# Part 30 – Atomic variable in Linux Device Driver

[ <https://embetronicx.com/tutorials/linux/device-drivers/atomic-variables-atomic-operation/> ]

## 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 |

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. Its not running as expected because 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 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 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() |
| 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 this things are required for single variable? Why don’t we have alternate method for single variable? Yes, obviously we have alternate mechanism for integer and long variables. That is **atomic operation**. If you use mutex/spinlock for just single variable, it will add overhead. In this tutorial we gonna see that 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 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 extra locking mechanism is not required when we are using atomic variables, since 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, atomic variable has defined like 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 like below.

typedef struct {

int counter;

} atomic\_t;

#ifdef CONFIG\_64BIT

typedef struct {

long counter;

} atomic64\_t;

#endif

You can read [here](https://github.com/spotify/linux/blob/master/Documentation/volatile-considered-harmful.txt) 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 initialize it to zero \*/

#### Reading atomic variables

##### atomic\_read

This function atomically reads the value 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 atomic variable.

**void atomic\_set(atomic\_t \*v, int i);**

where,  
**v** – pointer of type atomic\_t  
**i** – the value to be set to v

##### atomic\_add

This function atomically adds the value to atomic variable.

**void atomic\_add(int i, atomic\_t \*v);**

where,  
**i**– the value to be added to v  
**v** – pointer of type atomic\_t

##### atomic\_sub

This function atomically subtract the value from atomic variable.

**void atomic\_sub(int i, atomic\_t \*v);**

where,  
**i** – the value to be subtract from v  
**v** – 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** – 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**– pointer of type atomic\_t

##### atomic\_sub\_and\_test

This function atomically subtract the value from 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 subtract from v  
**v** – 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.

**void atomic\_dec\_and\_test(atomic\_t \*v);**

where,  
**v** – 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** – 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 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**– 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 atomic variable and return the value.

**void atomic\_add\_return(int i, atomic\_t \*v);**

where,  
**i** – the value to be added to v  
**v** – 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 atomic variable unless the number is a given value.

**atomic\_add\_unless (atomic\_t \*v, int a, int u);**

where,  
**v** – 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 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);

### Atomic Bitwise Operations

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

These bitwise functions operates on 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 |
| **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\_global\_variable), prev\_value);

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\_global\_variable), prev\_value);

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