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

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. Spinlock Tutorial in Linux Kernel – Part 1

Introduction

In our previous tutorial we have understood the use of Common Spinlock and its Implementation. If you have understood Spinlock then this Read Write Spinlock is also similar except some difference. We will cover those things below.

SpinLock

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. Read Write spinlock also does the same but it has separate locks for read and write operation.

Read Write Spinlocks

I told that spinlock and read write spinlock are similar in operation. That means spinlock is enough. Then why do we need Read Write Spinlock? Ridiculous right?

Okay now we will take one scenario. I have five threads. All those threads are accessing one global variable. So we can use spinlock over there. Am I right? Well, yes it's right. In those threads, thread-1's role is to write the data into that variable. Other four thread's roles are simply reading the data from that variable. This the case. I hope you guys understand the scenario. Now I can ask you guys an question. If you implement spinlock,

Question : Now we forgot the thread-1(writer thread). So we have 4 reader thread (thread-2 to thread-5). Now those all four threads are want to read the data from the variable at the same time. What will happen in this situation if you use spinlock? What about the processing speed and performance?
Let's assume thread-2 got the lock while other reading threads are trying hardly for the lock. That variable won't change even thread-2's access. Because no one is writing. But here only thread-2 is accessing. But other reading threads are simply wasting time to take lock since variable won't change. That means performance will be reduced. Isn't it? Yes you are correct.

In this case if you implement read and write lock, thread-2 will take the read lock and read the data. And other reading threads also will take the read lock without spinning (blocking) and read the data. Because no one is writing. So what about the performance now? There is no waiting between reading operations. Right? Then in this case read write spinlock is useful. Isn't It?

So, If multiple threads require read access to the same data, there is no reason why they should not be able to execute simultaneously. Spinlocks don't differentiate between read and read/write access. Thus spinlocks do not exploit this potential parallelism. To do so, read-write locks are required.

Working of Read Write Spinlock

- When there is no thread in the critical section, any reader or writer thread can enter into critical section by taking respective read or write lock. But only one thread can enter into critical section.
- If reader thread is in critical section, the new reader thread can enter arbitrarily, but the writer thread cannot enter. Writer thread has to wait until all the reader thread finish their process.
- If writer thread is in critical section, no reader thread or writer thread can enter.
- If one or more reader threads are in critical section by taking it's lock, the writer thread can of course not enter the critical section, but the writer thread cannot prevent the entry of the subsequent read thread. He has to wait until the critical section has a reader thread. So this read write spinlock is giving importance to reader thread and not writer thread. If you want to give importance to writer thread than reader thread, then another lock is available in linux which is seqlock.

Where to use Read Write Spinlock?

1. If you are only reading the data then you take read lock
2. If you are writing then go for write lock

***Note** : Many people can hold a read lock, but a writer must be sole holder. Read-Write locks are more useful in scenarios where the architecture is clearly divided into reader and writers, with more number of reads.*

In Read Write spinlock multiple readers are permitted at same time but only one writer. (i.e) If a writer has the lock, no reader is allowed to enter the critical section. If only a reader has the lock, then multiple readers are permitted in the critical section.

Read Write SpinLock in Linux Kernel Device Driver

Initialize

We can initialize Read Write Spinlock in two ways.

1. Static Method
2. Dynamic Method

Static Method

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

```
DEFINE_RWLOCK(etx_rwlock);
```

The macro given above will create rwlock_t variable in the name of etx_rwlock.

Dynamic Method

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

```
rwlock_tetx_rwlock;

rwlock_init(&etx_rwlock);
```

You can use any one of the methods.

After initializing the read/write spinlock, there are several ways to use read/write spinlock to lock or unlock, based on where the read/write 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.

Read Lock

Lock:

read_lock(rwlock_t *lock)

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

Unlock:

read_unlock(rwlock_t *lock)

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

Write Lock

Lock:

write_lock(rwlock_t *lock)

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

Unlock:

write_unlock(rwlock_t *lock)

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

Example

```
//Thread 1
int thread_function1(void *pv)
{
while(!kthread_should_stop()) {
write_lock(&etx_rwlock);
etx_global_variable++;
write_unlock(&etx_rwlock);
msleep(1000);
   }
return 0;
}
```

```
//Thread 2
int thread_function2(void *pv)
{
while(!kthread_should_stop()) {
read_lock(&etx_rwlock);
printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n",
etx_global_variable);
read_unlock(&etx_rwlock);
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.

Read Lock

Lock:

read_lock_bh(rwlock_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:

read_unlock_bh(rwlock_t *lock)

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

Write Lock

Lock:

write_lock_bh(rwlock_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:

write_unlock_bh(rwlock_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()) {
write_lock_bh(&etx_rwlock);
etx_global_variable++;
write_unlock_bh(&etx_rwlock);
msleep(1000);
    }
return 0;
}
/*Tasklet Function*/
voidtasklet_fn(unsigned long arg)
{
read_lock_bh(&etx_rwlock);
printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
read_unlock_bh(&etx_rwlock);
}
```

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.

Read Lock

Lock:

read_lock_irq(rwlock_t *lock)

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

Unlock:

read_unlock_irq(rwlock_t *lock)

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

Write Lock

Lock:

write_lock_irq(rwlock_t *lock)

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

Unlock:

write_unlock_irq(rwlock_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)
{
write_lock_irq(&etx_rwlock);
etx_global_variable++;
write_unlock_irq(&etx_rwlock);
}

//Interrupt handler for IRQ 11.
staticirqreturn_tirq_handler(intirq,void *dev_id) {
read_lock_irq(&etx_rwlock);
printk(KERN_INFO "Executing ISR Function : %lu\n", etx_global_variable);
read_unlock_irq(&etx_rwlock);
        /*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 read_lock_irq( )/write_lock_irq() and read_unlock_irq( )/write_unlock_irq() then you can use this approach.

Read Lock

Lock:

read_lock_irqsave(rwlock_t *lock, unsigned long flags);

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

Unlock:

read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);

This will releases the read/write spinlock and restores the interrupts using the flags argument.

Write Lock

Lock:

write_lock_irqsave(rwlock_t *lock, unsigned long flags);

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

Unlock:

write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);

This will releases the read/write 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 read/write spinlock. After that it will release the read/write 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>
```

```
//Static method to initialize the read write spinlock
static DEFINE_RWLOCK(etx_rwlock);

//Dynamic method to initialize the read write spinlock
//rwlock_tetx_rwlock;

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()) {
write_lock(&etx_rwlock);
etx_global_variable++;
write_unlock(&etx_rwlock);
msleep(1000);
    }
return 0;
}

int thread_function2(void *pv)
{
```

```
while(!kthread_should_stop()) {
read_lock(&etx_rwlock);
printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n",
etx_global_variable);
read_unlock(&etx_rwlock);
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;
        }

        //Dynamic method to initialize the read write spinlock
        //rwlock_init(&etx_rwlock);

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 - RW Spinlock");
MODULE_VERSION("1.19");
```

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
```