Synchronization in Linux

Aakarsh Nair

April 12, 2015

Contents

1 Introduction

Synchronization becomes necessary when the outcome of a computation depends on how two or more interleaved kernel control paths are nested. Critical regions are parts of code that must be executed by at most one kernel control path to completion before another kernel control path is allowed to execute it.

2 Kernel Preemption

Process running in Kernel Mode can be replaced by another process. The main reason for making a kernel preemptive is to reduce the dispatch latency of user mode processes (the time between when they become runnable and begin running). Process switches happens via the`switch_to_macro`.

Kernel premption is enabled and disabled via the `prempt_count` in the `thread_info`. A task is premptible if `thread_info()->preempt_count` is zero.

The `prempt_count` is greater than zero if

- The kernel is executing in an interrupt service routine.
- kernel is executing a tasklet of a softirq
- kernel preemption has been explicitly disabled setting it to a positive value

The `prempt_count` is an amalgamation of three separate counters meant to keep track number of times kernel preemption count as well as softirq and hardirq counts.

Bits	Description
0-7	Preemption counter (max=255)
8-15	Softirq counter (max=255)
16-27	Hardirq counter (max=4096)
28	`PREEMPT_ACTIVE` flag

Thus kernel is preempted only when its executing an exception handler and kernel preemption has not been explicitly disabled and the local CPU has local interrupts enabled.

Key Macors Dealing with preemption counter are given as

Bits	Description
`preempt_count()`	Select the `preempt count` from `thread_info`
`preempt_disable()`	Increase the value of the preemption counter.
`preempt_enable_no_resched()`	Decrease by one the value of the preemption counter
`preempt_enable()`	Decrease by one the value of preemption counter and call `preempt_schedule()` if `TIF_NEED_RESCHED` on `thread_info` is set
`get_cpu()`	Similar to `preempt_disable()` but also returns the number of local CPU
`put_cpu()`	Same as `preempt_enable()`
`put_cpu_no_resched()`	Same as `preempt_enable_no_resched()`

3 Kernel Synchronization Primitives

The linux kernel provides several synchronization primitives these include

- **Per-CPU variables** Use duplicate data structures for each cpu
- **Atomic Operations** Atomically read-modify-write instruction to a counter
- **Memory Barrier** Avoid instruction reordering
- **Spinlock** Lock with a busy wait
- **Semaphores** Lock with blocking wait/sleep.
- **Seqlocks** Lock based on access counter
- **Local interrupt disabling** Forbid interrupt handling on a single CPU
- **Local softirq disabling** Forbid deferrable function handling on a single CPU
- **Read Copy Update** Lock free access to shared data structures using pointers

Many of these synchronization constructs depend on implementation of atomic operations implemented at the chip level on CPUs. Thus are specific to the architecture they are executing on.

3.1 Processor guarantees

- On a particular CPU dependent memory accesses happen in order of issuance.
- Overlapping loads and stores within a particular CPU will appear to be ordered within a CPU

Things which cannot be assumed

- it **must not** be assumed that compilers will not reorder memory references not protected with `ACCESS_ONCE()`. Without `ACCESS_ONCE()` the compiler can perform transformations see memory barriers.
- it **must not** be assumed that independent loads and stored will be issued in any given order.
- it **must not** be assumed that overlapping memory accesses may be merged or discarded.

Things that we anti-guarantees (bad stuff will happen):

- Compilers will often generate code to modify bit fields using non atomic read-modify write sequences
- All fields in a given bit-field must be must be protected by one lock. Update of one field can be corrupted by another by compiler.
- Guarantees only apply to properly aligned and sized scalar variables. Same size as "char","short","int" and "long".

3.2 Memory Barrier

Memory barriers impose perceived partial ordering over memory operations on either side of the barrier.

Memory barriers provide a way to instruct the compiler and CPU to restrict the order in which instructions are executed. Performance optimizations of compilers and CPUs play havoc with synchronization primitives. These include compiler reordering instructions to optimize register usage, CPUs executing instructions in parallel, reordering memory accesses.

Types of performance tricks memory barriers may protect against are

- reordering and deferral of combination of memory operations
- speculative loads
- speculative branch prediction and caching

Memory barrier types include

- Write (or store) memory barrier

  A write memory barrier gives guarantee that all STORE operations specified before the barrier will happen before all STORE operations STORE operations specified after the barrier. wrt. other components in the system.

  A partial order on STORES only. Does not affect LOADS.

  CPU can be viewed as performing a commit. All stores before the write barrier will occur before all stores after the write barrier.

  Should not be paired with read or data dependency barrier.

- Data dependency barrier

  Weaker form of read barrier. Two loads such that the second depends on result of first (first load retrieves the address which to which second load is directed). Ensure the target of second load is updated before first load accessed (??)

  Partial ordering on inter-dependent loads only. No effect on independent loads or overlapping loads. Has no affect on stores.

  A data dependency barrier issued by the CPU is thus a line in the sand such that for any loads preceding it, if that load touchs one of a sequence of stores from another CPU then by the time the barrier completes(the line is crossed) the effect of all stores prior to that load will be perceptible by any loads issued after the data dependency barrier.

  Thus a dependency barrier is an optimization to prevent the requirement for a full read (or load) barrier.

- Read (or load) memory barrier

  A read load barrier is a data dependency barrier plus a guarantee that all LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier wrt other components in the system.

  A partial ordering on loads only; no required effect on stores.

  Imply data dependency barrier.

  Should be paired with write barriers ??

- General Memory barrier

  All LOAD and STORE operations before the barrier will happen before all LOAD and STORE operations specified after the barrier with respect to other components of the system.

  A partial ordering over both loads and stores.

- Implicit Barriers

  - ACQUIRE operations
  - RELEASE operations

3.3 Data Dependency Barrier

To better understand need for data dependency barriers consider the following sequence of instructions:

```
CPU 1          CPU 2
===============        ===============
{ A == 1, B == 2, C = 3, P == &A, Q == &C }
B = 4;
<write barrier>
```

```
  ACCESS_ONCE(P) = &B
            Q = ACCESS_ONCE(P);
            D = *Q;
```

While the write barrier will guarantee that all writes seen by both processors. After the barrier Q will be assigned either the address of B or the address of A depending on whether the update to P was seen by CPU2 or not.

D is assigned a dereference to Q. We thus expect the dereference to dereference A or B getting values of A=1 or B=4.

BUT, CPU 2's perception of P may be updated before its perception of B leading to the following situation.

```
(Q = &B) and (D = 2 ) ???
```

```
        CPU 1         CPU 2
  ===============       ===============
  { A == 1, B == 2, C = 3, P == &A, Q == &C }
  B = 4;
  <write barrier>
  ACCESS_ONCE(P) = &B
            Q = ACCESS_ONCE(P);
            <data dependency barrier>
            D = *Q;
```

The data dependency barrier forces the assignment of B to show up on processor 2 when Q is assigned a pointer to B.

3.4 Control Dependencies

Control dependencies are those which require a full read memory barrier where a data dependency barrier will simply not suffice.

```
q = ACCESS_ONCE(a);
if (q) {
  <data dependency barrier>  /* BUG: No data dependency!!! */
  p = ACCESS_ONCE(b);
}
```

Here CPU may attempt to short circuit in an attempt to predict the branch outcome. Thus the load from b may appear to happen before the load from a.

The solution for this is to impose a read barrier to ensure that b is read

While LOAD is speculated the STORES are NOT speculated. Thus the ordering semantics of following will not be affected.

```
  q = ACCESS_ONCE(a);
  if (q) {
    ACCESS_ONCE(b) = p;
  }
```

The `ACCESS_ONCE` directive is aimed at the compiler to prevent it from merging separate loads from 'a' and separate stores to 'b'.

TODO ......

3.5 Explicit Kernel Barriers

3.5.1 Compiler Barriers

Used primarily to prevent the compiler from reordering instructions the `barrier()` is a general barrier. The `ACCESS_ONCE()` macro is a weaker form of the barrier instruction which only affects instructions flagged by `ACCESS_ONCE()`.

Can be implemented using the
`barrier()` macro expanding to `asm volatile"":::memory`

Tells the compiler to insert empty assembly fragment. While the volatile keyword forbids the compiler from shuffling the instruction. The memory keyword forces the compiler to use memory locations instead of those stored in the register. The CPU can still mix assembly instruction

Effects of `barrier()`:

- Prevent compiler from reordering accesses following barrier() with those preceding them them.
- Within loop, foce compiler to lad variables used in loop conditional on each pass through loop.

Effects of `ACCESS_ONCE()` , prevent optimizations that though safe in single threaded code will break concurrent code.

- Provide cache coherence for accesses from multiple CPUs to single variable. Prevent compiler for reordering loads and stores to same variable.

  Thus

  ```
      a[0] = ACCESS_ONCE(x);
      a[1] = ACCESS_ONCE(x);
  ```

  Will prevent `a[1]` from receiving older value of x than `a[0]`.

- Prevent merging successive loads

  ```
      while (tmp = ACCESS_ONCE(a))
      do_something_with(tmp);
  ```

  Will prevent the compiler from optimizing the loop otherwise as

  ```
    if (tmp = a)
      for (;;)
        do_something_with(tmp);
  ```

-
-

It is not necessary to use `ACCESS_ONCE()` on variables marked volatile since it is implemented as a volatile cast.

It must be noted that compiler barriers ***DO NOT*** directly affect the CPU, which may still reorder instructions.

3.5.2 CPU Memory Barriers

All memory barriers except data dependency barriers imply compiler barrier. SMP memory barriers are reduced to compiler barriers on uni-processor compiled systems. All memory barriers except data dependency barriers imply a compiler barrier.

Type	Mandatory	SMP Conditional
GENERAL	`mb()`	`smb_mb()`
WRITE	`wmb()`	`smp_wmb()`
READ	`rmb()`	`smp_rmb()`
DATA DEPENDENCY	`read_barrier_depends()`	`smp_read_barrier_depends()`

3.5.3 Memory barrier and 80x86

In 80x86 list of serializing instructions which act as memory barriers:

- I/O port operations
- instructions with lock byte
- instructions affecting the IF flag in eflags register such as those instructions which write to registers
  - control registers (cli)
  - system registers (sti)
  - debug registers
- Some instructions introduced in Pentium 4
  - lfence - read barriers
  - sfence - write barries
  - mfence - read write barriers
- Speial instructions - iret terminating interrupt or exception handler

Read barriers maintain the serial order of read instructions, write barriers maitain serial order of write instructions.

Function/Macors	Description
`mb()`	Memory barrier for MP and UP
`rmb()`	Read memory barrier for MP and UP
`wmb()`	Write memory barrier for MP and UP
`smp_mb()`	Memory barrier for MP only
`smp_rmb()`	Read memory barrier for MP only
`smp_wmb()`	Write memory barrier for MP only

Macro expansions on 80x86

Function/Macors	Description
mb()	`asm volatile("mfence":::"memory")`

rmb()	`asm volatile ("lfence")`
	`or`
	`asm volatile ("lock;addl 0,0(%esp)":::"memory")`

`wmb()`	`barrier()` Intel never reorders write memory access so we get only a compiler barrier here.
`smp_mb()`	`mb()`
`smp_rmb()`	`barrier()` or `rmb()` depending on `CONFIG_X86_PPRO_FENCE`
`smp_wmb()`	`barrier()`

The usage of mandatory barrier may be used to control MMIO effects since they affect the order in which memory operations appear to device and prevent the compiler and CPU from reordering them.

3.6 Implicit Kernel Memory Barriers

Many locking and scheduling functions imply memory barriers. We consider the some of these implicit barriers.

3.6.1 Acquiring Functions

Following are examples of acquiring functions inside the kernel.

- Spin Locks
- Read/Write Spin locks
- Mutexes
- Semaphores
- Read/Write Semaphores
- RCU
- **ACQUIRE Implication**

  Post-ACQUIRE operations *will be completed* after after ACQUIRE completion.

  Pre-ACQUIRE operations *may be completed* after ACQUIRE completion.

- **RELEASE Implications**

  Pre-RELEASE operations *will be completed* before RELEASE completion.

Post-RELEASE operations *may be completed* before RELEASE completion.

- **ACQUIRE vs ACQUIRE Implication**

  All ACQUIRE operations will be completed before successive ACQUIRE operation .

- **ACQUIRE vs RELEASE Implication**

  All ACQUIRE operations issued before RELEASE operation will be completed before RElEASe operation.

- **Failed conditional ACQUIRE Implication**

  Failed lock operations dont imply any barrier.

Due to the possibility that pre-ACQUIRE operations may happen after ACQUIRE and post-RELEASE may happen before RELEASE thus accesses may cross.

```
*A = a;
ACQUIRE M
RELEASE M
*B = b;
```

May get run as

```
ACQUIRE M, STORE *B, STORE *A, RELEASE M
```

Also RELEASE followed by ACQUIRE does not imply a full memory barrier.

```
*A = a;
RELEASE M
ACQUIRE N
*B = b;
```

May run as :

```
ACQUIRE N, STORE *B, STORE *A, RELEASE M
```

3.6.2 Interrupt Disabling Functions

Functions disabling/enabling interrupts will only act as compiler barriers.

3.7 Atomic Operations

Many atomic operations imply full memory barriers and are heavily relied upon in the kernel.

Any atomic operation that modifies state in memory and returns information about the state implies SMP-conditional memory barrier. on each side of the operation. That is `smp_mb()`.

Atomic operations which use which imply SMP general memory barrier include:

Atomic Operations	
`xchg();`	
`cmpxchg();`	
`atomic_xchg();`	`atomic_long_xchg()`
`atomic_cmpxchg();`	`atomic_long_cmpxchg()`
`atomic_inc_return();`	`atomic_long_inc_return()`
`atomic_dec_return();`	`atomic_long_dec_return()`
`atomic_add_return();`	`atomic_long_add_return()`
`atomic_sub_return();`	`atomic_long_sub_return()`
`atomic_inc_and_test();`	`atomic_long_inc_and_test()`
`atomic_dec_and_test();`	`atomic_long_dec_and_test()`
`atomic_sub_and_test();`	`atomic_long_sub_and_test()`
`atomic_add_negative();`	`atomic_long_add_negative()`
`test_and_set_bit();`	
`test_and_clear_bit();`	
`test_and_change_bit();`	

These operations of aften used for implementing ACQUIRE and RELEASE operations and maintaining reference counters.

3.7.1 Atomic Operations 80x86

- Read-modify-write assembly instructions such as inc and dec that read data from memory and update it are atomic , provided stale data has not been read by another processor. This is the case in uniprocessor systems.
- Read-modify-write instructions whose opcode is prefixed by the *lock byte* (`0xf0`) are atomic on multiprocessor systems. The lock byte will lock access to the mememory bus until the locking instruction finishes its operation.
- Instructions prefixed by the *rep* byte `0xf2,0xf3` which forces instructions to be repeated are not atomic since the CPU checks interrupts before each iteration.

All atomic operations act as memory barriers since they use the lock byte.

3.8 Spinlock

Spinlock is implemented by the `spinlock_t` which consists of two fields:

- **slock** Encodes the spinlock state. Value 1 corresponds to unlocked state, Negative values and 0 denote locked state.

- **break_lock** Flag signaling that a process is busy waiting for the lock. Present on SMP systems with kernel preemption.

Some functions that initialize, acquire and releases spin locks are given in the table.

Function/Macors	Description
`spin_lock_init()`	Set spin lock to 1 (unlocked)
`spin_lock()`	Cycle until spinlock becomes 1(unlocked) then set it to 0 (locked)
`spin_unlock()`	Set the spin lock to 1 (unlocked)
`spin_unlock_wait()`	Wait until the spinlock becomes 1 (unlocked)
`spin_is_locked()`	Return 0 if the spinlock is set to 1(unlocked) ; 1 otherwise
`spin_trylock()`	Set the spin lock to 0 (locked), and return 1 if the previous value of the lock was 1; 0 otherwise

3.8.1 spin_lock with kernel preemption

- Invoke `preeempt_disable()` disable kernel preemption
- Invoke `_raw_spin_trylock()` atomic test and set on spinlock's `slock` field.

```
movb $0,%a1
xchgb %a1,slp->slock
```

  `xchg` exchange atomically content of 8-bit `%a1` with `slp->slock`. return 1 if old value was positive or 0 otherwise.

- If old value of spin lock was positive, we have acquired the spinlock.
- If failed then spin lock was not positive , invoke `preempt_enable()` decrements the preeempt counter. which if it goes to zero will allow the process to be scheduled out,
- set the `break_lock` field to one. Allow another process to release spin lock prematurely
- Execute the wait cycle

```
while(spin_is_locked(slp) && slp->break_lock)
    cpu_relax(); // special pause instruction Pentium 4
```

- Jump back to step 1

3.8.2 spin_lock (no kernel preemption)

The following tight busy wait is implemented. An atomic decrement of the spinlock is performed using the `lock` prefix on the decrement operation. A test is performed on sign flag if it is positive we continue with instruction 3. Otherwise tight loop at 2 is executed until the spinlock is positive. When spinlock attains positive value the execution will restart at label 1 where we will try to atomically decrement the spin lock.

```
1:  lock; decb slb->slock
    jns 3f
2:  pause
    cmpb $0,slp->slock
    jle 2b
    jmp 1b
3:
```

3.8.3 spin_unlock

This releases an acquired spin lock. Executes the assembly instruction:

```
movb $1,slp->slock
```

Then invoking `preempt_enable()`. on x86 the lock byte is not used since since write-only access in memory are atomically executed.

3.9 Semaphores

Semaphores are an alternative mechanism to spinlock to implement critical sections in kernel control path, in that they do not allow a process to proceed unless the lock is open. But unlike spinlocks whenever a kernel control path tries to acquire a busy resource, the corresponding process is suspended. Thus semaphors should not be accessed from non-suspendable control paths like interrupt handlers and deferred functions.

The `struct semaphore` contains

- `count`
  An `atomic_t` value. A positive value(greater than 0) indicates the resource is free. Negative value indicates there is at least one process waiting on the resource.

- `wait`
  A wait queue contianing list of processes sleeping on the semaphore.

- `sleepers`
  A flag indicating if there exists a process sleeping on semaphores.

Semaphores expected to be used enforce only single active control path in the critical region can be initalized using functions `init_MUTEX()` and `init_MUTEX_LOCKED()` or `DECLARE_MUTEX` , `DECLARE_MUTEX_LOCKED` for static allocation. The count in mutexes are not expected to exceed 1.

3.9.1 Releasing Semaphores

A process reseases a semaphore by invoking `up()`. Which on x86 performs equivalent of

```
  movl $sem->count,%ecx
  lock; incl (%ecx)
  jg 1f
  lea %ecx,%eax
  pushl %edx
  pushl %ecx
  call __up
  popl %ecx
  popl %edx
1:
```

Where `__up()` is the C function :

```
__attribute__((regparm(3))) void __up(struct semaphore *sem)
{
  wake_up(&sem->wait);
}
```

The incrementation of the semaphore count and the test for semaphore count greter than 0 happens under atomic `lock` ensuring the consistency of the value is mainted accorss multiple processors. If count is greater than 0 then one of the list of waiting process is woken up.

3.9.2 Acquiring Semaphores

Lock acquisition performs the equivalent of the following assembly instructions

```
  down:
   movl $sem->count,%ecx
   lock; decl (%ecx);
   jns 1f
   lea %ecx, %eax
   pushl %edx
   pushl %ecx
   call __down
   popl %ecx
   popl %edx
 1:
```

Where we decrement the semaphore count and test if the value is positive if not we have to queue the current process in the semaphore's wait list via the `__down` function.

```
__attribute__((regparm(3))) void __down(struct semaphore * sem)
{
  DECLARE_WAITQUEUE(wait, current);
  unsigned long flags;
  current->state = TASK_UNINTERRUPTIBLE;
  spin_lock_irqsave(&sem->wait.lock, flags);
  add_wait_queue_exclusive_locked(&sem->wait, &wait);
  sem->sleepers++;
```

```c
  for (;;) {
    if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
      sem->sleepers = 0;
      break;
    }
    sem->sleepers = 1;
    spin_unlock_irqrestore(&sem->wait.lock, flags);
    schedule();
    spin_lock_irqsave(&sem->wait.lock, flags);
    current->state = TASK_UNINTERRUPTIBLE;
  }
  remove_wait_queue_locked(&sem->wait, &wait);
  wake_up_locked(&sem->wait);
  spin_unlock_irqrestore(&sem->wait.lock, flags);
  current->state = TASK_RUNNING;
}
```

The decrement and test of the semaphore count happens under the `lock` instruction. If the count count is negative the process is suspended and added to the semaphores wait queue.

- Modify current process state from `TASK_RUNNING` to `TASK_UNINTERRUPTIBLE`
- Add the processs to the semaphore wait queue under a spin lock.
- The `sleepers` which in the fast path is 0 if no process is in the wait queue is incremented

  We consider the cases

  - MUTEX semaphore is open (count == 1 sleepers == 0)

    We set the count to 0. Skip over the execution of the `__down()`. This is considered the fast path.
  - MUTEX semaphore is closed, no sleeping process (count == 0 , sleepers == 0)

    We decrement the count(new value is -1) and invoke `__down()` . We enter the loop where if count is still -1 then set the sleepers to 1. Only we were sleeping thus we can go back to sleep. Otherwise if count is non-negative we can set the sleepers to 0. And begin the task of sceduling ourself for execution by changing process tate to `TASK_RUNNING`
  - MUTEX semaphore is closed with additional sleepers (count <= -1 , sleepers >= 1)

    We check if the semaphore got released before the `__down()`.

3.9.3 Interrupt Disabling

4 MESI Cache Coherency Protocol

Cache-coherency protocols manage cache-line states to prevent inconsistent states or lost data. MESI stats for "modified", "exclusive", "shared" and "invalid". Which represent the four states assigned to cache lines in this protocol.The MESI protocol is a protocol used to implement cache and memory coherency amongst multiple CPUs. [?]

Two bits are added to each cache line which represent the four states that a cache line can be in.

- Modified
  - Cache Line is present only on current CPU
  - Cache Line has been modified
  - Write back needs to be performed
- Exclusive
  - Cache Line is present only in current CPU
  - Cache Line is clean (matches main memory)
- Shared
  - Cache Line is present in multiple CPUs
  - Cache Line is Clean (matches main memory)
- Invalid

4.1 MESI Protocol messages

If the CPUs are on a single shared bus we only require the following messages on the bus.

- **Read** Content : physical address of cache line to be read.
- **Read Response** Response with data, Source : Memory or one of the other caches. If the other cache has data in modified state.
- **Invalidate** Content : Address of cache line to be invalidated. All other caches must invalidate cache line with this address.
- **Invalidate Acknowledge** From: CPU that has invalidated a cache line.
- **Read Invalidate** Content: Read cache line and take ownership of it getting the cache line removed from other processors. Combination of read and invalidate. Responses are read response and invalidate acknowledge

- **Writeback** contnet: Address of and data of write back to memory. This message might get snooped by other processors to mark cache lines as "modified"

Thus we see the fractal nature of distributed system where message passing is implemented at different levels of the systems architecture.

4.2 MESI Transitions Table

We give a tabular description of transitions involved in the MESI protocol.

#	Start	End	Descriptions
a	Modified	Exclusive	Cache line is written back to memory but CPU retains exclusive owner ship of and right to modify it. Requires "writeback" message
b	Exclusive	Modified	CPU writes to exclusive cache line. No messages required.
c	Modified	Invalid	CPU receives a "read invalidate" for a cache line it modified.It must invalidate the local copy, send an read response and a invalidate acknowledge
d	Invalid	Modified	An atomic read-modify-write operation on a data item not present in the cache. Transmits a "read invalidate" response and gets "read response". Cannot proceed until all cpus reply with "invalidate acknowledge" response.
e	Shared	Modified	CPU does read-modify write one data item that was read-only. Transmits a "invalidate" messagee. Wait for invalidate acknowlege from all other CPUs.
f	Modified	Shared	Some other CPU reads from our cache line suplied from this CPU. This CPU responds with a "read response" message.
g	Exclusive	Shared	Some other CPU reads data in this cache line supplied from this CPU or from memory. If this CPU retains a read-only copy. This cpu will respond with a read response with requested data.
h	Shared	Exclusive	This CPU is about to write to shared item. Transmits an invalidate message to other CPUs Waits for full "invalidate acknowledge" response. Or all other CPUs had to drop the cache line making this CPU the sole owner.
i	Exclusive	Invalid	Another CPU ran an atomic read-modify-write in cache line held by current CPU. Received a "read invalidate". This CPU will respond with a "read response" and "invalidate acknowledge" message.
j	Invalid	Exclusive	CPU does a store to data not in cache. Transmits a "read invalidate" message. The CPU must wait for all other processors to "invalidate acknowledge". An expected next state will be "modified"
k	Invalid	Shared	The CPU loads item not in cache and completes a transition on read a "read response"
l	Shared	Invalid	Some other cpu does a store to data item. On the reception of a "invalidate" message. Responds with a "invalidate acknowledge" message.

4.3 Summary & Acknowledgment

Lot of the text is adaptation of Chapter 5 of Linux Kernel Programming by Bovett and the mememory barrier documentation part of the linux kernel David Howells and Paul E. McKenney. Any errors are entirely my own. For people

looking for definitive and trustworthy accounts looking at these sources is recommended.

References

[1] *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*
Chapter 7.1: Locked Atomic Operations
Chapter 7.2: Memory Ordering
Chapter 7.4: Serializing Instructions

[2] *Unix Systems for Modern Architectures, Symmetric Multiprocessing and Caching for Kernel Programmers*
Chapter 13: Other Memory Models

[3] *MESI protocol*
`http://en.wikipedia.org/wiki/MESI_protocol`

[4] *Cache Coherency Primer Fabian "ryg" Giesen*
`https://fgiesen.wordpress.com/2014/07/07/cache-coherency/`

[5] *Atomic Operations - CSE 378 University of Washington*
`http://courses.cs.washington.edu/courses/cse378/07au/lectures/L25-Atomic-Operations.pdf`

[6] *Linux Kernel Memory Barriers*
`https://www.kernel.org/doc/Documentation/memory-barriers.txt`

[7] *Semantics and Behavior of Atomic and Bitmask Operations*
David S. Miller
`https://www.kernel.org/doc/Documentation/atomic_ops.txt`

[8] *Synchronization in Linux*
`http://www.cs.columbia.edu/~junfeng/10sp-w4118/lectures/l11-synch-linux.pdf`

[9] *Understanding the Linux Kernel - Chapter 5*
David Bovet and Marco Cesati

[10] *Structures and Design of Computers*
David E. Patterson and J.L. Hennessey

[11] *Why memory barriers?*
Paul McKay
`http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf`