Kernel Synchronization II

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
  - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization: concepts

Today: kernel synchronization II

- Atomic operations
- Spinlock, reader-writer spinlock (RWLock)
- Semaphore, mutex
- Sequential lock (seqlock)
- Completion variable

Atomic operations

- Provide instructions that execute *atomically without interruption*
- Non-atomic update: `i++`

Thread 1	Thread 2
`get i` (7)	`get i` (7)
`increment i` (7 -> 8)	—
—	`increment i` (7 -> 8)
`write back i` (8)	—
—	`write back i` (8)

Atomic operations

- Atomic update: `atomic_inc(&i)`

Thread 1	Thread 2
`increment & store i` (7 -> 8)	—
—	`increment & store i` (8 -> 9)

Or conversely

Thread 1	Thread 2
—	`increment & store` (7 -> 8)
`increment & store` (8 -> 9)	—

Atomic operations

- Examples
  - `fetch-and-add`: atomic increment
  - `test-and-set`: set a value at a memory location and return the previous value
  - `compare-and-swap`: modify the content of a memory location only if the previous content is equal to a given value
- Linux provides two APIs:
  - Integer atomic operations
  - Bitwise atomic operations

Atomic integer operations

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

typedef struct {
    long counter;
} atomic64_t;

/* API definition: linux/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) */
```

Atomic integer operations (32-bit)

Atomic integer operations (32-bit)

`int atomic_add_negative(int i, atomic_t *v)`	Atomically add `i` to `v` and return true if the result is negative; otherwise false.
`int atomic_add_return(int i, atomic_t *v)`	Atomically add `i` to `v` and return the result.
`int atomic_sub_return(int i, atomic_t *v)`	Atomically subtract `i` from `v` and return the result.
`int atomic_inc_return(int i, atomic_t *v)`	Atomically increment `v` by one and return the result.
`int atomic_dec_return(int i, atomic_t *v)`	Atomically decrement `v` by one and return the result.
`int atomic_dec_and_test(atomic_t *v)`	Atomically decrement `v` by one and return true if zero; false otherwise.
`int atomic_inc_and_test(atomic_t *v)`	Atomically increment `v` by one and return true if the result is zero; false otherwise.

Atomic integer operations (64-bit)

Atomic Integer Operation	Description
`ATOMIC64_INIT(long i)`	At declaration, initialize to `i`.
`long atomic64_read(atomic64_t *v)`	Atomically read the integer value of `v`.
`void atomic64_set(atomic64_t *v, int i)`	Atomically set `v` equal to `i`.
`void atomic64_add(int i, atomic64_t *v)`	Atomically add `i` to `v`.
`void atomic64_sub(int i, atomic64_t *v)`	Atomically subtract `i` from `v`.
`void atomic64_inc(atomic64_t *v)`	Atomically add one to `v`.
`void atomic64_dec(atomic64_t *v)`	Atomically subtract one from `v`.
`int atomic64_sub_and_test(int i, atomic64_t *v)`	Atomically subtract `i` from `v` and return true if the result is zero; otherwise false.
`int atomic64_add_negative(int i, atomic64_t *v)`	Atomically add `i` to `v` and return true if the result is negative; otherwise false.

Atomic integer operations (64-bit)

`long atomic64_add_return(int i, atomic64_t *v)`	Atomically add `i` to `v` and return the result.
`long atomic64_sub_return(int i, atomic64_t *v)`	Atomically subtract `i` from `v` and return the result.
`long atomic64_inc_return(int i, atomic64_t *v)`	Atomically increment `v` by one and return the result.
`long atomic64_dec_return(int i, atomic64_t *v)`	Atomically decrement `v` by one and return the result.
`int atomic64_dec_and_test(atomic64_t *v)`	Atomically decrement `v` by one and return true if zero; false otherwise.
`int atomic64_inc_and_test(atomic64_t *v)`	Atomically increment `v` by one and return true if the result is zero; false otherwise.

Atomic integer operations: usage 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/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);
}
```

Atomic integer operations: usage example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        printk(PRINT_PREF "counter: %d\n", atomic_read(&counter));
        msleep(500);
    }
    do_exit(0);
}

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

    atomic_set(&counter, 0);

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

    return 0;
}
```

Atomic integer operations: usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Atomic bitwise operations

```
/* API definition: include/linux/bitops.h */

/* Usage example */
unsigned long word = 0; /* 32 / 64 bits according to the system */

set_bit(0, &word);       /* bit zero is set atomically */
set_bit(1, &word);       /* bit one is set atomically */
printk("&ul\n", word);   /* print "3" */
clear_bit(1, &word);     /* bit one is unset atomically */
change_bit(0, &word);    /* flip bit zero atomically (now unset) */

/* set bit zero and return its previous value (atomically) */
if(test_and_set_bit(0, &word)) {
    /* not true in the case of our example */
}

/* you can mix atomic bit operations and normal C */
word = 7;
```

Atomic bitwise operations

Atomic Bitwise Operation	Description
`void set_bit(int nr, void *addr)`	Atomically set the `nr`-*th* bit starting from `addr`.
`void clear_bit(int nr, void *addr)`	Atomically clear the `nr`-*th* bit starting from `addr`.
`void change_bit(int nr, void *addr)`	Atomically flip the value of the `nr`-*th* bit starting from `addr`.
`int test_and_set_bit(int nr, void *addr)`	Atomically set the `nr`-*th* bit starting from `addr` and return the previous value.
`int test_and_clear_bit(int nr, void *addr)`	Atomically clear the `nr`-*th* bit starting from `addr` and return the previous value.
`int test_and_change_bit(int nr, void *addr)`	Atomically flip the `nr`-*th* bit starting from `addr` and return the previous value.
`int test_bit(int nr, void *addr)`	Atomically return the value of the `nr`-*th* bit starting from `addr`.

Atomic bitwise operations

- Non-atomic bitwise operations are also provided
  - prefixed with double underscore
  - Example: `test_bit()` vs `__test_bit()`
- If you do not require atomicity (say, for example, because a lock already protects your data), these variants of the bitwise functions might be faster.

Spinlocks

- The most common lock used in the kernel
- When a thread tries to acquire an already held lock, it spins while waiting for the lock become available.
    - Wasting processor time when spinning is too long
    - Spinlocks can be used in interrupt context, which a thread cannot sleep → **Kernel provides special spinlock API for data structures shared in interrupt context**
- In process context, do not sleep while holding a spinlock
    - Kernel preemption is disabled

Spinlocks: usage

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

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

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

Quiz #1: find a deadlock

```
/* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global_hashtbl */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock(&hashtbl_lock);
    /* What happen if an interrupt occurs
     * while a task executing here? -> Deadlock */
    spin_unlock(&hashtbl_lock);
}
```

Quiz #2: find a deadlock

```
/* WARNING!!! THIS CODE HAS A DEADLOCK!!! WARNING!!! */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler_1(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global_hashtbl */
    spin_unlock(&hashtbl_lock);
}

irqreturn_t irq_handler_2(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* What happen if an interrupt 1 occurs
     * while executing here? -> Deadlock */
    spin_unlock(&hashtbl_lock);
}
```

Spinlocks: usage in interrupt handlers

- Spin locks do not sleep so it is safe to use them in interrupt context
- **If a lock is used in an interrupt handler, you must also disable local interrupts before obtaining the lock.**
- Otherwise, it is possible for an interrupt handler to interrupt kernel code while the lock is held and attempt to reacquire the lock.
- The interrupt handler spins, waiting for the lock to become available. The lock holder, however, does not run until the interrupt handler completes.

  → **double-acquire deadlock**

Spinlocks: usage 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);
```

Spinlocks: usage in interrupt handlers

- Unconditional enabling/disabling local interrupt
  - If you always know before the fact that interrupts are initially enabled, there is no need to restore their previous state.

```
DEFINE_SPINLOCK(mr_lock);

/* Disable local interrupt and acquire lock */
spin_lock_irq(&mr_lock);

/* critical section ... */

/* Unlocks the given lock and enable local interrupt */
spin_unlock_irq(&mr_lock);
```

- Let’s check the code

Bug fix for usage #1

```
/* NOTE: BUG-FIXED VERSION OF USAGE #1 */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* It is okay not to disable interrupt here
     * because this is the only interrupt handler access
     * the shared data and this particular interrupt is
     * already disabled. */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    unsigned long flags;
    spin_lock_irqsave(&hashtbl_lock, flags);
    /* Interrupt is disabled here */
    spin_unlock_irqrestor(&hashtbl_lock, flags);
}
```

Bug fix for usage #2

```
/* NOTE: BUG-FIXED VERSION OF USAGE #2 */
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler_1(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock_irq(&hashtbl_lock);
    /* Need to disable interrupt here
     * to prevent irq_handler_1 from accessing the shared data */
    spin_unlock_irq(&hashtbl_lock);
}

irqreturn_t irq_handler_2(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock_irq(&hashtbl_lock);
    /* Need to disable interrupt here
     * to prevent irq_handler_1 from accessing the shared data */
    spin_unlock_irq(&hashtbl_lock);
}
```

Spinlock API

Table 10.4 **Spin Lock Methods**

Method	Description
`spin_lock()`	Acquires given lock
`spin_lock_irq()`	Disables local interrupts and acquires given lock
`spin_lock_irqsave()`	Saves current state of local interrupts, disables local interrupts, and acquires given lock
`spin_unlock()`	Releases given lock
`spin_unlock_irq()`	Releases given lock and enables local interrupts
`spin_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state
`spin_lock_init()`	Dynamically initializes given `spinlock_t`
`spin_trylock()`	Tries to acquire given lock; if unavailable, returns nonzero
`spin_is_locked()`	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Spinlocks and bottom halves

- `spin_lock_bh()` / `spin_unlock_bh()`
  - Obtains the given lock and disables all bottom halves
- Because a bottom half might preempt process context code, if data is shared between a bottom-half process context, you must protect the data in process context with both a lock and the disabling of bottom halves.
- Likewise, because an interrupt handler might preempt a bottom half, if data is shared between an interrupt handler and a bottom half, you must both obtain the appropriate lock and disable interrupts.

Quiz #3: interrupt context?

- Top-half: Interrupt handler
- Bottom-half: Softirq, Tasklet
- KProbe handler, timer handler
- Any handler
    - Ask whether it runs in interrupt context
    - If so ask which interrupts are disabled

Spinlock usage 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_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);
}
```

Spinlock usage example

```
static int read_function(void *data)
{
    while(!kthread_should_stop()) {
        spin_lock(&counter_lock);
        printk(PRINT_PREF "counter: %d\n", counter);
        spin_unlock(&counter_lock);
        msleep(500);
    }
    do_exit(0);
}

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

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

    return 0;
}
```

Spinlock usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Reader-writer spinlock (RWLock)

- Reader-writer spinlock allows multiple concurrent readers
- When entities accessing a 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 (RWLock)

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

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

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

/* Writer */
write_lock(&mr_rwlock);
/* 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); /* It will wait forever until there is no reader */
```

Reader-writer spinlock (RWLock)

- Linux reader-writer spinlocks favor readers over writers
  - If the read lock is held and a writer is waiting for exclusive access, readers that attempt to acquire the lock continue to succeed.
  - 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: usage 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: usage 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;
}
```

Reader-writer spinlock: usage example

```
static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread3);
    kthread_stop(read_thread2);
    kthread_stop(read_thread1);
    kthread_stop(write_thread);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

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 then 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 that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.

Semaphore

- Semaphores allow multiples holders
- `counter` initialized to a given value
  - Decremented each time a thread acquire 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: usage 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

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: usage 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: usage 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: usage 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: usage 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 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 trhough 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

```
#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);
        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);
}
```

Mutex: usage example

```
static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");
    counter = 0;
    mut = kmalloc(sizeof(struct mutex), GFP_KERNEL);
    if(!mut)
        return -1;
    mutex_init(mut);
    read_thread = kthread_run(read_function, NULL, "read-thread");
    write_thread = kthread_run(writer_function, NULL, "write-thread");
    return 0;
}

static void __exit my_mod_exit(void)
{
    kthread_stop(read_thread);
    kthread_stop(write_thread);
    kfree(mut);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
MODULE_LICENSE("GPL");
```

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

Completion variable

- 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 API

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: usage 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/completion.h>

#define PRINT_PREF "[SYNC_COMP]: "

/* shared data: */
unsigned int counter;
struct completion *comp;

struct task_struct *read_thread, *write_thread;

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

Completion variable: usage example

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

    do_exit(0);
}

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: usage example

```
static void __exit my_mod_exit(void)
{
    kfree(comp);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

Sequential lock (seqlock)

- A simple mechanism for reading and writing shared data
- Works by maintaining a sequence counter (or version number)
- Whenever the data in question is written to, a lock is obtained and a sequence number is incremented.
- Prior to 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();
```

Next lecture

- Kernel synchronization III
  - Memory ordering
  - Read-Copy-Update (RCU)