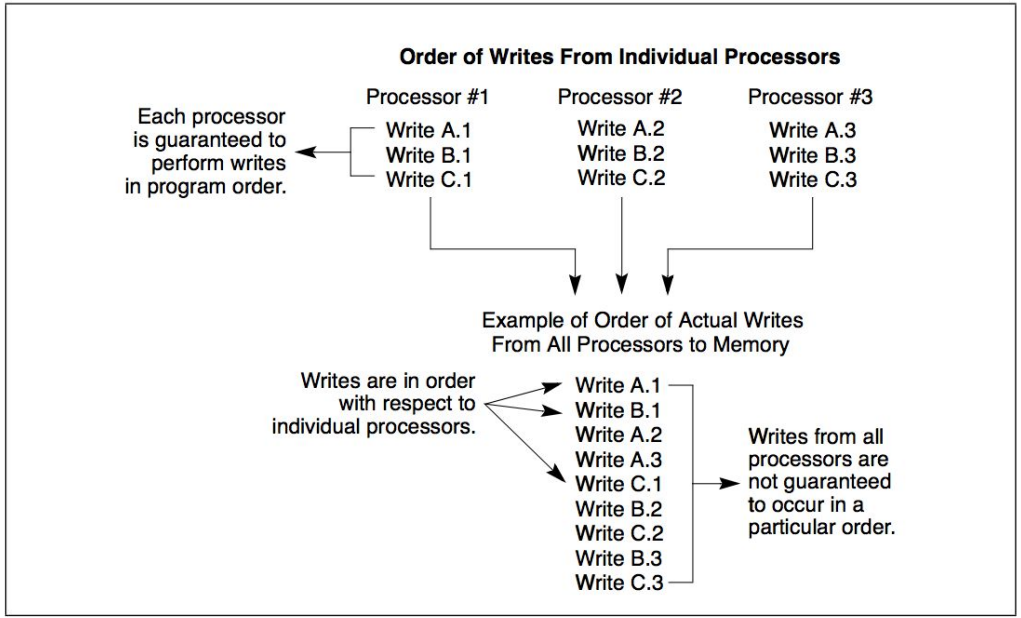
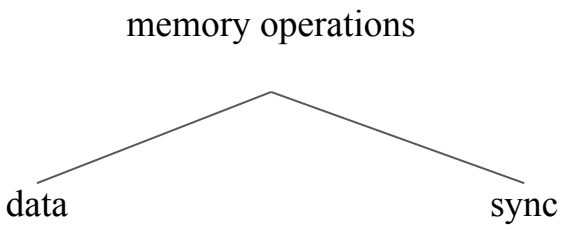


Figure 7-1. Example of Write Ordering in Multiple-Processor Systems

Weak Ordering [4.1]

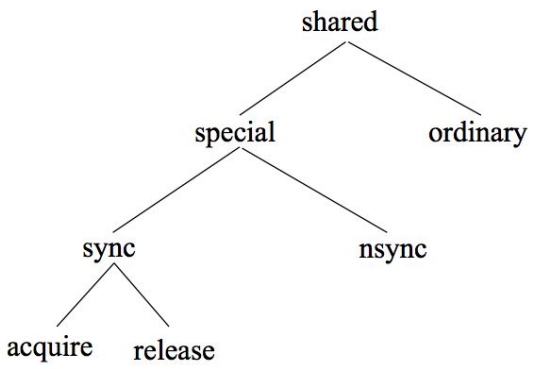
- Synchronization operations provide a safety net for enforcing program order
- Each processor must ensure that a synchronization operation is not issued until all previous operations are complete
- No operations are issued until the previous synchronization operation completes
- The weak ordering model ensures that writes always appear atomic to the programmer; therefore, no safety net is required for write atomicity



#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

Release Consistency [4.1]

- RCsc (Relase consistency sequencial consistency)
  - acquire → all
  - all → release
  - special → special.



- RCpc (Release consistency processor consistency)
  - acquire → all
  - all → release
  - special → special
    - except for a special write followed by a special read.

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

Memory Consistency Model of ARMv8-A and RISC-V

ARM: Memory Region and Memory Type

- Memory type in TLB setting
  - Normal memory
  - Device memory

ARM: Memory Sharability [2.1]

•

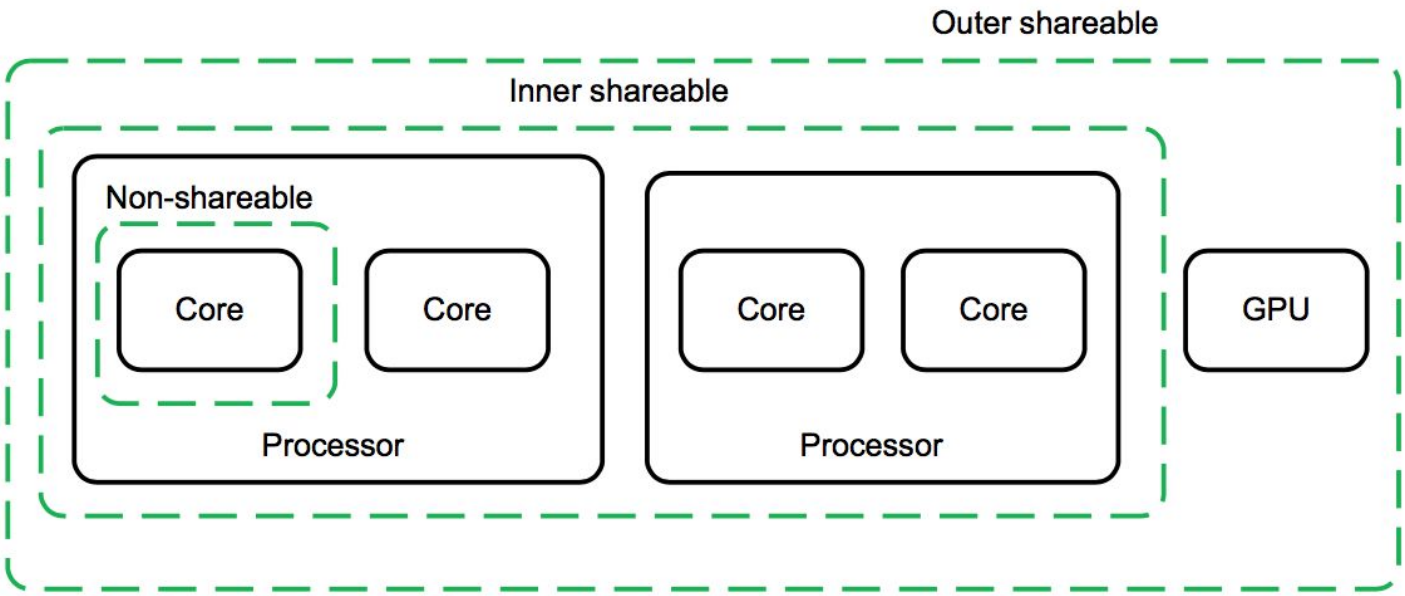


Figure 13-4 Inner and outer shareable domains

ARMv7-A Memory Barrier Instructions [2.1]

- dmb

SY	This is the default and means that the barrier applies to the full system, including all cores and peripherals.
ST	A barrier that waits only for stores to complete.
ISH	A barrier that applies only to the Inner Shareable domain.
ISHST	A barrier that combines ST and ISH. That is, it only stores to the Inner Shareable.
NSH	A barrier only to the Point of Unification (PoU). (See *Point of coherency and unification* on page 8-19).
NSHST	A barrier that waits only for stores to complete and only out to the point of unification.
OSH	Barrier operation only to the Outer Shareable domain.
OSHST	Barrier operation that waits only for stores to complete, and only to the Outer Shareable domain.

ARMv8-A Memory Barrier Instructions [2.1]

- dmb
- ldar / stlr
  - implicit sharability attribute from address
  - ldaxr / stlxr
- dsb
- isb

acquire-release

Table 13-1 Barrier parameters

<option>	Ordered Accesses (before – after)	Shareability Domain
OSHLD	Load – Load, Load – Store	Outer shareable
OSHST	Store – Store	
OSH	Any – Any	
NSHLD	Load – Load, Load – Store	Non-shareable
NSHST	Store – Store	
NSH	Any – Any	
ISHLD	Load –Load, Load – Store	Inner shareable
ISHST	Store – Store	
ISH	Any – Any	
LD	Load –Load, Load – Store	Full system
ST	Store – Store	
SY	Any – Any	

dmb

ARMv8.3-A

- Load-Acquire RCpc Register
  - LDAPR Wt|Xt, [Xn|SP {,#0}]

"The instruction has memory ordering semantics as described in Load-Acquire, Store-Release in the ARMv8-A Architecture Reference Manual, except that: There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed."
~ ARM Infocenter

RISC-V

- RV32I v2.0
  - Release consistency model
  - FENCE
  - FENCE.I
- “A” extension
  - aq / rl bit in atomic instructions

This section is somewhat out of date as the RISC-V memory model is currently under revision to ensure it can efficiently support current programming language memory models.

RISC-V: FENCE [1.1]

- Order access for
  - IO access
  - Memory access

31–28	27	26	25	24	23	22	21	20	19–15	14–12	11–7	6–0
0	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
0	predecessor				successor				0	FENCE	0	MISC-MEM

Example:

```
#define mmiowb()        __asm__ __volatile__ ("fence io,io" : : : "memory");
```

RISC-V: FENCE.I [1.1]

- FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart.

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	FENCE.I	0	MISC-MEM

RISC-V: aq/rl bit in Atomic Extension [1.1]

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO
AMOADD.W/D	ordering		src	addr	width	dest	AMO
AMOAND.W/D	ordering		src	addr	width	dest	AMO
AMOOR.W/D	ordering		src	addr	width	dest	AMO
AMOXOR.W/D	ordering		src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO

```
    li          t0, 1          # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez        t0, again      # Retry if held.
```

C11/C++11 Memory Model (TBD)

C11/C++11

- Add support to multithread model
  - multithreading support
  - Atomic support

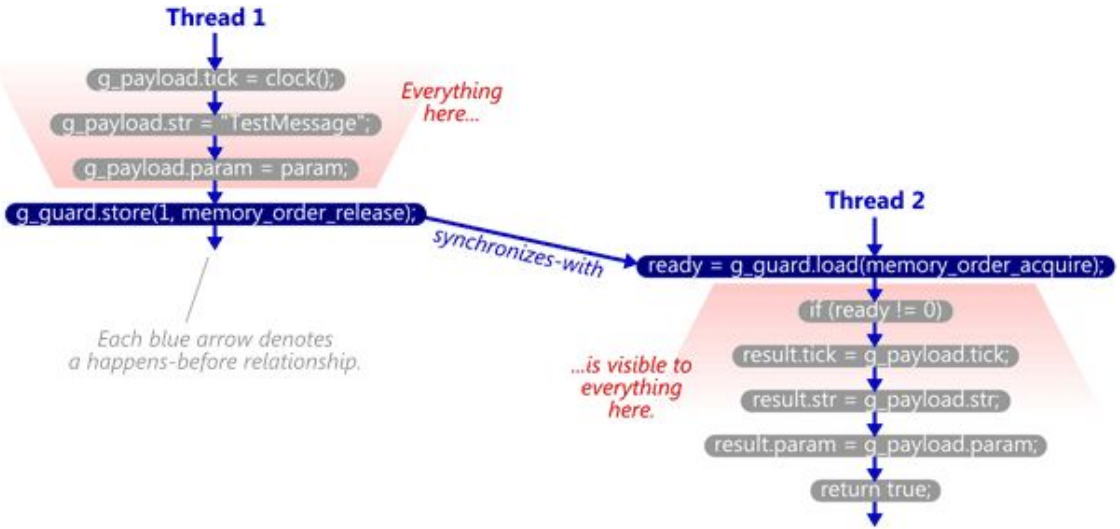
C99

- Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called **sequence points**, **all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.**
  - Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment.

Quiz: Where is the sequence point in a C program?
Hint: Annex C in C99 ISO/IEC 9899:201x

C11

- Evaluation
- Sequenced-before
- Happens-before
  - A is sequenced-before B
  - A inter-thread happens before B
- Synchronizes-with



http://preshing.com/20130823/the-synchronizes-with-relation/

Memory Order

- memory_order_relaxed

Memory Order

- memory_order_acquire
- memory_order_release

Memory Order

- memory_order_acq_rel

Memory Order

- memory_order_seq_cst
  - Different from memory_order_acq_rel
    - Essentially memory_order_acq_rel provides read and write orderings relative to t variable,
    - while memory_order_seq_cst provides read and write ordering globally.

- http://en.cppreference.com/w/cpp/atomic/memory_order

Memory Order

- memory_order_consume
  - Memory ordered by data dependency
  - RCU served as motivation for adding consume semantics to C++11 in the first place.

```
g = Guard.load(memory_order_consume);
if (g != nullptr)
    p = *g;
```

```
    movw    r3, :lower16:(_Guard-(L0+4))
    movt    r3, :upper16:(_Guard-(L0+4))
L0:
    add     r3, pc
    ldr     r4, [r3]        load from Guard
    cmp     r4, #0
    it      ne
    ldrne   r2, [r4]        load from *g
```

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

Linux Kernel Memory Model

(TBD)

Variable Access

- ACCESS_ONCE
  - volatile memory_order_relaxed
- smp_load_acquire
  - volatile memory_order_acquire
- smp_store_release
  - volatile memory_order_release

Memory Barriers

- barrier
- smp_mb / smp_rmb / smp_wmb
- smp_read_barrier_depends
  - Alpha only
- smp_mb__after_unlock_lock
  - RCpc

Locking Operation

- test_and_set_bit
- test_and_set_bit_lock (Lock-Barrier Operations)
  - memory_order_acquire semantics

Atomic Operations

Control Operations

- Avoid control-dependency-destroying compiler optimizations

RCU Grace-Period Relationships

Q & A

Appendix

References

1. RISC-V
   1.1. The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2
2. ARM Infocenter
   2.1. ARM Cortex-A Series Programmer's Guide for ARMv8-A
   2.2. ARM Cortex-A Series Programmer's Guide
   2.3. ARM® Synchronization Primitives
   2.4. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models
3. Linux kernel
   3.1. Is Parallel Programming Hard, And, If So, What Can You Do About It?
   3.2. Linux-Kernel Memory Model

References (Cont.)

4. Memory model concepts
   4.1. Shared Memory Consistency Models: A Tutorial
   4.2. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors
5. C11/C++11
   5.1. cppreference
6. x86
   6.1. Intel Architecture Software Developer's Manual Volume 3: System Programming
7. Articles from Preshing on Programming

Rights to Copy

- Taiwan Linux Kernel Hackers (TWLKH) is a group that focuses on Linux kernel development
  - Find us on Facebook and YouTube
- ARM is an registered trademark of ARM Holdings.
- RISC-V is a standard open architecture for industry implementations under the governance of the RISC-V Foundation.
- SPARC is a registered trademark of SPARC International, Inc
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
- The license of each graph belongs to each website listed individually.
- The others of my work in the slide is licensed under a CC-BY-SA 4.0 License.
  - License text: http://creativecommons.org/licenses/by-sa/4.0/legalcode