

## 第8章

# Linux设备驱动中的阻塞与非阻塞I/O

### 本章导读

阻塞和非阻塞I/O是设备访问的两种不同模式，驱动程序可以灵活地支持这两种用户空间对设备的访问方式。

8.1节讲述了阻塞和非阻塞I/O的区别，并讲解了实现阻塞I/O的等待队列机制，以及在globalfifo设备驱动中增加对阻塞I/O支持的方法，并进行了用户空间的验证。

8.2节讲述了设备驱动轮询(Poll)操作的概念和编程方法，轮询可以帮助用户了解是否能对设备进行无阻塞访问。

8.3节讲解在globalfifo中增加轮询操作的方法，并使用select、epoll在用户空间进行了验证。

## 8.1 阻塞与非阻塞I/O

阻塞操作是指在执行设备操作时，若不能获得资源，则挂起进程直到满足可操作的条件后再进行操作。被挂起的进程进入睡眠状态，被从调度器的运行队列移走，直到等待的条件被满足。而非阻塞操作的进程在不能进行设备操作时，并不挂起，它要么放弃，要么不停地查询，直至可以进行操作为止。

驱动程序通常需要提供这样的能力：当应用程序进行read()、write()等系统调用时，若设备的资源不能获取，而用户又希望以阻塞的方式访问设备，驱动程序应在设备驱动的xxx\_read()、xxx\_write()等操作中将进程阻塞直到资源可以获取，此后，应用程序的read()、write()等调用才返回，整个过程仍然进行了正确的设备访问，用户并没有感知到；若用户以非阻塞的方式访问设备文件，则当设备资源不可获取时，设备驱动的xxx\_read()、xxx\_write()等操作应立即返回，read()、write()等系统调用也随即被返回，应用程序收到-EAGAIN返回值。

如图8.1所示，在阻塞访问时，不能获取资源的进程将进入休眠，它将CPU资源“礼让”给其他进程。因为阻塞的进程会进入休眠状态，所以必须确保有一个地方能够唤醒休眠的进程，否则，进程就真的“寿终正寝”了。唤醒进程的地方最大可能发生在中断里面，因为在硬件资源获得的同时往往伴随着一个中断。而非阻塞的进程则不断尝试，直到可以进行I/O。

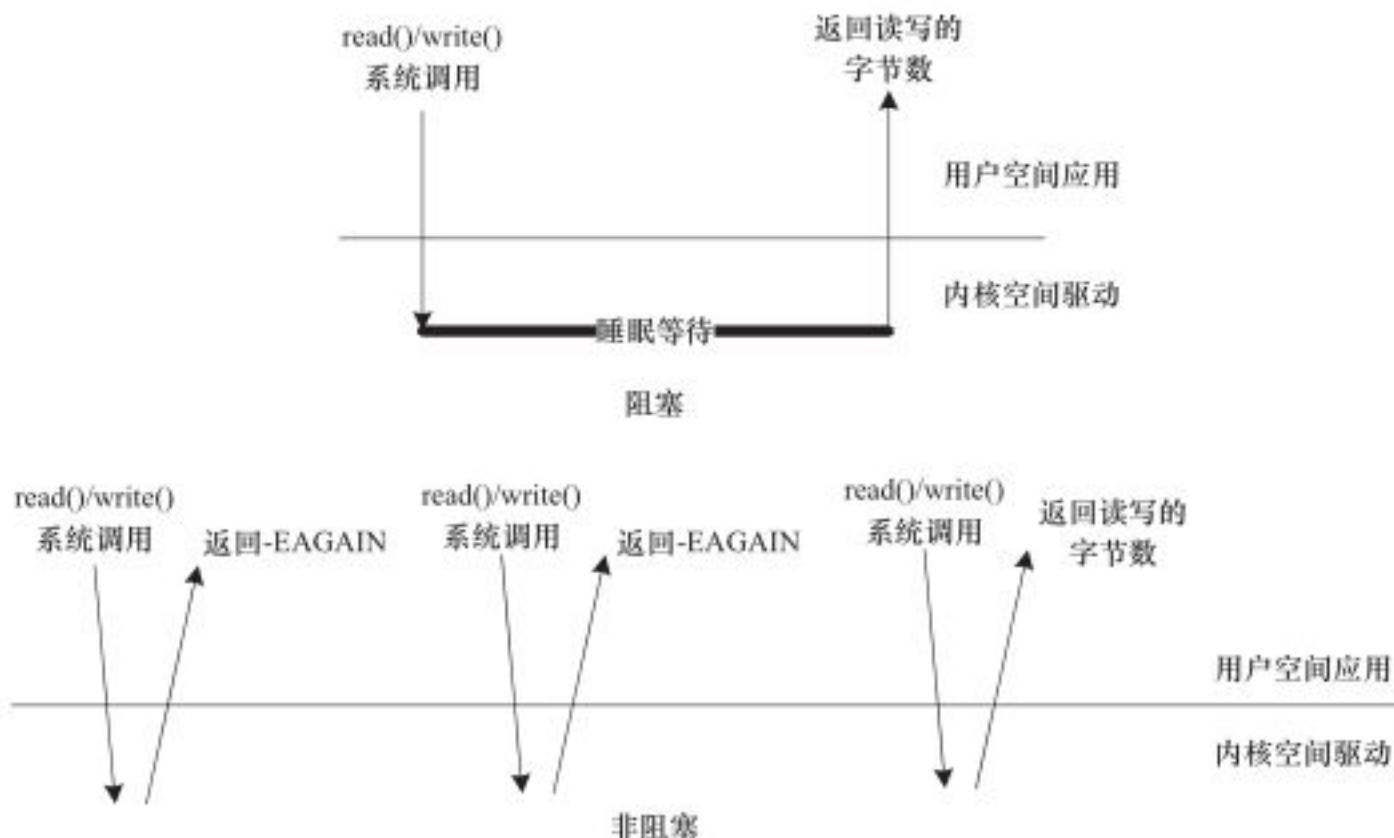


图 8.1 阻塞与非阻塞 I/O

代码清单 8.1 和 8.2 分别演示了以阻塞和非阻塞方式读取串口一个字符的代码。前者在打开文件的时候没有 `O_NONBLOCK` 标记，后者使用 `O_NONBLOCK` 标记打开文件。

---

#### 代码清单 8.1 阻塞地读串口一个字符

---

```
char buf;
fd = open("/dev/ttyS1", O_RDWR);
...
res = read(fd,&buf,1); /* 当串口上有输入时才返回 */
if(res==1)
printf("%c\n", buf);
```

---



---

#### 代码清单 8.2 非阻塞地读串口一个字符

---

```
char buf;
fd = open("/dev/ttyS1", O_RDWR| O_NONBLOCK);
...
while(read(fd,&buf,1)!=-1)
    continue; /* 串口上无输入也返回，因此要循环尝试读取串口 */
printf("%c\n", buf);
```

---

除了在打开文件时可以指定阻塞还是非阻塞方式以外，在文件打开后，也可以通过 `ioctl()` 和 `fcntl()` 改变读写的方式，如从阻塞变更为非阻塞或者从非阻塞变更为阻塞。例如，调用 `fcntl(fd, F_SETFL, O_NONBLOCK)` 可以设置 `fd` 对应的 I/O 为非阻塞。

### 8.1.1 等待队列

在Linux驱动程序中，可以使用等待队列（Wait Queue）来实现阻塞进程的唤醒。等待队列很早就作为一个基本的功能单位出现在Linux内核里了，它以队列为基础数据结构，与进程调度机制紧密结合，可以用来同步对系统资源的访问，第7章中所讲述的信号量在内核中也依赖等待队列来实现。

Linux内核提供了如下关于等待队列的操作。

#### 1. 定义“等待队列头部”

```
wait_queue_head_t my_queue;
```

`wait_queue_head_t` 是 `_wait_queue_head` 结构体的一个 `typedef`。

#### 2. 初始化“等待队列头部”

```
init_waitqueue_head(&my_queue);
```

而下面的 `DECLARE_WAIT_QUEUE_HEAD()` 宏可以作为定义并初始化等待队列头部的“快捷方式”。

```
DECLARE_WAIT_QUEUE_HEAD(name)
```

#### 3. 定义等待队列元素

```
DECLARE_WAITQUEUE(name, tsk)
```

该宏用于定义并初始化一个名为 `name` 的等待队列元素。

#### 4. 添加 / 移除等待队列

```
void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
```

`add_wait_queue()` 用于将等待队列元素 `wait` 添加到等待队列头部 `q` 指向的双向链表中，而 `remove_wait_queue()` 用于将等待队列元素 `wait` 从由 `q` 头部指向的链表中移除。

#### 5. 等待事件

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

等待第1个参数 `queue` 作为等待队列头部的队列被唤醒，而且第2个参数 `condition` 必须满足，否则继续阻塞。`wait_event()` 和 `wait_event_interruptible()` 的区别在于后者可以被信号打断，而前者不能。加上 `_timeout` 后的宏意味着阻塞等待的超时时间，以jiffy为单位，在第3个参数的 `timeout` 到达时，不论 `condition` 是否满足，均返回。

#### 6. 唤醒队列

```
void wake_up(wait_queue_head_t *queue);
```

```
void wake_up_interruptible(wait_queue_head_t *queue);
```

上述操作会唤醒以 queue 作为等待队列头部的队列中所有的进程。

wake\_up() 应该与 wait\_event() 或 wait\_event\_timeout() 成对使用，而 wake\_up\_interruptible() 则应与 wait\_event\_interruptible() 或 wait\_event\_interruptible\_timeout() 成对使用。wake\_up() 可唤醒处于 TASK\_INTERRUPTIBLE 和 TASK\_UNINTERRUPTIBLE 的进程，而 wake\_up\_interruptible() 只能唤醒处于 TASK\_INTERRUPTIBLE 的进程。

### 7. 在等待队列上睡眠

```
sleep_on(wait_queue_head_t *q);
interruptible_sleep_on(wait_queue_head_t *q);
```

sleep\_on() 函数的作用就是将目前进程的状态置成 TASK\_UNINTERRUPTIBLE，并定义一个等待队列元素，之后把它挂到等待队列头部 q 指向的双向链表，直到资源可获得，q 队列指向链接的进程被唤醒。

interruptible\_sleep\_on() 与 sleep\_on() 函数类似，其作用是将目前进程的状态置成 TASK\_INTERRUPTIBLE，并定义一个等待队列元素，之后把它附属到 q 指向的队列，直到资源可获得（q 指引的等待队列被唤醒）或者进程收到信号。

sleep\_on() 函数应该与 wake\_up() 成对使用，interruptible\_sleep\_on() 应该与 wake\_up\_interruptible() 成对使用。

代码清单 8.3 演示了一个在设备驱动中使用等待队列的模版，在进行写 I/O 操作的时候，判断设备是否可写，如果不可写且为阻塞 I/O，则进程睡眠并挂起到等待队列。

代码清单 8.3 在设备驱动中使用等待队列

---

```

1 static ssize_t xxx_write(struct file *file, const char *buffer, size_t count,
2                         loff_t *ppos)
3 {
4     ...
5     DECLARE_WAITQUEUE(wait, current);           /* 定义等待队列元素 */
6     add_wait_queue(&xxx_wait, &wait);          /* 添加元素到等待队列 */
7
8     /* 等待设备缓冲区可写 */
9     do {
10         avail = device_writable(...);
11         if (avail < 0) {
12             if (file->f_flags & O_NONBLOCK) {           /* 非阻塞 */
13                 ret = -EAGAIN;
14                 goto out;
15             }
16             __set_current_state(TASK_INTERRUPTIBLE);    /* 改变进程状态 */
17             schedule();                                /* 调度其他进程执行 */
18             if (signal_pending(current)) {            /* 如果是因为信号唤醒 */
19                 ret = -ERESTARTSYS;
20                 goto out;
21             }
22         }
23     } while (avail < 0);

```

```

24
25     /* 写设备缓冲区 */
26     device_write(...)
27     out:
28     remove_wait_queue(&xxx_wait, &wait);      /* 将元素移出 xxx_wait 指引的队列 */
29     set_current_state(TASK_RUNNING);          /* 设置进程状态为 TASK_RUNNING */
30     return ret;
31 }

```

读懂代码清单 8.3 对理解 Linux 进程状态切换非常重要，所以提请读者反复阅读此段代码（尤其注意其中黑体的部分），直至完全领悟，几个要点如下。

- 1) 如果是非阻塞访问 (`O_NONBLOCK` 被设置)，设备忙时，直接返回 “-EAGAIN”。
- 2) 对于阻塞访问，会调用 `_set_current_state(TASK_INTERRUPTIBLE)` 进行进程状态切换并显示通过 “`schedule()`” 调度其他进程执行。
- 3) 醒来的时候要注意，由于调度出去的时候，进程状态是 `TASK_INTERRUPTIBLE`，即浅度睡眠，所以唤醒它的有可能是信号，因此，我们首先通过 `signal_pending(current)` 了解是不是信号唤醒的，如果是，立即返回 “-ERESTARTSYS”。

`DECLARE_WAITQUEUE`、`add_wait_queue` 这两个动作加起来完成的效果如图 8.2 所示。在 `wait_queue_head_t` 指向的链表上，新定义的 `wait_queue` 元素被插入，而这个新插入的元素绑定了一个 `task_struct`（当前做 `xxx_write` 的 `current`，这也是 `DECLARE_WAITQUEUE` 使用 “`current`” 作为参数的原因）。

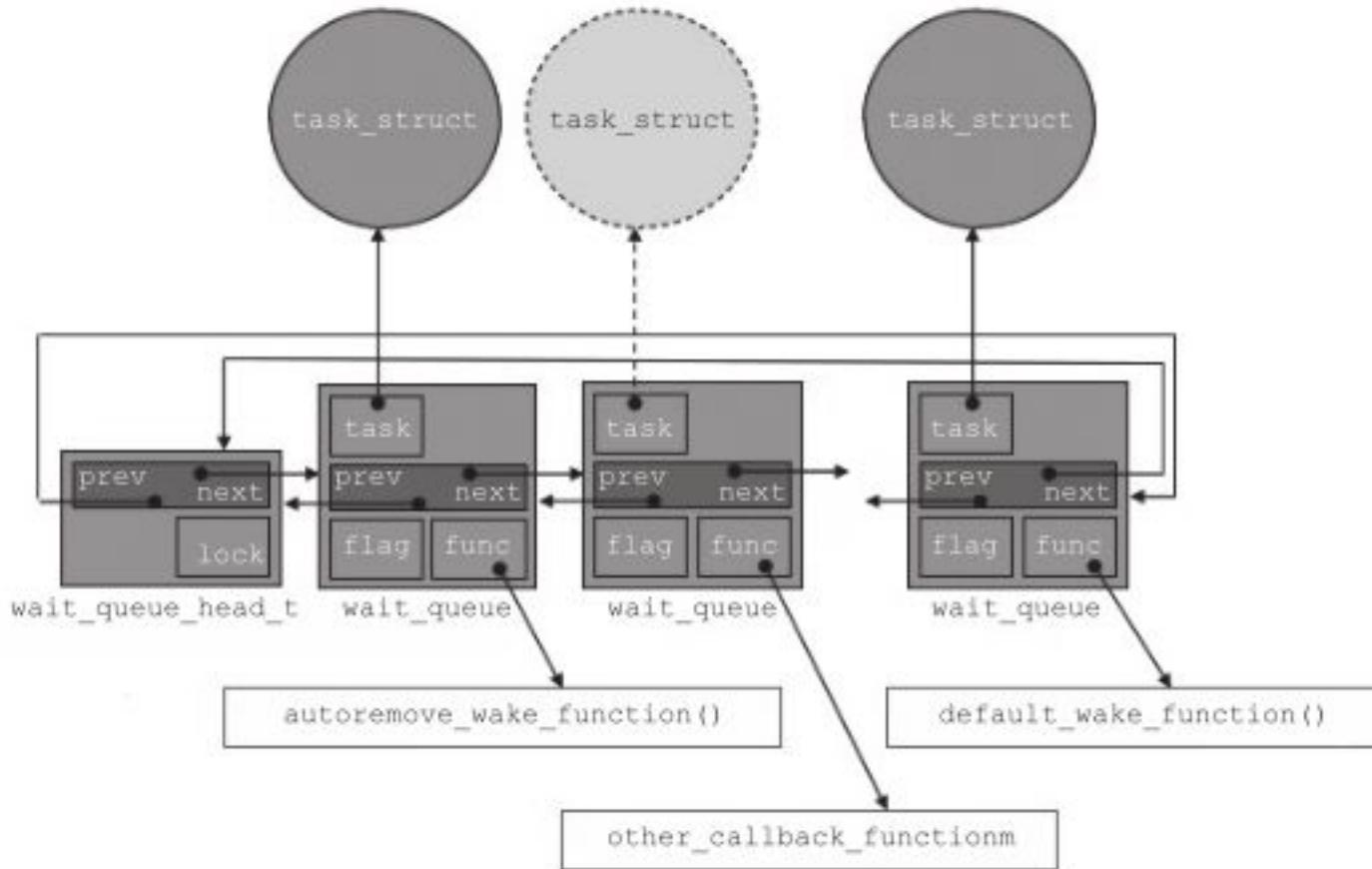


图 8.2 `wait_queue_head_t`、`wait_queue` 和 `task_struct` 之间的关系

### 8.1.2 支持阻塞操作的 globalfifo 设备驱动

现在我们给 globalmem 增加这样的约束：把 globalmem 中的全局内存变成一个 FIFO，只有当 FIFO 中有数据的时候（即有进程把数据写到这个 FIFO 而且没有被读进程读空），读进程才能把数据读出，而且读取后的数据会从 globalmem 的全局内存中被拿掉；只有当 FIFO 不是满的时（即还有一些空间未被写，或写满后被读进程从这个 FIFO 中读出了数据），写进程才能往这个 FIFO 中写入数据。

现在，将 globalmem 重命名为“globalfifo”，在 globalfifo 中，读 FIFO 将唤醒写 FIFO 的进程（如果之前 FIFO 正好是满的），而写 FIFO 也将唤醒读 FIFO 的进程（如果之前 FIFO 正好是空的）。首先，需要修改设备结构体，在其中增加两个等待队列头部，分别对应于读和写，如代码清单 8.4 所示。

代码清单 8.4 globalfifo 设备结构体

---

```

1 struct globalfifo_dev {
2     struct cdev cdev;
3     unsigned int current_len;
4     unsigned char mem[GLOBALFIFO_SIZE];
5     struct mutex mutex;
6     wait_queue_head_t r_wait;
7     wait_queue_head_t w_wait;
8 };

```

---

与 globalfifo 设备结构体的另一个不同是增加了 current\_len 成员以用于表征目前 FIFO 中有效数据的长度。current\_len 等于 0 意味着 FIFO 空，current\_len 等于 GLOBALFIFO\_SIZE 意味着 FIFO 满。

这两个等待队列头部需在设备驱动模块加载函数中调用 init\_waitqueue\_head() 被初始化，新的设备驱动模块加载函数如代码清单 8.5 所示。

代码清单 8.5 globalfifo 设备驱动模块加载函数

---

```

1 static int __init globalfifo_init(void)
2 {
3     int ret;
4     dev_t devno = MKDEV(globalfifo_major, 0);
5
6     if (globalfifo_major)
7         ret = register_chrdev_region(devno, 1, "globalfifo");
8     else {
9         ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");
10        globalfifo_major = MAJOR(devno);
11    }
12    if (ret < 0)
13        return ret;
14
15    globalfifo_devp = kzalloc(sizeof(struct globalfifo_dev), GFP_KERNEL);

```

---

```

16     if (!globalfifo_devp) {
17         ret = -ENOMEM;
18         goto fail_malloc;
19     }
20
21     globalfifo_setup_cdev(globalfifo_devp, 0);
22
23     mutex_init(&globalfifo_devp->mutex);
24     init_waitqueue_head(&globalfifo_devp->r_wait);
25     init_waitqueue_head(&globalfifo_devp->w_wait);
26
27     return 0;
28
29 fail_malloc:
30     unregister_chrdev_region(devno, 1);
31     return ret;
32 }
33 module_init(globalfifo_init);

```

设备驱动读写操作需要被修改，在读函数中需增加唤醒 globalfifo\_devp->w\_wait 的语句，而在写操作中唤醒 globalfifo\_devp->r\_wait，如代码清单 8.6 所示。

代码清单 8.6 增加等待队列后的 globalfifo 读写函数

```

1 static ssize_t globalfifo_read(struct file *filp, char __user *buf,
2                               size_t count, loff_t *ppos)
3 {
4     int ret;
5     struct globalfifo_dev *dev = filp->private_data;
6     DECLARE_WAITQUEUE(wait, current);
7
8     mutex_lock(&dev->mutex);
9     add_wait_queue(&dev->r_wait, &wait);
10
11    while (dev->current_len == 0) {
12        if (filp->f_flags & O_NONBLOCK) {
13            ret = -EAGAIN;
14            goto out;
15        }
16        __set_current_state(TASK_INTERRUPTIBLE);
17        mutex_unlock(&dev->mutex);
18
19        schedule();
20        if (signal_pending(current)) {
21            ret = -ERESTARTSYS;
22            goto out2;
23        }
24
25        mutex_lock(&dev->mutex);
26    }

```

```
27
28 if (count > dev->current_len)
29     count = dev->current_len;
30
31 if (copy_to_user(buf, dev->mem, count)) {
32     ret = -EFAULT;
33     goto out;
34 } else {
35     memcpy(dev->mem, dev->mem + count, dev->current_len - count);
36     dev->current_len -= count;
37     printk(KERN_INFO "read %d bytes(s), current_len:%d\n", count,
38           dev->current_len);
39
40     wake_up_interruptible(&dev->w_wait);
41
42     ret = count;
43 }
44 out:
45 mutex_unlock(&dev->mutex);
46 out2:
47 remove_wait_queue(&dev->w_wait, &wait);
48 set_current_state(TASK_RUNNING);
49 return ret;
50 }
51
52 static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
53                                 size_t count, loff_t *ppos)
54 {
55 struct globalfifo_dev *dev = filp->private_data;
56 int ret;
57 DECLARE_WAITQUEUE(wait, current);
58
59 mutex_lock(&dev->mutex);
60 add_wait_queue(&dev->w_wait, &wait);
61
62 while (dev->current_len == GLOBALFIFO_SIZE) {
63     if (filp->f_flags & O_NONBLOCK) {
64         ret = -EAGAIN;
65         goto out;
66     }
67     __set_current_state(TASK_INTERRUPTIBLE);
68
69     mutex_unlock(&dev->mutex);
70
71     schedule();
72     if (signal_pending(current)) {
73         ret = -ERESTARTSYS;
74         goto out2;
75     }
76
77     mutex_lock(&dev->mutex);
78 }
```

```

79
80 if (count > GLOBALFIFO_SIZE - dev->current_len)
81     count = GLOBALFIFO_SIZE - dev->current_len;
82
83 if (copy_from_user(dev->mem + dev->current_len, buf, count)) {
84     ret = -EFAULT;
85     goto out;
86 } else {
87     dev->current_len += count;
88     printk(KERN_INFO "written %d bytes(s), current_len:%d\n", count,
89            dev->current_len);
90
91     wake_up_interruptible(&dev->r_wait);
92
93     ret = count;
94 }
95
96 out:
97 mutex_unlock(&dev->mutex);;
98 out2:
99 remove_wait_queue(&dev->w_wait, &wait);
100 set_current_state(TASK_RUNNING);
101 return ret;
102 }

```

---

`globalfifo_read()` 通过第 6 行和第 9 行将自己加到了 `r_wait` 这个队列里面，但是此时读的进程并未睡眠，之后第 16 行调用 `_set_current_state(TASK_INTERRUPTIBLE)` 时，也只是标记了 `task_struct` 的一个浅度睡眠标记，并未真正睡眠，直到第 19 行调用 `schedule()`，读进程进入睡眠。进行完读操作后，第 40 行调用 `wake_up_interruptible(&dev->w_wait)` 唤醒可能阻塞的写进程。`globalfifo_write()` 的过程与此类似。

关注代码的第 17 行和 69 行，无论是读函数还是写函数，进入 `schedule()` 把自己切换出去之前，都主动释放了互斥体。原因是如果读进程阻塞，实际意味着 FIFO 空，必须依赖写的进程往 FIFO 里面写东西来唤醒它，但是写的进程为了写 FIFO，它也必须拿到这个互斥体来访问 FIFO 这个临界资源，如果读进程把自己调度出去之前不释放这个互斥体，那么读写进程之间就死锁了。所谓死锁，就是多个进程循环等待他方占有的资源而无限期地僵持下去。如果没有外力的作用，那么死锁涉及的各个进程都将永远处于封锁状态。因此，驱动工程师一定要注意：当多个等待队列、信号量、互斥体等机制同时出现时，谨防死锁！

现在回过来了看一下代码清单 8.6 的第 12 行和 63 行，发现在设备驱动的 `read()`、`write()` 等功能函数中，可以通过 `filp->f_flags` 标志获得用户空间是否要求非阻塞访问。驱动中可以依据此标志判断用户究竟要求阻塞还是非阻塞访问，从而进行不同的处理。

代码中还有一个关键点，就是无论读函数还是写函数，在进入真正的读写之前，都要再次判断设备是否可以读写，见第 11 行的 `while(dev->current_len == 0)` 和第 62 行的 `while (dev->current_len == GLOBALFIFO_SIZE)`。主要目的是为了让并发的读或者并发的写都正确。设

想如果两个读进程都阻塞在读上，写进程执行的 `wake_up_interruptible(&dev->r_wait)` 实际会同时唤醒它们，其中先执行的那个进程可能会率先将 FIFO 再次读空！

### 8.1.3 在用户空间验证 globalfifo 的读写

本书代码仓库的 `/kernel/drivers/globalfifo/ch8` 包含了 globalfifo 的驱动，运行“make”命令编译得到 `globalfifo.ko`。接着用 `insmod` 模块

```
# insmod globalfifo.ko
```

创建设备文件节点“`/dev/globalfifo`”，具体如下：

```
# mknod /dev/globalfifo c 231 0
```

启动两个进程，一个进程 `cat /dev/globalfifo &` 在后台执行，一个进程“`echo` 字符串 `/dev/globalfifo`”在前台执行：

```
# cat /dev/globalfifo &
[1] 20910

# echo 'I want to be' > /dev/globalfifo
I want to be

# echo 'a great Chinese Linux Kernel Developer' > /dev/globalfifo
a great Chinese Linux kernel Developer
```

每当 `echo` 进程向 `/dev/globalfifo` 写入一串数据，`cat` 进程就立即将该串数据显现出来，好的，让我们抱着这个信念“`I want to be a great Chinese Linux Kernel Developer`”继续前行吧！

往 `/dev/globalfifo` 里面 `echo` 需要 root 权限，直接运行“`sudo echo`”是不行的，可以先执行：

```
baohua@baohua-VirtualBox://sys/module/globalmem$ sudo su
[sudo] password for baohua:
```

这段代码的密码也是“`baohua`”。之后再进行 `echo`。

## 8.2 轮询操作

### 8.2.1 轮询的概念与作用

在用户程序中，`select()` 和 `poll()` 也是与设备阻塞与非阻塞访问息息相关的论题。使用非阻塞 I/O 的应用程序通常会使用 `select()` 和 `poll()` 系统调用查询是否可对设备进行无阻塞的访问。`select()` 和 `poll()` 系统调用最终会使设备驱动中的 `poll()` 函数被执行，在 Linux2.5.45 内核中还引入了 `epoll()`，即扩展的 `poll()`。

`select()` 和 `poll()` 系统调用的本质一样，前者在 BSD UNIX 中引入，后者在 System V 中引入。

### 8.2.2 应用程序中的轮询编程

应用程序中最广泛用到的是BSD UNIX中引入的select()系统调用，其原型为：

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
          struct timeval *timeout);
```

其中readfds、writefds、exceptfds分别是被select()监视的读、写和异常处理的文件描述符集合，numfds的值是需要检查的号码最高的fd加1。readfds文件集中的任何一个文件变得可读，select()返回；同理，writefds文件集中的任何一个文件变得可写，select也返回。

如图8.3所示，第一次对n个文件进行select()的时候，若任何一个文件满足要求，select()就直接返回；第2次再进行select()的时候，没有文件满足读写要求，select()的进程阻塞且睡眠。由于调用select()的时候，每个驱动的poll()接口都会被调用到，实际上执行select()的进程被挂到了每个驱动的等待队列上，可以被任何一个驱动唤醒。如果FD<sub>n</sub>变得可读写，select()返回。

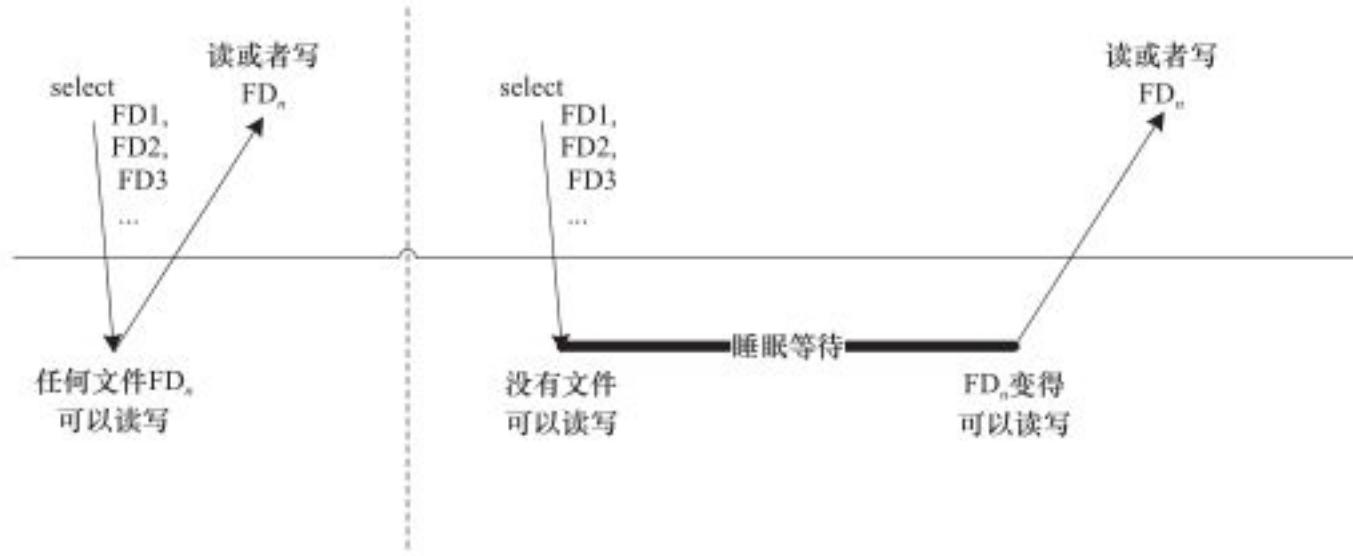


图8.3 多路复用select()

timeout参数是一个指向struct timeval类型的指针，它可以使select()在等待timeout时间后若仍然没有文件描述符准备好则超时返回。struct timeval数据结构的定义如代码清单8.7所示。

代码清单8.7 timeval结构体定义

---

```
1 struct timeval {
2     int tv_sec;      /* 秒 */
3     int tv_usec;    /* 微秒 */
4 };
```

---

下列操作用来设置、清除、判断文件描述符集合：

```
FD_ZERO(fd_set *set)
```

清除一个文件描述符集合；

```
FD_SET(int fd,fd_set *set)
```

将一个文件描述符加入文件描述符集合中；

```
FD_CLR(int fd,fd_set *set)
```

将一个文件描述符从文件描述符集合中清除；

```
FD_ISSET(int fd,fd_set *set)
```

判断文件描述符是否被置位。

`poll()` 的功能和实现原理与 `select()` 相似，其函数原型为：

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

当多路复用的文件数量庞大、I/O 流量频繁的时候，一般不太适合使用 `select()` 和 `poll()`，此种情况下，`select()` 和 `poll()` 的性能表现较差，我们宜使用 `epoll`。`epoll` 的最大好处是不会随着 `fd` 的数目增长而降低效率，`select()` 则会随着 `fd` 的数量增大性能下降明显。

与 `epoll` 相关的用户空间编程接口包括：

```
int epoll_create(int size);
```

创建一个 `epoll` 的句柄，`size` 用来告诉内核要监听多少个 `fd`。需要注意的是，当创建好 `epoll` 句柄后，它本身也会占用一个 `fd` 值，所以在使用完 `epoll` 后，必须调用 `close()` 关闭。

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

告诉内核要监听什么类型的事件。第 1 个参数是 `epoll_create()` 的返回值，第 2 个参数表示动作，包含：

`EPOLL_CTL_ADD`：注册新的 `fd` 到 `epfd` 中。

`EPOLL_CTL_MOD`：修改已经注册的 `fd` 的监听事件。

`EPOLL_CTL_DEL`：从 `epfd` 中删除一个 `fd`。

第 3 个参数是需要监听的 `fd`，第 4 个参数是告诉内核需要监听的事件类型，`struct epoll_event` 结构如下：

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

`events` 可以是以下几个宏的“或”：

`EPOLLIN`：表示对应的文件描述符可以读。

`EPOLLOUT`：表示对应的文件描述符可以写。

`EPOLLPRI`：表示对应的文件描述符有紧急的数据可读（这里应该表示的是有 socket 带

外数据到来)。

**EPOLLERR**: 表示对应的文件描述符发生错误。

**EPOLLHUP**: 表示对应的文件描述符被挂断。

**EPOLLET**: 将 epoll 设为边缘触发 (Edge Triggered) 模式, 这是相对于水平触发 (Level Triggered) 来说的。LT (Level Triggered) 是缺省的工作方式, 在 LT 情况下, 内核告诉用户一个 fd 是否就绪了, 之后用户可以对这个就绪的 fd 进行 I/O 操作。但是如果用户不进行任何操作, 该事件并不会丢失, 而 ET (Edge-Triggered) 是高速工作方式, 在这种模式下, 当 fd 从未就绪变为就绪时, 内核通过 epoll 告诉用户, 然后它会假设用户知道 fd 已经就绪, 并且不会再为那个 fd 发送更多的就绪通知。

**EPOLLONESHOT**: 意味着一次性监听, 当监听完这次事件之后, 如果还需要继续监听这个 fd 的话, 需要再次把这个 fd 加入到 epoll 队列里。

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

等待事件的产生, 其中 events 参数是输出参数, 用来从内核得到事件的集合, maxevents 告诉内核本次最多收多少事件, maxevents 的值不能大于创建 epoll\_create() 时的 size, 参数 timeout 是超时时间 (以毫秒为单位, 0 意味着立即返回, -1 意味着永久等待)。该函数的返回值是需要处理的事件数目, 如返回 0, 则表示已超时。

位于 <https://www.kernel.org/doc/ols/2004/ols2004v1-pages-215-226.pdf> 的文档《Comparing and Evaluating epoll, select, and poll Event Mechanisms》对比了 select、epoll、poll 之间的一些性能。一般来说, 当涉及的 fd 数量较少的时候, 使用 select 是合适的; 如果涉及的 fd 很多, 如在大规模并发的服务器中侦听许多 socket 的时候, 则不太适合选用 select, 而适合选用 epoll。

### 8.2.3 设备驱动中的轮询编程

设备驱动中 poll() 函数的原型是:

```
unsigned int(*poll)(struct file * filp, struct poll_table* wait);
```

第 1 个参数为 file 结构体指针, 第 2 个参数为轮询表指针。这个函数应该进行两项工作。

1) 对可能引起设备文件状态变化的等待队列调用 poll\_wait() 函数, 将对应的等待队列头部添加到 poll\_table 中。

2) 返回表示是否能对设备进行无阻塞读、写访问的掩码。

用于向 poll\_table 注册等待队列的关键 poll\_wait() 函数的原型如下:

```
void poll_wait(struct file *filp, wait_queue_head_t *queue, poll_table * wait);
```

poll\_wait() 函数的名称非常容易让人产生误会, 以为它和 wait\_event() 等一样, 会阻塞地等待某事件的发生, 其实这个函数并不会引起阻塞。poll\_wait() 函数所做的工作是把当前进

程添加到 wait 参数指定的等待列表（poll\_table）中，实际作用是让唤醒参数 queue 对应的等待队列可以唤醒因 select() 而睡眠的进程。

驱动程序 poll() 函数应该返回设备资源的可获取状态，即 POLLIN、POLLOUT、POLLPRI、POLLERR、POLLNVAL 等宏的位“或”结果。每个宏的含义都表明设备的一种状态，如 POLLIN（定义为 0x0001）意味着设备可以无阻塞地读，POLLOUT（定义为 0x0004）意味着设备可以无阻塞地写。

通过以上分析，可得出设备驱动中 poll() 函数的典型模板，如代码清单 8.8 所示。

代码清单 8.8 poll() 函数典型模板

---

```

1 static unsigned int xxx_poll(struct file *filp, poll_table *wait)
2 {
3     unsigned int mask = 0;
4     struct xxx_dev *dev = filp->private_data; /* 获得设备结构体指针 */
5
6     ...
7     poll_wait(filp, &dev->r_wait, wait);          /* 加入读等待队列 */
8     poll_wait(filp, &dev->w_wait, wait);          /* 加入写等待队列 */
9
10    if (...)                                     /* 可读 */
11        mask |= POLLIN | POLLRDNORM;             /* 标示数据可获得(对用户可读) */
12
13    if (...)                                     /* 可写 */
14        mask |= POLLOUT | POLLWRNORM;            /* 标示数据可写入 */
15    ...
16    return mask;
17 }
```

---

## 8.3 支持轮询操作的 globalfifo 驱动

### 8.3.1 在 globalfifo 驱动中增加轮询操作

在 globalfifo 的 poll() 函数中，首先将设备结构体中的 r\_wait 和 w\_wait 等待队列头部添加到等待列表中（意味着因调用 select 而阻塞的进程可以被 r\_wait 和 w\_wait 唤醒），然后通过判断 dev->current\_len 是否等于 0 来获得设备的可读状态，通过判断 dev->current\_len 是否等于 GLOBALFIFO\_SIZE 来获得设备的可写状态，如代码清单 8.9 所示。

代码清单 8.9 globalfifo 设备驱动的 poll() 函数

---

```

1 static unsigned int globalfifo_poll(struct file *filp, poll_table * wait)
2 {
3     unsigned int mask = 0;
4     struct globalfifo_dev *dev = filp->private_data;
5
6     mutex_lock(&dev->mutex);;
```

---

```

8     poll_wait(filp, &dev->r_wait, wait);
9     poll_wait(filp, &dev->w_wait, wait);
10
11    if (dev->current_len != 0) {
12        mask |= POLLIN | POLLRDNORM;
13    }
14
15    if (dev->current_len != GLOBALFIFO_SIZE) {
16        mask |= POLLOUT | POLLWRNORM;
17    }
18
19    mutex_unlock(&dev->mutex);
20
21    return mask;
22}

```

注意，要把 globalfifo\_poll 赋给 globalfifo\_fops 的 poll 成员：

```

static const struct file_operations globalfifo_fops = {
    ...
    .poll = globalfifo_poll,
    ...
};

```

### 8.3.2 在用户空间中验证 globalfifo 设备的轮询

编写一个应用程序 globalfifo\_poll.c，以用 select() 监控 globalfifo 的可读写状态，这个程序如代码清单 8.10 所示。

代码清单 8.10 使用 select 监控 globalfifo 是否可非阻塞读、写的应用程序

```

1 #define FIFO_CLEAR 0x1
2 #define BUFFER_LEN 20
3 void main(void)
4 {
5     int fd, num;
6     char rd_ch[BUFFER_LEN];
7     fd_set rfds, wfds; /* 读 / 写文件描述符集 */
8
9     /* 以非阻塞方式打开 /dev/globalfifo 设备文件 */
10    fd = open("/dev/globalfifo", O_RDONLY | O_NONBLOCK);
11    if (fd != -1) {
12        /* FIFO 清 0 */
13        if (ioctl(fd, FIFO_CLEAR, 0) < 0)
14            printf("ioctl command failed\n");
15
16        while (1) {
17            FD_ZERO(&rfds);
18            FD_ZERO(&wfds);
19            FD_SET(fd, &rfds);
20            FD_SET(fd, &wfds);

```

```

21
22         select(fd + 1, &rfds, &wfds, NULL, NULL);
23         /* 数据可获得 */
24         if (FD_ISSET(fd, &rfds))
25             printf("Poll monitor:can be read\n");
26         /* 数据可写入 */
27         if (FD_ISSET(fd, &wfd))
28             printf("Poll monitor:can be written\n");
29     }
30 } else {
31     printf("Device open failure\n");
32 }
33 }

```

在运行时可看到，当没有任何输入，即 FIFO 为空时，程序不断地输出 Poll monitor:can be written，当通过 echo 向 /dev/globalfifo 写入一些数据后，将输出 Poll monitor:can be read 和 Poll monitor:can be written，如果不间断地通过 echo 向 /dev/globalfifo 写入数据直至写满 FIFO，则发现 pollmonitor 程序将只输出 Poll monitor:can be read。对于 globalfifo 而言，不会出现既不能读，又不能写的情况。

编写一个应用程序 globalfifo\_epoll.c，以用 epoll 监控 globalfifo 的可读状态，这个程序如代码清单 8.11 所示。

**代码清单 8.11 使用 epoll 监控 globalfifo 是否可非阻塞读的应用程序**

```

1 #define FIFO_CLEAR 0x1
2 #define BUFFER_LEN 20
3 void main(void)
4 {
5     int fd;
6
7     fd = open("/dev/globalfifo", O_RDONLY | O_NONBLOCK);
8     if (fd != -1) {
9         struct epoll_event ev_globalfifo;
10        int err;
11        int epfd;
12
13        if (ioctl(fd, FIFO_CLEAR, 0) < 0)
14            printf("ioctl command failed\n");
15
16        epfd = epoll_create(1);
17        if (epfd < 0) {
18            perror("epoll_create()");
19            return;
20        }
21
22        bzero(&ev_globalfifo, sizeof(struct epoll_event));
23        ev_globalfifo.events = EPOLLIN | EPOLLPRI;
24

```

```

25         err = epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev_globalfifo);
26         if (err < 0) {
27             perror("epoll_ctl()");
28             return;
29         }
30         err = epoll_wait(epfd, &ev_globalfifo, 1, 15000);
31         if (err < 0) {
32             perror("epoll_wait()");
33         } else if (err == 0) {
34             printf("No data input in FIFO within 15 seconds.\n");
35         } else {
36             printf("FIFO is not empty\n");
37         }
38         err = epoll_ctl(epfd, EPOLL_CTL_DEL, fd, &ev_globalfifo);
39         if (err < 0)
40             perror("epoll_ctl()");
41     } else {
42         printf("Device open failure\n");
43     }
44 }
```

---

上述程序第 25 行 `epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev_globalfifo)` 将 `globalfifo` 对应的 `fd` 加入到了侦听的行列，第 23 行设置侦听读事件，第 30 行进行等待，若 15 秒内没有人写 `/dev/globalfifo`，该程序会打印 `No data input in FIFO within 15 seconds`，否则程序会打印 `FIFO is not empty`。

## 8.4 总结

阻塞与非阻塞访问是 I/O 操作的两种不同模式，前者在暂时不可进行 I/O 操作时会让进程睡眠，后者则不然。

在设备驱动中阻塞 I/O 一般基于等待队列或者基于等待队列的其他 Linux 内核 API 来实现，等待队列可用于同步驱动中事件发生的先后顺序。使用非阻塞 I/O 的应用程序也可借助轮询函数来查询设备是否能立即被访问，用户空间调用 `select()`、`poll()` 或者 `epoll` 接口，设备驱动提供 `poll()` 函数。设备驱动的 `poll()` 本身不会阻塞，但是与 `poll()`、`select()` 和 `epoll` 相关的系统调用则会阻塞地等待至少一个文件描述符集合可访问或超时。