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

Post Contents [hide]

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 by two or more processes simultaneously.

SpinLock

In Mutex concept, when 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_SMPand enabled CONFIG_PREEMPT then spinlock will simply disable preemption, which is sufficient to prevent any races.

Initialize

We can initialize Spinlock in two ways.

1. Static Method
2. Dynamic Method

Static Method

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

```
DEFINE_SPINLOCK(etx_spinlock);
```

The macro given above will create spinlock_t variable in the name of 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_tetx_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:

spin_lock(spinlock_t *lock)

This will take the lock if it is free, otherwise it'll spin until that lock is free (Keep trying).

Try Lock:

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 obtains the lock otherwise returns zero.

Unlock:

spin_unlock(spinlock_t *lock)

It does the reverse of lock. It will unlock which is locked by 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_variable);
spin_unlock(&etx_spinlock);
msleep(1000);
   }
return 0;
}

//Thread 2
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_variable);
spin_unlock(&etx_spinlock);
msleep(1000);
   }
return 0;
}
```

Approach 2 (Locking between Bottom Halves)

If you want to share data between two different Bottom halves or 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 is disabled by above call.

Example

```
//Thread
intthread_function(void *pv)
{
while(!kthread_should_stop()) {
spin_lock_bh(&etx_spinlock);
etx_global_variable++;
printk(KERN_INFO "In EmbeTronicX Thread Function %lu\n",
etx_global_variable);
spin_unlock_bh(&etx_spinlock);
msleep(1000);
   }
return 0;
}
/*Tasklet Function*/
voidtasklet_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:

spin_unlock_irq(spinlock_t *lock)

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

Example

```
/*Tasklet Function*/
voidtasklet_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.
staticirqreturn_tirq_handler(intirq,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 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 releases the spinlock and restores the interrupts using the flags argument.

Approach 6 (Locking between Hard IRQs)

If you want to share data between two different IRQs, then you should use Approach 5.

Example Programming

This code snippet explains how to create two threads that accesses 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_tetx_spinlock;
unsigned long etx_global_variable = 0;

dev_tdev = 0;
staticstruct class *dev_class;
staticstructcdevetx_cdev;

staticint __initetx_driver_init(void);
static void __exit etx_driver_exit(void);

staticstructtask_struct *etx_thread1;
staticstructtask_struct *etx_thread2;

/*************** Driver Fuctions **********************/
staticintetx_open(structinode *inode, struct file *file);
staticintetx_release(structinode *inode, struct file *file);
staticssize_tetx_read(struct file *filp,
char __user *buf, size_tlen,loff_t * off);
staticssize_tetx_write(struct file *filp,
const char *buf, size_tlen, loff_t * off);
 /******************************************************/

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

int thread_function1(void *pv)
{

while(!kthread_should_stop()) {
if(!spin_is_locked(&etx_spinlock)) {
printk(KERN_INFO "Spinlock is not locked in Thread Function1\n");
        }
spin_lock(&etx_spinlock);
if(spin_is_locked(&etx_spinlock)) {
printk(KERN_INFO "Spinlock is locked in Thread Function1\n");
        }
etx_global_variable++;
printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variable);
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_variable);
spin_unlock(&etx_spinlock);
msleep(1000);
    }
return 0;
}

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

staticintetx_open(structinode *inode, struct file *file)
{
```

```
printk(KERN_INFO "Device File Opened...!!!\n");
return 0;
}

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

staticssize_tetx_read(struct file *filp,
char __user *buf, size_tlen, loff_t *off)
{
printk(KERN_INFO "Read function\n");

return 0;
}
staticssize_tetx_write(struct file *filp,
const char __user *buf, size_tlen, loff_t *off)
{
printk(KERN_INFO "Write Function\n");
returnlen;
}

staticint __initetx_driver_init(void)
{
        /*Allocating Major number*/
if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
printk(KERN_INFO "Cannot allocate major number\n");
return -1;
        }
printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creatingcdev 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");
gotor_class;
        }

        /*Creatingstruct class*/
if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
```

```
printk(KERN_INFO "Cannot create the struct class\n");
gotor_class;
        }

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

void __exit etx_driver_exit(void)
{
```

```
kthread_stop(etx_thread1);
kthread_stop(etx_thread2);
device_destroy(dev_class,dev);
class_destroy(dev_class);
cdev_del(&etx_cdev);
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
```