# Part 19 – Kernel Thread

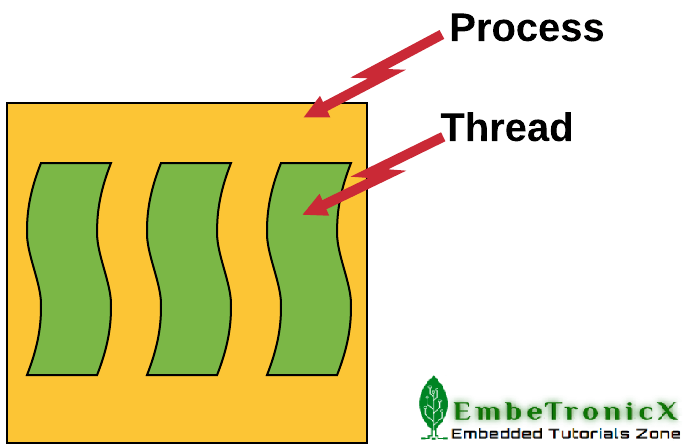
[ <https://embetronicx.com/tutorials/linux/device-drivers/linux-device-drivers-tutorial-kernel-thread/> ]

# 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 heavy weight process. 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 light weight 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 thread, it has to be managed 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, execution state need to be saved.
* Execution state in its simplest form is a program counter and CPU registers.
* 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 thread.

1. User Level Thread
2. Kernel Level Thread

## User Level Thread

In this type, 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 types of the threads have advantages and disadvantages too.

Now we will move into Kernel Thread Programming. First we will see the functions used in 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 noone 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 the structure to task control block.** |

## Start Kernel Thread

### wake\_up\_process

|  |
| --- |
| This is used to Wake up a specific process.  int wake\_up\_process (struct task\_struct \* p);  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. Its 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 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 which does both process (create and start). That is kthread\_run(). You can replace the 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). |

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;

}

#if 1

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;

}

#else

/\* Alternate method to create and run the thread (using one API) \*/

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 driver using sudo rmmod driver to stop the thread.