Synchronization
in the Linux Kernel (Part I)

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University
Outline

1. Race Condition Scenarios in the Linux Kernel
2. Synchronization for Single Core Hardware
3. Synchronization for Multicore Hardware
4. Q & A
Race Condition Scenario #1 in the Kernel

- System call and interrupt

```c
queue_t *shared_q;

my_irq_handler() { 
    tmp = io(..)
    push(shared_q, data);
}

my_read() { 
    n = length(shared_q);
    if(n>0) { 
        buf = kmalloc(1024);
        pop(shared_q, buf);
    }
}
```
Race Condition Scenario #2 in the Kernel

- System call and preemption

```c
queue_t *shared_q;

my_irq_handler() {
    tmp = io(...) 
    push(shared_q, data);
}

my_read() {
    n = length(shared_q);
    if(n>0) {
        buf = kmalloc(1024);
        pop(shared_q, buf);
    }
}
```
Synchronization Approaches in the Kernel

- **For single core hardware**
  - Disabling preemption or interrupts
    - Prevent other tasks or ISRs from running

- **For multicore hardware**
  - Atomic operations
    - Perform multiple actions at once
  - Locking primitives
    - Prevent other tasks from entering a critical section
    - Spinlocks, semaphores, and mutexes
Disabling Preemption

- Allow a task to complete its critical section without being interfered by other task.
Disabling Preemption in the Kernel

Three functions

- `preempt_disable()`
  - Disable kernel preemption by incrementing the preemption counter
- `preempt_enable()`
  - Decrement the preemption counter and checks and services any pending reschedules if the count is now zero
- `preempt_count()`
  - Return the preemption count

Preemption counter

- `preempt_count = 0` → preemptable
- `preempt_count > 0` → not preemptable
Limitations of Disabling Preemption

- Race condition in multicore hardware

- Race condition between a task and an ISR
Disabling Interrupts

- Allow a task to complete its critical section without being interfered by interrupts
  - Disabling interrupts also disables kernel preemption
Disabling Interrupts in the Kernel

- Simply disable and enable interrupts for the current processor
  - Clear and set interrupt flags of the processor
    ```c
    local_irq_disable();
    /* interrupts are disabled .. */
    local_irq_enable();
    ```

- Disable and enable interrupts saving and restoring the state of the interrupt system
  ```c
  local_irq_save(flags);
  /* ... */
  local_irq_restore(flags);
  ```
Disabling Interrupts in the Kernel

- Disable and enable only a specific interrupt line for the entire system
  - Mask out an interrupt line

```c
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
void synchronize_irq(unsigned int irq);
```

- `disable_irq()` does not return until any currently executing handler completes
- `disable_irq_nosync()` does not wait for current handlers to complete
- `synchronize_irq()` waits for a specific interrupt handler to exit
Atomic Operations

Atomic operations provide instructions that execute atomically—without interruption.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>get, increment, and store i (7 -&gt; 8)</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>get, increment, and store i (8 -&gt; 9)</td>
</tr>
<tr>
<td>get, increment, and store i (8 -&gt; 9)</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>get, increment, and store i (7 -&gt; 8)</td>
</tr>
<tr>
<td>get, increment, and store i (8 -&gt; 9)</td>
<td>—</td>
</tr>
</tbody>
</table>
Atomic Integer Operations

- A special data type `<linux/types.h>`

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

- The basic use

```c
atomic_t v;          /* define v */
atomic_t u = ATOMIC_INIT(0);  /* define u and initialize it to zero */
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 Add in ARM

- ARM LDREX and STREX are available in ARMv6 and above
## Atomic Integer Operations

<table>
<thead>
<tr>
<th>Operations and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMIC_INIT(int i): At declaration, initialize to I</td>
</tr>
<tr>
<td>int atomic_read(atomic_t *v): Atomically read the integer value of v</td>
</tr>
<tr>
<td>void atomic_set(atomic_t *v, int i): Atomically set v equal to i</td>
</tr>
<tr>
<td>void atomic_add(int i, atomic_t *v): Atomically add i to v</td>
</tr>
<tr>
<td>void atomic_sub(int i, atomic_t *v): Atomically subtract i from v</td>
</tr>
<tr>
<td>void atomic_inc(atomic_t *v): Atomically add one to v</td>
</tr>
<tr>
<td>void atomic_dec(atomic_t *v): Atomically subtract one from v</td>
</tr>
<tr>
<td>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</td>
</tr>
<tr>
<td>int atomic_add_negative(int i, atomic_t *v): Atomically add i to v and return true if the result is negative; otherwise false</td>
</tr>
<tr>
<td>int atomic_add_return(int i, atomic_t *v): Atomically add i to v and return the result</td>
</tr>
<tr>
<td>int atomic_sub_return(int i, atomic_t *v): Atomically subtract i from v and return the result</td>
</tr>
<tr>
<td>int atomic_inc_return(int i, atomic_t *v): Atomically increment v by one and return the result</td>
</tr>
<tr>
<td>int atomic_dec_return(int i, atomic_t *v): Atomically decrement v by one and return the result</td>
</tr>
<tr>
<td>int atomic_dec_and_test(atomic_t *v): Atomically decrement v by one and return true if zero; false otherwise</td>
</tr>
<tr>
<td>int atomic_inc_and_test(atomic_t *v): Atomically increment v by one and return true if the result is zero; false otherwise</td>
</tr>
</tbody>
</table>
Spin Locks

Spin lock is a mutual exclusion mechanism where a process spins (or busy-waits) until the lock becomes available.

Spin locks are architecture-dependent and implemented in assembly:
- The architecture-dependent code is defined in `<asm/spinlock.h>`
- The actual usable interfaces are defined in `<linux/spinlock.h>`
The Use of a Spin Lock

- Initializing and using a spin lock

```c
DEFINE_SPINLOCK(mr_lock);

spin_lock(&mr_lock);
/* critical region ... */
spin_unlock(&mr_lock);
```

- Unlike spin lock implementations in other operating systems and threading libraries, the Linux kernel’s spin locks are not recursive
  - This means that if the kernel attempts to acquire a lock it already holds, it will spin, waiting for itself to release the lock
  - But because it is busy spinning, it will never release the lock and it will deadlock
Acquiring a Spin Lock in an ISR

Spin locks can be used in interrupt handlers

- Semaphores cannot be used because they sleep

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

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

If a lock is used in an interrupt handler, you must also disable local interrupts

- Otherwise, it is possible for an interrupt handler to attempt to reacquire the lock (double-acquire deadlock)
Spin Lock Implementation (Single Core)

- If CONFIG_SMP is not set
  - `spin_lock() == preempt_disable()`
Spin Lock Implementation (Single Core)

- If CONFIG_SMP is not set
  - spin_lock_irqsave() == local_irqsave()
Spin Lock Implementation (Multicore)

- If CONFIG_SMP is set
  - `spin_lock() == preempt_disable() + arch_spin_lock()`
Spin Lock Implementation (Multicore)

If CONFIG_SMP is set

- `spin_lock_irqsave() == local_irqsave() + arch_spin_lock()`
Architecture Specific Implementation (ARM)

- Simple Test-and-Set Spin Lock
  - Lock is an integer (initialized to 0)
    - 0: unlocked
    - 1: locked

- Pseudo code

```c
arch_spin_lock(lock) {
    while (testAndSet(lock) == 1) {} 
}

arch_spin_unlock(lock) {
    lock = 0;
}
```

Atomic operation:
1. Set lock to 1
2. Return the previous value of lock
Architecture Specific Implementation (ARM)

- Real code

```c
static inline void arch_spin_lock(arch_spinlock_t *lock) {
    unsigned long tmp;
    __asm__ __volatile__(
        "1:
        ldrex %0, [%1]\n"
        "\n        teq %0, #0\n"
        WFE("ne")
        "\n        streseq %0, %2, [%1]\n"
        "\n        teqeq %0, #0\n"
        "\n        bne lb"
        : "=r" (tmp)
        : "r" (&lock->lock), "r" (1)
        : "cc"
    );
    smp_mb();
}
```
Semaphores

Semaphores are sleeping locks

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>down_interruptible (struct semaphore *)</td>
<td>Tries to acquire the given semaphore and enter interruptible sleep if it is contended</td>
</tr>
<tr>
<td>down(struct semaphore *)</td>
<td>Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended</td>
</tr>
<tr>
<td>down_trylock(struct semaphore *)</td>
<td>Tries to acquire the given semaphore and immediately return nonzero if it is contended</td>
</tr>
<tr>
<td>up(struct semaphore *)</td>
<td>Releases the given semaphore and wakes a waiting task, if any</td>
</tr>
</tbody>
</table>

- The function **down_interruptible**() attempts to acquire the given semaphore
- If the semaphore is unavailable, it places the calling process to sleep in the TASK_INTERRUPTIBLE state
- The task can be awakened with a signal
Semaphore Data Structure

```
#include/linux/semaphore.h

struct semaphore{
    unsigned int counter;
    spinlock_t lock;
    struct list_head wait_list;
};
```

- **count (= usage count)**
  The number of permissible simultaneous holders of semaphores

- **lock**
  Lock for wait queue

- **wait_list**
  wait queue for tasks which failed to acquire a semaphore
The **down()** and **up()** Functions

- Semaphore implementations rely on spinlocks

```c
void down(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    spin_unlock_irqrestore(&sem->lock, flags);
}

void up(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    else
        __up(sem);
    spin_unlock_irqrestore(&sem->lock, flags);
}
```
Constraints on Semaphores

- Semaphore cannot be used in interrupt context
- Semaphore cannot be used while holding a spin lock
**Mutexes**

- Mutexes are simpler sleeping locks
  - A mutex is a binary semaphore

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mutex_lock(struct mutex *)</code></td>
<td>Locks the given mutex; sleeps if the lock is unavailable</td>
</tr>
<tr>
<td><code>mutex_unlock(struct mutex *)</code></td>
<td>Unlocks the given mutex</td>
</tr>
<tr>
<td><code>mutex_trylock(struct mutex *)</code></td>
<td>Tries to acquire the given mutex; returns one if successful and the lock is acquired and zero otherwise</td>
</tr>
<tr>
<td><code>mutex_is_locked (struct mutex *)</code></td>
<td>Returns one if the lock is locked and zero otherwise</td>
</tr>
</tbody>
</table>
Mutex Data Structure

```c
#include/linux/mutex.h

struct mutex {
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
};
```

- **count** (= usage_count)
  - 1: unlocked
  - 0: locked
  - <0: locked, possible waiters

- **wait_lock**
  - Lock for wait queue

- **wait_list**
  - Wait queue for tasks which failed to acquire the mutex
Mutexes

- Semaphores are generic and do not impose many usage constraints
  - This makes them useful for managing exclusive access in obscure situations

- Mutexes have a strict, narrower use case
  - Whoever locked a mutex must unlock it
  - Recursive locks and unlocks are not allowed
  - A task cannot exit while holding a mutex
  - A mutex cannot be acquired by an interrupt handler or bottom half
  - A mutex can be managed only via the official API