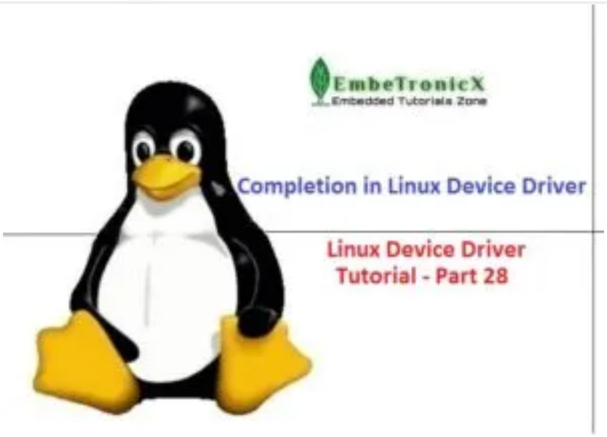
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 28 – Completion in Linux Device Driver**

Device Drivers



Linux Device Driver Tutorial Part 28 – Completion 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 28 – Completion in Linux Device Driver.

1

X

Post Contents [hide]

1

X

Prerequisites

In the example section, I had used kthread to explain this completion. 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. Waitqueue Tutorial in Linux Kernel

Completion

process finished then we need to wake up that thread which is sleeping. We can do this by using completion without race conditions.

These completions are a synchronization mechanism which is a good method in the above situation mentioned rather than using improper locks/semaphores and busy-loops.

Completion in Linux Device Driver

In the Linux kernel, Completions are developed by using waitqueue.

The advantage of using completions is that they have a well defined, focused purpose which makes it very easy to see the intent of the code, but they also result in more efficient code as all threads can continue execution until the result is actually needed, and both the waiting and the signaling is highly efficient using low-level scheduler sleep/wakeup facilities.

There are 5 important steps in Completions.

1. Initializing Completion
2. Re-Initializing Completion
3. Waiting for completion (The code is waiting and sleeping for

We have to include **<linux/completion.h>** and creating a variable of type **struct completion,** which has only two fields:

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

Where, **wait** is the waitqueue to place tasks on for waiting (if any). **done** is the completion flag for indicating whether it's completed or not.

We can create the struct variable in two ways.

1. Static Method
2. Dynamic Method

You can use any one of the methods.

Static Method

DECLARE_COMPLETION(data_read_done);

Where the "**data_read_done**" is the name of the struct which is going to create statically.

Dynamic Method

init_completion (struct completion * x);

Where, **x** – completion structure that is to be initialized

Example:

```
struct completion data_read_done;

init_completion(&data_read_done);
```

In this **init_completion** call we initialize the waitqueue and set **done** to

X

Where, **x** – completion structure that is to be reinitialized

Example:

```
reinit_completion(&data_read_done);
```

This function should be used to reinitialize a completion structure so it can be reused. This is especially important after **complete_all** is used. This simply resets the ->done field to 0 (“not done”), without touching the waitqueue. Callers of this function must make sure that there are no racy wait_for_completion() calls going on in parallel.

Waiting for completion

For a thread to wait for some concurrent activity to finish, it calls anyone of the function based on the use case.

wait_for_completion

This is used to make the function waits for completion of a task.

```
void wait_for_completion (struct completion * x);
```

Where, **x** – holds the state of this particular completion

This waits to be signaled for completion of a specific task. It is NOT interruptible and there is no timeout.

Example:

```
wait_for_completion (&data_read_done);
```

Note that `wait_for_completion()` is calling `spin_lock_irq()/spin_unlock_irq()`, so it can only be called safely when you know that interrupts are enabled. Calling it from IRQs-off atomic contexts will result in hard-to-detect spurious enabling of

X

wait_for_completion_timeout

This is used to make the function waits for completion of a task with a timeout. Timeouts are preferably calculated with `msecs_to_jiffies() or usecs_to_jiffies()`, to make the code largely HZ-invariant.

```
unsigned long wait_for_completion_timeout (struct completion * x, unsigned long timeout);
```

where, `x` – holds the state of this particular completion

`timeout` – timeout value in jiffies

This waits for either completion of a specific task to be signaled or for a specified timeout to expire. The timeout is in jiffies. It is not interruptible.

It **returns 0** if timed out, and **positive** (at least 1, or a number of jiffies left till timeout) if completed.

Example:

```
wait_for_completion_timeout (&data_read_done);
```

wait_for_completion_interruptible

This waits for completion of a specific task to be signaled. It is interruptible.

```
int wait_for_completion_interruptible (struct completion * x);
```

where, `x` – holds the state of this particular completion

It return **-ERESTARTSYS** if interrupted, **0** if completed.

1

X

wait_for_completion_interruptible_timeout

This waits for either completion of a specific task to be signaled or for a specified timeout to expire. It is interruptible. The timeout is in jiffies. Timeouts are preferably calculated with `msecs_to_jiffies()` or `usecs_to_jiffies()`, to make the code largely HZ-invariant.

```
long wait_for_completion_interruptible_timeout (struct completion
                  * x, unsigned long timeout);
```

where, `x` – holds the state of this particular completion

`timeout` – timeout value in jiffies

It returns **-ERESTARTSYS** if interrupted, **0** if timed out, positive (at least 1, or a number of jiffies left till timeout) if completed.

wait_for_completion_killable

This waits to be signaled for completion of a specific task. It can be interrupted by a kill signal.

```
int wait_for_completion_killable (struct completion * x);
```

X

wait_for_completion_killable_timeout

This waits for either completion of a specific task to be signaled or for a specified timeout to expire. It can be interrupted by a kill signal. The timeout is in jiffies. Timeouts are preferably calculated with msecs_to_jiffies() or usecs_to_jiffies(), to make the code largely HZ-invariant.

```
long wait_for_completion_killable_timeout (struct completion * x, unsigned long timeout);
```

where, `x` – holds the state of this particular completion

`timeout` – timeout value in jiffies

It returns **-ERESTARTSYS** if interrupted, **0** if timed out, positive (at least 1, or a number of jiffies left till timeout) if completed.

try_wait_for_completion

This function will not put the thread on the wait queue but rather returns false if it would need to enqueue (block) the thread, else it consumes one posted completion and returns true.

```
bool try_wait_for_completion (struct completion * x);
```

where, `x` – holds the state of this particular completion

It returns **0** if completion is not available **1** if it got succeeded.

This `try_wait_for_completion()` is safe to be called in IRQ or atomic context.

Waking Up Task

complete

X

where, **x** – holds the state of this particular completion

Example:

```
complete(&data_read_done);
```

complete_all

This will wake up all threads waiting on this particular completion event.

void complete_all (struct completion * x);

where, **x** – holds the state of this particular completion

Check the status

completion_done

This is the test to see if completion has any waiters.

bool completion_done (struct completion * x);

where, **x** – holds the state of this particular completion

It returns 0 if there are waiters (**wait_for_completion** in progress) **1** if there are no waiters.

This **completion_done()** is safe to be called in IRQ or atomic context.

Driver Source Code – Completion in Linux

First, I will explain to you the concept of driver code.

In this source code, two places we are sending the complete call. One from the read function and another one from driver exit function.

X

complete came from the read, it will print the read count and it will again wait. If it is from the exit function, it will exit from the thread.

Here I've added two versions of code.

1. Completion created by static method
2. Completion created by dynamic method

But operation wise, both are the same.

You can also find the source code here.

Completion created by static method

```
#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>
#include <linux/completion.h>                // Required for the completion


uint32_t read_count = 0;
static struct task_struct *wait_thread;

DECLARE_COMPLETION(data_read_done);

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

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

/*************** Driver Functions **********************/
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 * of
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o

static struct file_operations fops =
```

```c

static int wait_function(void *unused)
{

        while(1) {
                printk(KERN_INFO "Waiting For Event...\n");
                wait_for_completion (&data_read_done);
                if(completion_flag == 2) {
                        printk(KERN_INFO "Event Came From Exit Function\n");
                        return 0;
                }
                printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_cou
                completion_flag = 0;
        }
        do_exit(0);
        return 0;
}

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 *of
{
        printk(KERN_INFO "Read Function\n");
        completion_flag = 1;
        if(!completion_done (&data_read_done)) {
            complete (&data_read_done);
        }
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
{
        printk(KERN_INFO "Write function\n");
        return 0;
}

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);
```

```c
        }

        /*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 1\n");
            goto r_device;
        }

        //Create the kernel thread with name 'mythread'
        wait_thread = kthread_create(wait_function, NULL, "WaitThread");
        if (wait_thread) {
                printk("Thread Created successfully\n");
                wake_up_process(wait_thread);
        } else
                printk(KERN_INFO "Thread creation failed\n");

        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

void __exit etx_driver_exit(void)
{
        completion_flag = 2;
        if(!completion_done (&data_read_done)) {
            complete (&data_read_done);
        }
        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 or admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple device driver - Completion (Static Method)");
MODULE_VERSION("1.23");
```

Completion created by dynamic method

```
#include <linux/device.h>
#include <linux/slab.h>                 //kmalloc()
#include <linux/uaccess.h>              //copy_to/from_user()

#include <linux/kthread.h>
#include <linux/completion.h>               // Required for the completion


uint32_t read_count = 0;
static struct task_struct *wait_thread;

struct completion data_read_done;

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

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

/*************** Driver Functions **********************/
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 * of
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o

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

static int wait_function(void *unused)
{

        while(1) {
                printk(KERN_INFO "Waiting For Event...\n");
                wait_for_completion (&data_read_done);
                if(completion_flag == 2) {
                        printk(KERN_INFO "Event Came From Exit Function\n");
                        return 0;
                }
                printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_cou
                completion_flag = 0;
        }
        do_exit(0);
        return 0;
}

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

X

```
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
{
        printk(KERN_INFO "Read Function\n");
        completion_flag = 1;
        if(!completion_done (&data_read_done)) {
            complete (&data_read_done);
        }
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
{
        printk(KERN_INFO "Write function\n");
        return 0;
}

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);
        etx_cdev.owner = THIS_MODULE;
        etx_cdev.ops = &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 1\n");
            goto r_device;
        }

        //Create the kernel thread with name 'mythread'
        wait_thread = kthread_create(wait_function, NULL, "WaitThread");
        if (wait_thread) {
                printk("Thread Created successfully\n");
                wake_up_process(wait_thread);
        } else
                printk(KERN_INFO "Thread creation failed\n");
```

X

```
r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

void __exit etx_driver_exit(void)
{
        completion_flag = 2;
        if(!completion_done (&data_read_done)) {
            complete (&data_read_done);
        }
        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 or admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple device driver - Completion (Dynamic Method)");
MODULE_VERSION("1.24");
```

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
```

Building and Testing Driver

- Build the driver by using Makefile (***sudo make***)
- Load the driver using **`sudo insmod driver.ko`**
- Then Check the Dmesg

```
Major = 246 Minor = 0
Thread Created successfully
Device Driver Insert…Done!!!
Waiting For Event…
```

```
Read Function
Event Came From Read Function – 1
Waiting For Event…
Device File Closed…!!!
```

- We send the complete from the read function, So it will print the read count, and then again it will sleep. Now send the event from exit function by `sudo rmmod driver`

```
Event Came From Exit Function
Device Driver Remove…Done!!!
```

- Now the condition was 2. So it will return from the thread and remove the driver.

In our next tutorial, we will discuss how to export the function from one Linux device driver to another Linux device driver.

in Linux Kernel
In "Device Drivers"

Thread
In "Device Drivers"

Character Device Driver
In "Device Drivers"

1

X

Join the discussion

B I U S {} [+]

This site uses Akismet to reduce spam. Learn how your comment data is processed.

1 COMMENT

Oldest

Bijesh

May 2, 2020 4:50 AM

Saved my life. Thank you embetronicx.

Loading...

0 Reply

1

X

:)

1

X