Linux Kernel Programming
Kernel Synchronization

Pierre Olivier

Systems Software Research Group @ Virginia Tech

March 21, 2017

Outline

Outline

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:

```
int total = get_total_from_account();    /* total funds in user account */
int withdrawal = get_withdrawal_amount(); /* amount user asked to withdrawal */

/* check whether the user has enough funds in her account */
if(total < withdrawal) {
  error("Not enough money!");
  return -1;
}

/* The user has enough money, deduct the withdrawal amount from here total */
total -= withdrawal;
update_total_funds(total);

/* give the money to the user */
spit_out_money(withdrawal);
```

Introduction

Critical regions and race conditions (2)

```c
int total =
      get_total_from_account();
int withdrawal =
      get_withdrawal_amount();

if(total < withdrawal) {
  error("Not enough money!");
  return -1;
}

total -= withdrawal;
update_total_funds(total);

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)

```c
int total =
      get_total_from_account();
int withdrawal =
      get_withdrawal_amount();

if(total < withdrawal) {
  error("Not enough money!");
  return -1;
}

total -= withdrawal;
update_total_funds(total);

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:

```
get the current value of i and copy it into a register
add one to the value stored into the register
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 → 8)	-
`write back i` (8)	-
-	`get i` (8)
-	`increment i` (8 → 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` ($7 \rightarrow 8$)	-
-	`increment i` ($7 \rightarrow 8$)
`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	Thread 2
`increment & store i` ($7 \rightarrow 8$)	-
-	`increment and store i` ($8 \rightarrow 9$)

- Or:

Thread 1	Thread 2
-	`increment and store i` ($7 \rightarrow 8$)
`increment & store i` ($8 \rightarrow 9$)	-

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 $\rightarrow$ 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 $\rightarrow$ they are stuck
    - Real-life example: traffic deadlock
    - Self-deadlock (1 thread):

```
acquire lock
acquire lock again
waiting indefinitely ...
```

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

Thread 1	**Thread 2**
`acquire lock A`	`acquire lock B`
`try to acquire lock B`	`try to acquire lock A`
`wait for lock B...`	`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

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`:

```
typedef struct {
  int counter;
} atomic_t;
```

- API defined in `includes/asm/atomic.h`
- Usage:

```
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 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` 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(atomic_t *v)`	Atomically increment `v` by 1 and return the result
`int atomic_dec_return(atomic_t *v)`	Atomically decrement `v` by 1 and return the result
`int atomic_dec_and_test(atomic_t *v)`	Atomically decrement `v` by 1 and return true if the result is zero, false otherwise
`int atomic_inc_and_test(atomic_t *v)`	Atomically increment `v` by 1 and return true if the result is zero, false otherwise

Atomic operations

Atomic integer operations: 64-bits atomic operations

```
typedef struct {
  volatile long counter;
} 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
`int atomic64_add_return(int i, atomic64_t *v)`	Atomically add `i` to `v` and return the result
`int atomic64_sub_return(int i, atomic64_t *v)`	Atomically subtract `i` from `v` and return the result
`int atomic64_inc_return(atomic64_t *v)`	Atomically increment `v` by 1 and return the result
`int atomic64_dec_return(atomic64_t *v)`	Atomically decrement `v` by 1 and return the result
`int atomic64_dec_and_test(atomic64_t *v)`	Atomically decrement `v` by 1 and return true if the result is zero, false otherwise
`int atomic64_inc_and_test(atomic64_t *v)`	Atomically increment `v` by 1 and return true if the result is zero, false otherwise

Atomic operations

Atomic integer operations: 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/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);
}
```

```c
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 operations

Atomic integer operations: usage example(2)

```
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 operations

Atomic bitwise operations

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

```
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;
```

- 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
`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`

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

Outline

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`

```
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
- `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

```
DEFINE_SPINLOCK(my_lock); /* the spin lock */
unsigned long flags;      /* to save the interrupt state */

spin_lock_irqsave(&my_lock, flags);
/* critical region */
spin_unlock_irqrestore(&my_lock, flags);
```

  - If it is known that interrupts are initially enabled:

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

- Also true fron 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
`spin_lock()`	Acquires a lock
`spin_lock_irq()`	Disable local interrupts and acquire a lock
`spin_lock_irqsave()`	Save current state of local interrupts, disables local interrupts, and acquire a lock
`spin_unlock()`	Release a lock
`spin_unlock_irq()`	Release a lock and enable local interrupts
`spin_unlock_irqrestore()`	Release a lock and reset interrupts to previous state
`spin_lock_init()`	Dynamically initialize a `spinlock_t`
`spin_trylock()`	Try to acquire a lock and directly returns 0 if unavailable
`spin_is_locked()`	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

```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);
}
```

```c
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;
}
```

Spin locks

Usage example (2)

```c
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");
```

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**

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

- Reader code:

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

- Writer code:

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

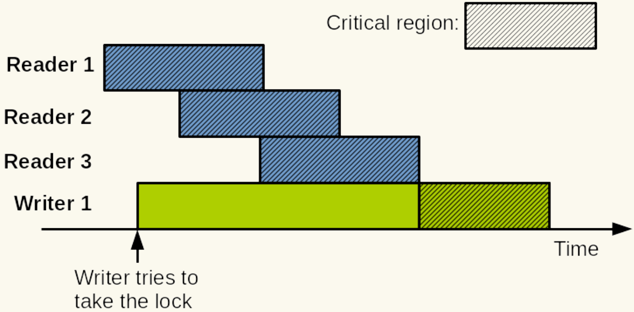
Spin locks

Reader-writer spin locks (2)

- Deadlock:

```
read_lock(&my_rwlock);
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

```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_RWSPINLOCK] "

unsigned int counter;  /* shared data: */
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);
}
```

```c
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;
}
```

Spin locks

Reader-writer spin locks: usage example (2)

```c
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");
```

Outline

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 awaken
- 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 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**
  - Counter initialized to:
    - 1 $\rightarrow$ initially available
    - 0 $\rightarrow$ initially disabled

Semaphores and mutexes

Semaphores usage

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

```c
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);
```

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

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

/* critical region */

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:

```
DECLARE_RWSEM(rwsem1);

init_rwsem(&rwsem1);

down_read(rwsem1);

/* critical (read) region */

up_read(&rwsem1);
```

```
struct rw_semaphore * rwsem2;

rwsem2 = kmalloc(sizeof(struct
        rw_semaphore), GFP_KERNEL);
if(!rwsem2)
  return -1;

init_rwsem(rwsem2);

down_write(rwsem2);
/* critical (write) region */
up_write(rwsem2);
```

- `downgrade_write()`
  - Convert an acquired write lock to a read one

Virginia Tech

Semaphores and mutexes

Reader-writer semaphore 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/rwsem.h>

#define PRINT_PREF "[SYNC_SEM] "

/* shared data: */
unsigned int counter;
struct rw_semaphore *counter_rwsemaphore;

struct task_struct *read_thread, *
      read_thread2, *write_thread;

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);
}
```

Semaphores and mutexes

Reader-writer semaphore usage example (2)

```
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;
}
```

```
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");
```

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 trhough 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`

```
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);
```

Semaphores and mutexes

Mutexes: 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;

static int writer_function(void *data)
{
  while(!kthread_should_stop()) {
    mutex_lock(mut);
    kfree(mut);        /* !!! */
    counter++;
    mutex_unlock(mut);
    msleep(500);
  }
  do_exit(0);
}
```

```c
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);
}

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);
```

Semaphores and mutexes

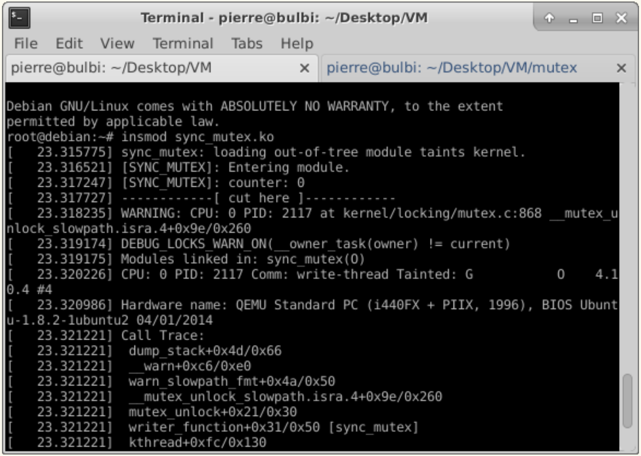
Mutexes: usage example (2)

```c
  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");
```

Semaphores and mutexes

Mutexes: usage example (3)

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




```
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(O)
[   23.320226] CPU: 0 PID: 2117 Comm: write-thread Tainted: G           O    4.1
0.4 #4
[   23.320986] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubunt
u-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

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:

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

- Thread A:

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

- Thread B:

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

Other synchronization mechanisms

Completion variables: 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/completion.h>

#define PRINT_PREF "[SYNC_COMP] "

unsigned int counter; /* shared data: */
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);
}
```

```c
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;
}
```

Other synchronization mechanisms

Completion variables: usage example (2)

```
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");
```

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:

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

```c
int cpu = get_cpu(); /* disable preemption and return current CPU id */

struct my_struct my_variable = per_cpu_structs_array[cpu];
/* manipulate my_variable */

put_cpu(); /* re-enable preemption */
```

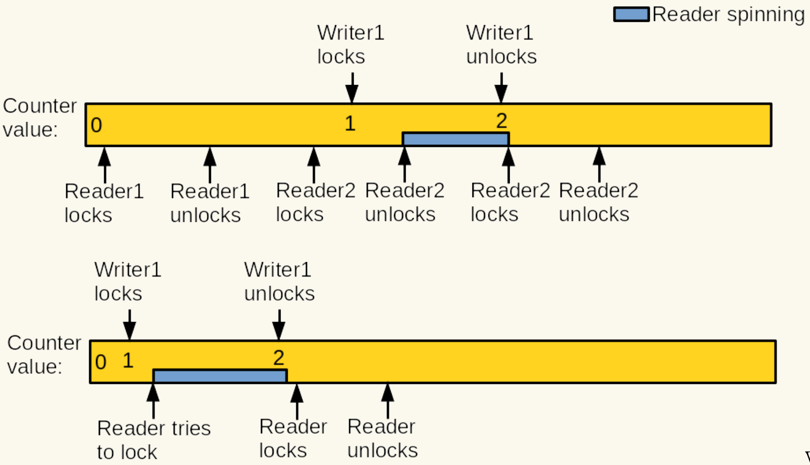
Other synchronization mechanisms

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:

```
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
- Example: `jiffies`

Outline

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:

```
a = 4;
b = 5;
```

- Not reordered:

```
a = 4;
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;`	`if(b == 4)`
`mb();`	`    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.