EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 10 – Waitqueue in Linux**

Device Drivers

Linux Device Driver Tutorial Part 10 – Waitqueue in Linux

This article is a continuation of the Series on Linux Device Driver and carries on the discussion on character drivers and their implementation. This is Part 10 of the Linux device driver tutorial. Now we will discuss Waitqueue in Linux.

Post Contents [hide]

Waitqueue in Linux

Introduction

When you write a Linux Driver or Module or Kernel Program, Some processes should 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 methods 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 for has arrived.

Wait queue is a mechanism provided in the kernel to implement the wait. As the name itself suggests, waitqueue 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 waitqueue.

1. Static method
2. Dynamic method

You can use any one of the methods.

Static Method

```
1 DECLARE_WAIT_QUEUE_HEAD(wq);
```

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

Dynamic Method

```
1 wait_queue_head_t wq;
2 init_waitqueue_head (&wq);
```

You can create waitqueue using any one of the above methods.

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 to you the concept of driver code.

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

I've created one thread (**wait_function**) which has **while(1)**. That thread will always wait for the event. It will sleep until it gets a wake_up call. When it gets the wake_up call, it will check the condition. If the condition is 1 then the wakeup came from the read function. If it is 2, then the wakeup came from an exit function. If wake_up came from the read function, it will print the read count and it will again wait. If it is from the 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 the same.

Waitqueue created by Static Method

```
1  #include <linux/kernel.h>
2  #include <linux/init.h>
3  #include <linux/module.h>
4  #include <linux/kdev_t.h>
5  #include <linux/fs.h>
6  #include <linux/cdev.h>
7  #include <linux/device.h>
8  #include <linux/slab.h>                 //kmalloc()
9  #include <linux/uaccess.h>              //copy_to/from_user()
10
11 #include <linux/kthread.h>
12 #include <linux/wait.h>                // Required for the wait queues
13
14
15 uint32_t read_count = 0;
16 static struct task_struct *wait_thread;
17
18 DECLARE_WAIT_QUEUE_HEAD(wait_queue_etx);
19
20 dev_t dev = 0;
21 static struct class *dev_class;
22 static struct cdev etx_cdev;
23 int wait_queue_flag = 0;
24
25 static int __init etx_driver_init(void);
26 static void __exit etx_driver_exit(void);
27
28 /*************** Driver Fuctions **********************/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
31 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,loff_t * of
```

```
32  static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o
33
34  static struct file_operations fops =
35  {
36          .owner          = THIS_MODULE,
37          .read           = etx_read,
38          .write          = etx_write,
39          .open           = etx_open,
40          .release        = etx_release,
41  };
42
43  static int wait_function(void *unused)
44  {
45
46          while(1) {
47                  printk(KERN_INFO "Waiting For Event...\n");
48                  wait_event_interruptible(wait_queue_etx, wait_queue_flag != 0 );
49                  if(wait_queue_flag == 2) {
50                          printk(KERN_INFO "Event Came From Exit Function\n");
51                          return 0;
52                  }
53                  printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_cou
54                  wait_queue_flag = 0;
55          }
56          do_exit(0);
57          return 0;
58  }
59
60
61
62  static int etx_open(struct inode *inode, struct file *file)
63  {
64          printk(KERN_INFO "Device File Opened...!!!\n");
65          return 0;
66  }
67
68  static int etx_release(struct inode *inode, struct file *file)
69  {
70          printk(KERN_INFO "Device File Closed...!!!\n");
71          return 0;
72  }
73
74  static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
75  {
76          printk(KERN_INFO "Read Function\n");
77          wait_queue_flag = 1;
78          wake_up_interruptible(&wait_queue_etx);
79          return 0;
80  }
81  static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
82  {
83          printk(KERN_INFO "Write function\n");
84          return 0;
85  }
86
87
88
89
90  static int __init etx_driver_init(void)
91  {
92          /*Allocating Major number*/
93          if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
94                  printk(KERN_INFO "Cannot allocate major number\n");
```

```
95                  return -1;
96          }
97          printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
98
99          /*Creating cdev structure*/
100         cdev_init(&etx_cdev,&fops);
101         etx_cdev.owner = THIS_MODULE;
102         etx_cdev.ops = &fops;
103
104         /*Adding character device to the system*/
105         if((cdev_add(&etx_cdev,dev,1)) < 0){
106             printk(KERN_INFO "Cannot add the device to the system\n");
107             goto r_class;
108         }
109
110         /*Creating struct class*/
111         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
112             printk(KERN_INFO "Cannot create the struct class\n");
113             goto r_class;
114         }
115
116         /*Creating device*/
117         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
118             printk(KERN_INFO "Cannot create the Device 1\n");
119             goto r_device;
120         }
121
122         //Initialize wait queue
123         init_waitqueue_head(&wait_queue_etx);
124
125         //Create the kernel thread with name 'mythread'
126         wait_thread = kthread_create(wait_function, NULL, "WaitThread");
127         if (wait_thread) {
128                 printk("Thread Created successfully\n");
129                 wake_up_process(wait_thread);
130         } else
131                 printk(KERN_INFO "Thread creation failed\n");
132
133         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
134     return 0;
135
136 r_device:
137         class_destroy(dev_class);
138 r_class:
139         unregister_chrdev_region(dev,1);
140         return -1;
141 }
142
143 void __exit etx_driver_exit(void)
144 {
145         wait_queue_flag = 2;
146         wake_up_interruptible(&wait_queue_etx);
147         device_destroy(dev_class,dev);
148         class_destroy(dev_class);
149         cdev_del(&etx_cdev);
150         unregister_chrdev_region(dev, 1);
151     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
152 }
153
154 module_init(etx_driver_init);
155 module_exit(etx_driver_exit);
156
157 MODULE_LICENSE("GPL");
```

```
158 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
159 MODULE_DESCRIPTION("A simple device driver");
160 MODULE_VERSION("1.7");
```

Waitqueue created by Dynamic Method

```
 1  #include <linux/kernel.h>
 2  #include <linux/init.h>
 3  #include <linux/module.h>
 4  #include <linux/kdev_t.h>
 5  #include <linux/fs.h>
 6  #include <linux/cdev.h>
 7  #include <linux/device.h>
 8  #include <linux/slab.h>                 //kmalloc()
 9  #include <linux/uaccess.h>              //copy_to/from_user()
10
11  #include <linux/kthread.h>
12  #include <linux/wait.h>                // Required for the wait queues
13
14
15  uint32_t read_count = 0;
16  static struct task_struct *wait_thread;
17
18  dev_t dev = 0;
19  static struct class *dev_class;
20  static struct cdev etx_cdev;
21  wait_queue_head_t wait_queue_etx;
22  int wait_queue_flag = 0;
23
24  static int __init etx_driver_init(void);
25  static void __exit etx_driver_exit(void);
26
27  /*************** Driver Fuctions **********************/
28  static int etx_open(struct inode *inode, struct file *file);
29  static int etx_release(struct inode *inode, struct file *file);
30  static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,loff_t * of
31  static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o
32
33
34  static struct file_operations fops =
35  {
36          .owner          = THIS_MODULE,
37          .read           = etx_read,
38          .write          = etx_write,
39          .open           = etx_open,
40          .release        = etx_release,
41  };
42
43
44  static int wait_function(void *unused)
453 {
46
47          while(1) {
48                  printk(KERN_INFO "Waiting For Event...\n");
49                  wait_event_interruptible(wait_queue_etx, wait_queue_flag != 0 );
50                  if(wait_queue_flag == 2) {
51                          printk(KERN_INFO "Event Came From Exit Function\n");
52                          return 0;
53                  }
54                  printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_cou
55                  wait_queue_flag = 0;
```

```
56          }
57          return 0;
58  }
59
60
61
62  static int etx_open(struct inode *inode, struct file *file)
63  {
64          printk(KERN_INFO "Device File Opened...!!!\n");
65          return 0;
66  }
67
68  static int etx_release(struct inode *inode, struct file *file)
69  {
70          printk(KERN_INFO "Device File Closed...!!!\n");
71          return 0;
72  }
73
74  static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
75  {
76          printk(KERN_INFO "Read Function\n");
77          wait_queue_flag = 1;
78          wake_up_interruptible(&wait_queue_etx);
79          return 0;
80  }
81  static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
82  {
83          printk(KERN_INFO "Write function\n");
84          return 0;
85  }
86
87
88
89
90  static int __init etx_driver_init(void)
91  {
92          /*Allocating Major number*/
93          if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
94                  printk(KERN_INFO "Cannot allocate major number\n");
95                  return -1;
96          }
97          printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
98
99          /*Creating cdev structure*/
100         cdev_init(&etx_cdev,&fops);
101
102         /*Adding character device to the system*/
103         if((cdev_add(&etx_cdev,dev,1)) < 0){
104             printk(KERN_INFO "Cannot add the device to the system\n");
105             goto r_class;
106         }
107
108         /*Creating struct class*/
109         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
110             printk(KERN_INFO "Cannot create the struct class\n");
111             goto r_class;
112         }
113
114         /*Creating device*/
115         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
116             printk(KERN_INFO "Cannot create the Device 1\n");
117             goto r_device;
118         }
```

```
119
120         //Initialize wait queue
121         init_waitqueue_head(&wait_queue_etx);
122
123         //Create the kernel thread with name 'mythread'
124         wait_thread = kthread_create(wait_function, NULL, "WaitThread");
125         if (wait_thread) {
126                 printk("Thread Created successfully\n");
127                 wake_up_process(wait_thread);
128         } else
129                 printk(KERN_INFO "Thread creation failed\n");
130
131         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
132     return 0;
133
134 r_device:
135         class_destroy(dev_class);
136 r_class:
137         unregister_chrdev_region(dev,1);
138         return -1;
139 }
140
141 void __exit etx_driver_exit(void)
142 {
143         wait_queue_flag = 2;
144         wake_up_interruptible(&wait_queue_etx);
145         device_destroy(dev_class,dev);
146         class_destroy(dev_class);
147         cdev_del(&etx_cdev);
148         unregister_chrdev_region(dev, 1);
149     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
150 }
151
152 module_init(etx_driver_init);
153 module_exit(etx_driver_exit);
154
155 MODULE_LICENSE("GPL");
156 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
157 MODULE_DESCRIPTION("A simple device driver");
158 MODULE_VERSION("1.8");
```

MakeFile

```
1 obj-m += driver.o
2 KDIR = /lib/modules/$(shell uname -r)/build
3 all:
4     make -C $(KDIR)  M=$(shell pwd) modules
5 clean:
6     make -C $(KDIR)  M=$(shell pwd) clean
```

3

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 the 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!!!
```

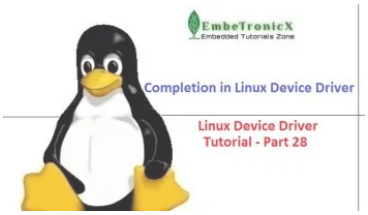
- Now the condition was 2. So it will return from the thread and remove the driver.

That’s all about waitqueue. In our next tutorial, we will discuss sysfs in the Linux device driver.

Like this:

Loading...

Related



Linux Device Driver Tutorial Part 28 – Completion in Linux Device Driver
In "Device Drivers"

Linux Device Driver Tutorial Part 14 – Workqueue in Linux Kernel Part 1
In "Device Drivers"

Linux Device Driver Part 1 : Introduction
In "Device Drivers"

3

3 COMMENTS Oldest

Anonymous

February 27, 2018 2:04 PM

Can you guys pls provide information on “container_of” structure, its implementation and usage when we work with kernel data Structures etc

Loading...

0 Reply

owl

Author March 9, 2018 2:00 PM

Hi,

You can find the container_of macro tutorial in the below link.

https://embetronicx.com/tutorials/linux/c-programming/understanding-of-container_of-macro-in-linux-kernel/

Loading...

0 Reply

Karishma

May 2, 2020 5:08 PM

Simple but worth. The way you ppl are explaining is just awesome. Anyone can easily understand. Thank you guys.

Loading...

0 Reply

3