# Part 16 – Workqueue in Linux Kernel Part 3

[ <https://embetronicx.com/tutorials/linux/device-drivers/work-queue-in-linux-own-workqueue/> ]

# Work queue in Linux Device Driver

In our previous ([Part 1](https://www.embetronicx.com/tutorials/linux/device-drivers/workqueue-in-linux-kernel/), [Part 2](https://www.embetronicx.com/tutorials/linux/device-drivers/workqueue-in-linux-dynamic-creation/)) 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 remove 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 creating workqueue when you want to create only a single thread for all the processor(s).

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 |

## 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](https://www.embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-part-13-interrupt-example-program-in-linux-kernel/)). 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;

}

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)

$ 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.