# Part 10 – WaitQueue in Linux

[ <https://embetronicx.com/tutorials/linux/device-drivers/waitqueue-in-linux-device-driver-tutorial/> ]

# Introduction

When you write a Linux  Driver or Module or Kernel Program, Some process should wait or sleep for some event. There are several ways of handling process / threads that sleep and wake-up in Linux, each suited to different needs. WaitQueue is one of the method to handle that (sleep / wakeup) case.

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses. After some time, the process will be woken up and will continue with its job when the event which we are waiting will be occurred.

Wait queue is a mechanism provided in kernel to implement the wait. As the name itself suggests, **wait queue is the list of processes waiting for an event.** In other words, A wait queue is used to wait for someone to wake you up when a certain condition is true. hey must be used carefully to ensure there is no race condition.

There are 3 important steps in Waitqueue.

1. Initializing Waitqueue
2. Queuing (Put the Task to sleep until the event comes)
3. Waking Up Queued Task

**Initializing Waitqueue**

Use this Header file for Waitqueue (include/linux/wait.h). There are two ways to initialize the Waitqueue.

1. Static method
2. Dynamic method

You can use any one of the method.

**Queuing**

Once the wait queue is declared and initialized, a process may use it to go to sleep. There are several macros are available for different uses. We will see one by one.

1. **wait\_event**
2. **wait\_event\_timeout**
3. **wait\_event\_cmd**
4. **wait\_event\_interruptible**
5. **wait\_event\_interruptible\_timeout**
6. **wait\_event\_killable**

Old kernel versions used the functions sleep\_on() and interruptible\_sleep\_on(), but those two functions can introduce bad race conditions and should not be used.

Whenever we use the above one of the macro, it will add that task to the waitqueue which is created by us. Then it will wait for the event.

## wait\_event

|  |
| --- |
| sleep until a condition gets true.  **wait\_event(wq, condition);**  wq – the waitqueue to wait on  condition – a C expression for the event to wait for  The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the *condition*  evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up. |

## wait\_event\_timeout

|  |
| --- |
| sleep until a condition gets true or a timeout elapses  **wait\_event\_timeout(wq, condition, timeout);**  wq –  the waitqueue to wait on  condtion – a C expression for the event to wait for  timeout –  timeout, in jiffies  The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the *condition* evaluates to true or timeout elapses.  The *condition* is checked each time the waitqueue *wq* is woken up.  It **returns**  **0** if the *condition* evaluated to false after the *timeout* elapsed,  **1** if the *condition* evaluated to true after the *timeout* elapsed,  or the **remaining jiffies** (at least 1) if the *condition* evaluated to true before the *timeout* elapsed. |

## wait\_event\_cmd

|  |
| --- |
| sleep until a condition gets true  **wait\_event\_cmd(wq, condition, cmd1, cmd2);**  wq –  the waitqueue to wait on  condtion – a C expression for the event to wait for  cmd1 – the command will be executed before sleep  cmd2 – the command will be executed after sleep  The process is put to sleep (TASK\_UNINTERRUPTIBLE) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up. |

## wait\_event\_interruptible

|  |
| --- |
| sleep until a condition gets true  **wait\_event\_interruptible(wq, condition);**  wq –  the waitqueue to wait on  condtion – a C expression for the event to wait for  The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.  The function will **return** -ERESTARTSYS if it was interrupted by a signal and 0 if *condition* evaluated to true. |

## wait\_event\_interruptible\_timeout

|  |
| --- |
| sleep until a condition gets true or a timeout elapses  **wait\_event\_interruptible\_timeout(wq, condition, timeout);**  wq –  the waitqueue to wait on  condtion – a C expression for the event to wait for  timeout –  timeout, in jiffies  The process is put to sleep (TASK\_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received or timeout elapes. The *condition* is checked each time the waitqueue *wq* is woken up.  It **returns**, **0** if the *condition* evaluated to false after the *timeout* elapsed, **1** if the *condition* evaluated to true after the *timeout* elapsed, the **remaining jiffies** (at least 1) if the *condition* evaluated to true before the *timeout* elapsed, or -ERESTARTSYS if it was interrupted by a signal. |

## wait\_event\_killable

|  |
| --- |
| sleep until a condition gets true  **wait\_event\_killable(wq, condition);**  wq –  the waitqueue to wait on  condtion – a C expression for the event to wait for  The process is put to sleep (TASK\_KILLABLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.  The function will **return** -ERESTARTSYS if it was interrupted by a signal and **0** if *condition* evaluated to true. |

**Waking Up Queued Task**

When some Tasks are in sleep mode because of waitqueue, then we can use the below function to wake up those tasks.

1. **wake\_up**
2. **wake\_up\_all**
3. **wake\_up\_interruptible**
4. **wake\_up\_sync and wake\_up\_interruptible\_sync**

## wake\_up

|  |
| --- |
| wakes up only one process from the wait queue  which is in non interruptible sleep.  **wake\_up(&wq);**  wq – the waitqueue to wake up |

## wake\_up\_all

|  |
| --- |
| wakes up all the processes on the wait queue  **wake\_up\_all(&wq);**  wq – the waitqueue to wake up |

## wake\_up\_interruptible

## 

|  |
| --- |
| wakes up only one process from the wait queue that is in interruptible sleep  **wake\_up\_interruptible(&wq);**  wq – the waitqueue to wake up |

## wake\_up\_sync & wake\_up\_interruptible\_sync

## 

|  |
| --- |
| wakes up only one process from the wait queue that is in interruptible sleep  **wake\_up\_sync(&wq);**  **wake\_up\_interruptible\_sync(&wq);**  wq – the waitqueue to wake up  Normally, a *wake\_up* call can cause an immediate reschedule to happen, meaning that other processes might run before *wake\_up* returns. The “synchronous” variants instead make any awakened processes runnable, but do not reschedule the CPU. This is used to avoid rescheduling when the current process is known to be going to sleep, thus forcing a reschedule anyway. Note that awakened processes could run immediately on a different processor, so these functions should not be expected to provide mutual exclusion. |

# Driver Source Code – WaitQueue in Linux

First i will explain you the concept of driver code.

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

I’ve created one thread (wait\_function) which has while(1). That thread will always wait for the event. It will be sleeping until it gets wake up call. When it gets the wakeup call, it will check the condition. If condition is 1 then the wakeup came from read function. If it is 2, then the wakeup came from exit function. If wake up came from read, it will print the read count  and it will again wait. If it’s from exit function, it will exit from the thread.

Here I’ve added two versions of code.

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

But operation wise both are same.

## Waitqueue 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/wait.h> // Required for the wait queues

uint32\_t read\_count = 0;

static struct task\_struct \*wait\_thread;

DECLARE\_WAIT\_QUEUE\_HEAD(wait\_queue\_etx);

dev\_t dev = 0;

static struct class \*dev\_class;

static struct cdev etx\_cdev;

int wait\_queue\_flag = 0;

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);

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\_event\_interruptible(wait\_queue\_etx, wait\_queue\_flag != 0 );

if(wait\_queue\_flag == 2) {

printk(KERN\_INFO "Event Came From Exit Function\n");

return 0;

}

printk(KERN\_INFO "Event Came From Read Function - %d\n", ++read\_count);

wait\_queue\_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 \*off)

{

printk(KERN\_INFO "Read Function\n");

wait\_queue\_flag = 1;

wake\_up\_interruptible(&wait\_queue\_etx);

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);

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;

}

//Initialize wait queue

init\_waitqueue\_head(&wait\_queue\_etx);

//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)

{

wait\_queue\_flag = 2;

wake\_up\_interruptible(&wait\_queue\_etx);

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");

MODULE\_VERSION("1.7");

## Waitqueue created by Dynamic 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/wait.h> // Required for the wait queues

uint32\_t read\_count = 0;

static struct task\_struct \*wait\_thread;

dev\_t dev = 0;

static struct class \*dev\_class;

static struct cdev etx\_cdev;

wait\_queue\_head\_t wait\_queue\_etx;

int wait\_queue\_flag = 0;

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);

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\_event\_interruptible(wait\_queue\_etx, wait\_queue\_flag != 0 );

if(wait\_queue\_flag == 2) {

printk(KERN\_INFO "Event Came From Exit Function\n");

return 0;

}

printk(KERN\_INFO "Event Came From Read Function - %d\n", ++read\_count);

wait\_queue\_flag = 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 \*off)

{

printk(KERN\_INFO "Read Function\n");

wait\_queue\_flag = 1;

wake\_up\_interruptible(&wait\_queue\_etx);

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;

}

//Initialize wait queue

init\_waitqueue\_head(&wait\_queue\_etx);

//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)

{

wait\_queue\_flag = 2;

wake\_up\_interruptible(&wait\_queue\_etx);

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");

MODULE\_VERSION("1.7");

**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…*

* So that thread is waiting for the event. Now we will send the event by reading the driver using sudo cat /dev/etx\_device
* Now check the dmesg

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

* We send the wake up from 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.