5. Lock

NXKR_YoungSikYang

Exported on 03/07/2024

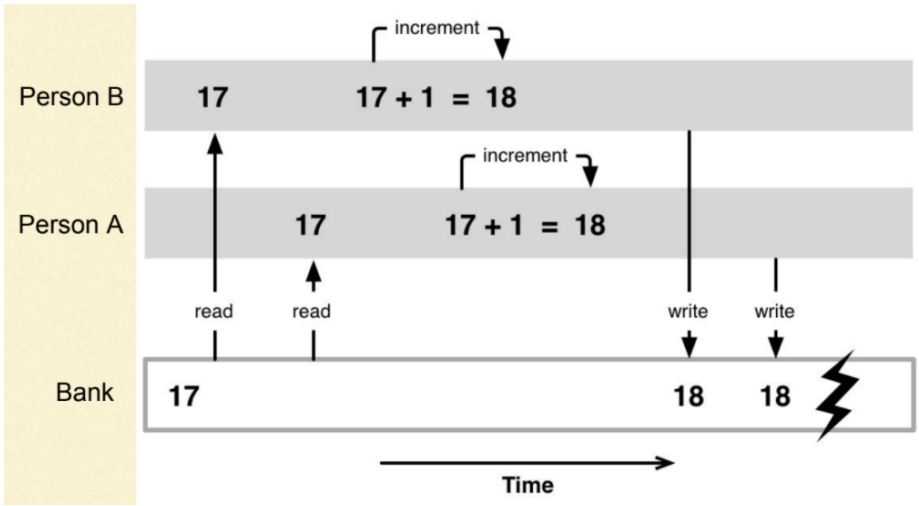
Table of Contents

1. Terminology

- **Data consistency** refers to whether the same data kept at different places do or do not match.
- **Critical section** is a code segment where shared resources are accessed.
    - Example of the critical section:

```
P0
{
Read(A);
A=A+1;
Write(A);
}
```

- **Race condition** is a situation where multiple processes are accessing a shared resource(**critical section**) simultaneously, which might affect **data consistency**.
    - This picture shows how the race condition occurs:



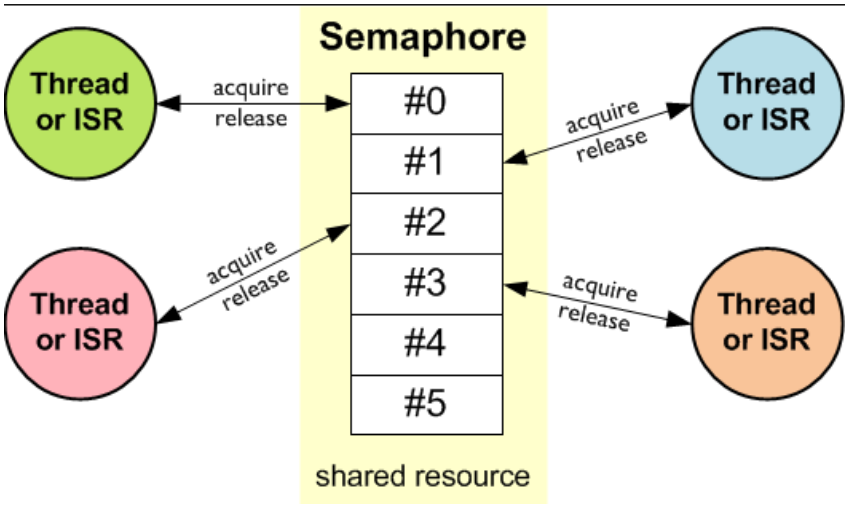
- **Process Synchronization** refers to aligning the execution timing of processes to ensure that shared resources are accessed by only one process at a time to deal with the **race condition** and maintain **data consistency**.

2. Lock mechanisms

Lock mechanisms are used to protect shared resources from concurrent access.

- **Mutex(mutual exclusion)** : Used to ensure that a shared resource is accessed only by one thread at a time.
  - It has two states: locked and unlocked. If a thread tries to lock a mutex that is already locked by another thread, the second thread waits until it is unlocked.
  - Visualization of the mutex:

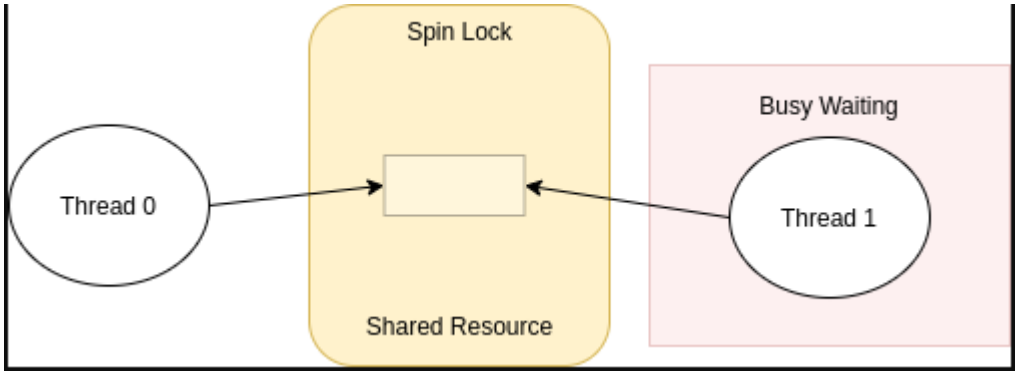
- **Semaphore** : A semaphore has multiple locks. A mutex can be considered a binary semaphore.
  - A semaphore can have multiple locks, thereby allowing more than one thread to access a shared resource but limits the number.
  - Visualization of the semaphore:



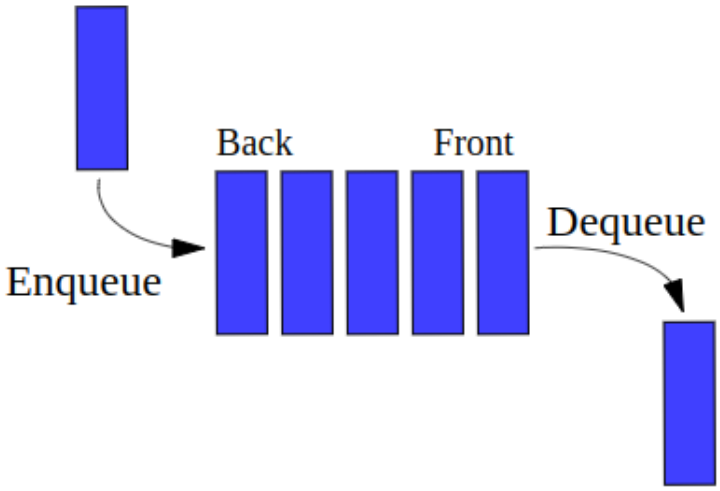
- **Monitor**: Unlike the mutex, where a thread must **explicitly** lock and unlock, monitors manage locks **implicitly**. When a thread enters a monitor's method, it automatically acquires the lock, and when it exits the method (either normally or by waiting on a condition variable), it automatically releases the lock. Monitors can use a condition variables, which are used to block a thread until a particular condition is met.

3. How to implement a lock

- **Spinlock(busy waiting)**: When a thread attempts to acquire a spinlock that is already held by another thread, it will continuously check (or "spin") until the lock becomes available.
    - **When to use?**: This approach can be efficient if the expected wait time is short since the thread remains active and does not require a context switch.
    - **When not to use?**: However, it wastes CPU cycles if the lock is held for a long time. The **process queue** is more appropriate in this case.
    - Visualization of the spinlock:

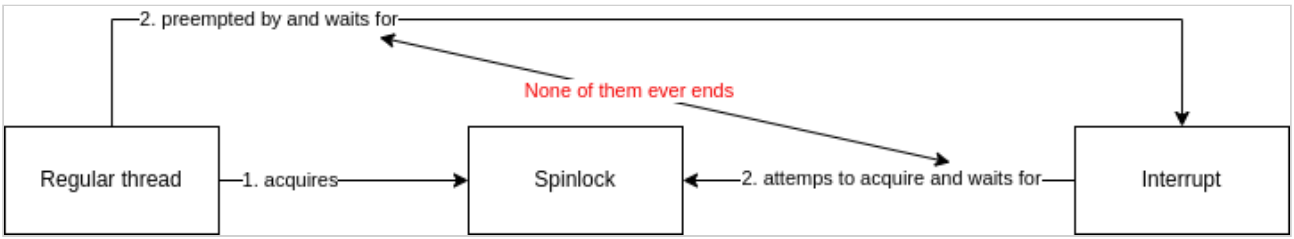


- **Process queue**: The PCB of a process is put in a queue and sent to sleep. It is popped when the shared resource becomes available.
    - Example of PCBs enqueued for sleep and dequeued to wake up for the context switch:



3. Lock in the interrupt context

This section describes how use of a regular spinlock leads to deadlock.



1. The regular thread acquires the spin lock to access the shared resource.
2. While the regular thread holds the spin lock, it is preempted by an interrupt.
3. The interrupt handler, which also needs to access the same shared resource, attempts to acquire the spin lock.
4. **The system is in a deadlock**: the interrupt handler cannot proceed because it's waiting for the spin lock, and the regular thread cannot proceed because it's interrupted by the handler, which does not finish.

This is why **spin_lock_irqsave()** and **spin_unlock_irqrestore()** are used for the lock in the interrupt context.

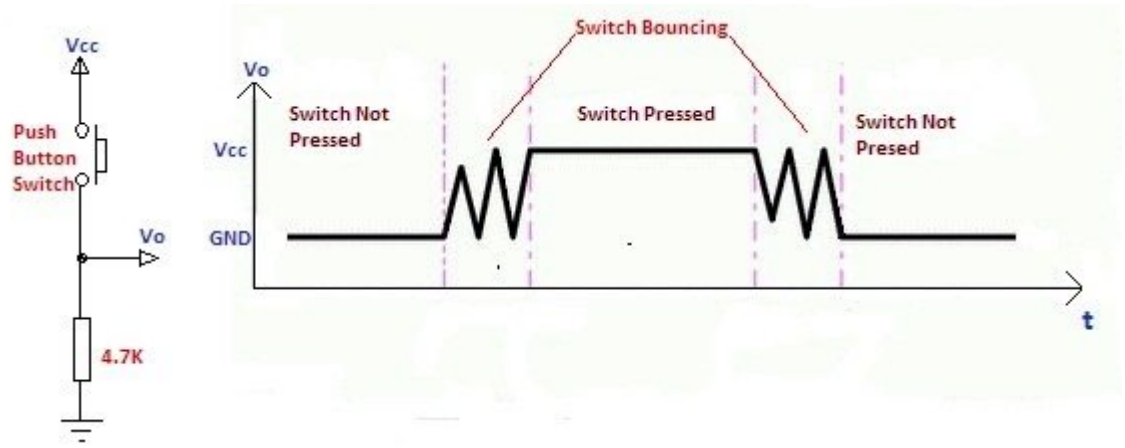
- **spin_lock_irqsave():** disables the interrupt and saves its state
- **spin_lock_irqrestore():** restores the interrupt state

4. Practice

4.1 Chattering and debouncing

The hardware switch GPIO_B4, used in the practice code in `section 4.2,` experiences chattering and lacks debouncing, requiring a programmatic solution.

- **Chattering**, also known as bounce, refers to the rapid, undesired opening and closing of a switch contact in a very short duration when it is pushed or released.. This phenomenon can cause multiple signals to be sent by a single actuation, leading to unpredictable behavior in electronic circuits.
- **Debouncing** is the process of removing the unwanted chattering effect from the signal generated by a mechanical switch to stabilize the input signal. This can be achieved either through software or hardware solutions.
    - Visualization of debouncing:



    - In **software** debouncing, a timer can be used to ignore additional state changes that occur within a short window of time after the first change is detected.
    - In **hardware** debouncing, **capacitors** can be used to filter out the noise or more complex circuits like flip-flops to stabilize the signal.
- Example of the software debouncing

```
#include <linux/delay.h>
static bool debounce_flag = false;
static void foo()
{
    unsigned int gpio_value, flags;
    if (debounce_flag)
        return IRQ_HANDLED;
    // Ignore chattering interrupts
```

```
    debounce_flag = true;
    // Enable the button after Debounce time
    mdelay(200);
    debounce_flag = false;
    // Continue code
}
```

4.2 Code flow

1. Register 2 button interrupts
2. Create an interrupt handler that includes a critical section where a specific value is added to the shared resource and waits for the amount of time equal to the shared resource.
3. Press the 2 buttons at the same time and check the result.

4.3 Example of race condition without a lock

4.3.1 Source code

In this example without a spinlock, The ISR triggered by the second button press is not blocked until the first button press finishes.

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

#define button 3 // Example GPIO number
#define button2 36
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
static bool debounce_flag = false; // Used to debounce the button
static spinlock_t my_lock;
static int shared_resource = 0;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}
```

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    unsigned int gpio_value, flags;
    if (debounce_flag)
        return IRQ_HANDLED;
    // Ignore chattering interrupts
    debounce_flag = true;
    // Enable the button after Debounce time
    mdelay(100);
    debounce_flag = false;
    // Critical section
    // spin_lock_irqsave(&my_lock, flags);
    pr_info("\nButton pressed! irq = %d, num = %d\n", irq, shared_resource);
    shared_resource += 1;
    mdelay(shared_resource * 1000);
    pr_info("Unlocked! irq = %d, num = %d\n", irq, shared_resource);
    shared_resource = 0;
    // spin_unlock_irqrestore(&my_lock, flags);
    return IRQ_HANDLED;
}

static void set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
}

static int __init btn_irq_init(void)
{
    // Request GPIO
    int result_btn;
    result_btn = gpio_request(button, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the LED GPIO\n");
        return 0;
    }
    // Initialize the spinlock
    // spin_lock_init(&my_lock);
    // Set the direction of button GPIO
    gpio_direction_input(button);
```

```
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    set_irq(button2, bottom_half, IRQF_TRIGGER_RISING);
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

4.3.2 Result

```
Button pressed! irq = 106, shared_resource = 0

Button pressed! irq = 107, shared_resource = 1
Unlocked! irq = 106, shared_resource = 2
Unlocked! irq = 107, shared_resource = 0
```

Simultaneous access to **shared_resource** is not blocked, which leads to having unpredictable values.

4.4 Example of race condition with a lock

4.4.1 Source code

In this example with a spinlock, The ISR triggered by the second button press is blocked until the first button press finishes.

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

#define button 3 // Example GPIO number
#define button2 36
```

```
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
static bool debounce_flag = false; // Used to debounce the button
static spinlock_t my_lock;
static int shared_resource = 0;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t bottom_half(int irq, void *dev_id)
{
    unsigned int gpio_value, flags;
    if (debounce_flag)
        return IRQ_HANDLED;
    // Ignore chattering interrupts
    debounce_flag = true;
    // Enable the button after Debounce time
    mdelay(200);
    debounce_flag = false;
    // Critical section
    spin_lock_irqsave(&my_lock, flags);
    pr_info("\nButton pressed! irq = %d, shared_resource = %d\n", irq,
shared_resource);
    shared_resource += 1;
    mdelay(shared_resource * 1000);
    pr_info("Unlocked! irq = %d, shared_resource = %d\n", irq, shared_resource);
    shared_resource = 0;
    spin_unlock_irqrestore(&my_lock, flags);
    return IRQ_HANDLED;
}

static void set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
```

```
    }
}

static int __init btn_irq_init(void)
{
    // Request GPIO
    int result_btn;
    result_btn = gpio_request(button, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    result_btn = gpio_request(button2, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Initialize the spinlock
    spin_lock_init(&my_lock);
    // Set the direction of button GPIO
    gpio_direction_input(button);
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    set_irq(button2, bottom_half, IRQF_TRIGGER_RISING);
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

4.4.2 Result

Simultaneous access to **shared_resource** is not blocked, which leads to having stable values.

```
Button pressed! irq = 107, shared_resource = 0
Unlocked! irq = 107, shared_resource = 1

Button pressed! irq = 106, shared_resource = 0
Unlocked! irq = 106, shared_resource = 1
```