Kernel Synchronization II

Xiaoguang Wang

UIC COMPUTER SCIENCE

Recap: updating a single variable

A single C statement

```
/* C code */
01: i++;
```

It can be translated into multiple machine instructions

```
/* Machine instructions */
01: get the current value of i and copy it into a register
02: add one to the value stored in the register
03: write back to memory the new value of i
```

Now, check what happens if two threads concurrently update `i`

Recap: atomic integer operations

```c
/* Type definition: include/linux/types.h */
typedef struct {
    int counter;
} atomic_t;

typedef struct {
    long counter;
} atomic64_t;

/* API definition: include/linux/atomic.h */
/* Usage example */
atomic_t v;                     /* define v */
atomic_t u = ATOMIC_INIT(0); /* define and initialize u to 0 */

atomic_set(&v, 4);              /* v = 4 (atomically) */
atomic_add(2, &v);              /* v = v + 2 == 6 (atomically) */
atomic_inc(&v);                 /* v = v + 1 == 7 (atomically) */
```

Recap: atomic int operations example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/types.h>

#define PRINT_PREF "[SYNC_ATOMIC] "
atomic_t counter; /* shared data: */
struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        atomic_inc(&counter);
        msleep(500);
    }
    do_exit(0);
}
```

Recap: spinlocks

```
/* include/linux/spinlock_types.h */
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* critical region */
spin_unlock(&my_lock);
```

Lock/unlock methods **disable/enable kernel preemption** and **acquire/release the lock**

Lock is compiled away on uniprocessor systems

- Still needs do disabled/re-enable preemption to prevent interleaving of task execution

`spin_lock()` is not recursive! → self-deadlock

Recap: spinlock usage example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#define PRINT_PREF "[SYNC_SPINLOCK] "

unsigned int counter;   /* shared data: */
DEFINE_SPINLOCK(counter_lock);
struct task_struct *read_thread, *write_thread;

static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        spin_lock(&counter_lock);
        counter++;
        spin_unlock(&counter_lock);
        msleep(500);
    }
    do_exit(0);
}
```

Recap: spinlocks in interrupt handlers

Conditional enabling/disabling local interrupt

```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;

/* Saves the current state of interrupts, disables them locally, and
then obtains the given lock */
spin_lock_irqsave(&mr_lock, flags);

/* critical region ... */

/* Unlocks the given lock and returns interrupts to their previous state */
spin_unlock_irqrestore(&mr_lock, flags);
```

Checklist for locking

- Is the data global?
- Can a thread of execution other than the current one access it?
- Is the data shared between process context and interrupt context?
- Is it shared between two different interrupt handlers?
- If a process is preempted while accessing this data, can the newly scheduled process access the same data?
- If the current process sleeps on anything, in what state does that leave any shared data?
- What happens if this function is called again on another processor?

Agenda for today

Reader-writer locks

Semaphore, mutex

Sequential lock (seqlock)

Completion variable

RCU

Reader-writer spinlock

Reader-writer spinlock (RWLock) allows multiple concurrent readers

- When entities accessing shared data can be clearly divided into readers and writers

Example: list updated (write) and searched (read)

- When updated, no other entity should update nor search
- When searched, no other entity should update
  - **Safe to allow multiple readers in parallel**
  - **Can improve scalability by allowing parallel readers**

Reader-write lock == shared-exclusive lock == concurrent-exclusive lock

Reader-writer spinlock

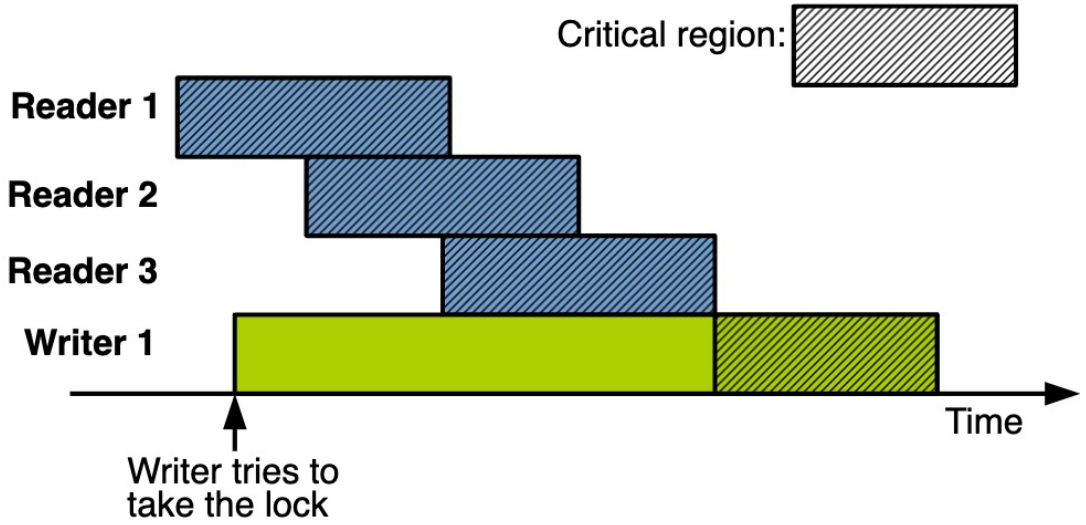
```
#include <linux/spinlock.h>

/* Define reader-writer spinlock */
DEFINE_RWLOCK(mr_rwlock);

read_lock(&mr_rwlock);          /* Reader */
/* critical section (read only) ... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);         /* Writer */
/* critical section (read and write) ... */
write_unlock(&mr_lock);

/* You cannot upgrade a read lock to a write lock.
 * Following code has a deadlock: */
read_lock(&mr_rwlock);
write_lock(&mr_lock);
```

Q: Who benefits in the rwlock? Reader? Writer?

Reader-writer spinlock

Linux reader-writer spinlocks **favor readers over writers**

- If the read lock is held and a writer waits for exclusive access, readers who attempt to acquire the lock continue succeeding.
- Therefore, a sufficient number of readers can starve pending writers.



Reader-writer spinlock API

Table 10.5 **Reader-Writer Spin Lock Methods**

Method	Description
`read_lock()`	Acquires given lock for reading
`read_lock_irq()`	Disables local interrupts and acquires given lock for reading
`read_lock_irqsave()`	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
`read_unlock()`	Releases given lock for reading
`read_unlock_irq()`	Releases given lock and enables local interrupts
`read_unlock_ irqrestore()`	Releases given lock and restores local interrupts to the given previous state
`write_lock()`	Acquires given lock for writing
`write_lock_irq()`	Disables local interrupts and acquires the given lock for writing
`write_lock_irqsave()`	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing

Reader-writer spinlock API

`write_unlock()`	Releases given lock
`write_unlock_irq()`	Releases given lock and enables local interrupts
`write_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state
`write_trylock()`	Tries to acquire given lock for writing; if unavailable, returns nonzero
`rwlock_init()`	Initializes given `rwlock_t`

Reader-writer spinlock: example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#define PRINT_PREF "[SYNC_RWSPINLOCK]: "
/* shared data: */
unsigned int counter;
DEFINE_RWLOCK(counter_lock);
struct task_struct *read_thread1, *read_thread2, *read_thread3, *write_thread;
static int writer_function(void *data) {
    while(!kthread_should_stop()) {
        write_lock(&counter_lock);
        counter++;
        write_unlock(&counter_lock);

        msleep(500);
    }
    do_exit(0);
}
```

Reader-writer spinlock: example

```
static int read_function(void *data) {
    while(!kthread_should_stop()) {
        read_lock(&counter_lock);
        printk(PRINT_PREF "counter: %d\n", counter);
        read_unlock(&counter_lock);

        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    read_thread1 = kthread_run(read_function, NULL, "read-thread1");
    read_thread2 = kthread_run(read_function, NULL, "read-thread2");
    read_thread3 = kthread_run(read_function, NULL, "read-thread3");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Semaphore

Sleeping locks → not usable in interrupt context

When a task attempts to acquire a semaphore that is unavailable, the semaphore **places the task onto a wait queue** and **puts the task to sleep**.

- → The processor is free to execute other code.

When a task releases the semaphore, one of the tasks on the wait queue is awakened so that it can then acquire the semaphore.

Semaphores are not optimal for locks held for short periods

- The overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.

Semaphore

Semaphores allow multiple holders

`counter` initialized to a given value

- Decremented each time a thread acquires the semaphore
- The semaphore becomes unavailable when the counter reaches 0

In the kernel, most of the semaphores used are **binary semaphores** (or mutex)

Semaphore: example

```
struct semaphore *sem1;

sem1 = kmalloc(sizeof(struct semaphore), GFP_KERNEL);
if(!sem1)
    return -1;

/* counter == 1: binary semaphore */
sema_init(&sema, 1);

down(sem1);
/* critical region */
up(sem1);
```

```
/* Binary semaphore static declaration */
DECLARE_MUTEX(sem2);

if(down_interruptible(&sem2)) {
    /* signal received, semaphore not acquired */
}

/* critical region */

up(sem2);
```

Semaphore API

Method	Description
`sema_init(struct semaphore *, int)`	Initializes the dynamically created semaphore to the given count
`init_MUTEX(struct semaphore *)`	Initializes the dynamically created semaphore with a count of one
`init_MUTEX_LOCKED(struct semaphore *)`	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
`down_interruptible (struct semaphore *)`	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
`down(struct semaphore *)`	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
`down_trylock(struct semaphore *)`	Tries to acquire the given semaphore and immediately return nonzero if it is contended
`up(struct semaphore *)`	Releases the given semaphore and wakes a waiting task, if any

Semaphore API

include/linux/semaphore.h

```
21 #define __SEMAPHORE_INITIALIZER(name, n)                                \
22 {                                                                       \
23         .lock           = __RAW_SPIN_LOCK_UNLOCKED((name).lock),       \
24         .count          = n,                                           \
25         .wait_list      = LIST_HEAD_INIT((name).wait_list),            \
26 }
27
28 #define DEFINE_SEMAPHORE(name)  \
29         struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)
30
31 static inline void sema_init(struct semaphore *sem, int val)
32 {
33         static struct lock_class_key __key;
34         *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
35         lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
36 }
```

Reader-writer semaphores

Reader-writer flavor of semaphore like reader-writer spinlock

```
#include <linux/rwsem.h>

/* declare reader-writer semaphore */
static DECLARE_RWSEM(mr_rwsem); /* or use init_rwsem(struct rw_semaphore *) */

/* attempt to acquire the semaphore for reading ... */
down_read(&mr_rwsem);
/* critical region (read only) ... */
/* release the semaphore */
up_read(&mr_rwsem);

/* ... */

/* attempt to acquire the semaphore for writing ... */
down_write(&mr_rwsem);
/* critical region (read and write) ... */
/* release the semaphore */
up_write(&mr_sem);
```

Reader-writer semaphores

`down_read_trylock()`, `down_write_trylock()`

- try to acquire read/write lock
- returns 1 if successful, 0 if contention

`downgrade_write()`

- atomically converts an acquired write lock to a read lock

Reader-writer semaphore: example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/rwsem.h>
#define PRINT_PREF "[SYNC_RWSEM] "

/* shared data: */
unsigned int counter;
struct rw_semaphore *counter_rwsemaphore;
struct task_struct *read_thread, *read_thread2, *write_thread;
```

Reader-writer semaphore: example

```
static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        down_write(counter_rwsemaphore);
        counter++;
        downgrade_write(counter_rwsemaphore);
        printk(PRINT_PREF "(writer) counter: %d\n", counter);
        up_read(counter_rwsemaphore);
        msleep(500);
    }
    do_exit(0);
}
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        down_read(counter_rwsemaphore);
        printk(PRINT_PREF "counter: %d\n", counter);
        up_read(counter_rwsemaphore);
        msleep(500);
    }

    do_exit(0);
}
```

Reader-writer semaphore: example

```
static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    counter_rwsemaphore = kmalloc(sizeof(struct rw_semaphore), GFP_KERNEL);
    if(!counter_rwsemaphore)
        return -1;

    init_rwsem(counter_rwsemaphore);

    read_thread = kthread_run(read_function, NULL, "read-thread");
    read_thread2 = kthread_run(read_function, NULL, "read-thread2");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Reader-writer semaphore: example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    kthread_stop(read_thread2);

    kfree(counter_rwsemaphore);

    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Mutex

Mutexes are binary semaphore with stricter use cases:

- Only one thread can hold the mutex at a time
- A thread locking a mutex must unlock it
- No recursive (nested) lock and unlock operations
- A thread cannot exit while holding a mutex
- A mutex cannot be acquired in interrupt context
- A mutex can be managed only through the API

Semaphore vs Mutex?

- Start with a mutex and move to a semaphore only if you have to

Mutex

```
#include <linux/mutex.h>

DEFINE_MUTEX(mut1); /* static */

struct mutex *mut2 = kmalloc(sizeof(struct mutex), GFP_KERNEL); /* dynamic */
if(!mut2)
    return -1;

mutex_init(mut2);

mutex_lock(&mut1);
/* critical region */
mutex_unlock(&mut1);
```

Mutex API

Method	Description
`mutex_lock(struct mutex *)`	Locks the given mutex; sleeps if the lock is unavailable
`mutex_unlock(struct mutex *)`	Unlocks the given mutex
`mutex_trylock(struct mutex *)`	Tries to acquire the given mutex; returns one if successful and the lock is acquired and zero otherwise
`mutex_is_locked (struct mutex *)`	Returns one if the lock is locked and zero otherwise

Mutex: usage example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/mutex.h>

#define PRINT_PREF "[SYNC_MUTEX]: "
/* shared data: */
unsigned int counter;
struct mutex *mut;
struct task_struct *read_thread, *write_thread;
```

Mutex: usage example

```
static int writer_function(void *data)
{
    while(!kthread_should_stop()) {
        mutex_lock(mut);
        kfree(mut);        /* !!! */
        counter++;
        mutex_unlock(mut);
        msleep(500);
    }
    do_exit(0);
}
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        mutex_lock(mut);
        printk(PRINT_PREF "counter: %d\n", counter);
        mutex_unlock(mut);
        msleep(500);
    }

    do_exit(0);
}
```

Spinlock vs Mutex

Requirement	Recommended lock
Low overhead locking	Spin lock is preferred
Short lock hold time	Spin lock is preferred
Long lock hold time	Mutex is preferred
Need to lock from interrupt context	Spin lock is required
Need to sleep while holding lock	Mutex is required

Quiz

Completion variables

Completion variables are used when one task needs to signal to the other that an event has occurred.

```
#include <linux/completion.h>

/* Declaration / initialization */
DECLARE_COMPLETION(comp1); /* static */
struct completion *comp2 = kmalloc(sizeof(struct completion), GFP_KERNEL);
if(!comp2)
    return -1;
init_completion(comp2);

/* Thread 1 */
/* signal event: */
complete(comp1);

/* Thread 2 */
/* wait for signal: */
wait_for_completion(comp1);
```

Completion variable APIs

Method	Description
`init_completion(struct completion *)`	Initializes the given dynamically created completion variable
`wait_for_completion(struct completion *)`	Waits for the given completion variable to be signaled
`complete(struct completion *)`	Signals any waiting tasks to wake up

Completion variable: example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/completion.h>

#define PRINT_PREF "[SYNC_COMP]: "

/* shared data: */
unsigned int counter;
struct completion *comp;
struct task_struct *read_thread, *write_thread;
```

Completion variable: example

```
static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;

    comp = kmalloc(sizeof(struct completion), GFP_KERNEL);
    if(!comp)
        return -1;

    init_completion(comp);

    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");

    return 0;
}
```

Completion variable: example

```
static int read_function(void *data)
{
    wait_for_completion(comp);
    printk(PRINT_PREF "counter: %d\n", counter);

    do_exit(0);
}
static int writer_function(void *data)
{
    while(counter != 1234)
        counter++;
    complete(comp);
    do_exit(0);
}
```

Sequential lock (seqlock)

A simple mechanism for reading and writing shared data

Works by maintaining a sequence counter (or version number)

- When **a write lock is obtained**, a sequence number is incremented.

Before and after reading the data, the sequence number is read. If the values are the same, a write did not begin in the middle of the read.

Further, if the values are even, a write is not underway.

- Grabbing the **write lock** makes the value **odd**, whereas releasing it makes it even because the lock starts at zero.

Sequential lock (seqlock)

```
/* define a seq lock */
seqlock_t my_seq_lock = DEFINE_SEQLOCK(my_seq_lock);

/* Write path */
write_seqlock(&my_seq_lock);
/* critical (write) region */
write_sequnlock(&my_seq_lock);

/* Read path */
unsigned long seq;
do {
    seq = read_seqbegin(&my_seq_lock);
    /* read data here ... */
} while(read_seqretry(&my_seq_lock, seq));
```

Seq locks are useful when

- There are many readers and few writers
- Writers should be favored over readers

Preemption disabling

When a spin lock is held preemption is disabled

Some situations need to disable preemption without involving spin locks

- Example: manipulating per-processor data:

```
task A manipulates per-processor variable foo, which is not protected by a lock
task A is preempted
task B is scheduled
task B manipulates variable foo
task B completes
task A is rescheduled
task A continues manipulating variable foo
```

Preemption disabling

Function	Description
`preempt_disable()`	Disables kernel preemption by incrementing the preemption counter
`preempt_enable()`	Decrements the preemption counter and checks and services any pending reschedules if the count is now zero
`preempt_enable_no_resched()`	Enables kernel preemption but does not check for any pending reschedules
`preempt_count()`	Returns the preemption count

Preemption disabling

For per-processor data

```
int cpu;
/* disable kernel preemption and set “cpu” to the current processor */
cpu = get_cpu();
/* manipulate per-processor data ... */
/* reenable kernel preemption, “cpu” can change and so is no longer valid */
put_cpu();
```

Ordering and barriers

Memory reads (load) and write (store) operations can be reordered for performance reasons

- by the compiler at compile time: compiler optimization
- by the CPU at run time: TSO (total store ordering), PSO (partial store ordering)

Ordering and barriers

```
/* Following code can be reordered
 * by a compiler (optimization) or a processor (out-of-order execution)
 *
 * Your code              Compiled code
 * ============           ============= */
   a = 4;                 b = 5;
   b = 5;                 a = 4;
```

Ordering and barriers

```
/* Following code can be reordered
 * by a compiler (optimization) or a processor (out-of-order execution)
 *
 * Your code                Compiled code
 * ============             ============= */
   a = 4;                   b = 5;
   b = 5;                   a = 4;

/* Following code will never be reorderd because
 * there is a dependency between a and b.
 *
 * Your code                Compiled code
 * ============             ============= */
   a = 1;                   a = 1;
   b = a;                   b = a;
```

Memory barriers

Instruct a compiler or a processor not to reorder instructions around a given point

`barrier()` (a.k.a., compiler barrier)

- Prevents the compiler from reordering stores or loads across the barrier

```
/* Compiler does not reorder store operations of a and b
 * However, a processor may reorder the store operations for performance */
a = 4;
barrier();
b = 5;
```

Memory barrier instructions

`rmb()`: prevents mem reads from being reordered across the barrier

`wmb()`: prevents mem writes from being reordered across the barrier

`mb()`: prevents both mem reads and writes from being reordered across the barrier

`read_barrier_depends()`: prevent data-dependent loads to be reordered across the barrier

- On some architectures, it is much faster than `rmb()` because it is not needed and is, thus, a **noop**

Memory barrier example

Initial value: a = 1 and b = 2

Thread 1	Thread 2
`a = 3;`	—
`mb();`	—
`b = 4;`	`c = b;`
—	`rmb();`
—	`d = a;`

`mb()` ensures that a is written before b

`rmb()` ensures that b is read before a