Linux Device Driver Tutorial Part 16 – Workqueue in Linux Kernel Part 3 (Own Workqueue)

Post Contents [hide]

Work queue in Linux Device Driver

In our previous (Part 1, Part 2) tutorials we haven't created any of the workqueue. We were just creating work and scheduling that work to the global workqueue. Now we are going to create our own workqueue. Let's get into the tutorial.

The core work queue is represented by structure struct workqueue_struct, which is the structure onto which work is placed. This work is added to queue in the top half (Interrupt context) and execution of this work happened in the bottom half (Kernel context).

The work is represented by structure struct work_struct, which identifies the work and the deferral function.

Create and destroy work queue structure

Work queues are created through a macro called **create_workqueue**, which returns a **workqueue_struct** reference. You can remote this work queue later (if needed) through a call to the destroy_workqueue function.

struct workqueue_struct *create_workqueue(name);
void destroy_workqueue(struct workqueue_struct *);

You should use create_singlethread_workqueue() for create workqueue when you want to create only a single thread for all the processor..

Since **create_workqueue** and **create_singlethread_workqueue()** are macros. Both are using the **alloc_workqueue** function in background.

```
#define create_workqueue(name)
      alloc_workqueue("%s", WQ_MEM_RECLAIM, 1, (name))
#define create_singlethread_workqueue(name)
      alloc_workqueue("%s", WQ_UNBOUND | WQ_MEM_RECLAIM, 1, (name))
```

alloc_workqueue

Allocate a workqueue with the specified parameters.

```
alloc_workqueue ( fmt, flags, max_active );
```

`fmt`– printf format for the name of the workqueue
`flags` – WQ_* flags
`max_active` – max in-flight work items, 0 for default
This will return Pointer to the allocated workqueue on success, `NULL` on failure.

WQ_* flags

This is the second argument of alloc_workqueue.

WQ_UNBOUND

Work items queued to an unbound wq are served by the special worker-pools which host workers which are not bound to any specific CPU. This makes the wq behave as a simple execution context provider without concurrency management. The unbound worker-pools try to start execution of work items as soon as possible. Unbound wq sacrifices locality but is useful for the following cases.

- Wide fluctuation in the concurrency level requirement is expected and using bound wq may end up creating large number of mostly unused workers across different CPUs as the issuer hops through different CPUs.
- Long running CPU intensive workloads which can be better managed by the system scheduler.

WQ_FREEZABLE

A freezable wq participates in the freeze phase of the system suspend operations. Work items on the wq are drained and no new work item starts execution until thawed.

WQ_MEM_RECLAIM

All wq which might be used in the memory reclaim paths **MUST** have this flag set. The wq is guaranteed to have at least one execution context regardless of memory pressure.

WQ_HIGHPRI

Work items of a highpri wq are queued to the highpri worker-pool of the target cpu. Highpri worker-pools are served by worker threads with elevated nice level.

Note that normal and highpri worker-pools don’t interact with each other. Each maintain its separate pool of workers and implements concurrency management among its workers.

WQ_CPU_INTENSIVE

Work items of a CPU intensive wq do not contribute to the concurrency level. In other words, runnable CPU intensive work items will not prevent other work items in the same worker-pool from starting execution. This is useful for bound work items which are expected to hog CPU cycles so that their execution is regulated by the system scheduler.

Although CPU intensive work items don't contribute to the concurrency level, start of their executions is still regulated by the concurrency management and runnable non-CPU-intensive work items can delay execution of CPU intensive work items.

This flag is meaningless for unbound wq.

Queuing Work to workqueue

With the work structure initialized, the next step is enqueuing the work on a work queue. You can do this in a few ways.

queue_work

This will queue the work to the CPU on which it was submitted, but if the CPU dies it can be processed by another CPU.

```
int queue_work( struct workqueue_struct *wq, struct work_struct *work );
```

Where,

`wq` – workqueue to use
`work` – work to queue
It returns `false` if *`work`* was already on a queue, `true` otherwise.

queue_work_on

This puts a work on a specific cpu.

```
int queue_work_on( int cpu, struct workqueue_struct *wq,
                        struct work_struct *work );
```

Where,

`cpu`– cpu to put the work task on
`wq` – workqueue to use
`work`– job to be done

queue_delayed_work

After waiting for a given time this function puts a work in the workqueue.

```
int queue_delayed_work( struct workqueue_struct *wq,
  struct delayed_work *dwork, unsigned long delay );
```

Where,

`wq` – workqueue to use
`dwork` – work to queue
delay – number of jiffies to wait before queueing or 0 for immediate execution

queue_delayed_work_on

After waiting for a given time this puts a job in the workqueue on the specified CPU.

```
int queue_delayed_work_on( int cpu, struct workqueue_struct *wq,
  struct delayed_work *dwork, unsigned long delay );
```

Where,

cpu– cpu to put the work task on
`wq` – workqueue to use
`dwork` – work to queue
delay – number of jiffies to wait before queueing or 0 for immediate execution

Programming

Driver Source Code

In that source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm scheduling the work to the workqueue. I'm not going to do any job in both interrupt handler and workqueue function, since it is a tutorial post. But in real workqueues, this function can be used to carry out any operations that need to be scheduled.
We have created workqueue "own_wq" in init function.

Let's go through the 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/sysfs.h>
#include<linux/kobject.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include <linux/workqueue.h>            // Required for workqueues


#define IRQ_NO 11
```

```
static struct workqueue_struct *own_workqueue;

static void workqueue_fn(struct work_struct *work);

static DECLARE_WORK(work, workqueue_fn);


/*Workqueue Function*/
static void workqueue_fn(struct work_struct *work)
{
   printk(KERN_INFO "Executing Workqueue Function\n");
   return;

}


//Interrupt handler for IRQ 11.
static irqreturn_t irq_handler(int irq,void *dev_id) {
       printk(KERN_INFO "Shared IRQ: Interrupt Occurred\n");
       /*Allocating work to queue*/
       queue_work(own_workqueue, &work);

       return IRQ_HANDLED;
}


volatile int etx_value = 0;


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

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

/*************** 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);

/*************** Sysfs Fuctions **********************/
static ssize_t sysfs_show(struct kobject *kobj,
```

```
            struct kobj_attribute *attr, char *buf);
static ssize_t sysfs_store(struct kobject *kobj,
            struct kobj_attribute *attr,const char *buf, size_t count);

struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);

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

static ssize_t sysfs_show(struct kobject *kobj,
            struct kobj_attribute *attr, char *buf)
{
      printk(KERN_INFO "Sysfs - Read!!!\n");
      return sprintf(buf, "%d", etx_value);
}

static ssize_t sysfs_store(struct kobject *kobj,
            struct kobj_attribute *attr,const char *buf, size_t count)
{
      printk(KERN_INFO "Sysfs - Write!!!\n");
      sscanf(buf,"%d",&etx_value);
      return count;
}

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");
      asm("int $0x3B");  // Corresponding to irq 11
      return 0;
```

```c
}
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 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);

      /*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;
      }

      /*Creating a directory in /sys/kernel/ */
      kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);

      /*Creating sysfs file for etx_value*/
      if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
            printk(KERN_INFO"Cannot create sysfs file......\n");
            goto r_sysfs;
      }
      if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handler))) {
```

```
            printk(KERN_INFO "my_device: cannot register IRQ \n");
                goto irq;
        }

        /*Creating workqueue */
        own_workqueue = create_workqueue("own_wq");

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

irq:
        free_irq(IRQ_NO,(void *)(irq_handler));

r_sysfs:
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);

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)
{
        /* Delete workqueue */
        destroy_workqueue(own_workqueue);
        free_irq(IRQ_NO,(void *)(irq_handler));
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
        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 - Workqueue part 3");
MODULE_VERSION("1.12");
```

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`
- To trigger interrupt read device file (`sudo cat /dev/etx_device`)
- Now see the Dmesg (`dmesg`)

[2562.609446] Major = 246 Minor = 0
[2562.649362] Device Driver Insert...Done!!!
[2565.133204] Device File Opened...!!!
[2565.133225] Read function
[2565.133248] Shared IRQ: Interrupt Occurred
[2565.133267] Executing Workqueue Function
[2565.140284] Device File Closed...!!!

- We can able to see the print "**Shared IRQ: Interrupt Occurred**" and "**Executing Workqueue Function**"
- Use "`ps -aef`" command to see our workqueue. You can able to see our workqueue which is "`own_wq`"

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	3516	2	0	21:35	?	00:00:00	[own_wq]

- Unload the module using `sudo rmmod driver`

Difference between Schedule_work and queue_work

- If you want to use your own dedicated workqueue you should create workqueue using `create_workqueue`. In that time you need to put work on your workqueue by using `queue_work` function.
- If you don't want to create any own workqueue, you can use kernel global workqueue. In that condition, you can use `schedule_work` function to put your work to global workqueue.