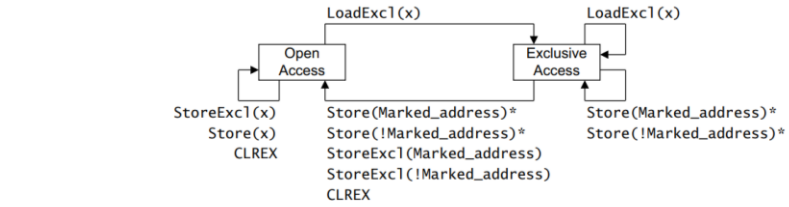
Linux 内核的同步机制

--ARM 原子操作 atomic

一、内存独占监视器(local/global exclusive monitor)和内存独占指令(load-exclusive/store-exclusive)

对于单核(UP)系统，那么要想保证读-改-写的操作不被打断，只需要在读之前关闭中断就可以了。随着多核(SMP)系统的普及，大幅的提升了系统的性能，但是随之而来的并发导致的数据同步的问题就变得极为重要。

针对这一情况，各 CPU 架构提供了基于各自架构的解决方案。对于 ARM 来说，通过内存独占监视器(local/global exclusive monitor)配合内存独占操作指令(ldxr/stxr)来保证读-改-写操作时的同步问题。



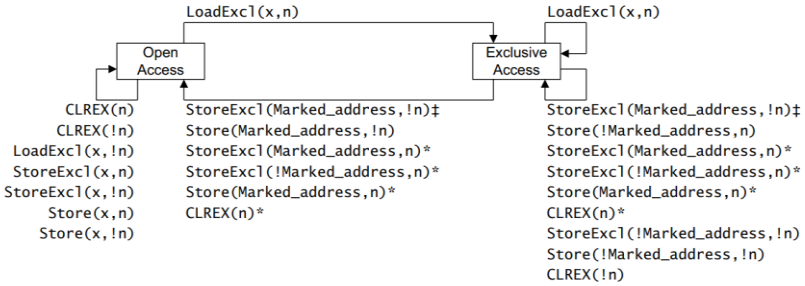
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-4 Local monitor state machine diagram

local exclusive monitor 状态机迁移

Figure B2-5 shows the state machine for PE(n) in a global monitor.



‡StoreExcl(Marked_address,!n) clears the monitor only if the StoreExcl updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

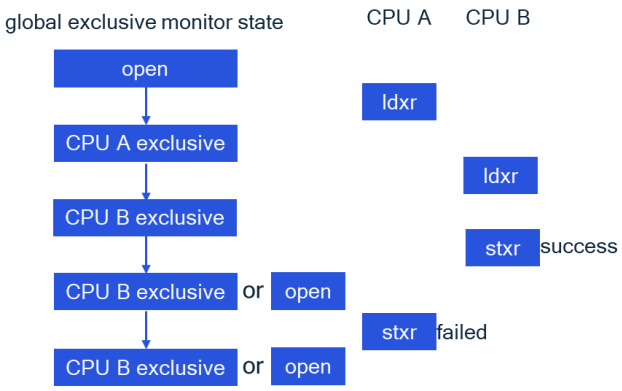
Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

global exclusive monitor 状态机迁移

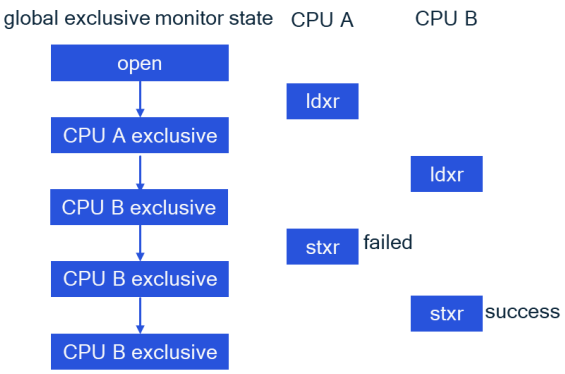
ldxr 指令执行的过程中会将所操作的地址 x（物理地址）的本地和全局内存独占监视

器为独占状态。全局监视器此时还会记录下标记内存独占状态的 CPU 号。stxr 指令执行的过程中，只有监视器为独占状态且全局监视器记录的 CPU 与读数据的时候一致才会写成功。下面对照上面两幅状态机的迁移图看具体的执行过程。

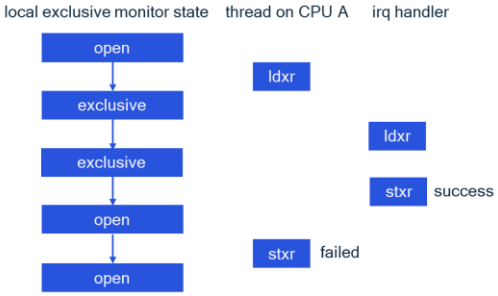
CPU A 调用 ldxr 指令执行的过程中会将所操作的地址 x（物理地址）的本地和全局内存独占监视器为独占状态。全局监视器此时还会记录下标记内存独占状态的 CPU 为 CPU A。考虑情形一，在 CPU A 写数据之前发生了线程调度且在 CPU B 上运行，CPU B 也调用 ldxr 指令读取地址 x，此时地址 x 的本地和全局内存独占监视器依然为独占状态，但是全局监视器此时记录的标记内存独占状态的 CPU 为 CPU B。然后 CPU B 调用 stxr 指令成功的更新地址 x 的值，此时全局监视器的状态可以根据架构设计为独占状态或者开放状态。线程再次切换到 CPU A 上，这时 CPUA 通过 stxr 指令写地址 x。如果全局监视器是开放状态，不会写入数据，同时返回 1，表明指令写失败；如果全局监视器是独占状态，但是记录的标记独占状态的 cpu 是 CPU B，同样不会写入数据，同时返回 1，表明指令写失败。考虑情形二，在 CPU A 写数据之前发生了线程调度且在 CPU B 上运行，CPU B 也调用 ldxr 指令读取地址 x，此时地址 x 的本地和全局内存独占监视器依然为独占状态，但是全局监视器此时记录的标记内存独占状态的 CPU 为 CPU B。然后又切换到 CPU A 调用 stxr 指令更新地址 x 上的值，由于监视器记录的标记独占状态的 cpu 是 CPU B，不会写入数据，同时返回 1，表明指令写失败。线程再次切换到 CPU B 上，这时 CPUB 通过 stxr 指令可以成功的更新地址 x 上的值。考虑情形三，在 CPU A 写数据之前 C P U A 收到了中断，中断处理函数也调用 ldxr 指令读取地址 x，此时地址 x 的本地和全局内存独占监视器依然为独占状态，然后中断处理函数调用 stxr 指令更新地址 x 上的值，此时本地监视器的状态会变为开放状态。中断结束之后 CPUA 的线程继续执行，这时 CPUA 调用 stxr 指令会发现本地监视器不是独占状态，不会写入数据，同时返回 1，表明指令写失败。



情形一



情形二



情形三

内存独占指令除了上文中的 ldxr/stxr 之外还有针对其他不同字节长度的变量指令。

Table B2-3 Synchronization primitives and associated instruction, A64 instruction set

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	LDXRB	STXRB	-
	Load-Acquire/Store-Release	LDAXRB	STLXRB	-
Halfword	-	LDXRH	STXRH	-
	Load-Acquire/Store-Release	LDAXRH	STLXRH	-
Register[b]	-	LDXR	STXR	-
	Load-Acquire/Store-Release	LDAXR	STLXR	-
Pair[b]	-	LDXP	STXP	-
	Load-Acquire/Store-Release	LDAXP	STLXP	-
None	Clear-Exclusive	-	-	CLREX

二、ARM64 架构的原子操作在 linux 中的代码实现

在 ARM64 的 arch 层代码中(kernel/arch/arm64/include/asm/atomic_ll_sc.h)通过宏 ATOMIC_OP 来实现原子操作函数。宏展开之后核心就是 ldxr 和 stxr 指令，在两条指令之间加了对变量的 add/sub 指令。

```
113 ATOMIC_OPS(add, add)
102 #define ATOMIC_OPS(...)                                         \
103     ATOMIC_OP(__VA_ARGS__)
```

```
    /* ATOMIC_OPS(add, add)展开为:
    * static inline void atomic_add(int i, atomic_t *v)
    * {
    *       xxx
    * }
    *  此处 op = add, asm_op = add
    */
40  #define ATOMIC_OP(op, asm_op)                                                \
41  __LL_SC_INLINE void                                                          \
42  __LL_SC_PREFIX(atomic_##op(int i, atomic_t *v))                              \
43  {                                                                            \
44      unsigned long tmp;                                                       \
45      int result;                                                              \
46                                                                               \
    /* asm 关键字提示 gcc 此处为内联汇编代码
     * volatile 关键字用来防止 gcc 对代码做优化。如果使用优化选项（-O）进行编译,
     * 对于那些没有声明__volatile__的代码，编译器有可能会对代码进行优化，编译的结果可能不
     * 是原来你撰写的汇编代码，但是如果使用 volatile/__volatile__的关键字，也就是告诉编译器，
     * 不要随便动我的嵌入汇编代码哦。
     */
47      asm volatile("// atomic_" #op "\n"                                       \
    /* 加载 v->vounter 到 L1 cache */
48  "   prfm    pstl1strm, %2\n"                                                 \
    /* 1:   result = v-> counter;   ldxr 指令同时还会设置独占 monitor */
49  "1: ldxr    %w0, %2\n"                                                       \
    /* result = result + i; */
50  "   " #asm_op "     %w0, %w0, %w3\n"                                         \
    /* v->counter = result; tmp 保存 stxr 指令的返回值，执行失败 tmp==1，成功 tmp==0 */
51  "   stxr    %w1, %w0, %2\n"                                                  \
    /* if(tmp != 0)
    *    goto label 1;   如果 stxr 执行失败，表明变量的值已经被别人修改，跳转到标号 1 重新读值
    */
52  "   cbnz    %w1, 1b"                                                         \
    /* 修饰符含义:
    * =：只写，+：可读写，&用来修饰输出操作数，表示不能和输入操作数寄存器相同
    * r：使用通用寄存器(x0~x30)保存变量，I：整数，Q：内存地址
    */
53      : "=&r" (result), "=&r" (tmp), "+Q" (v->counter)                         \
54      : "Ir" (i));                                                             \
55  }
```