



Sidebar▼

[Home](#) → [Tutorials](#) → [Linux](#) → [Device Drivers](#) → **Linux Device Driver Tutorial Part 22 - Mutex in Linux Kernel**

Device Drivers



Linux Device Driver Tutorial Part 22 - Mutex 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 22 - Mutex in Linux Kernel.

3

Apple Music Turns 5 as It Continues Rivalry With Spotify

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

- 1 Prerequisites
- 2 Introduction
 - 2.1 Example Problems
- 3 Race Condition
- 4 Mutex
- 5 Mutex in Linux Kernel
 - 5.1 Initializing Mutex
 - 5.1.1 Static Method
 - 5.1.2 Dynamic Method
 - 5.1.2.1 Example
 - 5.2 Mutex Lock
 - 5.2.1 mutex_lock
 - 5.2.2 mutex_lock_interruptible
 - 5.2.3 mutex_trylock
 - 5.3 Mutex Unlock
 - 5.4 Mutex Status
- 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](#)

Introduction

Before getting to know about Mutex, let's take an analogy first.

Let us assume we have a car designed to accommodate only one person at any instance of time, while all the four doors of the car are open. But, if any more than one person tries to enter the car, a bomb will set off an explosion! (Quite a fancy car manufactured by EmbeTronicX!! 😊) Now that four doors of the car are open, the car is vulnerable to the explosion as anyone can enter through one of the four doors.

Now how we can solve this issue? Yes, correct. We can provide a key for the car. So, the person who wants to enter the car must have access to the key. If they don't have keys, they have to wait until that key is available or they can do some other work instead of waiting for the key.

Example Problems

Let's correlate the analogy above to what happens in our software. Let's explore situations like these through examples.

1. You have one SPI connection. What if one thread wants to write something into that SPI device and another thread wants to read from that SPI device at the same time?
2. You have one LED display. What if one thread is writing data at a different position of Display and another thread is writing different data at a different position of Display at the same time?
3. You have one Linked List. What if one thread wants to insert something into the list and another one wants to delete something on the same Linked List at the same time?

In all the scenarios above, the problem encountered is the same. At any given point two threads are accessing a single resource. Now we will

relate the above scenarios to our car example.

1. In SPI example, CAR = SPI, Person = Threads, Blast = Software Crash/Software may get wrong data.
2. In the LED display example, CAR = LED Display, Person = Threads, Blast = Display will show some unwanted junks.
3. In Linked List example, CAR = Linked List, Person = Threads, Blast = Software Crash/Software may get wrong data.

The cases above are termed as Race Condition.

Race Condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, we don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

To avoid race conditions, we have many ways like Semaphore, Spinlock, and Mutex. In this tutorial, we will concentrate on Mutex.

Mutex

A *mutex* is a mutual exclusion lock. Only one thread can hold the lock.

A mutex can be used to prevent the simultaneous execution of a block of code by multiple threads that are running in a single or multiple processes.

Mutex is used as a synchronization primitive in situations where a resource has to be shared by multiple threads simultaneously.

A mutex has ownership. The thread which locks a Mutex must also unlock it.

So whenever you are accessing a shared resource that time first we lock mutex and then access the shared resource. When we are finished with

that shared resource then we unlock the Mutex.

I hope you got some idea about Mutex. Now, let us look at Mutex in the Linux Kernel.

Mutex in Linux Kernel

Today most major operating systems employ multitasking. Multitasking is where multiple threads can execute in parallel and thereby utilizing the CPU in optimum way. Even though, multitasking is useful, if not implemented cautiously can lead to concurrency issues (Race condition), which can be very difficult to handle.

The actual mutex type (minus debugging fields) is quite simple:

```
1 struct mutex {  
2     atomic_t      count;  
3     spinlock_t    wait_lock;  
4     struct list_head wait_list;  
5 };
```

We will be using this structure for Mutex. Refer to [Linux/include/linux/mutex.h](#)

Initializing Mutex

We can initialize Mutex in two ways

1. Static Method
2. Dynamic Method

Static Method

This method will be useful while using global Mutex. This macro is defined below.

3

```
DEFINE_MUTEX(name);
```

This call *defines and initializes* a mutex. Refer to [Linux/include/linux/mutex.h](#)

Dynamic Method

This method will be useful for per-object mutexes when the mutex is just a field in a heap-allocated object. This macro is defined below.

```
mutex_init(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be initialized.

This call *initializes* already allocated mutex. Initialize the mutex to the unlocked state.

It is not allowed to initialize an already locked mutex.

Example

```
1 struct mutex etx_mutex;  
2 mutex_init(&etx_mutex);
```

Mutex Lock

Once a mutex has been initialized, it can be locked by anyone of them explained below.

mutex_lock

This is used to lock/acquire the mutex exclusively for the current task. If the mutex is not available, the current task will sleep until it acquires the Mutex.

The mutex must, later on, be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locked. **memset**-ing the mutex to 0 is not allowed.

```
void mutex_lock(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be acquired

mutex_lock_interruptible

Locks the mutex like **mutex_lock**, and returns 0 if the mutex has been acquired or sleep until the mutex becomes available. If a signal arrives while waiting for the lock then this function returns **-EINTR**.

```
int mutex_lock_interruptible(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be acquired

mutex_trylock

This will try to acquire the mutex, without waiting (will attempt to obtain the lock, but will not sleep). Returns 1 if the mutex has been acquired successfully, and 0 on contention.

```
int mutex_trylock(struct mutex *lock);
```

3

**Argument:**

struct mutex *lock – the mutex to be acquired

This function must not be used in interrupt context. The mutex must be released by the same task that acquired it.

Mutex Unlock

This is used to unlock/release a mutex that has been locked by a task previously.

This function must not be used in interrupt context. Unlocking of a not locked mutex is not allowed.

```
void mutex_unlock(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be released

Mutex Status

This function is used to check whether mutex has been locked or not.

```
int mutex_is_locked(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to check the status.

Returns 1 if the mutex is locked, 0 if unlocked.

Example Programming

This code snippet explains how to create two threads that access a global variable (**mtx_gloabl_variable**). So before accessing the variable, it should lock the mutex. After that, it will release the mutex.

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/mutex.h>
14
15 struct mutex etx_mutex;
16 unsigned long etx_global_variable = 0;
17
18 dev_t dev = 0;
19 static struct class *dev_class;
20 static struct cdev etx_cdev;
21
22 static int __init etx_driver_init(void);
23 static void __exit etx_driver_exit(void);
24
25 static struct task_struct *etx_thread1;
26 static struct task_struct *etx_thread2;
27
28 /***** Driver Fuctions *****/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
31 static ssize_t etx_read(struct file *filp,
32                         char __user *buf, size_t len, loff_t * off);
33 static ssize_t etx_write(struct file *filp,
34                          const char *buf, size_t len, loff_t * off);
35 /*****/
36
37 int thread_function1(void *pv);
38 int thread_function2(void *pv);
39
40 int thread_function1(void *pv)
41 {
42
43     while(!kthread_should_stop()) {
44         mutex_lock(&etx_mutex);
45         etx_global_variable++;
46         printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variab
47         mutex_unlock(&etx_mutex);
48         msleep(1000);
49     }
50     return 0;
51 }
52
53 int thread_function2(void *pv)
54 {
55     while(!kthread_should_stop()) {
56         mutex_lock(&etx_mutex);
57         etx_global_variable++;
58         printk(KERN_INFO "In EmbeTronicX Thread Function2 %lu\n", etx_global_variab
59         mutex_unlock(&etx_mutex);
60         msleep(1000);
61     }
62     return 0;
63 }
64
65 static struct file_operations fops =
66 {
67     .owner          = THIS_MODULE,
68     .read           = etx_read,

```

```
69     .write          = etx_write,
70     .open           = etx_open,
71     .release        = etx_release,
72 };
73
74 static int etx_open(struct inode *inode, struct file *file)
75 {
76     printk(KERN_INFO "Device File Opened...!!!\n");
77     return 0;
78 }
79
80 static int etx_release(struct inode *inode, struct file *file)
81 {
82     printk(KERN_INFO "Device File Closed...!!!\n");
83     return 0;
84 }
85
86 static ssize_t etx_read(struct file *filp,
87                         char __user *buf, size_t len, loff_t *off)
88 {
89     printk(KERN_INFO "Read function\n");
90
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     mutex_init(&etx_mutex);
132 }
```

```

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     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
151     return 0;
152
153
154 r_device:
155     class_destroy(dev_class);
156 r_class:
157     unregister_chrdev_region(dev, 1);
158     cdev_del(&etx_cdev);
159     return -1;
160 }
161
162 void __exit etx_driver_exit(void)
163 {
164     kthread_stop(etx_thread1);
165     kthread_stop(etx_thread2);
166     device_destroy(dev_class, dev);
167     class_destroy(dev_class);
168     cdev_del(&etx_cdev);
169     unregister_chrdev_region(dev, 1);
170     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
171 }
172
173 module_init(etx_driver_init);
174 module_exit(etx_driver_exit);
175
176 MODULE_LICENSE("GPL");
177 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
178 MODULE_DESCRIPTION("A simple device driver - Mutex");
179 MODULE_VERSION("1.17");

```

3 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 spinlock in the Linux device driver.

0

Article Rating

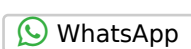


Share this:



Post

Tweet



Share

9



Like this:

Loading...

Related



[Linux Device Driver
Tutorial Part 23 – Spinlock
in Linux Kernel Part 1](#)
In "Device Drivers"



[Linux Device Driver
Tutorial Part 29 –
EXPORT_SYMBOL in Linux
Device Driver](#)
In "Device Drivers"

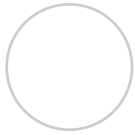


[Linux Device Driver
Tutorial Part 31 – Seqlock
in Linux Kernel](#)
In "Device Drivers"

3

 ³Subscribe ▼

Connect with | [Login](#)



Join the discussion

B *I* U    “ ” </>  {} [+]



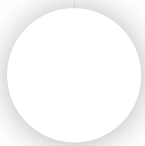
This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

3 COMMENTS



Oldest ▼

3





neel

July 15, 2019 4:43 AM

mutex_init should be done before threads are spawned

Loading...




0



Reply

EmbeTronicX

 Reply to neel

July 15, 2019 5:21 AM

Hi Neel,

Thanks for your input. We have updated. Keep supporting us.

Loading...



0



Reply



Kannan

May 2, 2020 5:37 PM

Thanks for your post.

Loading...



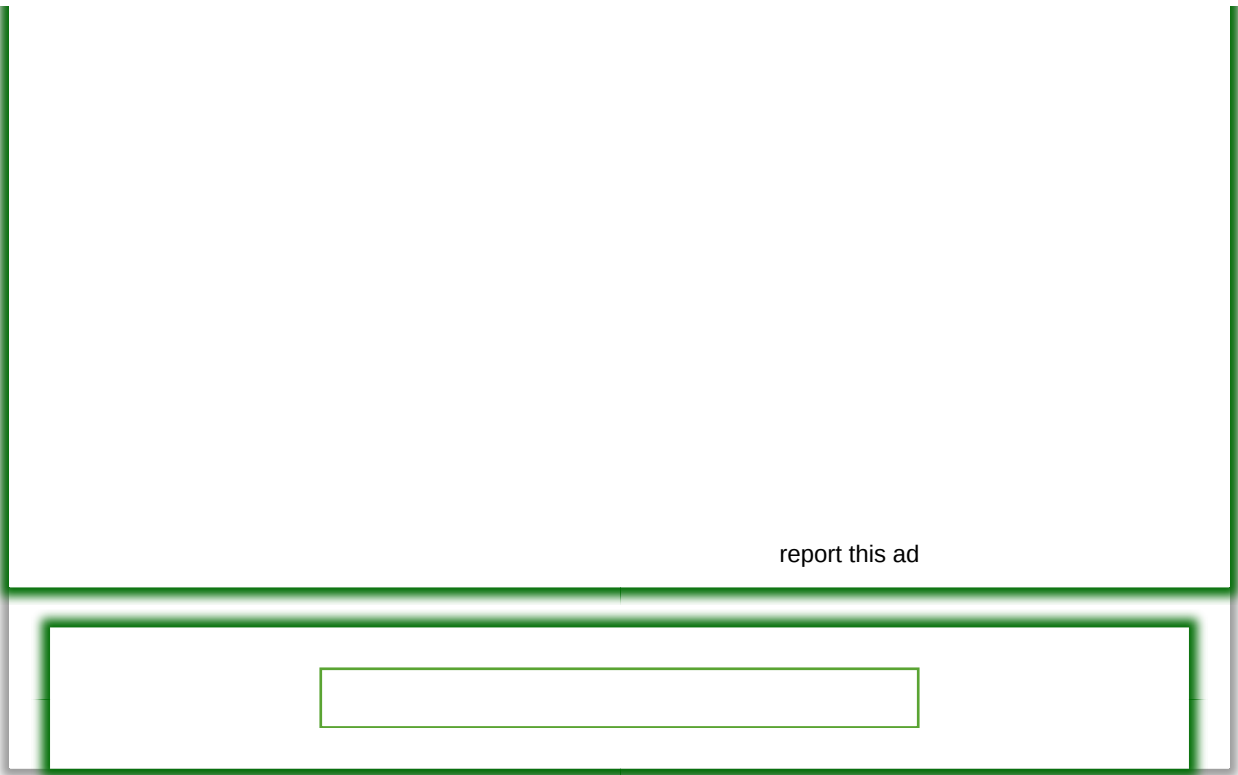
0



Reply

3





3

