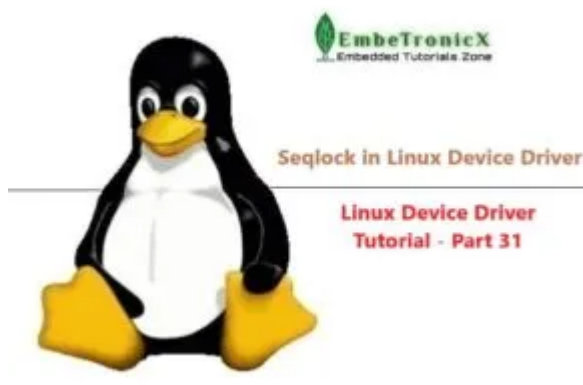


[Sidebar](#) ▼

[Home](#) → [Tutorials](#) → [Linux](#) → [Device Drivers](#) → **Linux Device Driver Tutorial Part 31 - Seqlock in Linux Kernel**

📁 Device Drivers



Linux Device Driver Tutorial Part 31 - Seqlock in Linux Kernel

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 31 - Seqlock in Linux Kernel.

Apple Music Turns 5 as It Continues Rivalry With Spotify

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

- 1 Prerequisites
- 2 Seqlock in Linux Kernel
 - 2.1 Working of seqlock
 - 2.2 When we have to use seqlock
- 3 Seqlock in Linux Kernel – API
 - 3.1 Init Seqlock
 - 3.1.1 Example
 - 3.2 Write operation
 - 3.2.1 Write Lock
 - 3.2.1.1 write_seqlock
 - 3.2.1.2 write_tryseqlock
 - 3.2.1.3 write_seqlock_irqsave
 - 3.2.1.4 write_seqlock_irq
 - 3.2.1.5 write_seqlock_bh
 - 3.2.2 Write unlock
 - 3.2.2.1 write_sequnlock
 - 3.2.2.2 write_sequnlock_irqrestore
 - 3.2.2.3 write_sequnlock_irq
 - 3.2.2.4 write_sequnlock_bh
 - 3.2.3 Example write operation
 - 3.3 Read operation
 - 3.3.1 read_seqbegin

- 3.3.2 read_seqbegin_irqsave
- 3.3.3 read_seqretry
- 3.3.4 read_seqretry_irqrestore
- 3.3.5 Example read operation
- 4 Example Programming
 - 4.1 Driver Source Code
 - 4.2 MakeFile
 - 4.3 Share this:
 - 4.4 Like this:
 - 4.5 Related

Prerequisites

In the below-mentioned posts, we are using spinlock and mutex, atomic variable for synchronization. I would recommend you explore that by using the below link.



Put your testing on auto-pilot

Build generic test sessions in few minutes with our high level web automation framework.

das

- [Mutex Tutorial in Linux Device Driver](#)
- [Spinlock in Linux Device Driver - Part 1](#)
- [Spinlock in Linux Device Driver - Part 2](#)
- [Atomic Variable in Linux Device Driver](#)

Sealock in Linux Kernel

In our previous tutorials, we have seen some locking methods like a [mutex](#), [spinlock](#), etc. In short, When you want to treat both write and reader operation equally, then you have to use [spinlock](#). In some situations, we may have to give importance to readers. In such a case, we can use [read write spinlock](#).

Likewise, is there any mechanism that gives importance to writers? Yeah, it is there in Linux. Seqlock is the one that gives importance to writers. We'll continue to see about seqlock.

The 2.5.60 kernel added a new type of lock called a seqlock. Seqlock is a short form of **sequential lock**. It is a reader-writer consistent mechanism which is giving importance to the writer. So this avoids the problem of writer starvation. You can read [here](#) how the writer is starving while using read write spinlock. So How this seqlock is giving importance to the writer? Is it really useful in all situations? We will see one by one.

Working of seqlock

1. When no one is in a critical section then one writer can enter into a critical section by acquiring its lock. Once it took its lock then the writer will increment the sequence number by one. Currently, the sequence number is an odd value. Once done with the writing, again it will increment the sequence number by one. Now the number is an even value. So, when the sequence number is an odd value, writing is happening. When the sequence number is an even value, writing has done. Only one writer thread will be allowed in the critical section. So

other writers will be waiting for the lock.

2. When the reader wants to read the data, first it will read the sequence number. If it is an even value, then it will go to a critical section and reads the data. If it is an odd value (the writer is writing something), the reader will wait for the writer to finish (the sequence number becomes an even number). The value of the sequence number while entering into the critical section is called an old sequence number.
3. After reading the data, again it will check the sequence number. If it is equal to the old sequence number, then everything is okay. Otherwise, it will repeat step 2 again. In this case, readers simply retry (using a loop) until they read the same even sequence number before and after. The reader never blocks, but it may have to retry if a write is in progress.
4. When only the reader is reading the data and no writer is in the critical section, any time one writer can enter into a critical section by taking lock without blocking. This means the writer cannot be blocked for the reader and the reader has to re-read the data when the writer is writing. This means seqlock is giving importance to a writer, not the reader (the reader may have to wait but not the writer).

When we have to use seqlock

We cannot use this seqlock in any situations like normal spinlock or mutex. Because this will not be effective in such situations other than the situations mentioned below.

- where read operations are more frequent than write.
- where write access is rare but must be fast.
- That data is simple (no pointers) that needs to be protected. Seqlocks generally cannot be used to protect data structures involving pointers, because the reader may be following a pointer that is invalid while the writer is changing the data structure.

Seqlock in Linux Kernel - API

Init Seqlock

This API is used to initialize the seqlock.

```
seqlock_init(seqlock_t *lock);
```

Example

```
1 #include <linux/seqlock.h>
2
3 seqlock_t lock;
4
5 seqlock_init(&lock);
```

Write operation

Before writing to the protected data, the writers must take exclusive access to enter the critical section. This write lock is implemented by using spinlock. Let's see the API used for that.

Write Lock

write_seqlock

```
void write_seqlock(seqlock_t *lock);
```

When you call this API, it locks the spinlock and increments the sequence number. Now you can access the protected data. Once you are done with that, you can release the lock using the below API.

write_tryseqlock

```
int write_tryseqlock(seqlock_t *lock);
```

This API won't wait for the lock. It will return non zero if it took the lock. Otherwise, it will return 0. That means some other writer is accessing the data.

write_seqlock_irqsave

```
void write_seqlock_irqsave(seqlock_t *lock, long flags);
```

This will save whether interrupts were ON or OFF in a **flags** word and grab the lock. This API is used in interrupt context.

write_seqlock_irq

```
void write_seqlock_irq(seqlock_t *lock);
```

This will disable interrupts on that CPU, and take the lock while writing. This API is used in interrupt context.

write_seqlock_bh

```
void write_seqlock_bh(seqlock_t *lock);
```

This is similar to **write_seqlock**, but when you try to write from the bottom halves you can use this call.

Write unlock

write_sequnlock

```
void write_sequnlock(seqlock_t *lock);
```

This API will increments the sequence number again and release the spinlock.

write_sequnlock_irqrestore

```
void write_sequnlock_irqrestore(seqlock_t *lock, long flags);
```

This will release the lock and restores the interrupts using the **flags** argument. This API is used in interrupt context.

write_sequnlock_irq

```
void write_sequnlock_irq(seqlock_t *lock);
```

This will release the lock and re-enable interrupts on that CPU, which is disabled by **write_seqlock_irq** call. This API is used in interrupt context.

write_sequnlock_bh

```
void write_sequnlock_bh(seqlock_t *lock);
```

This will be used from the bottom halves while reading.

Example write operation

This example is for locking between user context. Use other variants based on the context(bottom half or IRQ).

```
1 write_seqlock(&lock);  
2 /* Write data */  
3 write_sequnlock(&lock);
```

Read operation

There is no locking needed for reading the protected data. But we have to implement the below steps in our code.

1. Begin the read and get the initial sequence number.
2. Read the data.
3. Once the reading is done, compare the current sequence number with an initial sequence number. If the current sequence number is an odd value or current sequence number is not matching with the initial sequence number means writing is going on. So the reader has to retry, which means the reader has to again go to step 1 and do the process again.

Let's see the APIs used for reading.

read_seqbegin

```
unsigned int read_seqbegin(seqlock_t *lock);
```

This API will begin the read and return the sequence number. This API is used for the above step 1.

read_seqbegin_irqsave

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock, long  
flags);
```

This will save whether interrupts were ON or OFF in a **flags** word and return the sequence number.

read_seqretry

```
int read_seqretry(seqlock_t *lock, unsigned int seq_no);
```


This API will compare the current sequence number with the provided sequence number (argument 2). If the current sequence number is an odd value or current sequence number is not matching with the initial sequence number (argument 2) means writing is going on. So it will return 1. Otherwise, it will return 0.

read_seqretry_irqrestore

```
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int
                             seq_no, long flags);
```

This will restore the interrupt using flags, and work like **read_seqretry**.

Example read operation


This example is for the user context. Use other variants based on the context(bottom half or IRQ).Example reading snippet is given below.

```
1 unsigned int seq_no;
2
3 do {
4     seq_no = read_seqbegin(&lock);
5     /* Read the data */
6 } while ( read_seqretry(&lock, seq_no) );
```

Example Programming

This code snippet explains how to create two threads that access a global variable (**etx_gloabl_variable**). Thread 1 is for writing and Thread 2 is for reading. Before writing to the variable, the writer should take the seqlock. After that, it will release the seqlock. The reader will check the sequence number. If it is not a valid sequence number, then again the reader will retry.

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 #include <linux/seqlock.h>
14
15 //Seqlock variable
16 seqlock_t etx_seq_lock;
17
18 unsigned long etx_global_variable = 0;
19 dev_t dev = 0;
20 static struct class *dev_class;
21 static struct cdev etx_cdev;
22
23 static int __init etx_driver_init(void);
24 static void __exit etx_driver_exit(void);
25
26 static struct task_struct *etx_thread1;
27 static struct task_struct *etx_thread2;
28
29 /***** Driver Fuctions *****/
30 static int etx_open(struct inode *inode, struct file *file);
31 static int etx_release(struct inode *inode, struct file *file);
32 static ssize_t etx_read(struct file *filp,
33                         char __user *buf, size_t len, loff_t * off);
34 static ssize_t etx_write(struct file *filp,
35                          const char *buf, size_t len, loff_t * off);
36 /*****/
37
38 int thread_function1(void *pv);
39 int thread_function2(void *pv);
40
41 //Thread used for writing
42 int thread_function1(void *pv)
43 {
44     while(!kthread_should_stop()) {
45         write_seqlock(&etx_seq_lock);
46         etx_global_variable++;
47         write_sequnlock(&etx_seq_lock);
48         msleep(1000);
49     }
50     return 0;
51 }
52
53 //Thread used for reading
54 int thread_function2(void *pv)
55 {
56     unsigned int seq_no;
57     unsigned long read_value;
58     while(!kthread_should_stop()) {
59         do {
60             seq_no = read_seqbegin(&etx_seq_lock);
61             read_value = etx_global_variable;
62         } while (read_seqretry(&etx_seq_lock, seq_no));
63         printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n", read_value);
64         msleep(1000);
65     }
66     return 0;
67 }
68

```

```
69 static struct file_operations fops =
70 {
71     .owner          = THIS_MODULE,
72     .read            = etx_read,
73     .write           = etx_write,
74     .open            = etx_open,
75     .release         = etx_release,
76 };
77
78 static int etx_open(struct inode *inode, struct file *file)
79 {
80     printk(KERN_INFO "Device File Opened...!!!\n");
81     return 0;
82 }
83
84 static int etx_release(struct inode *inode, struct file *file)
85 {
86     printk(KERN_INFO "Device File Closed...!!!\n");
87     return 0;
88 }
89
90 static ssize_t etx_read(struct file *filp,
91                         char __user *buf, size_t len, loff_t *off)
92 {
93     printk(KERN_INFO "Read function\n");
94
95     return 0;
96 }
97
98 static ssize_t etx_write(struct file *filp,
99                          const char __user *buf, size_t len, loff_t *off)
100 {
101     printk(KERN_INFO "Write Function\n");
102     return len;
103 }
104
105 static int __init etx_driver_init(void)
106 {
107     /*Allocating Major number*/
108     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
109         printk(KERN_INFO "Cannot allocate major number\n");
110         return -1;
111     }
112     printk(KERN_INFO "Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));
113
114     /*Creating cdev structure*/
115     cdev_init(&etx_cdev, &fops);
116
117     /*Adding character device to the system*/
118     if((cdev_add(&etx_cdev, dev, 1)) < 0){
119         printk(KERN_INFO "Cannot add the device to the system\n");
120         goto r_class;
121     }
122
123     /*Creating struct class*/
124     if((dev_class = class_create(THIS_MODULE, "etx_class")) == NULL){
125         printk(KERN_INFO "Cannot create the struct class\n");
126         goto r_class;
127     }
128
129     /*Creating device*/
130     if((device_create(dev_class, NULL, dev, NULL, "etx_device")) == NULL){
131         printk(KERN_INFO "Cannot create the Device \n");
132         goto r_device;
133     }
134 }
```

```

132     }
133
134
135     /* Creating Thread 1 */
136     etx_thread1 = kthread_run(thread_function1, NULL, "eTx Thread1");
137     if(etx_thread1) {
138         printk(KERN_ERR "Kthread1 Created Successfully...\n");
139     } else {
140         printk(KERN_ERR "Cannot create kthread1\n");
141         goto r_device;
142     }
143
144     /* Creating Thread 2 */
145     etx_thread2 = kthread_run(thread_function2, NULL, "eTx Thread2");
146     if(etx_thread2) {
147         printk(KERN_ERR "Kthread2 Created Successfully...\n");
148     } else {
149         printk(KERN_ERR "Cannot create kthread2\n");
150         goto r_device;
151     }
152
153     //Initialize the seqlock
154     seqlock_init(&etx_seq_lock);
155
156     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
157     return 0;
158
159
160 r_device:
161     class_destroy(dev_class);
162 r_class:
163     unregister_chrdev_region(dev, 1);
164     cdev_del(&etx_cdev);
165     return -1;
166 }
167
168 void __exit etx_driver_exit(void)
169 {
170     kthread_stop(etx_thread1);
171     kthread_stop(etx_thread2);
172     device_destroy(dev_class, dev);
173     class_destroy(dev_class);
174     cdev_del(&etx_cdev);
175     unregister_chrdev_region(dev, 1);
176     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
177 }
178
179 module_init(etx_driver_init);
180 module_exit(etx_driver_exit);
181
182 MODULE_LICENSE("GPL");
183 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
184 MODULE_DESCRIPTION("A simple device driver - Seqlock");
185 MODULE_VERSION("1.28");

```



```

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 the misc device drivers in the Linux device driver.

5

Article Rating

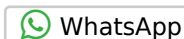


Share this:



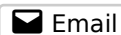
Post

Tweet



Share

2



Like this:

Loading...

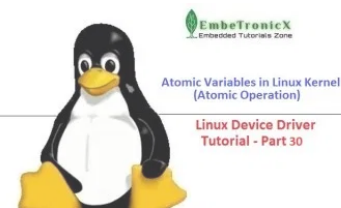
Related



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



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



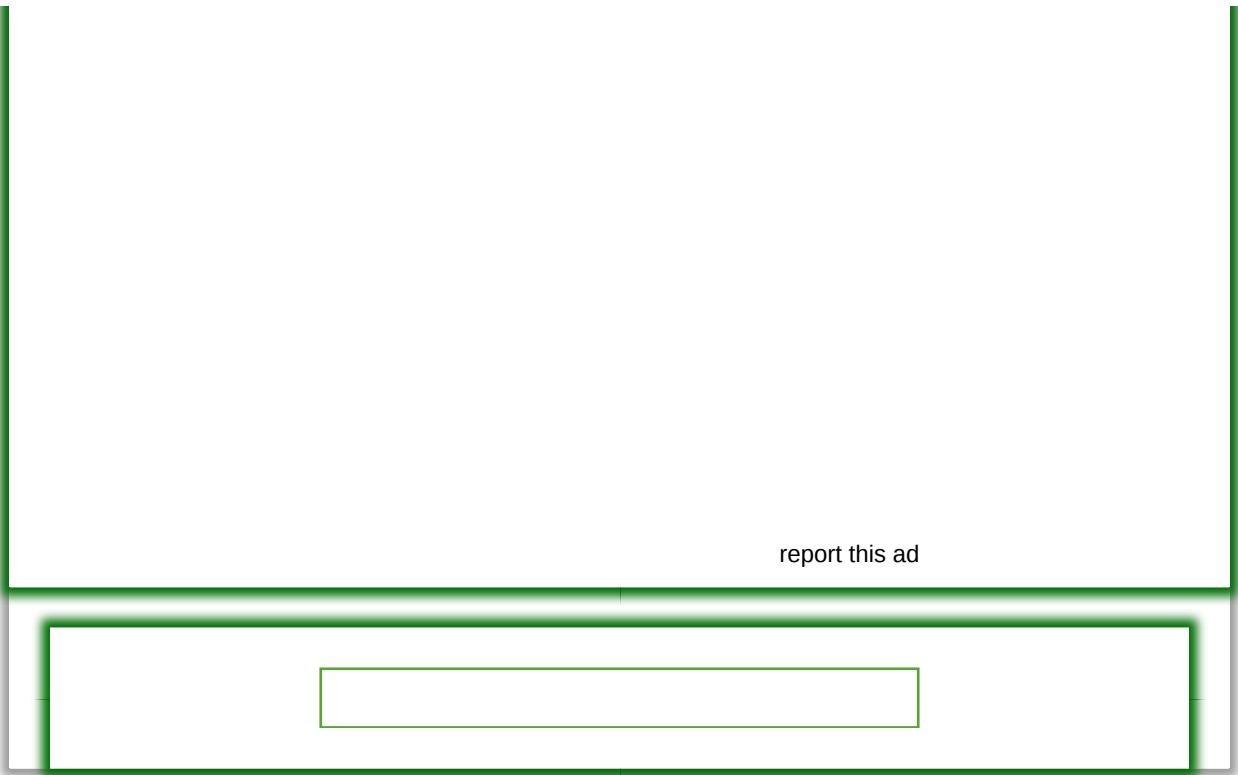
Linux Device Driver
Tutorial Part 30 – Atomic
variable in Linux Device
Driver
In "Device Drivers"

Connect with | [Login](#)



0 COMMENTS





2

