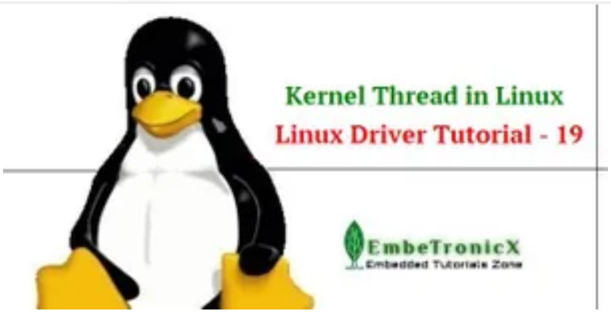
Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 19 – Kernel Thread**

Device Drivers




# Linux Device Driver Tutorial Part 19 – Kernel Thread

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 19 – Kernel Thread.

**Post Contents** [hide]

# Process

An executing instance of a program is called a process. Some operating systems use the term ‘task‘ to refer to a program that is being executed. ***Process*** is a heavyweight process. The context switch between the process is time-consuming.

## Threads

A *thread* is an independent flow of control that operates within the same address space as other independent flows of control within a process.

One process can have multiple threads, with each thread executing different code concurrently, while sharing data and synchronizing much more easily than cooperating processes. Threads require fewer system resources than processes and can start more quickly. Threads, also known as lightweight processes.

Some of the advantages of the thread, is that since all the threads within the processes share the same address space, the communication between the threads is far easier and less time consuming as compared to processes. This approach has one disadvantage also. It leads to several concurrency issues and require the synchronization mechanisms to handle the same.

## Thread Management

Whenever we are creating a thread, it has to manage by someone. So that management follows like below.

- A thread is a sequence of instructions.
- CPU can handle one instruction at a time.
- To switch between instructions on parallel threads, the execution state needs to be saved.
- Execution state in its simplest form is a program counter and CPU registers.
- The program counter tells us what instruction to execute next.
- CPU registers hold execution arguments, for example, addition operands.
- This alternation between threads requires management.
- Management includes saving state, restoring state, deciding what thread to pick next.

## Types of Thread

There are two types of threads.

1. User Level Thread
2. Kernel Level Thread

## User Level Thread

In this type, the kernel is not aware of these threads. Everything is maintained by the user thread library. That thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. So all will be in User Space.

## Kernel Level Thread

Kernel level threads are managed by the OS, therefore, thread operations are implemented in the kernel code. There is no thread management code in the application area.

Anyhow each type of thread has advantages and disadvantages too.

Now we will move into Kernel Thread Programming. First, we will see the functions used in a kernel thread.

## Kernel Thread Management Functions

There are many functions used in Kernel Thread. We will see one by one. We can classify those functions based on functionalities.

- Create Kernel Thread
- Start Kernel Thread
- Stop Kernel Thread
- Other functions in Kernel Thread

For use the below functions you should include `linux/kthread.h` header file.

# Create Kernel Thread

## kthread_create

create a kthread.

```
struct task_struct * kthread_create (int (* threadfn(void *data),
                                     void *data, const char namefmt[], ...);
```

Where,

**threadfn** – the function to run until signal_pending(current).

**data** – data ptr for threadfn.

**namefmt[]** – printf-style name for the thread.

**...** – variable arguments

This helper function creates and names a kernel thread. But we need to wake up that thread manually. When woken, the thread will run **threadfn()** with data as its argument.

*threadfn* can either call **do_exit** directly if it is a standalone thread for which no one will call **kthread_stop**, or return when '**kthread_should_stop**' is true (which means **kthread_stop** has been called). The return value should be zero or a negative error number; it will be passed to **kthread_stop**.

It **Returns** `task_struct` or `ERR_PTR(-ENOMEM)`.

# Start Kernel Thread

## wake_up_process

This is used to Wake up a specific process.

```
int wake_up_process (struct task_struct * p);
```

4

Where,

`p` – The process to be woken up.

Attempt to wake up the nominated process and move it to the set of runnable processes.

It **returns 1** if the process was woken up, **0** if it was already running.

It may be assumed that this function implies a write memory barrier before changing the task state if and only if any tasks are woken up.

# Stop Kernel Thread

## kthread_stop

It stops a thread created by `kthread_create`.

```
int kthread_stop ( struct task_struct *k);
```

Where,

`k` – thread created by `kthread_create`.

Sets `kthread_should_stop` for `k` to return `true`, wakes it and waits for it to exit. Your `threadfn` must not call `do_exit` itself if you use this function! This can also be called after `kthread_create` instead of calling `wake_up_process`: the thread will exit without calling `threadfn`.

It **Returns** the result of `threadfn`, or –**EINTR** if `wake_up_process` was never called.

# Other functions in Kernel Thread

## kthread_should_stop

should this kthread return now?

```
int kthread_should_stop (void);
```

When someone calls `kthread_stop` on your kthread, it will be woken and this will return true. You should then return, and your return value will be passed through to `kthread_stop`.

## kthread_bind

This is used to bind a just-created kthread to a cpu.

**`void kthread_bind (struct task_struct *k, unsigned int cpu);`**

Where,

**`k`** – thread created by kthread_create.

**`cpu`** – cpu (might not be online, must be possible) for **`k`** to run on.

# Implementation

# Thread Function

First, we have to create our thread which has the argument of **`void *`** and should return **`int`** value.  We should follow some conditions in our thread function. It is advisable.

- If that thread is a long run thread, we need to check **`kthread_should_stop()`** every time as because any function may call **`kthread_stop`**. If any function called **`kthread_stop`**, that time **`kthread_should_stop`** will return **`true`**. We have to exit our thread function if **`true`** value been returned by **`kthread_should_stop`**.
- But if your thread function is not running long, then let that thread finish its task and kill itself using **`do_exit`**.

In my thread function, lets print something every minute and it is a continuous process. So let's check the **`kthread_should_stop`** every time. See the below snippet to understand.

```
int thread_function(void *pv)
{
    int i=0;
    while(!kthread_should_stop()) {
        printk(KERN_INFO "In EmbeTronicX Thread Function %d\n", i++);
        msleep(1000);
    }
    return 0;
}
```

# Creating and Starting Kernel Thread

So as of now, we have our thread function to run. Now, we will create kernel thread using kthread_create and start the kernel thread using **wake_up_process**.

```
static struct task_struct *etx_thread;

etx_thread = kthread_create(thread_function,NULL,"eTx Thread");
if(etx_thread) {
    wake_up_process(etx_thread);
} else {
    printk(KERN_ERR "Cannot create kthread\n");
}
```

There is another function that does both processes (create and start). That is **kthread_run**(). You can replace both **kthread_create** and **wake_up_process** using this function.

## kthread_run

This is used to create and wake a thread.

**kthread_run (threadfn, data, namefmt, ...);**

Where,

**threadfn** – the function to run until signal_pending(current).

**data** – data ptr for threadfn.

**namefmt** – printf-style name for the thread.

**...** – variable arguments

Convenient wrapper for kthread_create followed by wake_up_process.

It **returns** the **kthread** or **ERR_PTR(-ENOMEM).**

4

You can see the below snippet which is using kthread_run.

```
static struct task_struct *etx_thread;

etx_thread = kthread_run(thread_function,NULL,"eTx Thread");
if(etx_thread) {
    printk(KERN_ERR "Kthread Created Successfully...\n");
} else {
```

```
    printk(KERN_ERR "Cannot create kthread\n");
}
```

## Stop Kernel Thread

You can stop the kernel thread using **kthread_stop**. Use the below snippet to stop.

```
kthread_stop(etx_thread);
```

## Driver Source Code – Kthread in Linux

Kernel thread will start when we insert the kernel module. It will print something every second. When we remove the module that time it stops the kernel thread. Let’s see the 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>


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_thread;


/*************** 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_function(void *pv);

int thread_function(void *pv)
{
    int i=0;
    while(!kthread_should_stop()) {
```

```
        printk(KERN_INFO "In EmbeTronicX Thread Function %d\n", i++);
        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;
        }

        etx_thread = kthread_create(thread_function,NULL,"eTx Thread");
        if(etx_thread) {
            wake_up_process(etx_thread);
        } else {
            printk(KERN_ERR "Cannot create kthread\n");
            goto r_device;
        }
#if 0
        /* You can use this method to create and run the thread */
        etx_thread = kthread_run(thread_function,NULL,"eTx Thread");
        if(etx_thread) {
            printk(KERN_ERR "Kthread Created Successfully...\n");
        } else {
            printk(KERN_ERR "Cannot create kthread\n");
             goto r_device;
        }
#endif
        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_thread);
        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 - Kernel Thread");
MODULE_VERSION("1.14");
```

# 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

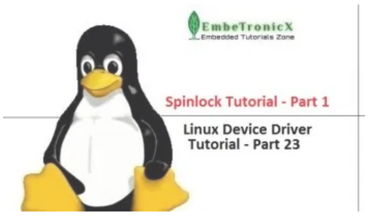
- Build the driver by using Makefile (***sudo make***)
- Load the driver using **`sudo insmod driver.ko`**
- Then Check the **`dmesg`**

```
Major = 246 Minor = 0
Device Driver Insert...Done!!!
In EmbeTronicX Thread Function 0
In EmbeTronicX Thread Function 1
In EmbeTronicX Thread Function 2
In EmbeTronicX Thread Function 3
```

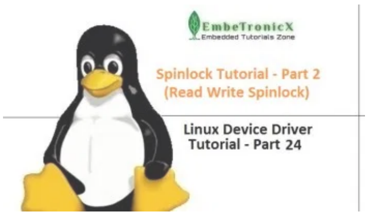
- So our thread is running now.
- Remove the driver using **`sudo rmmod driver`** to stop the thread.

In our next tutorial, we will discuss tasklet in the Linux device driver.

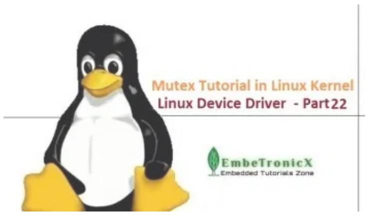
**Related**




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




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




Linux Device Driver Tutorial Part 22 – Mutex in Linux Kernel
In "Device Drivers"

4

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.

**4 COMMENTS** Oldest

madhab

August 14, 2019 3:54 AM

Thank you Embetronicx for providing this type of tutorial.
I want to make multiple kernel thread with proper synchronisation. so can you please update the code or procedure.

Loading...

0 Reply

Umesh

May 2, 2020 6:55 PM

please add linux driver using hardware

Loading...

0 Reply

4

Kunapareddy Jeevan

June 20, 2020 6:33 PM

what is pv argument in thread_function

Loading...

0 Reply

EmbeTronicX

*Reply to Kunapareddy Jeevan* June 20, 2020 6:54 PM

That is the argument of the thread function. Let's say I want to pass the value 10 to the thread function while started running. We can do that like below.

```
int32_t temp_var = 10;
kthread_create(thread_function, (void *) &temp_var, "eTx Thread")
```

Now you are passing the address to the 2nd arg of the kernel thread create API. So this address will be passed to the thread_function like below.

```
int thread_function(void *pv)
{
    printk(KERN_INFO "Value = %d\n", *(int32_t*)pv);
    return 0;
}
```

Loading...

0 Reply

4

4