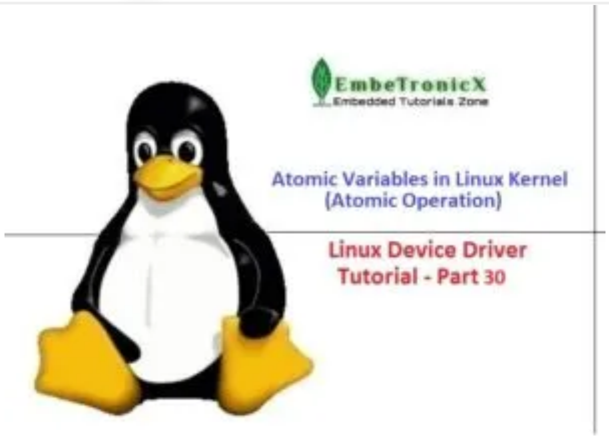
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 30 – Atomic variable in Linux Device Driver**

Device Drivers


EmbeTronicX
Embedded Tutorials Zone
Atomic Variables in Linux Kernel (Atomic Operation)
Linux Device Driver Tutorial - Part 30

Linux Device Driver Tutorial Part 30 – Atomic variable in Linux Device Driver

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 30 – atomic variable in Linux Device Driver (atomic operations).

1

Post Contents [hide]

1

atomic variables in Linux Device Driver

Prerequisites

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

- Mutex Tutorial in Linux Device Driver
- Spinlock in Linux Device Driver – Part 1
- Spinlock in Linux Device Driver – Part 2

Introduction

Before looking into atomic variables, we will see one example.

I have one integer or long variable named **`etx_global_variable`** which is shared between two threads. Two threads are just incrementing the variable like below.

Thread 1:

```
etx_global_variable++; //Accessing the variable
```

Thread 2:

```
etx_global_variable++; //Accessing the variable
```

1

Now we will see how it is incrementing internally in each instruction when both threads are running concurrently. Assume the initial value of **`etx_global_variable`** is 0.

Thread 1	Thread 2

get the value of **etx_global_variable** from memory (0)	get the value of **etx_global_variable** from memory (0)
increment **etx_global_variable** (0 -> 1)	—
—	increment **etx_global_variable** (0 -> 1)
write back the **etx_global_variable** value to memory (1)	—
—	write back the **etx_global_variable** value to memory (1)

Now the value of **etx_global_variable** is 1 after the two threads are processed. Are we expecting the same value which is 1? Nope. We are expecting the value of **etx_global_variable** should be 2. It is not running as expected because the global variable is sharing between two concurrent threads. So we need to implement synchronization because both the threads are accessing (writing/reading) the variable. We can implement synchronization like below using locks.

Thread 1:

```
lock(); //spinlock or mutex
etx_global_variable++; //Accessing the variable
unlock();
```

Thread 2:

```
lock(); //spinlock or mutex
etx_global_variable++; //Accessing the variable
unlock();
```

After this synchronization, we will see how it is incrementing internally when both threads are running concurrently. Assume the initial value of **etx_global_variable** is 0.

Thread 1	**Thread 2**

lock()	—
get the value of **`etx_global_variable`** from memory (0)	lock() (it will be stuck here because the lock is already taken by thread 1)
increment **`etx_global_variable`** (0 -> 1)	—
write back the **`etx_global_variable`** value to memory (1)	—
unlock()	—
—	get the value of **`etx_global_variable`** from memory (1)
—	increment **`etx_global_variable`** (1 -> 2)
—	write back the **`etx_global_variable`** value to memory (2)
—	unlock()

This will be the one possibility to run. Another possibility is mentioned below.

Thread 1	**Thread 2**
—	lock()
lock() (it will be stuck here because the lock is already taken by thread 2)	get the value of **`etx_global_variable`** from memory (0)
—	increment **`etx_global_variable`** (0 -> 1)
—	write back the **`etx_global_variable`** value to memory (1)
—	unlock()

1

get the value of `etx_global_variable` from memory (1)	—
increment `etx_global_variable` (1 -> 2)	—
write back the `etx_global_variable` value to memory (2)	—
unlock()	—

Great. That's all. Now we are getting 2 in the two methods mentioned above. But have anyone thought anytime that, why these things are required for a single variable? Why don't we have an alternate method for a single variable? Yes, obviously we have an alternate mechanism for integer and long variables. That is **the atomic operation**. If you use mutex/spinlock for just a single variable, it will add overhead. In this tutorial, we gonna see the atomic variable, atomic operation, and its usage.

atomic variables

The read, write and arithmetic operations on the atomic variables will be done in one instruction without interrupt.

So again we will take the same example mentioned above to explain the atomic variable operations. When we use the atomic method, that will work like below.

Thread 1	Thread 2
get, increment and store `etx_global_variable` (0 -> 1)	—
—	get, increment and store `etx_global_variable` (1 -> 2)

and another possibility will be,

Thread 1	Thread 2

—	get, increment and store **etx_global_variable** (0 -> 1)
get, increment and store **etx_global_variable** (1 -> 2)	—

So the extra locking mechanism is not required when we are using atomic variables since the operation is happening in one machine instruction.

An **atomic_t** holds an **int** value and **atomic64_t** holds the **long** value on all supported architectures.

In Linux Kernel Version 2.6, the atomic variable has defined below.

```
typedef struct {
volatile int counter;
} atomic_t;

#ifdef CONFIG_64BIT
typedef struct {
volatile long counter;
} atomic64_t;
#endif
```

Then later, they have removed **volatile** and defined as below.

```
typedef struct {
int counter;
} atomic_t;

#ifdef CONFIG_64BIT
typedef struct {
long counter;
} atomic64_t;
#endif
```

You can read here why they have removed volatile.

Types of atomic variables

Two different atomic variables are there.

- Atomic variables who operates on Integers
- Atomic variables who operates on Individual Bits

Atomic Integer Operations

When we are doing atomic operations, that variable should be created using **atomic_t** or **atomic64_t**. So we have separate special functions for reading, writing, and arithmetic operations, and those are explained below.

The declarations are needed to use the atomic integer operations are in **<asm/atomic.h>**. Some architectures provide additional methods that are unique to that architecture, but all architectures provide at least a minimum set of operations that are used throughout the kernel. When you write kernel code, you can ensure that these operations are correctly implemented on all architectures.

Creating atomic variables

```
atomic_t etx_global_variable; /* define etx_global_variable */

or

atomic_t etx_global_variable = ATOMIC_INIT(0); /* define etx_global_variable and init
```

Reading atomic variables

atomic_read

This function atomically reads the value of the given atomic variable.

int atomic_read(atomic_t *v);

where,

v – pointer of type atomic_t

Return: It returns the integer value.

Other operations on atomic variables

atomic_set

This function atomically sets the value to the atomic variable.

```
void atomic_set(atomic_t *v, int i);
```

where,
v – the pointer of type atomic_t
i – the value to be set to v

atomic_add

This function atomically adds the value to the atomic variable.

```
void atomic_add(int i, atomic_t *v);
```

where,
i – the value to be added to v
v – the pointer of type atomic_t

atomic_sub

This function atomically subtracts the value from the atomic variable.

```
void atomic_sub(int i, atomic_t *v);
```

where,
i – the value to be subtracted from v
v – the pointer of type atomic_t

atomic_inc

This function atomically increments the value of the atomic variable by

1.

```
void atomic_inc (atomic_t *v);
```

where,
v – the pointer of type atomic_t

atomic_dec

This function atomically decrements the value of the atomic variable by 1.

```
void atomic_dec (atomic_t *v);
```

where,
v – the pointer of type atomic_t

atomic_sub_and_test

This function atomically subtract the value from the atomic variable and test the result is zero or not.

```
void atomic_sub_and_test(int i, atomic_t *v);
```

where,
i – the value to be subtracted from v
v – the pointer of type atomic_t

Return: It returns true if the result is zero, or false for all other cases.

atomic_dec_and_test

This function atomically decrements the value of the atomic variable by 1 and test the result is zero or not.

1

```
void atomic_dec_and_test(atomic_t *v);
```

where,
v – the pointer of type atomic_t

Return: It returns true if the result is zero, or false for all other cases.

atomic_inc_and_test

This function atomically increments the value of the atomic variable by 1 and test the result is zero or not.

```
void atomic_inc_and_test(atomic_t *v);
```

where,
v – the pointer of type atomic_t

Return: It returns true if the result is zero, or false for all other cases.

atomic_add_negative

This function atomically adds the value to the atomic variable and test the result is negative or not.

```
void atomic_add_negative(int i, atomic_t *v);
```

where,
i – the value to be added to v
v – the pointer of type atomic_t

Return: It returns true if the result is negative, or false for all other cases.

atomic_add_return

This function atomically adds the value to the atomic variable and returns the value.

```
void atomic_add_return(int i, atomic_t *v);
```

1

where,
i – the value to be added to v
v – the pointer of type atomic_t

Return : It returns true if the result the value (i + v).

Like this, other functions also there. Those are,

Function	Description
`int atomic_sub_return(int i, atomic_t *v)`	Atomically subtract **i** from **v** and return the result
`int atomic_inc_return(int i, atomic_t *v)`	Atomically increments **v** by one and return the result
`int atomic_dec_return(int i, atomic_t *v)`	Atomically decrements **v** by one and return the result

atomic_add_unless

This function atomically adds the value to the atomic variable unless the number is a given value.

```
atomic_add_unless (atomic_t *v, int a, int u);
```

where,
v – the pointer of type atomic_t

a – the amount to add to v…

u – …unless v is equal to u.

Return: It returns non-zero if v was not u, and zero otherwise.

There is a 64-bit version also available. Unlike **atomic_t**, that will be operates on 64 bits. This 64-bit version also have similar function like above, the only change is we have to use 64.

Example

```
atomic64_t etx_global_variable = ATOMIC64_INIT(0);
long atomic64_read(atomic64_t *v);
void atomic64_set(atomic64_t *v, int i);
void atomic64_add(int i, atomic64_t *v);
void atomic64_sub(int i, atomic64_t *v);
void atomic64_inc(atomic64_t *v);
void atomic64_dec(atomic64_t *v);
int atomic64_sub_and_test(int i, atomic64_t *v);
int atomic64_add_negative(int i, atomic64_t *v);
long atomic64_add_return(int i, atomic64_t *v);
long atomic64_sub_return(int i, atomic64_t *v);
long atomic64_inc_return(int i, atomic64_t *v);
long atomic64_dec_return(int i, atomic64_t *v);
int atomic64_dec_and_test(atomic64_t *v);
int atomic64_inc_and_test(atomic64_t *v);
```

But all the operations are the same as **atomic_t**.

Atomic Bitwise Operations

Atomic_t is good when we are working on integer arithmetic. But when it comes to bitwise atomic operation, it doesn't work well. So kernel offers separate functions to achieve that. Atomic bit operations are very fast. These functions are architecture dependent and are declared in **<asm/bitops.h>**.

These bitwise functions operate on a generic pointer. So **atomic_t** / **atomic64_t** is not required. So we can work with a pointer to whatever data we want.

The below functions are available for atomic bit operations.

Function	Description
`void set_bit(int nr, void *addr)`	Atomically set the `nr`-th bit starting from `addr`
`void clear_bit(int nr, void *addr)`	Atomically clear the `nr`-th bit starting from `addr`
`void change_bit(int nr, void *addr)`	Atomically flip the value of the `nr`-th bit starting from `addr`
`int test_and_set_bit(int nr, void *addr)`	Atomically set the `nr`-th bit starting from `addr` and return the previous value
`int test_and_clear_bit(int nr, void *addr)`	Atomically clear the `nr`-th bit starting from `addr` and return the previous value

1

`int test_and_change_bit(int nr, void *addr)`	Atomically flip the `nr`-th bit starting from `addr` and return the previous value
`int test_bit(int nr, void *addr)`	Atomically return the value of the `nr`-th bit starting from `addr`
`int find_first_zero_bit(unsigned long *addr, unsigned int size)`	Atomically returns the bit-number of the first zero bit, not the number of the byte containing a bit
`int find_first_bit(unsigned long *addr, unsigned int size)`	Atomically returns the bit-number of the first set bit, not the number of the byte containing a bit

And also non-atomic bit operations also available. What is the use of that when we have atomic bit operations? When we have code which is already locked by **mutex/spinlock** then we can go for this non-atomic version. This might be faster in that case. The below functions are available for non-atomic bit operations.

Function	**Description**
`void _set_bit(int nr, void *addr)`	Non-atomically set the `nr`-th bit starting from `addr`
`void _clear_bit(int nr, void *addr)`	Non-atomically clear the `nr`-th bit starting from `addr`
`void _change_bit(int nr, void *addr)`	Non-atomically flip the value of the `nr`-th bit starting from `addr`
`int _test_and_set_bit(int nr, void *addr)`	Non-atomically set the `nr`-th bit starting from `addr` and return the previous value
`int _test_and_clear_bit(int nr, void *addr)`	Non-atomically clear the `nr`-th bit starting from `addr` and return the previous value
`int _test_and_change_bit(int nr, void *addr)`	Non-atomically flip the `nr`-th bit starting from `addr` and return the

	previous value
int _test_bit(int nr, void *addr)	Non-atomically return the value of the **nr**-th bit starting from **addr**

Example Programming

In this program, we have two threads called **thread_function1** and **thread_function2**. Both will be accessing the atomic variables.

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>

atomic_t etx_global_variable = ATOMIC_INIT(0);      //Atomic integer variable
unsigned int etc_bit_check = 0;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

static struct task_struct *etx_thread1;
static struct task_struct *etx_thread2;

/*************** Driver Fuctions **********************/
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len,loff_t * off);
static ssize_t etx_write(struct file *filp,
                const char *buf, size_t len, loff_t * off);
 /******************************************************/

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

int thread_function1(void *pv)
{
    unsigned int prev_value = 0;

    while(!kthread_should_stop()) {
        atomic_inc(&etx_global_variable);
        prev_value = test_and_change_bit(1, (void*)&etc_bit_check);
        printk(KERN_INFO "Function1 [value : %u] [bit:%u]\n", atomic_read(&etx_glob
        msleep(1000);
    }
    return 0;
}

int thread_function2(void *pv)
{
    unsigned int prev_value = 0;
    while(!kthread_should_stop()) {
        atomic_inc(&etx_global_variable);
        prev_value = test_and_change_bit(1,(void*) &etc_bit_check);
        printk(KERN_INFO "Function2 [value : %u] [bit:%u]\n", atomic_read(&etx_glob
        msleep(1000);
    }
    return 0;
}

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

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

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

static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Read function\n");

        return 0;
}
static ssize_t etx_write(struct file *filp,
                const char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Write Function\n");
        return len;
}

static int __init etx_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));

        /*Creating cdev 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");
            goto r_class;
        }

        /*Creating struct class*/
        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
            printk(KERN_INFO "Cannot create the struct class\n");
            goto r_class;
        }

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

        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 - Atomic Variables");
MODULE_VERSION("1.27");
```

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
```

In our next tutorial, we can discuss Seqlock.

0

Article Rating

☆☆☆☆☆

Share this:

Share 7 Post Tweet SHARE Print WhatsApp Share

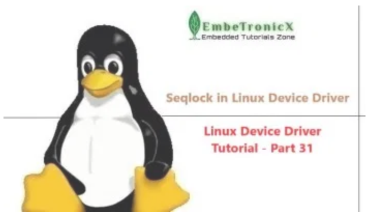
1 Email Telegram More

Like this:

Loading...

Related

Linux Device Driver



Linux Device Driver

Linux Device Driver

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

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

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

1

Subscribe

Connect with | Login

Join the discussion

B *I* U ~~S~~ {} [+]

This site uses Akismet to reduce spam. [Learn how your comment data is processed](https://akismet.com/privacy/).

1 COMMENT

Oldest

Himanshu

March 17, 2019 11:44 PM

Hello sir,

This is the best explanation i got on Atomic variable. Just need to know when you will upload i2c device driver tutorial for linux.

Loading...

0 Reply

1

report this ad

1