Linux Device Driver Tutorial Part 10 – WaitQueue in Linux

WaitQueue in Linux

Introduction

When you write a Linux Driver or Module or Kernel Program, Some process should be wait or sleep for some event. There are several ways of handling sleeping and waking up in Linux, each suited to different needs. Waitqueue also one of the method to handle that 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. They 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.

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.

Static Method

DECLARE_WAIT_QUEUE_HEAD(wq);

Where the “wq” is the name of the queue on which task will be put to sleep.

Dynamic Method

wait_queue_head_t wq;
init_waitqueue_head (&wq);

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 and wake_up_interruptible_sync

wake_up_sync(&wq);
wake_up_interruptible_sync(&wq);

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 sleep until it gets wake up call. When it gets the wake up call, it will check the condition. If condition is 1 then the wakeup came from read function. It 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 its 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.