EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 23 – Spinlock in Linux Kernel Part 1**

Device Drivers

EmbeTronicX
Embedded Tutorials Zone
Spinlock Tutorial - Part 1
Linux Device Driver Tutorial - Part 23

Linux Device Driver Tutorial Part 23 – Spinlock in Linux Kernel Part 1

2

X

provide easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 23 – Spinlock in Linux Kernel Part 1.

Apple Music Turns 5 as It Continues Rivalry With Spotify

Post Contents [hide]

1 Prerequisites
2 Introduction
3 SpinLock
4 SpinLock in Linux Kernel Device Driver
  4.1 Initialize
    4.1.1 Static Method
    4.1.2 Dynamic Method
  4.2 Approach 1 (Locking between User context)
    4.2.1 Example
  4.3 Approach 2 (Locking between Bottom Halves)
  4.4 Approach 3 (Locking between User context and Bottom Halves)
    4.4.1 Example
  4.5 Approach 4 (Locking between Hard IRQ and Bottom Halves)
    4.5.1 Example
  4.6 Approach 5 (Alternative way of Approach 4)
  4.7 Approach 6 (Locking between Hard IRQs)

2

X

5.2.1 Share this:
5.2.2 Like this:
5.2.3 Related

Prerequisites

In the example section, I had used Kthread to explain Mutex. If you don't know what is Kthread and How to use it, then I would recommend you to explore that by using below link.

1. Kthread Tutorial in Linux Kernel
2. Mutex Tutorial in Linux Kernel

Introduction

In our previous tutorial, we have understood the use of Mutex and its Implementation. If you have understood Mutex then Spinlock is also similar. Both are used to protect a shared resource from being modified

X

In the Mutex concept, when the thread is trying to lock or acquire the Mutex which is not available then that thread will go to sleep until that Mutex is available. Whereas in Spinlock it is different. The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock. This simplicity creates a small and fast lock.

Like Mutex, there are two possible states in Spinlock: **Locked** or **Unlocked**.

SpinLock in Linux Kernel Device Driver

If the kernel is running on a uniprocessor and `CONFIG_SMP`, `CONFIG_PREEMPT` aren't enabled while compiling the kernel then spinlock will not be available. Because there is no reason to have a lock when no one else can run at the same time.

But if you have disabled `CONFIG_SMP` and enabled `CONFIG_PREEMPT` then spinlock will simply disable preemption, which is sufficient to prevent any races.

Initialize

We can initialize Spinlock in Linux kernel in two ways.

X

Static Method

You can statically initialize a Spinlock using the macro given below.

```
DEFINE_SPINLOCK(etx_spinlock);
```

The macro given above will create a **spinlock_t** variable in the name of **etx_spinlock** and initialize to **UNLOCKED STATE**. Take a look at the expansion of DEFINE_SPINLOCK below.

```
#define DEFINE_SPINLOCK(x)      spinlock_t x = __SPIN_LOCK_UNLOCKED(x)
```

Dynamic Method

If you want to initialize dynamically you can use the method as given below.

```
spinlock_t etx_spinlock;
spin_lock_init(&etx_spinlock);
```

You can use any one of the methods.

After initializing the spinlock, there are several ways to use spinlock to lock or unlock, based on where the spinlock is used; either in user context or interrupt context. Let's look at the approaches with these situations.

Approach 1 (Locking between User context)

If you share data with user context (between Kernel Threads), then you can use this approach.

Lock:

2

spin_lock(spinlock_t *lock)

X

`spin_trylock(spinlock_t *lock)`

Locks the spinlock if it is not already locked. If unable to obtain the lock it exits with an error and do not spin. It **returns** non-zero if it obtains the lock otherwise returns zero.

Unlock:

`spin_unlock(spinlock_t *lock)`

It does the reverse of the lock. It will unlock which is locked by the above call.

Checking Lock:

`spin_is_locked(spinlock_t *lock)`

This is used to check whether the lock is available or not. It **returns** non-zero if the lock is currently acquired. otherwise returns zero.

Example

```
//Thread 1
int thread_function1(void *pv)
{
    while(!kthread_should_stop()) {
        spin_lock(&etx_spinlock);
        etx_global_variable++;
        printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variabl
        spin_unlock(&etx_spinlock);
        msleep(1000);
    }
    return 0;
}

//Thread 2
int thread_function2(void *pv)
```

X

```
    }
    return 0;
}
```

Approach 2 (Locking between Bottom Halves)

If you want to share data between two different Bottom halves or the same bottom halves, then you can use the Approach 1.

Approach 3 (Locking between User context and Bottom Halves)

If you share data with a bottom half and user context (like Kernel Thread), then this approach will be useful.

Lock:

`spin_lock_bh(spinlock_t *lock)`

It disables soft interrupts on that CPU, then grabs the lock. This has the effect of preventing softirqs, tasklets, and bottom halves from running on the local CPU. Here the suffix '`_bh`' refers to "**Bottom Halves**".

Unlock:

`spin_unlock_bh(spinlock_t *lock)`

It will release the lock and re-enables the soft interrupts which are disabled by the above call.

Example

```
//Thread
int thread_function(void *pv)
{
    while(!kthread_should_stop()) {
        spin_lock_bh(&etx_spinlock);
        etx_global_variable++;
```

X

```
void tasklet_fn(unsigned long arg)
{
        spin_lock_bh(&etx_spinlock);
        etx_global_variable++;
        printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
        spin_unlock_bh(&etx_spinlock);
}
```

Approach 4 (Locking between Hard IRQ and Bottom Halves)

If you share data between Hardware ISR and Bottom halves then you have to disable the IRQ before lock. Because the bottom halves processing can be interrupted by a hardware interrupt. So this will be used in that scenario.

Lock:

`spin_lock_irq(spinlock_t *lock)`

This will disable interrupts on that cpu, then grab the lock.

Unlock:

2

`spin_unlock_irq(spinlock_t *lock)`

It will release the lock and re-enables the interrupts which are disabled

X

```
/*Tasklet Function*/
void tasklet_fn(unsigned long arg)
{
        spin_lock_irq(&etx_spinlock);
        etx_global_variable++;
        printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
        spin_unlock_irq(&etx_spinlock);
}

//Interrupt handler for IRQ 11.
static irqreturn_t irq_handler(int irq,void *dev_id) {
        spin_lock_irq(&etx_spinlock);
        etx_global_variable++;
        printk(KERN_INFO "Executing ISR Function : %lu\n", etx_global_variable);
        spin_unlock_irq(&etx_spinlock);
        /*Scheduling Task to Tasklet*/
        tasklet_schedule(tasklet);
        return IRQ_HANDLED;
}
```

Approach 5 (Alternative way of Approach 4)

If you want to use a different variant rather than using **spin_lock_irq()** and **spin_unlock_irq()** then you can use this approach.

Lock:

```
spin_lock_irqsave( spinlock_t *lock, unsigned long flags );
```

This will save whether interrupts were on or off in a **flags** word and grab the lock.

Unlock:

```
spin_unlock_irqrestore( spinlock_t *lock, unsigned long flags );
```

This will release the spinlock and restores the interrupts using the **flags** argument.

2

Approach 6 (Locking between Hard IRQs)

X

programming

This code snippet explains how to create two threads that access a global variable (**`etx_gloabl_variable`**). So before accessing the variable, it should lock the spinlock. After that, it will release the spinlock. This example is using Approach 1.

Driver Source Code

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include <linux/kthread.h>             //kernel threads
#include <linux/sched.h>               //task_struct
#include <linux/delay.h>

DEFINE_SPINLOCK(etx_spinlock);
//spinlock_t etx_spinlock;
unsigned long etx_global_variable = 0;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

static struct task_struct *etx_thread1;
static struct task_struct *etx_thread2;

/*************** Driver Fuctions **********************/
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len,loff_t * off);
static ssize_t etx_write(struct file *filp,
                const char *buf, size_t len, loff_t * off);
 /******************************************************/

int thread_function1(void *pv);
int thread_function2(void *pv);

int thread_function1(void *pv)
{
```

```
            printk(KERN_INFO "Spinlock is locked in Thread Function1\n");
        }
        etx_global_variable++;
        printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variab
        spin_unlock(&etx_spinlock);
        msleep(1000);
    }
    return 0;
}

int thread_function2(void *pv)
{
    while(!kthread_should_stop()) {
        spin_lock(&etx_spinlock);
        etx_global_variable++;
        printk(KERN_INFO "In EmbeTronicX Thread Function2 %lu\n", etx_global_variab
        spin_unlock(&etx_spinlock);
        msleep(1000);
    }
    return 0;
}

static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .release        = etx_release,
};

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Read function\n");

        return 0;
}
static ssize_t etx_write(struct file *filp,
                const char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Write Function\n");
        return len;
}
```

```
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*Adding character device to the system*/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
            printk(KERN_INFO "Cannot add the device to the system\n");
            goto r_class;
        }

        /*Creating struct class*/
        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
            printk(KERN_INFO "Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
            printk(KERN_INFO "Cannot create the Device \n");
            goto r_device;
        }


        /* Creating Thread 1 */
        etx_thread1 = kthread_run(thread_function1,NULL,"eTx Thread1");
        if(etx_thread1) {
            printk(KERN_ERR "Kthread1 Created Successfully...\n");
        } else {
            printk(KERN_ERR "Cannot create kthread1\n");
             goto r_device;
        }

         /* Creating Thread 2 */
        etx_thread2 = kthread_run(thread_function2,NULL,"eTx Thread2");
        if(etx_thread2) {
            printk(KERN_ERR "Kthread2 Created Successfully...\n");
        } else {
            printk(KERN_ERR "Cannot create kthread2\n");
             goto r_device;
        }
        //spin_lock_init(&etx_spinlock);

        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;


r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        cdev_del(&etx_cdev);
        return -1;
}

```

```
        unregister_chrdev_region(dev, 1);
        printk(KERN_INFO "Device Driver Remove...Done!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("A simple device driver - Spinlock");
MODULE_VERSION("1.18");
```

MakeFile

```
obj-m += driver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:
    make -C $(KDIR) M=$(shell pwd) modules

clean:
    make -C $(KDIR) M=$(shell pwd) clean
```

In our next part of the tutorial, we will see another variant of the spinlock (**Reader/writer spinlocks**).

0

Article Rating

☆☆☆☆☆

Share this:

Share 10 Post Tweet SHARE Print WhatsApp Share

1 Email Telegram More

Like this:

Loading...

X

Linux Device Driver Tutorial Part 22 – Mutex in Linux Kernel
In "Device Drivers"

Linux Device Driver Tutorial Part 24 – Read Write Spinlock in Linux Kernel (Spinlock Part 2)
In "Device Drivers"

Linux Device Driver Tutorial Part 31 – Seqlock in Linux Kernel
In "Device Drivers"

2

X

Subscribe Connect with | Login

Join the discussion

B I U S {} [+]

This site uses Akismet to reduce spam. Learn how your comment data is processed.

2 COMMENTS Oldest

Riddhi

May 23, 2019 6:31 AM

Why you need to disable IRQ while sharing data between ISR, hard IRQ (Apporch 6) ?
Spin lock is enough.....

Loading...

0 Reply

riddhi patel

May 23, 2019 6:50 AM

Why you need to disable IRQ while sharing data between ISR (hard IRQ , ... Apporch 6) ?
During ISR execution interrupts on particular core are already disabled... So Spin lock is enough.....

2

X

report this ad

2

X