

Linux Kernel Programming **Kernel Synchronization**

Pierre Olivier

Systems Software Research Group @ Virginia Tech

March 21, 2017





Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Introduction

Critical regions and race conditions

- ▶ The kernel is programmed using the shared memory model
 - ▶ **Shared data must be protected against concurrent access**
 - ▶ Interruption/preemption on a single core
 - ▶ Pure concurrent access on a multi-core CPU (SMP)
- ▶ **Critical region/section:** part of the code manipulating shared data
 - ▶ Must execute *atomically*, i.e. without interruption
 - ▶ Should not be executed in parallel on SMP
- ▶ **Race condition:** two threads concurrently executing the same critical region
 - ▶ It's a bug!

Introduction

Critical regions and race conditions: why protecting shared data?

► ATM example:

```
1 int total = get_total_from_account(); /* total funds in user account */
2 int withdrawal = get_withdrawal_amount(); /* amount user asked to withdrawal */
3
4 /* check whether the user has enough funds in her account */
5 if(total < withdrawal) {
6     error("Not enough money!");
7     return -1;
8 }
9
10 /* The user has enough money, deduct the withdrawal amount from here total */
11 total -= withdrawal;
12 update_total_funds(total);
13
14 /* give the money to the user */
15 spit_out_money(withdrawal);
```

Introduction

Critical regions and race conditions (2)

```
1 int total =
2     get_total_from_account();
3 int withdrawal =
4     get_withdrawal_amount();
5
6 if(total < withdrawal) {
7     error("Not enough money!");
8     return -1;
9 }
10 total -= withdrawal;
11 update_total_funds(total);
12 spit_out_money(withdrawal);
```

- ▶ Assume two transactions are happening nearly at the same time
 - ▶ Ex: shared credit card account
- ▶ Assume
 - total == 105,
 - withdrawal1 == 100,
 - withdrawal2 == 10
- ▶ Should fail as !(100+10 > 105)

Introduction

Critical regions and race conditions (3)

```
1 int total =
2     get_total_from_account();
3 int withdrawal =
4     get_withdrawal_amount();
5
6 if(total < withdrawal) {
7     error("Not enough money!");
8     return -1;
9 }
10 total -= withdrawal;
11 update_total_funds(total);
12 spit_out_money(withdrawal);
```

► Assume:

total == 105, withdrawal1 == 100,
withdrawal2 == 10

► Possible scenario:

- ① Threads check that $100 < 105$ and
 $10 < 105$

► All good

- ② Thread 1 updates

total = 105 - 100 = 5

- ③ Thread 2 updates

total = 105 - 10 = 95

► Total withdrawal: 110, and there is 95 left on the account!

Introduction

Critical regions and race conditions: single variable example

- ▶ Consider this C instruction: `i++;`
 - ▶ It might translates into machine code as:

```
1 get the current value of i and copy it into a register  
2 add one to the value stored into the register  
3 write back to memory the new value of i
```

- ▶ Assume `i == 7` is shared between two threads, both wanting to increment it:

Thread 1	Thread 2
get i (7)	-
increment i ($7 \rightarrow 8$)	-
write back i (8)	-
-	get i (8)
-	increment i ($8 \rightarrow 9$)
-	write back i (9)

Introduction

Critical regions and race conditions: single variable example (2)

- ▶ Race condition:

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)

Introduction

Critical regions and race conditions: single variable example (3)

- ▶ A solution is to use **atomic instructions**
 - ▶ Instructions provided by the CPU that cannot interleave
 - ▶ ex: *increment and store*

Thread 1

increment & store i ($7 \rightarrow 8$)
-

Thread 2

-
increment and store i ($8 \rightarrow 9$)

- ▶ Or:

Thread 1

-
increment & store i ($8 \rightarrow 9$)

Thread 2

increment and store i ($7 \rightarrow 8$)
-

Introduction

Locking

- ▶ Atomic operations are not sufficient for protecting shared data in long and complex critical regions
 - ▶ Example: a shared stack (data structure) with multiple pushing and popping threads
- ▶ Need a mechanism to assure **a critical region is executed atomically by only one core at the same time → locks.**

Introduction

Locking (2)

- ▶ Example - stack protected by a lock:

Thread 1	Thread 2
try to lock the stack	try to lock the stack
success: lock acquired	failure: waiting...
access stack...	waiting...
unlock the stack...	waiting ...
...	success: lock acquired
...	access stack
...	unlock the stack

- ▶ Locking is implemented by the programmer *voluntarily*
 - ▶ No indication from the compiler!
 - ▶ No protection generally ends up in data corruption → inconsistent behavior for the program → **difficult to debug and trace back the source of the issue**
- ▶ Locking/unlocking primitives are implemented through atomic operations

Introduction

Causes of concurrency

- ▶ From a single core standpoint: **interleaving asynchronous execution threads**
 - ▶ For example preemption or interrupts
 - ▶ **pseudo-concurrency**
- ▶ On a multi-core: **true concurrency**
- ▶ **Sources of concurrency in the kernel:**
 - ① Interrupts
 - ② Softirqs and tasklets
 - ③ Kernel preemption
 - ④ Sleeping and synchronization
 - ⑤ Symmetrical multiprocessing
 - ▶ Need to understand and prepare for these: **identifying shared data and related critical regions**
 - ▶ Needs to be done from the start as **concurrency bugs are difficult to detect and solve**

Introduction

Causes of concurrency (2)

- ▶ Naming:
 - ▶ Code safe from access from an interrupt handler: **interrupt-safe**
 - ▶ *This code can be interrupted by an interrupt handler and this will not cause any issue*
 - ▶ Code safe from access from multiple cores: **SMP-safe**
 - ▶ *This code can be executed on multiple cores at the same time without issue*
 - ▶ Code safe from concurrency with kernel preemption: **preempt-safe**
 - ▶ *This code can be preempted without issue*

Introduction

What to protect?

- ▶ When writing some code, **observe the data manipulated by the code**
 - ▶ *If anyone else (thread/handler) can see it, lock it*
- ▶ Questions to ask when writing kernel code:
 - ▶ Is the data global?
 - ▶ Is the data shared between process and interrupt context?
 - ▶ If the process is preempted while accessing the data, can the newly scheduled process access the same data?
 - ▶ Can the code blocks on anything? If so, in what state does that leave any shared data?
 - ▶ What prevents the data from being freed out from under me?
 - ▶ What happens if this function is called again on another core?

Introduction

Deadlocks

► Deadlock

- ▶ Situations in which one or several threads are waiting on locks for one or several resources that will never be freed → they are stuck
 - ▶ Real-life example: traffic deadlock
 - ▶ Self-deadlock (1 thread):

```
1 acquire lock
2 acquire lock again
3 waiting indefinitely ...
```

- ▶ Deadly embrace (n threads and n locks):

Thread 1

acquire lock A
try to acquire lock B
wait for lock B...

Thread 2

acquire lock B
try to acquire lock A
wait for lock A...

Introduction

Deadlocks: how to prevent them

- ▶ **Implement lock ordering**

- ▶ Nested lock must always be obtained in the same order
- ▶ Document lock usage in comments

Thread 1	Thread 2
acquire lock cat	acquire lock fox
acquire lock dog	try to acquire lock dog
try to acquire lock fox	wait for lock dog...
wait for lock fox...	wait for lock dog...

- ▶ **Do not double-acquire the same lock**

Introduction

Contention and scalability

- ▶ A lock is said to be **contented** when there are often threads waiting for it
- ▶ A highly contented lock can become a bottleneck for the system performance
- ▶ **Coarse vs fine-grained locking**
 - ▶ Coarse lock example: protecting an entire subsystem' shared data structures
 - ▶ Bottleneck on high-core count machines
 - ▶ Fine-grained locks:
 - ▶ Overhead on low-core count machines
- ▶ Start simple and grow in complexity if needed

Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Atomic operations

- ▶ **Atomic operations** perform (simple) operations in memory and either succeed or fail in their entirety
 - ▶ Regardless of what operations are executed on other cores
 - ▶ Without interruption
- ▶ Examples:
 - ▶ Atomic increment (*fetch-and-add*)
 - ▶ Set a value at a memory location and return the previous value (*test-and-set*)
 - ▶ Modify the content of a memory location only if the previous content is equal to a given value (*compare-and-swap*)
- ▶ Linux provides two APIs:
 - ▶ Integers atomic operations
 - ▶ Bitwise atomic operations

Atomic operations

Atomic integer operations

- ▶ includes/linux/types.h:

```
1 typedef struct {  
2     int counter;  
3 } atomic_t;
```

- ▶ API defined in includes/asm/atomic.h

- ▶ Usage:

```
1 atomic_t v;                      /* define v */  
2 atomic_t u = ATOMIC_INIT(0);      /* define and initialize u to 0 */  
3  
4 atomic_set(&v, 4); /* v = 4 (atomically) */  
5 atomic_add(2, &v); /* v = v + 2 == 6 (atomically) */  
6 atomic_inc(&v);   /* v = v + 1 == 7 (atomically) */
```

Atomic operations

Atomic integer operations (2)

► API:

Atomic integer operation	Description
ATOMIC_INIT(int i)	Declare and initialize to i
int atomic_read(atomic_t *v)	Atomically read the value of v
void atomic_set(atomic_t *v, int i)	Atomically set v to i
void atomic_add(int i, atomic_t *v)	Atomically add i to v
void atomic_sub(int i, atomic_t *v)	Atomically subtract i from v
void atomic_inc(atomic_t *v)	Atomically add 1 to v
void atomic_dec(atomic_t *v)	Atomically subtract 1 from v
int atomic_sub_and_test(int i, atomic_t *v)	Atomically subtract i from v and return true if the result is zero, otherwise false
int atomic_add_negative(int i, atomic_t *v)	Atomically add i to v and return true if the result is negative, otherwise false

Atomic operations

Atomic integer operations (3)

► API (continued):

Atomic integer operation	Description
int atomic_add_return(int i, atomic_t *v)	Atomically add <i>i</i> to <i>v</i> and return the result
int atomic_sub_return(int i, atomic_t *v)	Atomically subtract <i>i</i> from <i>v</i> and return the result
int atomic_inc_return(atomic_t *v)	Atomically increment <i>v</i> by 1 and return the result
int atomic_dec_return(atomic_t *v)	Atomically decrement <i>v</i> by 1 and return the result
int atomic_dec_and_test(atomic_t *v)	Atomically decrement <i>v</i> by 1 and return true if the result is zero, false otherwise
int atomic_inc_and_test(atomic_t *v)	Atomically increment <i>v</i> by 1 and return true if the result is zero, false otherwise



Atomic operations

Atomic integer operations: 64-bits atomic operations

```
1 typedef struct {
2     volatile long counter;
3 } atomic64_t;
```

Atomic integer operation	Description
ATOMIC_INIT(int i)	Declare and initialize to i
int atomic64_read(atomic64_t *v)	Atomically read the value of v
void atomic64_set(atomic64_t *v, int i)	Atomically set v 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 1 to v
void atomic64_dec(atomic64_t *v)	Atomically subtract 1 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 operations

Atomic integer operations: 64-bits atomic operations (2)

Atomic integer operation	Description
<code>int atomic64_add_return(int i, atomic64_t *v)</code>	Atomically add <code>i</code> to <code>v</code> and return the result
<code>int atomic64_sub_return(int i, atomic64_t *v)</code>	Atomically subtract <code>i</code> from <code>v</code> and return the result
<code>int atomic64_inc_return(atomic64_t *v)</code>	Atomically increment <code>v</code> by 1 and return the result
<code>int atomic64_dec_return(atomic64_t *v)</code>	Atomically decrement <code>v</code> by 1 and return the result
<code>int atomic64_dec_and_test(atomic64_t *v)</code>	Atomically decrement <code>v</code> by 1 and return true if the result is zero, false otherwise
<code>int atomic64_inc_and_test(atomic64_t *v)</code>	Atomically increment <code>v</code> by 1 and return true if the result is zero, false otherwise

Atomic operations

Atomic integer operations: usage example

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/delay.h>
6 #include <linux/kthread.h>
7 #include <linux/sched.h>
8 #include <linux/types.h>
9
10 #define PRINT_PREF "[SYNC_ATOMIC] "
11 atomic_t counter; /* shared data: */
12 struct task_struct *read_thread, *
13     write_thread;
14
15 static int writer_function(void *data)
16 {
17     while(!kthread_should_stop()) {
18         atomic_inc(&counter);
19         msleep(500);
20     }
21     do_exit(0);
22 }
```

```

22 static int read_function(void *data)
23 {
24     while(!kthread_should_stop()) {
25
26         printk(PRINT_PREF "counter: %d\n",
27                atomic_read(&counter));
28         msleep(500);
29     }
30     do_exit(0);
31 }
32
33 static int __init my_mod_init(void)
34 {
35     printk(PRINT_PREF "Entering module.\n");
36
37     atomic_set(&counter, 0);
38
39     read_thread = kthread_run(read_function,
40                               NULL, "read-thread");
41     write_thread = kthread_run(
42                     writer_function, NULL, "write-thread");
43
44     return 0;
45 }
```

Atomic operations

Atomic integer operations: usage example(2)

```
43 static void __exit my_mod_exit(void)
44 {
45     kthread_stop(read_thread);
46     kthread_stop(write_thread);
47     printk(KERN_INFO "Exiting module.\n");
48 }
49
50 module_init(my_mod_init);
51 module_exit(my_mod_exit);
52
53 MODULE_LICENSE("GPL");
```

Atomic operations

Atomic bitwise operations

► Atomic **bitwise operations** (include/linux/bitops.h)

```
1 unsigned long word = 0; /* 32 / 64 bits according to the system */
2
3 set_bit(0, &word); /* bit zero is set atomically */
4 set_bit(1, &word); /* bit one is set atomically */
5 printk("%ul\n", word); /* print "3" */
6 clear_bit(1, &word); /* bit one is unset atomically */
7 change_bit(0, &word); /* flip bit zero atomically (now unset) */
8
9 /* set bit zero and return its previous value (atomically) */
10 if(test_and_set_bit(0, &word)) {
11     /* not true in the case of our example */
12 }
13
14 /* you can mix atomic bit operations and normal C */
15 word = 7;
```

- API function operate on generic pointers (void *)
- Example with `long` on 32-bits systems:
 - Bit 31 is the most significant bit
 - Bit 0 is the least significant bit

Atomic operations

Atomic bitwise operations: API

► API:

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 nr-th bit starting from addr
void 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

Atomic operations

Atomic bitwise operations: API(2)

- ▶ **API (continued):**

Atomic bitwise operation	Description
<code>int test_and_change_bit(int nr, void *addr)</code>	Atomically flip the <code>nr</code> -th bit starting from <code>addr</code> and return the previous value
<code>int test_bit(int nr, void *addr)</code>	Atomically return the value of the <code>nr</code> -th bit starting from <code>addr</code>

- ▶ Non-atomic bitwise operations (can be slightly faster according to the architecture) , prefixed with '`_`'
 - ▶ Example: `_test_bit()`

Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Spin locks

Presentation

- ▶ The most common lock used in the kernel: **spin lock**
- ▶ Can be **held by at most one thread of execution**
- ▶ When a thread tries to acquire an already held lock:
 - ▶ **Active waiting (spinning)**
 - ▶ Hurts performance when spinning for too long
 - ▶ However spinlocks are needed in context where one cannot sleep (interrupt)
 - ▶ As opposed to putting the thread to sleep (semaphores/mutexes)
- ▶ **In process context, do not sleep while holding a spinlock**
 - ▶ Another thread trying to acquire the spinlock hangs the CPU, preventing you to wake up
 - ▶ Deadlock

Spin locks

Usage

- ▶ **Usage:** (API in `include/linux/spinlock.h`)

```
1 DEFINE_SPINLOCK(my_lock);  
2  
3 spin_lock(&my_lock);  
4 /* critical region */  
5 spin_unlock(&my_lock);
```

- ▶ Lock/unlock methods disable/enable kernel preemption and acquire/release the lock
- ▶ `spin_lock()` is not recursive!
 - ▶ A thread calling `spin_lock()` twice on the same lock self-deadlocks
- ▶ Lock is compiled away on uniprocessor systems
 - ▶ Still needs do disabled/re-enable preemption

Spin locks

Usage: interrupt handlers

- ▶ Spin locks do not sleep: it is safe to use them in interrupt context
- ▶ **In an interrupt handler, need to disable local interrupts before taking the lock!**
 - ▶ Otherwise, risk of deadlock if interrupted by another handler accessing the same lock

```
1 DEFINE_SPINLOCK(my_lock); /* the spin lock */  
2 unsigned long flags; /* to save the interrupt state */  
3  
4 spin_lock_irqsave(&my_lock, flags);  
5 /* critical region */  
6 spin_unlock_irqrestore(&my_lock, flags);
```

- ▶ If it is known that interrupts are initially enabled:

```
1 spin_lock_irq(&my_lock);  
2 /* critical region */  
3 spin_unlock_irq(&my_lock);
```

- ▶ Also true from process context sharing data with interrupt handler
- ▶ **Debugging spin locks:** CONFIG_DEBUG_SPINLOCKS [2],
CONFIG_DEBUG_LOCK_ALLOC [1]

Spin locks

Other spin locks methods

Method	Description
<code>spin_lock()</code>	Acquires a lock
<code>spin_lock_irq()</code>	Disable local interrupts and acquire a lock
<code>spin_lock_irqsave()</code>	Save current state of local interrupts, disables local interrupts, and acquire a lock
<code>spin_unlock()</code>	Release a lock
<code>spin_unlock_irq()</code>	Release a lock and enable local interrupts
<code>spin_unlock_irqrestore()</code>	Release a lock and reset interrupts to previous state
<code>spin_lock_init()</code>	Dynamically initialize a <code>spinlock_t</code>
<code>spin_trylock()</code>	Try to acquire a lock and directly returns 0 if unavailable
<code>spin_is_locked()</code>	Return nonzero if the lock is currently acquired, otherwise return 0

Spin locks

Spin locks and bottom halves

- ▶ `spin_lock_bh()`/`spin_unlock_bh()`:
 - ▶ Disable softirqs (and thus tasklets) before taking the lock
- ▶ In process context:
 - ▶ Data shared with bottom-half context?
 - ▶ Disable bottom-halves + lock
 - ▶ Data shared with interrupt handler?
 - ▶ Disable interrupts + lock

Spin locks

Usage example

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/delay.h>
6 #include <linux/spinlock.h>
7 #include <linux/kthread.h>
8 #include <linux/sched.h>
9
10 #define PRINT_PREF "[SYNC_SPINLOCK] "
11
12 unsigned int counter; /* shared data: */
13 DEFINE_SPINLOCK(counter_lock);
14 struct task_struct *read_thread, *
15     write_thread;
16
17 static int writer_function(void *data)
18 {
19     while(!kthread_should_stop()) {
20         spin_lock(&counter_lock);
21         counter++;
22         spin_unlock(&counter_lock);
23         msleep(500);
24     }
25 }
```

```
26 static int read_function(void *data)
27 {
28     while(!kthread_should_stop()) {
29         spin_lock(&counter_lock);
30         printk(PRINT_PREF "counter: %d\n",
31                counter);
32         spin_unlock(&counter_lock);
33         msleep(500);
34     }
35     do_exit(0);
36 }
37 static int __init my_mod_init(void)
38 {
39     printk(PRINT_PREF "Entering module.\n");
40     counter = 0;
41
42     read_thread = kthread_run(read_function,
43                               NULL, "read-thread");
43     write_thread = kthread_run(
44         writer_function, NULL, "write-thread");
45
46 }
```

Spin locks

Usage example (2)

```
47 static void __exit my_mod_exit(void)
48 {
49     kthread_stop(read_thread);
50     kthread_stop(write_thread);
51     printk(KERN_INFO "Exiting module.\n");
52 }
53
54 module_init(my_mod_init);
55 module_exit(my_mod_exit);
56
57 MODULE_LICENSE("GPL");
```

Spin locks

Reader-writer spin locks

- ▶ 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**

```
1 DEFINE_RWLOCK(my_rwlock); /* declaration & initialization */
```

▶ Reader code:

```
1 read_lock(&my_rwlock);
2 /* critical region */
3 read_unlock(&my_rwlock);
```

▶ Writer code:

```
1 write_lock(&my_rwlock);
2 /* critical region */
3 write_unlock(&my_rwlock);
```

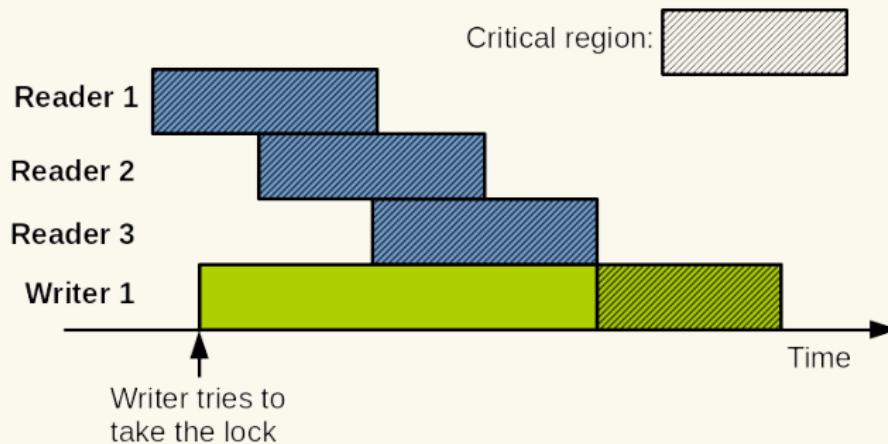
Spin locks

Reader-writer spin locks (2)

- Deadlock:

```
1 read_lock(&my_rwlock);  
2 write_lock(&my_rwlock);
```

- RW spinlocks favor readers over writers:



Spin locks

Reader-writer spin locks: methods

► Reader-writer spin locks methods:

- `read_lock()`
- `read_lock_irq()`
- `read_lock_irqsave()`
- `read_unlock()`
- `read_unlock_irq()`
- `read_unlock_irqrestore()`
- `write_lock()`
- `write_lock_irq()`
- `write_lock_irqsave()`
- `write_unlock()`
- `write_unlock_irq()`
- `write_unlock_irqrestore()`
- `write_trylock()`
- `rwlock_init()`

Spin locks

Reader-writer spin locks: usage example

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/delay.h>
6 #include <linux/spinlock.h>
7 #include <linux/kthread.h>
8 #include <linux/sched.h>
9
10 #define PRINT_PREF "[SYNC_RWSPINLOCK] "
11
12 unsigned int counter; /* shared data: */
13 DEFINE_RWLOCK(counter_lock);
14 struct task_struct *read_thread1, *
15     read_thread2, *read_thread3, *
16     write_thread;
17
18 static int writer_function(void *data)
19 {
20     while(!kthread_should_stop()) {
21         write_lock(&counter_lock);
22         counter++;
23         write_unlock(&counter_lock);
24         msleep(500);
25     }
26     do_exit(0);
27 }
```

```

26 static int read_function(void *data)
27 {
28     while(!kthread_should_stop()) {
29         read_lock(&counter_lock);
30         printk(PRINT_PREF "counter: %d\n",
31             counter);
32         read_unlock(&counter_lock);
33         msleep(500);
34     }
35     do_exit(0);
36 }
37
38 static int __init my_mod_init(void)
39 {
40     printk(PRINT_PREF "Entering module.\n");
41     counter = 0;
42
43     read_thread1 = kthread_run(read_function,
44         NULL, "read-thread1");
45     read_thread2 = kthread_run(read_function,
46         NULL, "read-thread2");
47     read_thread3 = kthread_run(read_function,
48         NULL, "read-thread3");
49     write_thread = kthread_run(writer_function,
50         NULL, "write-thread");
51
52     return 0;
53 }
```

Spin locks

Reader-writer spin locks: usage example (2)

```
48 static void __exit my_mod_exit(void)
49 {
50     kthread_stop(read_thread3);
51     kthread_stop(read_thread2);
52     kthread_stop(read_thread1);
53     kthread_stop(write_thread);
54     printk(KERN_INFO "Exiting module.\n");
55 }
56
57 module_init(my_mod_init);
58 module_exit(my_mod_exit);
59
60 MODULE_LICENSE("GPL");
```

Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Semaphores and mutexes

Semaphores presentation

► **Semaphores: sleeping locks**

- ▶ A thread trying to acquire an already held lock is put on a waitqueue
- ▶ When the semaphore becomes available, one task on the waitqueue is awoken
- ▶ Well suited towards locks held for a long time
 - ▶ On the contrary, large overhead for locks held for short periods

► **No usable in interrupt context**

- ▶ A thread can sleep while holding a semaphore
 - ▶ Another thread trying to acquire it will sleep and let you continue
- ▶ A thread cannot hold a spinlock while trying to acquire a semaphore
 - ▶ Might sleep!

Semaphores and mutexes

Semaphores presentation: counting vs binary semaphores

- ▶ Contrary to spin locks, 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**
 - ▶ Counter initialized to:
 - ▶ 1 → initially available
 - ▶ 0 → initially disabled

Semaphores and mutexes

Semaphores usage

- ▶ API in `includes/linux/semaphore.h`

```

1 struct semaphore *sem1;
2
3 sem1 = kmalloc(sizeof(struct semaphore),
4                 GFP_KERNEL);
4 if(!sem1)
5     return -1;
6
7 /* counter == 1: binary semaphore */
8 sema_init(&sema, 1);
9
10 down(sem1);
11 /* critical region */
12 up(sem1);

```

```

1 /* Binary semaphore static declaration */
2 DECLARE_MUTEX(sem2);
3
4 if(down_interruptible(&sem2)) {
5     /* signal received, semaphore not
6      acquired */
7 }
8
9 /* critical region */
10 up(sem2);

```

- ▶ `down()` puts the thread to sleep in `TASK_UNINTERRUPTIBLE` mode
- ▶ `down_interruptible` uses `TASK_INTERRUPTIBLE`

Semaphores and mutexes

Semaphores usage: methods

- ▶ `sema_init(struct semaphore *, int)`
 - ▶ Initializes the dynamically created semaphore with the given count
- ▶ `init_MUTEX(struct semaphore *)`
 - ▶ Initializes the dynamically created semaphore with the count of 1
- ▶ `init_MUTEX_LOCKED(struct semaphore *)`
 - ▶ Initializes the dynamically created semaphore with the count of 0
- ▶ `down_interruptible(struct semaphore *)`
 - ▶ Try to acquire the semaphore and goes into interruptible sleep if it is not available
- ▶ `down(struct semaphore *)`
 - ▶ Try to acquire the semaphore and goes into uninterruptible sleep if it is not available
 - ▶ **Deprecated** → prefer the use of `down_interruptible`

Semaphores and mutexes

Semaphores usage (2)

- ▶ `down_trylock(struct semaphore *)`
 - ▶ Try to acquire the semaphore and immediately return 0 if acquired, otherwise 1
- ▶ `down_timeout(struct semaphore *, long timeout)`
 - ▶ Try to acquire the semaphore and goes to sleep if not available. If the semaphore is not released after `timeout` jiffies, returns -ETIME
- ▶ `up(struct semaphore *)`
 - ▶ Release the semaphore and wake up a waiting thread if needed

Semaphores and mutexes

Reader-writer semaphores

- ▶ Example:

```
1 DECLARE_RWSEM(rwsem1);
2
3 init_rwsem(&rwsem1);
4
5 down_read(rwsem1);
6
7 /* critical (read) region */
8
9 up_read(&rwsem1);
```

```
1 struct rw_semaphore * rwsem2;
2
3 rwsem2 = kmalloc(sizeof(struct
4                   rw_semaphore), GFP_KERNEL);
5 if(!rwsem2)
6     return -1;
7
8 init_rwsem(rwsem2);
9
10 down_write(rwsem2);
11 /* critical (write) region */
12 up_write(rwsem2);
```

- ▶ downgrade_write()

- ▶ Convert an acquired write lock to a read one

Semaphores and mutexes

Reader-writer semaphore usage example

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/delay.h>
6 #include <linux/kthread.h>
7 #include <linux/sched.h>
8 #include <linux/rwsem.h>
9
10 #define PRINT_PREF "[SYNC_SEM] "
11
12 /* shared data: */
13 unsigned int counter;
14 struct rw_semaphore *counter_rwsemaphore;
15
16 struct task_struct *read_thread, *
17     read_thread2, *write_thread;
18
19 static int writer_function(void *data)
20 {
21     while(!kthread_should_stop()) {
22         down_write(counter_rwsemaphore);
23         counter++;
24     }
25 }
26
27 static int read_function(void *data)
28 {
29     up_read(counter_rwsemaphore);
30     msleep(500);
31 }
32
33 do_exit(0);
34
35 static int read_function(void *data)
36 {
37     while(!kthread_should_stop()) {
38         down_read(counter_rwsemaphore);
39         printk(PRINT_PREF "counter: %d\n",
40             counter);
41         up_read(counter_rwsemaphore);
42         msleep(500);
43     }
44 }
45
46 do_exit(0);
47
48 static int writer_function(void *data)
49 {
50     while(!kthread_should_stop()) {
51         down_write(counter_rwsemaphore);
52         counter++;
53     }
54 }
55
56 do_exit(0);
57
58 static int read_function(void *data)
59 {
60     up_read(counter_rwsemaphore);
61     msleep(500);
62 }
63
64 do_exit(0);
65
66 static int writer_function(void *data)
67 {
68     while(!kthread_should_stop()) {
69         down_write(counter_rwsemaphore);
70         counter++;
71     }
72 }
73
74 do_exit(0);
75
76 static int read_function(void *data)
77 {
78     up_read(counter_rwsemaphore);
79     msleep(500);
80 }
81
82 do_exit(0);
83
84 static int writer_function(void *data)
85 {
86     while(!kthread_should_stop()) {
87         down_write(counter_rwsemaphore);
88         counter++;
89     }
90 }
91
92 do_exit(0);
93
94 static int read_function(void *data)
95 {
96     up_read(counter_rwsemaphore);
97     msleep(500);
98 }
99
100 do_exit(0);
101
102 static int writer_function(void *data)
103 {
104     while(!kthread_should_stop()) {
105         down_write(counter_rwsemaphore);
106         counter++;
107     }
108 }
109
110 do_exit(0);
111
112 static int read_function(void *data)
113 {
114     up_read(counter_rwsemaphore);
115     msleep(500);
116 }
117
118 do_exit(0);
119
120 static int writer_function(void *data)
121 {
122     while(!kthread_should_stop()) {
123         down_write(counter_rwsemaphore);
124         counter++;
125     }
126 }
127
128 do_exit(0);
129
130 static int read_function(void *data)
131 {
132     up_read(counter_rwsemaphore);
133     msleep(500);
134 }
135
136 do_exit(0);
137
138 static int writer_function(void *data)
139 {
140     while(!kthread_should_stop()) {
141         down_write(counter_rwsemaphore);
142         counter++;
143     }
144 }
145
146 do_exit(0);
147
148 static int read_function(void *data)
149 {
150     up_read(counter_rwsemaphore);
151     msleep(500);
152 }
153
154 do_exit(0);
155
156 static int writer_function(void *data)
157 {
158     while(!kthread_should_stop()) {
159         down_write(counter_rwsemaphore);
160         counter++;
161     }
162 }
163
164 do_exit(0);
165
166 static int read_function(void *data)
167 {
168     up_read(counter_rwsemaphore);
169     msleep(500);
170 }
171
172 do_exit(0);
173
174 static int writer_function(void *data)
175 {
176     while(!kthread_should_stop()) {
177         down_write(counter_rwsemaphore);
178         counter++;
179     }
180 }
181
182 do_exit(0);
183
184 static int read_function(void *data)
185 {
186     up_read(counter_rwsemaphore);
187     msleep(500);
188 }
189
190 do_exit(0);
191
192 static int writer_function(void *data)
193 {
194     while(!kthread_should_stop()) {
195         down_write(counter_rwsemaphore);
196         counter++;
197     }
198 }
199
200 do_exit(0);
201
202 static int read_function(void *data)
203 {
204     up_read(counter_rwsemaphore);
205     msleep(500);
206 }
207
208 do_exit(0);
209
210 static int writer_function(void *data)
211 {
212     while(!kthread_should_stop()) {
213         down_write(counter_rwsemaphore);
214         counter++;
215     }
216 }
217
218 do_exit(0);
219
220 static int read_function(void *data)
221 {
222     up_read(counter_rwsemaphore);
223     msleep(500);
224 }
225
226 do_exit(0);
227
228 static int writer_function(void *data)
229 {
230     while(!kthread_should_stop()) {
231         down_write(counter_rwsemaphore);
232         counter++;
233     }
234 }
235
236 do_exit(0);
237
238 static int read_function(void *data)
239 {
240     up_read(counter_rwsemaphore);
241     msleep(500);
242 }
243
244 do_exit(0);
245
246 static int writer_function(void *data)
247 {
248     while(!kthread_should_stop()) {
249         down_write(counter_rwsemaphore);
250         counter++;
251     }
252 }
253
254 do_exit(0);
255
256 static int read_function(void *data)
257 {
258     up_read(counter_rwsemaphore);
259     msleep(500);
260 }
261
262 do_exit(0);
263
264 static int writer_function(void *data)
265 {
266     while(!kthread_should_stop()) {
267         down_write(counter_rwsemaphore);
268         counter++;
269     }
270 }
271
272 do_exit(0);
273
274 static int read_function(void *data)
275 {
276     up_read(counter_rwsemaphore);
277     msleep(500);
278 }
279
280 do_exit(0);
281
282 static int writer_function(void *data)
283 {
284     while(!kthread_should_stop()) {
285         down_write(counter_rwsemaphore);
286         counter++;
287     }
288 }
289
290 do_exit(0);
291
292 static int read_function(void *data)
293 {
294     up_read(counter_rwsemaphore);
295     msleep(500);
296 }
297
298 do_exit(0);
299
300 static int writer_function(void *data)
301 {
302     while(!kthread_should_stop()) {
303         down_write(counter_rwsemaphore);
304         counter++;
305     }
306 }
307
308 do_exit(0);
309
310 static int read_function(void *data)
311 {
312     up_read(counter_rwsemaphore);
313     msleep(500);
314 }
315
316 do_exit(0);
317
318 static int writer_function(void *data)
319 {
320     while(!kthread_should_stop()) {
321         down_write(counter_rwsemaphore);
322         counter++;
323     }
324 }
325
326 do_exit(0);
327
328 static int read_function(void *data)
329 {
330     up_read(counter_rwsemaphore);
331     msleep(500);
332 }
333
334 do_exit(0);
335
336 static int writer_function(void *data)
337 {
338     while(!kthread_should_stop()) {
339         down_write(counter_rwsemaphore);
340         counter++;
341     }
342 }
343
344 do_exit(0);
345
346 static int read_function(void *data)
347 {
348     up_read(counter_rwsemaphore);
349     msleep(500);
350 }
351
352 do_exit(0);
353
354 static int writer_function(void *data)
355 {
356     while(!kthread_should_stop()) {
357         down_write(counter_rwsemaphore);
358         counter++;
359     }
360 }
361
362 do_exit(0);
363
364 static int read_function(void *data)
365 {
366     up_read(counter_rwsemaphore);
367     msleep(500);
368 }
369
370 do_exit(0);
371
372 static int writer_function(void *data)
373 {
374     while(!kthread_should_stop()) {
375         down_write(counter_rwsemaphore);
376         counter++;
377     }
378 }
379
380 do_exit(0);
381
382 static int read_function(void *data)
383 {
384     up_read(counter_rwsemaphore);
385     msleep(500);
386 }
387
388 do_exit(0);
389
390 static int writer_function(void *data)
391 {
392     while(!kthread_should_stop()) {
393         down_write(counter_rwsemaphore);
394         counter++;
395     }
396 }
397
398 do_exit(0);
399
400 static int read_function(void *data)
401 {
402     up_read(counter_rwsemaphore);
403     msleep(500);
404 }
405
406 do_exit(0);
407
408 static int writer_function(void *data)
409 {
410     while(!kthread_should_stop()) {
411         down_write(counter_rwsemaphore);
412         counter++;
413     }
414 }
415
416 do_exit(0);
417
418 static int read_function(void *data)
419 {
420     up_read(counter_rwsemaphore);
421     msleep(500);
422 }
423
424 do_exit(0);
425
426 static int writer_function(void *data)
427 {
428     while(!kthread_should_stop()) {
429         down_write(counter_rwsemaphore);
430         counter++;
431     }
432 }
433
434 do_exit(0);
435
436 static int read_function(void *data)
437 {
438     up_read(counter_rwsemaphore);
439     msleep(500);
440 }
441
442 do_exit(0);
443
444 static int writer_function(void *data)
445 {
446     while(!kthread_should_stop()) {
447         down_write(counter_rwsemaphore);
448         counter++;
449     }
450 }
451
452 do_exit(0);
453
454 static int read_function(void *data)
455 {
456     up_read(counter_rwsemaphore);
457     msleep(500);
458 }
459
460 do_exit(0);
461
462 static int writer_function(void *data)
463 {
464     while(!kthread_should_stop()) {
465         down_write(counter_rwsemaphore);
466         counter++;
467     }
468 }
469
470 do_exit(0);
471
472 static int read_function(void *data)
473 {
474     up_read(counter_rwsemaphore);
475     msleep(500);
476 }
477
478 do_exit(0);
479
480 static int writer_function(void *data)
481 {
482     while(!kthread_should_stop()) {
483         down_write(counter_rwsemaphore);
484         counter++;
485     }
486 }
487
488 do_exit(0);
489
490 static int read_function(void *data)
491 {
492     up_read(counter_rwsemaphore);
493     msleep(500);
494 }
495
496 do_exit(0);
497
498 static int writer_function(void *data)
499 {
500     while(!kthread_should_stop()) {
501         down_write(counter_rwsemaphore);
502         counter++;
503     }
504 }
505
506 do_exit(0);
507
508 static int read_function(void *data)
509 {
510     up_read(counter_rwsemaphore);
511     msleep(500);
512 }
513
514 do_exit(0);
515
516 static int writer_function(void *data)
517 {
518     while(!kthread_should_stop()) {
519         down_write(counter_rwsemaphore);
520         counter++;
521     }
522 }
523
524 do_exit(0);
525
526 static int read_function(void *data)
527 {
528     up_read(counter_rwsemaphore);
529     msleep(500);
530 }
531
532 do_exit(0);
533
534 static int writer_function(void *data)
535 {
536     while(!kthread_should_stop()) {
537         down_write(counter_rwsemaphore);
538         counter++;
539     }
540 }
541
542 do_exit(0);
543
544 static int read_function(void *data)
545 {
546     up_read(counter_rwsemaphore);
547     msleep(500);
548 }
549
550 do_exit(0);
551
552 static int writer_function(void *data)
553 {
554     while(!kthread_should_stop()) {
555         down_write(counter_rwsemaphore);
556         counter++;
557     }
558 }
559
560 do_exit(0);
561
562 static int read_function(void *data)
563 {
564     up_read(counter_rwsemaphore);
565     msleep(500);
566 }
567
568 do_exit(0);
569
570 static int writer_function(void *data)
571 {
572     while(!kthread_should_stop()) {
573         down_write(counter_rwsemaphore);
574         counter++;
575     }
576 }
577
578 do_exit(0);
579
580 static int read_function(void *data)
581 {
582     up_read(counter_rwsemaphore);
583     msleep(500);
584 }
585
586 do_exit(0);
587
588 static int writer_function(void *data)
589 {
590     while(!kthread_should_stop()) {
591         down_write(counter_rwsemaphore);
592         counter++;
593     }
594 }
595
596 do_exit(0);
597
598 static int read_function(void *data)
599 {
600     up_read(counter_rwsemaphore);
601     msleep(500);
602 }
603
604 do_exit(0);
605
606 static int writer_function(void *data)
607 {
608     while(!kthread_should_stop()) {
609         down_write(counter_rwsemaphore);
610         counter++;
611     }
612 }
613
614 do_exit(0);
615
616 static int read_function(void *data)
617 {
618     up_read(counter_rwsemaphore);
619     msleep(500);
620 }
621
622 do_exit(0);
623
624 static int writer_function(void *data)
625 {
626     while(!kthread_should_stop()) {
627         down_write(counter_rwsemaphore);
628         counter++;
629     }
630 }
631
632 do_exit(0);
633
634 static int read_function(void *data)
635 {
636     up_read(counter_rwsemaphore);
637     msleep(500);
638 }
639
640 do_exit(0);
641
642 static int writer_function(void *data)
643 {
644     while(!kthread_should_stop()) {
645         down_write(counter_rwsemaphore);
646         counter++;
647     }
648 }
649
650 do_exit(0);
651
652 static int read_function(void *data)
653 {
654     up_read(counter_rwsemaphore);
655     msleep(500);
656 }
657
658 do_exit(0);
659
660 static int writer_function(void *data)
661 {
662     while(!kthread_should_stop()) {
663         down_write(counter_rwsemaphore);
664         counter++;
665     }
666 }
667
668 do_exit(0);
669
670 static int read_function(void *data)
671 {
672     up_read(counter_rwsemaphore);
673     msleep(500);
674 }
675
676 do_exit(0);
677
678 static int writer_function(void *data)
679 {
680     while(!kthread_should_stop()) {
681         down_write(counter_rwsemaphore);
682         counter++;
683     }
684 }
685
686 do_exit(0);
687
688 static int read_function(void *data)
689 {
690     up_read(counter_rwsemaphore);
691     msleep(500);
692 }
693
694 do_exit(0);
695
696 static int writer_function(void *data)
697 {
698     while(!kthread_should_stop()) {
699         down_write(counter_rwsemaphore);
700         counter++;
701     }
702 }
703
704 do_exit(0);
705
706 static int read_function(void *data)
707 {
708     up_read(counter_rwsemaphore);
709     msleep(500);
710 }
711
712 do_exit(0);
713
714 static int writer_function(void *data)
715 {
716     while(!kthread_should_stop()) {
717         down_write(counter_rwsemaphore);
718         counter++;
719     }
720 }
721
722 do_exit(0);
723
724 static int read_function(void *data)
725 {
726     up_read(counter_rwsemaphore);
727     msleep(500);
728 }
729
730 do_exit(0);
731
732 static int writer_function(void *data)
733 {
734     while(!kthread_should_stop()) {
735         down_write(counter_rwsemaphore);
736         counter++;
737     }
738 }
739
740 do_exit(0);
741
742 static int read_function(void *data)
743 {
744     up_read(counter_rwsemaphore);
745     msleep(500);
746 }
747
748 do_exit(0);
749
750 static int writer_function(void *data)
751 {
752     while(!kthread_should_stop()) {
753         down_write(counter_rwsemaphore);
754         counter++;
755     }
756 }
757
758 do_exit(0);
759
760 static int read_function(void *data)
761 {
762     up_read(counter_rwsemaphore);
763     msleep(500);
764 }
765
766 do_exit(0);
767
768 static int writer_function(void *data)
769 {
770     while(!kthread_should_stop()) {
771         down_write(counter_rwsemaphore);
772         counter++;
773     }
774 }
775
776 do_exit(0);
777
778 static int read_function(void *data)
779 {
780     up_read(counter_rwsemaphore);
781     msleep(500);
782 }
783
784 do_exit(0);
785
786 static int writer_function(void *data)
787 {
788     while(!kthread_should_stop()) {
789         down_write(counter_rwsemaphore);
790         counter++;
791     }
792 }
793
794 do_exit(0);
795
796 static int read_function(void *data)
797 {
798     up_read(counter_rwsemaphore);
799     msleep(500);
800 }
801
802 do_exit(0);
803
804 static int writer_function(void *data)
805 {
806     while(!kthread_should_stop()) {
807         down_write(counter_rwsemaphore);
808         counter++;
809     }
810 }
811
812 do_exit(0);
813
814 static int read_function(void *data)
815 {
816     up_read(counter_rwsemaphore);
817     msleep(500);
818 }
819
820 do_exit(0);
821
822 static int writer_function(void *data)
823 {
824     while(!kthread_should_stop()) {
825         down_write(counter_rwsemaphore);
826         counter++;
827     }
828 }
829
830 do_exit(0);
831
832 static int read_function(void *data)
833 {
834     up_read(counter_rwsemaphore);
835     msleep(500);
836 }
837
838 do_exit(0);
839
840 static int writer_function(void *data)
841 {
842     while(!kthread_should_stop()) {
843         down_write(counter_rwsemaphore);
844         counter++;
845     }
846 }
847
848 do_exit(0);
849
850 static int read_function(void *data)
851 {
852     up_read(counter_rwsemaphore);
853     msleep(500);
854 }
855
856 do_exit(0);
857
858 static int writer_function(void *data)
859 {
860     while(!kthread_should_stop()) {
861         down_write(counter_rwsemaphore);
862         counter++;
863     }
864 }
865
866 do_exit(0);
867
868 static int read_function(void *data)
869 {
870     up_read(counter_rwsemaphore);
871     msleep(500);
872 }
873
874 do_exit(0);
875
876 static int writer_function(void *data)
877 {
878     while(!kthread_should_stop()) {
879         down_write(counter_rwsemaphore);
880         counter++;
881     }
882 }
883
884 do_exit(0);
885
886 static int read_function(void *data)
887 {
888     up_read(counter_rwsemaphore);
889     msleep(500);
890 }
891
892 do_exit(0);
893
894 static int writer_function(void *data)
895 {
896     while(!kthread_should_stop()) {
897         down_write(counter_rwsemaphore);
898         counter++;
899     }
900 }
901
902 do_exit(0);
903
904 static int read_function(void *data)
905 {
906     up_read(counter_rwsemaphore);
907     msleep(500);
908 }
909
910 do_exit(0);
911
912 static int writer_function(void *data)
913 {
914     while(!kthread_should_stop()) {
915         down_write(counter_rwsemaphore);
916         counter++;
917     }
918 }
919
920 do_exit(0);
921
922 static int read_function(void *data)
923 {
924     up_read(counter_rwsemaphore);
925     msleep(500);
926 }
927
928 do_exit(0);
929
930 static int writer_function(void *data)
931 {
932     while(!kthread_should_stop()) {
933         down_write(counter_rwsemaphore);
934         counter++;
935     }
936 }
937
938 do_exit(0);
939
940 static int read_function(void *data)
941 {
942     up_read(counter_rwsemaphore);
943     msleep(500);
944 }
945
946 do_exit(0);
947
948 static int writer_function(void *data)
949 {
950     while(!kthread_should_stop()) {
951         down_write(counter_rwsemaphore);
952         counter++;
953     }
954 }
955
956 do_exit(0);
957
958 static int read_function(void *data)
959 {
960     up_read(counter_rwsemaphore);
961     msleep(500);
962 }
963
964 do_exit(0);
965
966 static int writer_function(void *data)
967 {
968     while(!kthread_should_stop()) {
969         down_write(counter_rwsemaphore);
970         counter++;
971     }
972 }
973
974 do_exit(0);
975
976 static int read_function(void *data)
977 {
978     up_read(counter_rwsemaphore);
979     msleep(500);
980 }
981
982 do_exit(0);
983
984 static int writer_function(void *data)
985 {
986     while(!kthread_should_stop()) {
987         down_write(counter_rwsemaphore);
988         counter++;
989     }
990 }
991
992 do_exit(0);
993
994 static int read_function(void *data)
995 {
996     up_read(counter_rwsemaphore);
997     msleep(500);
998 }
999
1000 do_exit(0);

```

Semaphores and mutexes

Reader-writer semaphore usage example (2)

```

44 static int __init my_mod_init(void)
45 {
46     printk(PRINT_PREF "Entering module.\n");
47     counter = 0;
48
49     counter_rwsemaphore = kmalloc(sizeof(
50         struct rw_semaphore), GFP_KERNEL);
51     if(!counter_rwsemaphore)
52         return -1;
53
54     init_rwsem(counter_rwsemaphore);
55
56     read_thread = kthread_run(read_function,
57         NULL, "read-thread");
58     read_thread2 = kthread_run(read_function
59         , NULL, "read-thread2");
60     write_thread = kthread_run(
61         writer_function, NULL, "write-thread
62         ");
63
64     return 0;
65 }
```

```

61 static void __exit my_mod_exit(void)
62 {
63     kthread_stop(read_thread);
64     kthread_stop(write_thread);
65     kthread_stop(read_thread2);
66
67     kfree(counter_rwsemaphore);
68
69     printk(KERN_INFO "Exiting module.\n");
70 }
71
72 module_init(my_mod_init);
73 module_exit(my_mod_exit);
74
75 MODULE_LICENSE("GPL");
```

Semaphores and mutexes

Mutexes

- ▶ **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 through the API
- ▶ With special debugging mode: (`CONFIG_DEBUG_MUTEXES`)
 - ▶ **The kernel can check and warn if these constraints are not met**
- ▶ Mutex vs semaphore use?
 - ▶ If these constraints disallow the use of mutexes, use semaphores
 - ▶ **Otherwise always use mutexes**

Semaphores and mutexes

Mutexes: usage

- ▶ API in `include/linux/mutex.h`

```
1 DEFINE_MUTEX(mut1); /* static */
2
3 struct mutex *mut2 = kmalloc(sizeof(struct mutex), GFP_KERNEL); /* dynamic */
4 if(!mut2)
5     return -1;
6
7 mutex_init(mut2);
8
9 mutex_lock(&mut1);
10
11 /* critical region */
12
13 mutex_unlock(&mut1);
```

Semaphores and mutexes

Mutexes: usage example

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/delay.h>
6 #include <linux/kthread.h>
7 #include <linux/sched.h>
8 #include <linux/mutex.h>
9
10 #define PRINT_PREF "[SYNC_MUTEX]: "
11 /* shared data: */
12 unsigned int counter;
13 struct mutex *mut;
14 struct task_struct *read_thread, *
15     write_thread;
16
17 static int writer_function(void *data)
18 {
19     while(!kthread_should_stop()) {
20         mutex_lock(mut);
21         kfree(mut);      /* !!! */
22         counter++;
23         mutex_unlock(mut);
24         msleep(500);
25     }
26     do_exit(0);
}

```

```

27 static int read_function(void *data)
28 {
29     while(!kthread_should_stop()) {
30         mutex_lock(mut);
31         printk(PRINT_PREF "counter: %d\n",
32                counter);
33         mutex_unlock(mut);
34         msleep(500);
35     }
36     do_exit(0);
37 }
38
39 static int __init my_mod_init(void)
40 {
41     printk(PRINT_PREF "Entering module.\n");
42     counter = 0;
43
44     mut = kmalloc(sizeof(struct mutex),
45                  GFP_KERNEL);
46     if(!mut)
47         return -1;
48     mutex_init(mut);
}

```

Semaphores and mutexes

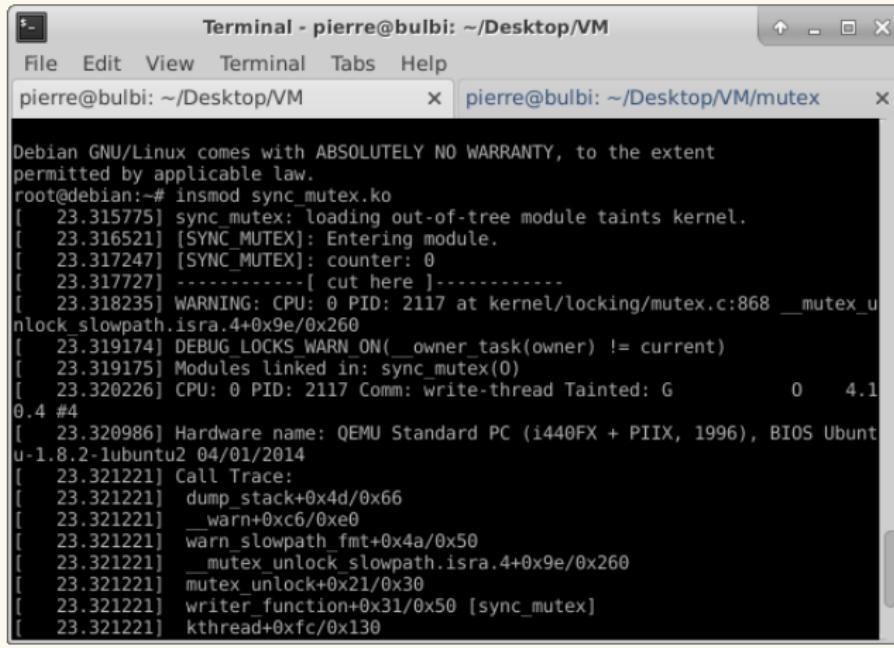
Mutexes: usage example (2)

```
49     read_thread = kthread_run(read_function,
50         NULL, "read-thread");
51     write_thread = kthread_run(
52         writer_function, NULL, "write-thread
53         ");
54
55     return 0;
56 }
57
58 static void __exit my_mod_exit(void)
59 {
60     kthread_stop(read_thread);
61     kthread_stop(write_thread);
62     kfree(mut);
63     printk(KERN_INFO "Exiting module.\n");
64 }
65
66 module_init(my_mod_init);
67 module_exit(my_mod_exit);
68
69 MODULE_LICENSE("GPL");
```

Semaphores and mutexes

Mutexes: usage example (3)

- On a kernel compiled with **CONFIG_DEBUG_MUTEXES**:



The screenshot shows a terminal window titled "Terminal - pierre@bulbi: ~/Desktop/VM". It contains two tabs: "pierre@bulbi: ~/Desktop/VM" and "pierre@bulbi: ~/Desktop/VM/mutex". The "mutex" tab is active and displays the following kernel log output:

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@debian:~# insmod sync_mutex.ko
[ 23.315775] sync_mutex: loading out-of-tree module taints kernel.
[ 23.316521] [SYNC_MUTEX]: Entering module.
[ 23.317247] [SYNC_MUTEX]: counter: 0
[ 23.317727] -----[ cut here ]-----
[ 23.318235] WARNING: CPU: 0 PID: 2117 at kernel/locking/mutex.c:868 __mutex_u
nlock_slowpath.isra.4+0x9e/0x260
[ 23.319174] DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current)
[ 23.319175] Modules linked in: sync_mutex(0)
[ 23.320226] CPU: 0 PID: 2117 Comm: write-thread Tainted: G      0      4.1
0.4 #4
[ 23.320986] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubuntu-1.8.2-1ubuntu2 04/01/2014
[ 23.321221] Call Trace:
[ 23.321221]  dump_stack+0x4d/0x66
[ 23.321221]  warn+0xc6/0xe0
[ 23.321221]  warn_slowpath_fmt+0x4a/0x50
[ 23.321221]  __mutex_unlock_slowpath.isra.4+0x9e/0x260
[ 23.321221]  mutex_unlock+0x21/0x30
[ 23.321221]  writer_function+0x31/0x50 [sync_mutex]
[ 23.321221]  kthread+0xfc/0x130
```

Semaphores and mutexes

Spin lock vs mutex usage

- ▶ Low overhead locking needed? use **spin lock**
- ▶ Short lock hold time? use **spin lock**
- ▶ Long lock hold time? use **mutex**
- ▶ Need to lock in interrupt context? use **spin lock**
- ▶ Need to sleep while holding? use **mutex**

Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Other synchronization mechanisms

Completion variables

- ▶ **Completion variables** are used when a thread need to signal another one of some event
 - ▶ Waiting thread **sleeps**
- ▶ API in `include/linux/completion.h`
- ▶ Declaration / initialization:

```
1 DECLARE_COMPLETION(comp1); /* static */
2 struct completion *comp2 = kmalloc(sizeof(struct completion), GFP_KERNEL); /* dynamic */
3 if(!comp2)
4     return -1;
5 init_completion(comp2);
```

- ▶ Thread A:

```
1 /* signal event: */
2 complete(comp1);
```

- ▶ Thread B:

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

Other synchronization mechanisms

Completion variables: usage example

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/delay.h>
6 #include <linux/kthread.h>
7 #include <linux/sched.h>
8 #include <linux/completion.h>
9
10 #define PRINT_PREF "[SYNC_COMP] "
11
12 unsigned int counter; /* shared data: */
13 struct completion *comp;
14 struct task_struct *read_thread, *
15     write_thread;
16
17 static int writer_function(void *data)
18 {
19     while(counter != 1234)
20         counter++;
21     complete(comp);
22
23     do_exit(0);
}

```

```

24     static int read_function(void *data)
25     {
26         wait_for_completion(comp);
27         printk(PRINT_PREF "counter: %d\n", counter)
28         ;
29         do_exit(0);
30     }
31
32     static int __init my_mod_init(void)
33     {
34         printk(PRINT_PREF "Entering module.\n");
35         counter = 0;
36
37         comp = kmalloc(sizeof(struct completion),
38                         GFP_KERNEL);
39         if(!comp) return -1;
40
41         init_completion(comp);
42         read_thread = kthread_run(read_function,
43             NULL, "read-thread");
44         write_thread = kthread_run(writer_function,
45             NULL, "write-thread");
46
47         return 0;
48     }

```

Other synchronization mechanisms

Completion variables: usage example (2)

```
47 static void __exit my_mod_exit(void)
48 {
49     kfree(comp);
50     printk(KERN_INFO "Exiting module.\n");
51 }
52
53 module_init(my_mod_init);
54 module_exit(my_mod_exit);
55
56 MODULE_LICENSE("GPL");
```

Other synchronization mechanisms

Preemption disabling

- ▶ When a spin lock is held preemption is disabled
- ▶ Some situations require preemption disabling without involving spin locks
 - ▶ Example: manipulating per-processor data:

```
1 task A manipulates per-processor data foo (not protected by a lock)
2 task A is preempted and task B is scheduled (on the same CPU)
3 task B manipulates variable foo
4 task B completes and task A is rescheduled
5 task A continues manipulating variable foo
```

- ▶ Might lead to inconsistent state for foo
- ▶ API to **disable kernel preemption**
 - ▶ Can nest, implemented through a counter
 - ▶ `preempt_disable()`
 - ▶ Disabled kernel preemption, increment preemption counter
 - ▶ `preempt_enable()`
 - ▶ Decrement counter and enable preemption if it reaches 0

Other synchronization mechanisms

Preemption disabling (2)

► API (continued):

- ▶ preempt_enable_no_resched()
 - ▶ Enable kernel preemption, does not check for any pending reschedule
- ▶ preempt_count()
 - ▶ Return preemption counter
- ▶ get_cpu()
 - ▶ Disable preemption and return the current CPU id

```
1 int cpu = get_cpu(); /* disable preemption and return current CPU id */
2
3 struct my_struct my_variable = per_cpu_structs_array[cpu];
4 /* manipulate my_variable */
5
6 put_cpu(); /* re-enable preemption */
```

Other synchronization mechanisms

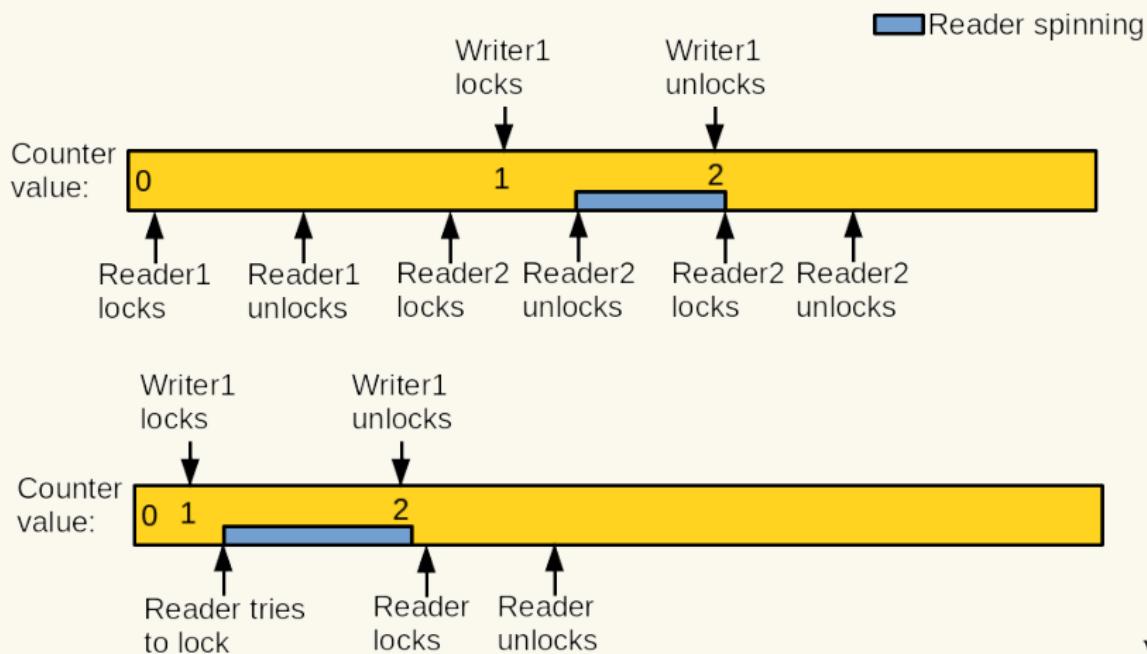
Sequential locks

- ▶ **Sequential lock / seq lock**

- ▶ Reader-writer spinlock **scaling to many readers and favoring writers**
- ▶ Implemented with a counter (sequence number)
 - ▶ Initialized to 0
 - ▶ Incremented by 1 each time a writer takes and releases the lock
- ▶ Before and after reading the data the counter is checked
 - ▶ If different, a write operation happened and the read operation must be repeated
 - ▶ Prior to the read operation, if the counter is odd a write is underway
- ▶ API in `include/linux/seqlock.h`

Other synchronization mechanisms

Sequential locks (2)



Other synchronization mechanisms

Sequential locks (3)

- ▶ Usage:

```
1 seqlock_t my_seq_lock = DEFINE_SEQLOCK(my_seq_lock);
```

- ▶ Write path:

```
1 write_seqlock(&my_seq_lock);
2 /* critical (write) region */
3 write_sequnlock(&my_seq_lock);
```

- ▶ Read path:

```
1 unsigned long seq;
2 do {
3     seq = read_seqbegin(&my_seq_lock);
4     /* read data here ... */
5 } 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

- ▶ Example: jiffies

Outline

- 1 Introduction
- 2 Atomic operations
- 3 Spin locks
- 4 Semaphores and mutexes
- 5 Other synchronization mechanisms
- 6 Ordering and memory barriers

Ordering and memory barriers

Context

- ▶ **Memory reads (load) and write (store) operations can be reordered**

- ▶ By the compiler (compile time)
- ▶ By the CPU (run time)

- ▶ Could be reordered:

```
1 a = 4;  
2 b = 5;
```

- ▶ Not reordered:

```
1 a = 4;  
2 b = a; /* dependency b <- a */
```

- ▶ CPU/compiler are not aware about code in other context

- ▶ Communication with hardware
- ▶ Symmetric multiprocessing

- ▶ **Memory barriers instruction** allow to force the actual execution of load and stores at some point in the program

Ordering and memory barriers

Usage

- ▶ **rmb ()** (read memory barrier):
 - ▶ No load prior to the code will be reordered after the call
 - ▶ No load after the call will be reordered before the call
 - ▶ i.e. **commit all pending loads before continuing**
- ▶ **wmb ()** (write memory barrier):
 - ▶ Same as `rmb ()` with stores instead of loads
- ▶ **mb ()**:
 - ▶ Concerns loads *and* stores
- ▶ **barrier ()**:
 - ▶ Same as `mb ()` but only for the compiler
- ▶ **read_barrier_depends ()**
 - ▶ Prevent data-dependent loads ($b = a$) to be reordered across the barrier
 - ▶ Less costly than `rmb ()` as we block only on a subset of pending loads

Ordering and memory barriers

Usage: example

- ▶ Initially $a = 1, b = 2$

Thread 1	Thread 2
$a = 3;$	$if(b == 4)$
$b = 4;$	$assert(a == 3);$

- ▶ Cannot assume $a == 3$ in this example

Ordering and memory barriers

Usage: example (2)

- ▶ **Correct version:**

- ▶ Initially $a = 1, b = 2$

Thread 1	Thread 2
$a = 3;$	$\text{if}(b == 4)$
$\text{mb}();$	$\text{assert}(a == 3);$
$b = 4;$	

- ▶ **Concrete example of barrier usage:**

- ① Thread 1 initializes a data structure
- ② Thread 1 spawns thread 2
- ③ Thread 2 access the data structure

- ▶ Intuitively, no synchronization needed, but a `wmb()` is needed after the data structure initialization



Ordering and memory barriers

Usage: SMP optimizations

► **SMP optimizations:**

- ▶ `smp_rmb ()`:
 - ▶ `rmb()` on SMP and `barrier()` on UP
- ▶ `smp_read_barrier_depends ()`:
 - ▶ `read_barrier_depends()` on SMP and `barrier()` on UP
- ▶ `smp_wmb ()`:
 - ▶ `wmb()` on SMP and `barrier()` on UP
- ▶ `smp_mb ()`:
 - ▶ `mb()` on SMP and `barrier()` on UP

► More info on barriers:

- ▶ Documentation/memory-barriers.txt

Bibliography I

[1] Config.debug.lock.alloc: Lock debugging: detect incorrect freeing of live locks.

http://cateee.net/lkddb/web-lkddb/DEBUG_LOCK_ALLOC.html.

Accessed: 2017-03-14.

[2] Stack overflow - how to use lockdep feature in linux kernel for deadlock detection.

<http://stackoverflow.com/questions/20892822/>

how-to-use-lockdep-feature-in-linux-kernel-for-deadlock-detection.

Accessed: 2017-03-14.