Linux Kernel Synchronization

Paul Chu

Hao-Ran Liu

Term Definitions

- Critical sections
  - Code paths that access shared data
- Race condition
  - If two context access the same critical section at the same time
- Synchronization
  - We use synchronization primitives in the kernel to ensure that race conditions do not happen

Examples of race condition

```
int a = 0, b = 0; /* a+b must be 0  always  */
```

```
int thread_one(int argc, char** argv) {
        a++;
        do_io_block(); // dead in non-preemptive kernel
        b--;
}
```

```
int thread_two(int argc, char** argv) {
        a++;
        b--; // dead in preemptive kernel or in SMP
}
```

```
int thread_x(int argc, char** argv) {
        do_actions(a, b); // asuming a+b == 0
}
```

```
int isr(int argc, char** argv) {
        a++;
        b--; // dead if other threads do not disable irq
}
```

Source of concurrency in the Linux kernel

- Interrupt handling (pseudo concurrency)
  - Interrupt handlers
  - Bottom halves
- Kernel preemption (pseudo concurrency)
  - Cooperative: tasks invoke the scueduler for sleeping or synchronization
  - Noncooperative: one task in the kernel can preempt another when the kernel is preemptive
- Symmetrical multiprocessing
  - Kernel code can be executed by two or more processors

Locking

- To prevent concurrency execution of a critical section
- Locking protocol
  - Acquire a lock before accessing shared data
  - If the lock is already held, you must wait
  - Release the lock after completing the access
- Where to use
  - Identify what data needs protection
  - Locks go with data, not code

Deadlocks

- A condition when threads hold the locks that others are waiting for and they themselves are waiting for locks held by others
- Deadlock can be prevented if any of the following condition is not true
  - Mutual exclusion, Hold and wait, No preemption, Circular waiting
- Strategies
  - Enforce a specific locking order
  - Reduce the number of locks to hold at the same time
  - Prevent starvation

Lock contention and scalability

- A highly contended lock can slow down a system's performance
  - Because a lock's job is to serialize access to a resource
  - This becomes worse when the number of processors is increased
- Solution
  - Divide a coarse lock into fine-grained lock
  - Eliminate the needs to lock by separating data
    - Per processor data

Atomic Operations

- Atomicity
  - Not dividable by interrupts
    - Eliminate pseudo concurrency
    - May be mimic by disabling interrupt during operations
  - Not dividable by other processors
    - Bus locking capability in hardware must be supported
- When to use
  - Sharing simple data types; e.g. integer, bits
  - No consistency requirement on two or more variables
  - Better efficiency than complicated locking mechanisms
- As the building blocks of complicated locking mechanisms

Overhead of atomic operations

- Disable/enable local interrupt or lock/unlock bus is not without cost
  - Implicit memory barrier cause CPU to flush its pipeline
- Data caches invalidation of frequently modified variables shared between different CPUs
- These are the overheads RCU avoids

Atomic Operations in Linux kernel

- Atomic integer operations
  - atomic_t ensures variables not be processed by non-atomic routines

```
ATOMIC_INIT(int i)
int atomic_read(atomic_t *v) / void atomic_set(atomic_t *v, int i)
void atomic_add(int i, atomic_t *v) / void atomic_sub(int i, atomic_t *v)
void atomic_inc(v) / void atomic_dec(v)
int atomic_dec_and_test(atomic_t *v) / int acomic_inc_and_test (atomic_t *v)
atomic_add_negative(int i, atomic_t *v)
```

- Atomic bitwise operations

```
int set_bit(int nr, void *addr) / int clear_bit(int nr, void *addr)
int test_bit(int nr, void *addr)
int change_bit(int bit, void *addr)
test_and_set_bit(int nr, void *addr) / test_and_clear_bit(int nr, void *addr)
test_and_change_bit(int nr, void *addr)
```

  - Non-atomic operations: __test_bit(), and etc.
  - Local-only atomic operations: local_add(local_t), and etc.
  - The only portable way to set a specific bit (endianess)

Atomic operations on x86

- The processor use 3 interdependent mechanisms for carrying out locked atomic operations
  - Guaranteed atomic operations
    - Reading or writing a byte, a word aligned on 16-bit boundary, a doubleword aligned on 32-bit boundary
  - Bus locking
    - Automatic locking: accessing memory with XCHG instruction
    - Software-controlled locking: use the prefix LOCK with certain instructions
  - Cache coherency protocols
    - The area of memory being locked might be cached in the processor

Implementing atomic operations on x86

- Accessing a doubleword is guaranteed to be atomic

```
#ifdef CONFIG_SMP
#define LOCK "lock ; "
#else
#define LOCK ""
#endif

#define atomic_read(v)          ((v)->counter)

#define atomic_set(v,i)         (((v)->counter) = (i))

static __inline__ void atomic_add(int i, atomic_t *v)
{
        __asm__ __volatile__(
                LOCK "addl %1,%0"
                :"=m" (v->counter)
                :"ir" (i), "m" (v->counter));
}
```

Memory barriers

- Both compilers and processors reorder instructions to get better runtime performance
  - Compilers reorder instructions at compile time (e.g. to increase the throughput of pipelining)
  - CPUs reorder instructions at runtime (e.g. to fill execution units in a superscalar processor)
- Sometimes, we need memory read (load) and write (store) issued in the order specified in our program
  - Issuing I/O commands to hardware
  - Synchronized threads running on different processors
  - Protecting instructions in a critical section from bleeding out
- A memory barrier primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive

Memory barriers in Linux kernel

- Compiler barrier
  - barrier(): prevents the compiler from optimizing stores/loads across it
- Hardware barrier + compiler barrier
  - read_barrier_depends(): prevents data-dependent loads from being reordered across it
  - rmb(): prevents loads from being reorder across it
  - wmb(): prevents stores from being reordered across it
  - mb(): prevents loads or stores from being reordered across it
- These macros provide a memory barrier on SMP, and provide a compiler barrier on UP*
  - smp_read_barrier_depends()
  - smp_rmb()
  - smp_wmb()
  - smp_mb()

* memory order observed by processes on the same CPU is guaranteed by processor (*precise interrupt*)

Example of using memory barriers

- Without memory barriers, it is possible that c gets the new value of b, whereas d receives the old value of a

```
Thread 1                    Thread 2

a = 3;                      c = b;
mb();                       rmb();
b = 4;                      d = a;
```

- Without memory barriers, it is possible for b to be set to pp before pp was set to p

```
Thread 1                    Thread 2

a = 3;                      pp = p;
mb();                       read_barrier_depends();
p = &a;                     b = *pp;
```

Memory ordering of various CPUs

- x86 does not support out-of-order stores (except few string operations)
- Atomic instructions on x86 comes with implicit memory barriers.

	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?
Alpha	Yes	Yes	Yes	Yes	Yes	Yes	Yes
AMD64	Yes			Yes			
IA64	Yes	Yes	Yes	Yes	Yes	Yes	
PowerPC	Yes	Yes	Yes	Yes	Yes	Yes	
x86	Yes	Yes		Yes			

Copy from ”Memory ordering in modern processors, part I”, Paul McKenny, Linux Journal #136, 2005

Memory barriers instructions for x86

- Serializing instructions (implicit memory barriers)
  - All instructions that operate on I/O ports
  - All instructions prefixed by the *lock* byte
  - All instructions that write into control registers, system registers or debug registers (e.g. cli and sti)
  - A few special instructions (invd, invlpg, wbinvd, iret …)
- Memory ordering Instructions (explicit memory barriers)
  - lfence, serializing all load operations
  - sfence, serializing all store operations
  - mfence, serializing all load and store operations

Implementing memory barriers on x86

- The __volatile __ tells gcc that the instruction has important side effects. Do not delete the instruction or reschedule other instructions across it
- The “memory” tells gcc that the instruction changes memory, so that it does not cache variables in registers across the instruction.

```
#define barrier() __asm__ __volatile__("": : :"memory")
#define mb() alternative("lock; addl $0,0(%%esp)", "mfence", X86_...)
#define rmb() alternative("lock; addl $0,0(%%esp)", "lfence", X86_...)
#define read_barrier_depends()    do { } while(0)
#define wmb()    __asm__ __volatile__ ("": : :"memory")

#ifdef CONFIG_SMP
#define smp_mb()          mb()
#define smp_rmb()         rmb()
#define smp_wmb()         wmb()
#define smp_read_barrier_depends()        read_barrier_depends()
#else
#define smp_mb()          barrier()
#define smp_rmb()         barrier()
#define smp_wmb()         barrier()
#define smp_read_barrier_depends()        do { } while(0)
#endif
```

Disabling Interrupts

- Disable interrupts
  - Eliminate pseudo concurrency on single processor
    - Coupled with spinlock if sharing data between multiple processors
  - Lead to longer interrupt latency
  - When to use
    - Normal path shares data with interrupt handlers
    - Interrupt handlers share data with other interrupt handlers
    - One interrupt handler for different IRQs share data within it; the interrupt handler might be reentrant
    - Shorter duration of critical sections
  - Need not to use
    - Sharing data within an interrupt handler of a IRQ; interrupt handler of a IRQ is not reentrant in SMP

Interrupt Control Routines

- local_irq_disable() / local_irq_enable()
  - Disable or enable all interrupts of current CPU
- local_irq_save(flags) / local_irq_restore(flags)
  - Save current IRQ state and disable IRQ
  - Restore IRQ state instead of enabling it directly
  - When a routine is reached both with and without interrupts enabled
- disable_irq(irq) / enable_irq(irq)
  - Disable or enable a specific IRQ line for all CPUs
  - Return only when the specific handler is not being executed
- disable_irq_nosync(unsigned int irq)
  - Disable a specific IRQ line without waiting it (SMP)
- State checking
  - irqs_disabled(): if all local IRQs are disabled
  - in_interrupt(): if being executed in interrupt context
  - in_irq(): if being executed in an interrupt handler

Disabling preemption

- Context switches can happened at any time with a preemptive kernel even when a process is in the kernel mode
  - Critical sections must disable preemption to avoid race condition
- preempt_disable() / preempt_enable() is nestable; kernel maintain a preempt count for every processes.
- Preemption-related functions

```
#define preempt_count()  (current_thread_info()->preempt_count)
#define preempt_disable() \
do {     inc_preempt_count(); \
         barrier(); \
} while (0)
#define preempt_enable_no_resched() \
do {     barrier(); \
         dec_preempt_count(); \
} while (0)
#define preempt_enable() \
do {     preempt_enable_no_resched(); \
         preempt_check_resched(); \
} while (0)
```

Spin Locks

- Disabling interrupts cannot stop other processors
- Spin lock busy waits a shared lock to be release
  - Lightweight single-holder lock, all other threads will be busy looping to poll the shared lock
- When it's UP system
  - Markers to disable kernel preemption (scheduling latency)
  - Or, be removed at compile time if no kernel preemption
- When to use
  - Sharing data among threads running on processors of SMP system
  - Sharing data among preempt-able kernel threads
  - Shorter duration of critical sections

Spin lock functions

- spin_lock_init()
  - Runtime initializing given spinlock_t
- spin_lock() / spin_unlock()
  - Acquire or release given lock
- spin_lock_irq() / spin_unlock_irq()
  - Disable local interrupts and acquire given lock
  - Release given lock and enable local interrupts
- spin_lock_irqsave() / spin_unlock_irqrestore()
  - Save current state of local interrupts, disable local interrupts and acquire given lock
  - Release given lock and restore local interrupts to given previous state
- spin_trylock()
  - Try to acquire given lock; if unavailable, returns zero
- spin_islocked()
  - Return nonzero if the given lock is currently acquired

Spin lock implementation on x86

- Implementation for SMP and preemptive kernel

```
#define spin_lock(lock)                 _spin_lock(lock)
void __lockfunc _spin_lock(spinlock_t *lock) {
        preempt_disable();
        if (unlikely(!_raw_spin_trylock(lock)))
                __preempt_spin_lock(lock);
}
```

- *xchgb* will lock the bus; it acts as a memory barrier

```
static inline int _raw_spin_trylock(spinlock_t *lock) {
        char oldval;
        __asm__ __volatile__(
                "xchgb %b0,%1"
                :"=q" (oldval), "=m" (lock->lock)
                :"0" (0) : "memory");
        return oldval > 0;
}
```

Spin lock implementation on x86

```
#define spin_is_locked(x) (*(volatile signed char *)(&(x)->lock) <= 0)

/* This could be a long-held lock.  If another CPU holds it for a long time,
 * and that CPU is not asked to reschedule then *this* CPU will spin on the
 * lock for a long time, even if *this* CPU is asked to reschedule.
 * So what we do here, in the slow (contended) path is to spin on the lock by
 * hand while permitting preemption. */
static inline void __preempt_spin_lock(spinlock_t *lock)  {
        if (preempt_count() > 1) {
                _raw_spin_lock(lock);
                return;
        }
        do {
                preempt_enable();
                while (spin_is_locked(lock))
                        cpu_relax();
                preempt_disable();
        } while (!_raw_spin_trylock(lock));
}
```

Spin lock implementation on x86

```
#define spin_lock_string \
        "\n1:\t" \
        "lock ; decb %0\n\t" \              /* lock bus, memory barrier */
        "jns 3f\n" \                        /* jump if we acquire the lock */
        "2:\t" \                            /* spin lock loop below */
        "rep;nop\n\t" \                     /* = cpu_relax() */
        "cmpb $0,%0\n\t" \                  /* check if lock is available */
        "jle 2b\n\t" \                      /* jump if lock not available */
        "jmp 1b\n" \                        /* lock available, try lock again */
        "3:\n\t"                            /* lock is acquired */

static inline void _raw_spin_lock(spinlock_t *lock) {
        __asm__ __volatile__(
                spin_lock_string
                :"=m" (lock->lock) : : "memory");
}
```

Spin lock implementation on x86

```
#define spin_unlock_string \
        "movb $1,%0" \
                :"=m" (lock->lock) : : "memory“

static inline void _raw_spin_unlock(spinlock_t *lock) {
        __asm__ __volatile__(
                spin_unlock_string
        );
}
#define spin_unlock(lock) _spin_unlock(lock)
void __lockfunc _spin_unlock(spinlock_t *lock) {
        _raw_spin_unlock(lock);
        preempt_enable();
}
```

- Conclusion about spin lock
  - Spin lock implementation is composed of provided by atomic operations, memory barriers and (preemption/bottom halve/interrupt) disabling

Reader-writer spin locks

- Multiple concurrent accesses to shared data are read-only
- Multiple read locks can be granted, but write lock is allowed only when there is no any lock.
- Favor readers over writers: writers starvation
- Operations
  - rw_lock_init(), rw_is_locked()
  - read_lock(), read_lock_irq(), and so on.
  - write_lock(), write_lock_irq(), and so on.

Sequence Locks

- What are sequence locks (seqlock_t)
  - Similar to reader-writer spin locks
  - But favor writers over readers
  - A writer accesses data after acquiring a lock, while readers check the lock by polling the sequence count
    - Read needs retry if the count differs or it is an odd number
- How to use
  - Writers

```
write_seqlock(seqlock);                    // acquire lock & increment count
/* write the shared data */
write_sequnlock(seqlock);                  // release lock & increment again
```

  - Readers

```
do {
        seq = read_seqbegin(seqlock);      // get sequence count
        /* read the shared data */
} while (read_seqretry(seqlock, seq);      // check if write lock is obtained
```

Sequence lock implementation

- Sequence lock data structure

```
typedef struct {
        unsigned sequence;
        spinlock_t lock;
} seqlock_t;
```

- Sequence lock functions for writers

```
static inline void write_seqlock(seqlock_t *sl) {
        spin_lock(&sl->lock);
        ++sl->sequence;
        smp_wmb();
}
static inline void write_sequnlock(seqlock_t *sl) {
        smp_wmb();
        sl->sequence++;
        spin_unlock(&sl->lock);
}
```

Sequence lock implementation

- Sequence lock functions for readers

```
static inline unsigned read_seqbegin(const seqlock_t *sl)
{
        unsigned ret = sl->sequence;
        smp_rmb();
        return ret;
}
static inline int read_seqretry(const seqlock_t *sl, unsigned iv)
{
        smp_rmb();
        return (iv & 1) | (sl->sequence ^ iv);
}
```

volatile keyword v.s. memory barriers

- Small quiz
  - ***jiffies*** *variable in Linux kernel is declared with* ***volatile*** *keyword. The keyword tells the compiler that the value of this variable may change at any time and disables compiler optimization on it.*
  - *The question is: Why is not the field* ***sequence*** *in* ***seqlock_t*** *declared as* ***volatile****? (hint: the purpose of* ***smp_rmb()*** *and* ***smp_wmb()****)*

Semaphores

- Sleeping locks
  - The locking thread is put to sleep and be woken up when the lock is released
- When to use
  - The Lock is to be held for a long time
    - the overhead of sleeping outweigh the lock hold time
  - Can be used only in process context
  - Shared data among threads
- Notes
  - Do not hold spin lock before acquire a semaphore
  - Thread holding semaphore might be preempted
- Types of semaphores
  - Binary semaphore and mutex
  - Counting semaphore
    - More than one semaphore holders are allowed
    - When the shared resources are more than one

More About Semaphores

- Semaphores operations

```
sema_init (semaphore*, int)
init_MUTEX(semaphore*) / init_MUTEX_LOCKED(semaphore*)
down(semaphore*) / down_interruptible(semaphore*)
down_trylock(semaphore*)
up(semaphore*)
```

- Reader-writer semaphores (rw_semaphore)
  - Same as reader-writer spin locks
  - All uninterruptible sleep
  - Converting acquired write lock to read lock

```
init_rwsem(rw_semaphore* )
down_read(rw_semaphore*) / down_write(rw_semaphore*)
up_read(rw_semaphore*) / up_write(rw_semaphore *)
down_read_trylock(rw_semaphore*) / down_write_trylock(rw_semaphore*)
downgrade_write(rw_semaphore *)
```

Other Synchronization Controls

- Bottom halves disabling
  - Sharing data with softirqs and tasklets
    - local_bh_disable() / local_bh_enable()
    - spin_lock_bh() / spin_unlock_bh()
- Completion variables
  - One thread waits another thread to complete some tasks
    - wait_for_completion() / complete ()
- Big kernel lock (BKL)
  - Locking the whole kernel; being discouraged.
    - lock_kernel() / unlock_kernel() / kernel_locked()

Read-Copy Update

- Goal
  - A high performance and scaling algorithm for read-mostly situations
  - Reader must not required to acquire locks, execute atomic operations, or disable interrupts
- How
  - Writers create new versions atomically
    - Create new or delete old elements
  - Readers can access old versions independently of subsequent writers
  - Old versions are garbage-collected by poor man's GC, deferring destruction
  - Readers must signal "GC" when done

The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

Read-Copy Update

- Why
  - Readers are not permitted to block in read-side critical sections
  - Once an element is deleted, any CPU that subsequently performs a context switch cannot possibly gain a reference to this element
- Overhead might incurred
  - Readers incur little or no overhead (read_barrier_depends)
  - Writers incur substantial overhead
    - Writers must synchronize with each other
    - Writers must defer destructive actions until readers are done
    - The poor man's GC also incurs some overhead

The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

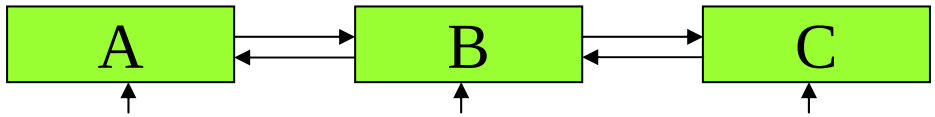
Read-Copy Update terms

- Quiescent state
  - Context switch is defined as the quiescent state
  - Quiescent state cannot appear in a read-side critical section
  - CPU in quiescent state are guaranteed to have completed all preceding read-side critical section
- Grace period
  - Any time period during which all CPUs pass through a quiescent state
  - A CPU may free up an element (destructive action) after a grace period has elapsed from the time that it deletes the element

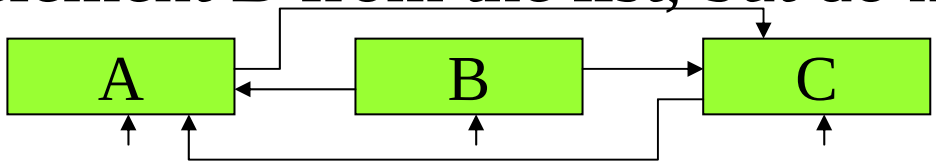
The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

Read-Copy Update example

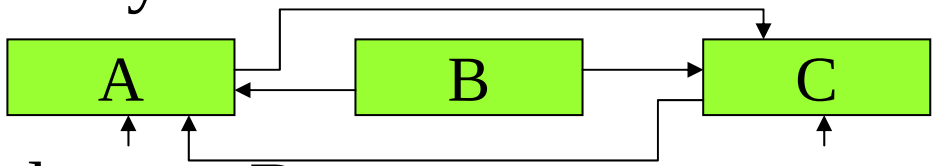
- Initial linked list



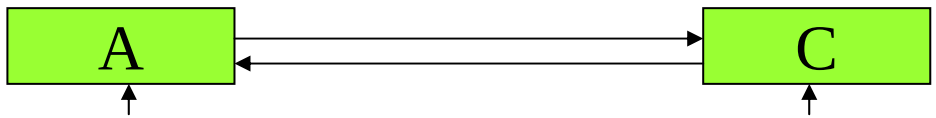
- Unlink element B from the list, but do not free it



- At this point, each CPU has performed one context switch after element B has been unlinked. Thus, there cannot be any more references to element B



- Free up element B



The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

Read-Copy Update primitives

- rcu_read_lock() / rcu_read_unlock()
  - Mark the begin and end of a read-side critical section
  - NULL on non-preemptive kernel; disable/enable preemption on preemptive kernel

```
#define rcu_read_lock()          preempt_disable()
#define rcu_read_unlock()        preempt_enable()
```

- synchronize_rcu()
  - Mark the end of updater code and the beginning of reclaimer code
  - Wait until all pre-existing RCU read-side critical sections complete
  - Subsequently started RCU read-side critical sections not waited for
- call_rcu(struct rcu_head *, void (*func)(struct rcu_head *))
  - Asynchronous form of synchronize_rcu()
  - Instead of blocking, it registers a callback function which are invoked after all ongoing RCU read-side critical sections have completed

Read-Copy Update primitives

- rcu_assign_pointer(p, v)
  - uses this function to assign a new value to an RCU-protected pointer
  - It returns the new value and executes any memory-barrier instructions required for a given CPU architecture

```
#define rcu_assign_pointer(p, v)   ({ smp_wmb(); (p) = (v); })
```

- rcu_dereference(p)
  - Protect an RCU-protected pointer for later safe-dereferencing; it executes any needed memory-barrier instructions for a given CPU architecture

```
#define rcu_dereference(p)    ({   typeof(p) _p1 = p; \
                                   smp_read_barrier_depends(); (_p1); })
```

RCU for the Linux *list* API

```
static inline void __list_add_rcu(struct list_head * new,
                struct list_head * prev, struct list_head * next)
{
        new->next = next;
        new->prev = prev;
        smp_wmb();
        next->prev = new;
        prev->next = new;
}

static inline void list_add_rcu(struct list_head *new, struct list_head *head)
{
        __list_add_rcu(new, head, head->next);
}

#define __list_for_each_rcu(pos, head) \
        for (pos = (head)->next; pos != (head); \
        pos = rcu_dereference(pos->next))
```

rwlock, seqlock and RCU

- All of these locks divide provide different interfaces for readers and writers
- RCU can be used only for algorithms that can tolerate concurrent accesses and updates
- For read-mostly situation
  - Use RCU if applicable; it avoids atomic operations (cache bouncing) and memory barrier* (pipeline stall) overhead
  - Use seqlock if applicable; it has memory barrier overhead
  - Do not use rwlock if read-side critical section is short

* True for all architectures except Alpha

Reader-writer lock versus RCU

- Mapping the primitives between rwlock and RCU

Reader-writer lock	**Read-Copy Update**
rwlock_t	spinlock_t
read_lock()	rcu_read_lock()
read_unlock()	rcu_read_unlock()
write_lock()	spin_lock()
write_unlock()	spin_unlock()

Final Remarks

- Design protection when you start everything
- Identify the sources of concurrency
  - Callback functions
  - Event and interrupt handlers
  - Instances of kernel threads
  - When will be blocked and put to sleep
  - SMP and preemptive kernel
- Lock goes with data structures not code segments
- Keep things simple when starting
- Use the right synchronization tool for the job

References

- Linux Kernel Development, 2nd edition, Robert Love, 2005
- Understanding the Linux Kernel, Bovet & Cesati, O'REILLY, 2002
- Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, 2005
- Linux 2.6.10 kernel source
- RCU papers by Paul E. McKenney, http://www.rdrop.com/users/paulmck/RCU/