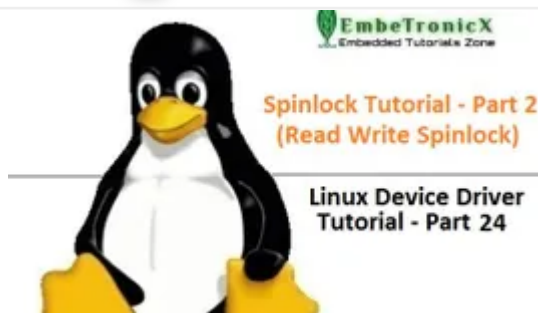




Sidebar▼

[Home](#) → [Tutorials](#) → [Linux](#) → [Device Drivers](#) → **Linux Device Driver Tutorial Part 24 - Read Write Spinlock in Linux Kernel (Spinlock Part 2)**

Device Drivers



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

This is the [Series on Linux Device Driver](#). The aim of this series is to provide easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 24 - Read-Write Spinlock in Linux Kernel (Spinlock Part 2).

2

Apple Music Turns 5 as It Continues Rivalry With Spotify

Post Contents [\[hide\]](#)

- 1 Prerequisites
- 2 Introduction
- 3 SpinLock
- 4 Read Write Spinlocks
 - 4.1 Working of Read Write Spinlock
 - 4.2 Where to use Read Write Spinlock?
- 5 Read-Write SpinLock in Linux Kernel Device Driver
 - 5.1 Initialize
 - 5.1.1 Static Method
 - 5.1.2 Dynamic Method
 - 5.2 Approach 1 (Locking between User context)
 - 5.2.1 Read Lock
 - 5.2.2 Write Lock
 - 5.2.3 Example
 - 5.3 Approach 2 (Locking between Bottom Halves)
 - 5.4 Approach 3 (Locking between User context and Bottom Halves)
 - 2 5.4.1 Read Lock
 - 5.4.2 Write Lock
 - 5.4.3 Example
 - 5.5 Approach 4 (Locking between Hard IRQ and Bottom Halves)
 - 5.5.1 Read Lock
 - 5.5.2 Write Lock

- 5.5.3 Example
- 5.6 Approach 5 (Alternative way of Approach 4)
 - 5.6.1 Read Lock
 - 5.6.2 Write Lock
- 5.7 Approach 6 (Locking between Hard IRQs)
- 6 Example Programming
 - 6.1 Driver Source Code
 - 6.2 MakeFile
 - 6.2.1 Share this:
 - 6.2.2 Like this:
 - 6.2.3 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 the 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 the read and the 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 a 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. The 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 a question. If you implement spinlock,

Question: Now we forgot the thread-1(writer thread). So we have 4 reader threads (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 hard 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, a 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 the 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 finishes their process.
- If the writer thread is in the critical section, no reader thread or writer thread can enter.
- If one or more reader threads are in critical section by taking its 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.

```
1 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.

```
1 rwlock_t etx_rwlock;  
2 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 the 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 the lock. It will unlock which is locked by above call.

Example

```
1 //Thread 1
2 int thread_function1(void *pv)
3 {
4     while(!kthread_should_stop()) {
5         write_lock(&etx_rwlock);
6         etx_global_variable++;
7         write_unlock(&etx_rwlock);
8         msleep(1000);
9     }
10    return 0;
11 }
12
13 //Thread 2
14 int thread_function2(void *pv)
15 {
16     while(!kthread_should_stop()) {
17         read_lock(&etx_rwlock);
18         printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n", etx_g
19         read_unlock(&etx_rwlock);
20         msleep(1000);
21     }
22    return 0;
23 }
```

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 the above call.

Example

```
1 //Thread
2 int thread_function(void *pv)
3 {
4     while(!kthread_should_stop()) {
5         write_lock_bh(&etx_rwlock);
6         etx_global_variable++;
7         write_unlock_bh(&etx_rwlock);
8         msleep(1000);
9     }
10    return 0;
11 }
12 /*Tasklet Function*/
13 void tasklet_fn(unsigned long arg)
14 {
15     read_lock_bh(&etx_rwlock);
16     printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
17     read_unlock_bh(&etx_rwlock);
18 }
```

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 the above call.

Example

```
1 /*Tasklet Function*/
2 void tasklet_fn(unsigned long arg)
3 {
4     write_lock_irq(&etx_rwlock);
5     etx_global_variable++;
6     write_unlock_irq(&etx_rwlock);
7 }
8
9 //Interrupt handler for IRQ 11.
10 static irqreturn_t irq_handler(int irq,void *dev_id) {
11     read_lock_irq(&etx_rwlock);
12     printk(KERN_INFO "Executing ISR Function : %lu\n", etx_global_variable);
13     read_unlock_irq(&etx_rwlock);
14 }
```

```
14     /*Scheduling Task to Tasklet*/  
15     tasklet_schedule(tasklet);  
16     return IRQ_HANDLED;  
17 }
```

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 release 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 release 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 access 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

```

1  #include <linux/kernel.h>
2  #include <linux/init.h>
3  #include <linux/module.h>
4  #include <linux/kdev_t.h>
5  #include <linux/fs.h>
6  #include <linux/cdev.h>
7  #include <linux/device.h>
8  #include <linux/slab.h>           //kmalloc()
9  #include <linux/uaccess.h>       //copy_to/from_user()
10 #include <linux/kthread.h>       //kernel threads
11 #include <linux/sched.h>         //task_struct
12 #include <linux/delay.h>
13
14
15 //Static method to initialize the read write spinlock
16 static DEFINE_RWLOCK(etx_rwlock);
17
18 //Dynamic method to initialize the read write spinlock
19 //rwlock_t etx_rwlock;
20
21 unsigned long etx_global_variable = 0;
22 dev_t dev = 0;
23 static struct class *dev_class;
24 static struct cdev etx_cdev;
25
26 static int __init etx_driver_init(void);
27 static void __exit etx_driver_exit(void);
28
29 static struct task_struct *etx_thread1;
30 static struct task_struct *etx_thread2;
31
32 /******* Driver Fuctions *****/
33 static int etx_open(struct inode *inode, struct file *file);
34 static int etx_release(struct inode *inode, struct file *file);
35 static ssize_t etx_read(struct file *filp,
36                        char __user *buf, size_t len, loff_t * off);
37 static ssize_t etx_write(struct file *filp,
38                         const char *buf, size_t len, loff_t * off);

```

```
39  /*****/
40
41  int thread_function1(void *pv);
42  int thread_function2(void *pv);
43
44  int thread_function1(void *pv)
45  {
46      while(!kthread_should_stop()) {
47          write_lock(&etx_rwlock);
48          etx_global_variable++;
49          write_unlock(&etx_rwlock);
50          msleep(1000);
51      }
52      return 0;
53  }
54
55  int thread_function2(void *pv)
56  {
57      while(!kthread_should_stop()) {
58          read_lock(&etx_rwlock);
59          printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n", etx_
60          read_unlock(&etx_rwlock);
61          msleep(1000);
62      }
63      return 0;
64  }
65
66  static struct file_operations fops =
67  {
68      .owner          = THIS_MODULE,
69      .read            = etx_read,
70      .write           = etx_write,
71      .open            = etx_open,
72      .release         = etx_release,
73  };
74
75  static int etx_open(struct inode *inode, struct file *file)
76  {
77      printk(KERN_INFO "Device File Opened...!!!\n");
78      return 0;
79  }
80
81  static int etx_release(struct inode *inode, struct file *file)
82  {
83      printk(KERN_INFO "Device File Closed...!!!\n");
84      return 0;
85  }
86
87  static ssize_t etx_read(struct file *filp,
88                          char __user *buf, size_t len, loff_t *off)
89  {
90      printk(KERN_INFO "Read function\n");
91      return 0;
92  }
93  }
94  static ssize_t etx_write(struct file *filp,
95                           const char __user *buf, size_t len, loff_t *off)
96  {
97      printk(KERN_INFO "Write Function\n");
98      return len;
99  }
100
101  static int  init etx driver init(void)
```

```

102 {
103     /*Allocating Major number*/
104     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
105         printk(KERN_INFO "Cannot allocate major number\n");
106         return -1;
107     }
108     printk(KERN_INFO "Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));
109
110     /*Creating cdev structure*/
111     cdev_init(&etx_cdev, &fops);
112
113     /*Adding character device to the system*/
114     if((cdev_add(&etx_cdev, dev, 1)) < 0){
115         printk(KERN_INFO "Cannot add the device to the system\n");
116         goto r_class;
117     }
118
119     /*Creating struct class*/
120     if((dev_class = class_create(THIS_MODULE, "etx_class")) == NULL){
121         printk(KERN_INFO "Cannot create the struct class\n");
122         goto r_class;
123     }
124
125     /*Creating device*/
126     if((device_create(dev_class, NULL, dev, NULL, "etx_device")) == NULL){
127         printk(KERN_INFO "Cannot create the Device \n");
128         goto r_device;
129     }
130
131
132     /* Creating Thread 1 */
133     etx_thread1 = kthread_run(thread_function1, NULL, "eTx Thread1");
134     if(etx_thread1) {
135         printk(KERN_ERR "Kthread1 Created Successfully...\n");
136     } else {
137         printk(KERN_ERR "Cannot create kthread1\n");
138         goto r_device;
139     }
140
141     /* Creating Thread 2 */
142     etx_thread2 = kthread_run(thread_function2, NULL, "eTx Thread2");
143     if(etx_thread2) {
144         printk(KERN_ERR "Kthread2 Created Successfully...\n");
145     } else {
146         printk(KERN_ERR "Cannot create kthread2\n");
147         goto r_device;
148     }
149
150     //Dynamic method to initialize the read write spinlock
151     //rwlock_init(&etx_rwlock);
152
153     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
154     return 0;
155
156
157 r_device:
158     class_destroy(dev_class);
159 r_class:
160     unregister_chrdev_region(dev, 1);
161     cdev_del(&etx_cdev);
162     return -1;
163 }
164

```

```
165 void __exit etx_driver_exit(void)
166 {
167     kthread_stop(etx_thread1);
168     kthread_stop(etx_thread2);
169     device_destroy(dev_class,dev);
170     class_destroy(dev_class);
171     cdev_del(&etx_cdev);
172     unregister_chrdev_region(dev, 1);
173     printk(KERN_INFO "Device Driver Remove...Done!!\n");
174 }
175
176 module_init(etx_driver_init);
177 module_exit(etx_driver_exit);
178
179 MODULE_LICENSE("GPL");
180 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
181 MODULE_DESCRIPTION("A simple device driver - RW Spinlock");
182 MODULE_VERSION("1.19");
```

MakeFile

```
1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean
```

In our [next tutorial](#), we will discuss How to send signals in the Linux device driver.

0

Article Rating

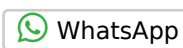


Share this:



Post

Tweet



Share



Like this:

Loading...

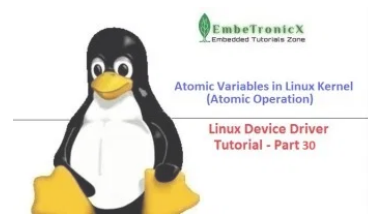
Related



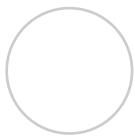
Linux Device Driver
Tutorial Part 23 – Spinlock
in Linux Kernel Part 1
In "Device Drivers"



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



Linux Device Driver
Tutorial Part 30 – Atomic
variable in Linux Device
Driver
In "Device Drivers"

☒ Subscribe ▼Connect with | [Login](#)*Join the discussion***B** *I* U        

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

2 COMMENTS

Oldest ▼

2





madhab

August 20, 2019 4:54 AM

i have small doubt here. As you mentioned in note point that "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." so i have created 5 threads. 1 is write thread and remaining 4 is read thread. so after 1st write thread remaining 4 threads should run and again write thread should come. but it is not happening. logs: Major = 246 Minor... [Read more »](#)



0



 Reply



madhab

August 20, 2019 4:54 AM

i have small doubt here. As you mentioned in note point that "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." so i have created 5 threads. 1 is write thread and remaining 4 is read thread. so after 1st write thread remaining 4 threads should run and again write thread should come. but it is not happening. logs: Major = 246 Minor... [Read more »](#)



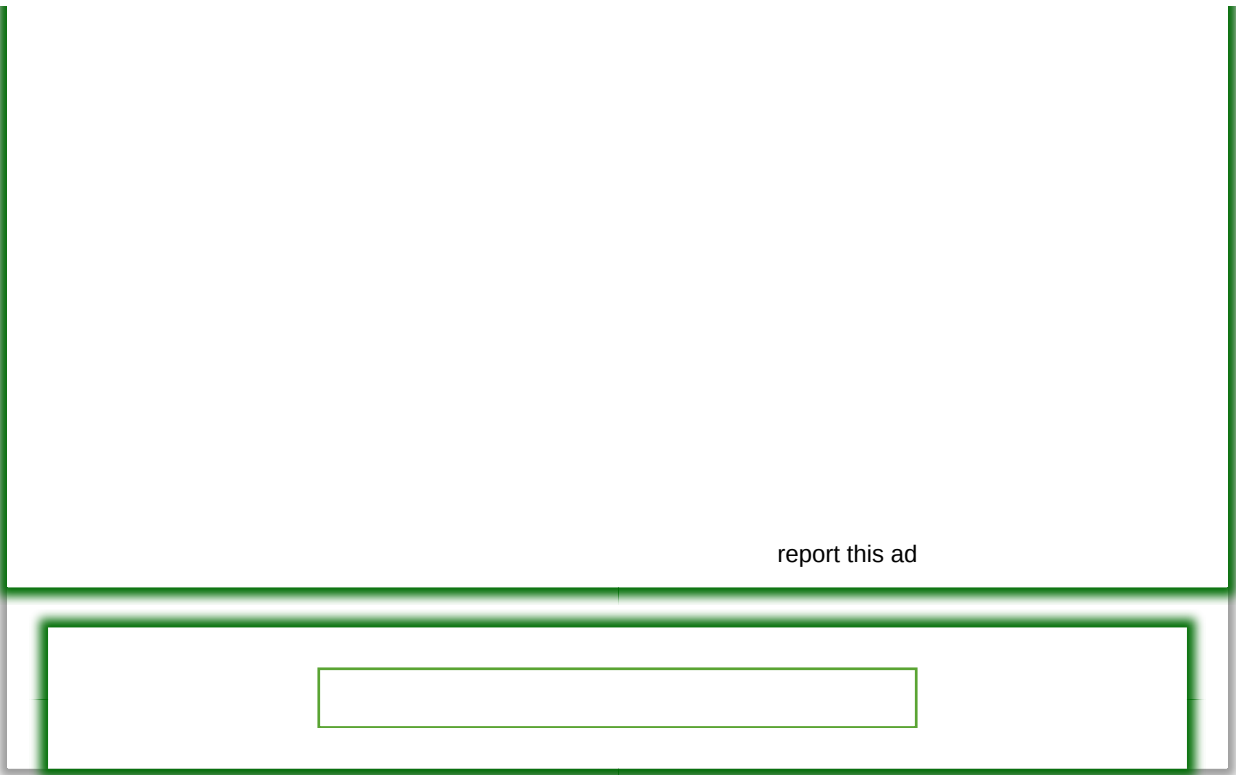
0



 Reply

2





2

