LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 8 章 Linux 设备驱动中的阻塞与非阻塞 I/O

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用户空间对设备的这两种访问方式。

8.1 节讲解阻塞 I/O 和非阻塞 I/O 的区别，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 节讲解设备驱动的轮询（poll）操作的概念和编程方法，poll 操作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 poll 操作的方法，并进行了用户空间的验证。

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。被挂起的进程进入休眠状态，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，它或者放弃，或者不停地查询，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。

阻塞从字面上听起来似乎意味着低效率，实则不然，如果设备驱动不阻塞，则用户想获取设备资源只能不停地查询，这反而会无谓地耗费 CPU 资源。而阻塞访问时，不能获取资源的进程将进入休眠，它将 CPU 资源让给其他进程。

因为阻塞的进程会进入休眠状态，因此，必须确保有一个地方能够唤醒休眠的进程。唤醒进程的地方最大可能发生在中断里面，因为硬件资源获得的同时往往伴随着一个中断。

代码清单 8.1 和代码清单 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。实际的串口编程中，若使用非阻塞模式，还可借助信号（sigaction）以异步方式访问串口以提高 CPU 利用率，而这里仅仅是为了说明阻塞与非阻塞的区别。

代码清单 8.1　阻塞地读取串口一个字符

```
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　非阻塞地读取串口一个字符

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

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。wait queue 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，能够用于实现内核中的异步事件通知机制。等待队列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 2.6 提供如下关于等待队列的操作。

1．定义“等待队列头”。

```
wait_queue_head_t my_queue;
```

2．初始化“等待队列头”。

```
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的“快捷方式”。

```
DECLARE_WAIT_QUEUE_HEAD (name)
```

3．定义等待队列。

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

该宏用于定义并初始化一个名为 name 的等待队列。

4．添加/移除等待队列。

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

add_wait_queue()用于将等待队列 wait 添加到等待队列头 q 指向的等待队列链表中，而 remove_wait_queue()用于将等待队列 wait 从附属的等待队列头 q 指向的等待队列链表中移除。

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)
```

等待第一个参数 queue 作为等待队列头的等待队列被唤醒，而且第二个参数 condition 必须满足，否则阻塞。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，在第三个参数的 timeout 到达时，不论 condition 是否满足，均返回。

wait()的定义如代码清单 8.3 所示，从其源代码可以看出，当 condition 满足时，

wait_event()会立即返回，否则，阻塞等待 condition 满足。

代码清单 8.3 wait_event()函数

```
1  #define wait_event(wq, condition)                          \
2  do {                                                 \
3   if (condition)   /*条件满足立即返回*/                  \
4       break;                                          \
5   __wait_event(wq, condition);/*添加等待队列并阻塞*/
6  } while (0)
7
8  #define __wait_event(wq, condition)                        \
9  do {                                                 \
10  DEFINE_WAIT(__wait);                                    \
11                                                      \
12  for (;;) {                                          \
13      prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);   \
14      if (condition)                                  \
15          break;                                      \
16      schedule();/*放弃 CPU*/                             \
17  }                                                   \
18  finish_wait(&wq, &__wait);                              \
19 } while (0)
20
21 void fastcall
22 prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)
23 {
24  unsigned long flags;
25
26  wait->flags &= ~WQ_FLAG_EXCLUSIVE;
27  spin_lock_irqsave(&q->lock, flags);
28  if (list_empty(&wait->task_list))
29      __add_wait_queue(q, wait);//添加等待队列
30  if (is_sync_wait(wait))
31      set_current_state(state);//改变当前进程的状态为休眠
32  spin_unlock_irqrestore(&q->lock, flags);
33 }
34
35 void fastcall finish_wait(wait_queue_head_t *q, wait_queue_t *wait)
36 {
37  unsigned long flags;
```

```
    38
    39  __set_current_state(TASK_RUNNING);// 恢复当前进程的状态为
TASK_RUNNING
    40  if (!list_empty_careful(&wait->task_list)) {
    41      spin_lock_irqsave(&q->lock, flags);
    42      list_del_init(&wait->task_list);
    43      spin_unlock_irqrestore(&q->lock, flags);
    44  }
    45 }
```

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 作为等待队列头的所有等待队列中所有属于该等待队列头的等待队列对应的进程。

wake_up() 应与 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，之后把它附属到等待队列头 q，直到资源可获得，q 引导的等待队列被唤醒。

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，之后把它附属到等待队列头 q，直到资源可获得，q 引导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 8.4 和代码清单 8.5 分别列出了 sleep_on()和 interruptible_sleep_on()函数的源代码。

代码清单 8.4 sleep_on()函数

```
1  void fastcall __sched interruptible_sleep_on(wait_queue_head_t *q)
2  {
3   SLEEP_ON_VAR
4    /* #define SLEEP_ON_VAR                    \
5      unsigned long flags;                     \
6      wait_queue_t wait;                       \
7      init_waitqueue_entry(&wait, current); */
8   current->state = TASK_UNINTERRUPTIBLE;//改变当前进程状态
9
10  SLEEP_ON_HEAD
11   /* #define SLEEP_ON_HEAD                   \
12    spin_lock_irqsave(&q->lock,flags);        \
13    __add_wait_queue(q, &wait);               \
14    spin_unlock(&q->lock);*/
15
16  schedule();//放弃 CPU
17  SLEEP_ON_TAIL
18   /* #define SLEEP_ON_TAIL                   \
19     spin_lock_irq(&q->lock);                 \
20     __remove_wait_queue(q, &wait);           \
21     spin_unlock_irqrestore(&q->lock, flags); */
22 }
```

代码清单 8.5 interruptible_sleep_on()函数

```
1  void fastcall __sched interruptible_sleep_on(wait_queue_head_t *q)
2  {
3   SLEEP_ON_VAR
4
5   current->state = TASK_INTERRUPTIBLE;//改变当前进程状态
6
7   SLEEP_ON_HEAD
8   schedule();//放弃 CPU
9   SLEEP_ON_TAIL
10 }
```

从代码清单 8.4 和代码清单 8.5 可以看出，不论是 sleep_on()还是 interruptible_sleep_on()，其流程都如下所示。

（1）定义并初始化一个等待队列，将进程状态改变为 TASK_UNINTERRUPTIBLE（不能被信号打断）或 TASK_INTERRUPTIBLE（可以被信号打断），并将等待队列添加到等待队列头。

（2）通过 schedule()放弃 CPU，调度其他进程执行。

（3）进程被其他地方唤醒，将等待队列移出等待队列头。

在内核中使用 set_current_state()函数或__add_wait_queue()函数来实现目前进程状态的改变，直接采用 current->state = TASK_UNINTERRUPTIBLE 类似的赋值语句也是可行的。通常而言，set_current_state()函数在任何环境下都可以使用，不会存在并发问题，但是效率要低于__add_ wait_queue()。

因此，在许多设备驱动中，并不调用 sleep_on()或 interruptible_sleep_on()，而是亲自进行进程的状态改变和切换，如代码清单 8.6 所示。

代码清单 8.6 在驱动程序中改变进程状态并调用 schedule()

```
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
6    __add_wait_queue(&xxx_wait, &wait);  //添加等待队列
7
8    ret = count;
9    /* 等待设备缓冲区可写 */
10   do
11   {
12     avail = device_writable(...);
13     if (avail < 0)
14       __set_current_state(TASK_INTERRUPTIBLE);//改变进程状态
15
16     if (avail < 0)
17     {
18       if (file->f_flags &O_NONBLOCK)  //非阻塞
19       {
20         if (!ret)
21           ret =  - EAGAIN;
22         goto out;
23       }
24       schedule();   //调度其他进程执行
25       if (signal_pending(current))//如果是因为信号唤醒
26       {
27         if (!ret)
28           ret =  - ERESTARTSYS;
29         goto out;
30       }
31     }
32   }while (avail < 0);
33
34   /* 写设备缓冲区 */
35   device_write(...)
36   out:
37   remove_wait_queue(&xxx_wait, &wait);//将等待队列移出等待队列头
38   set_current_state(TASK_RUNNING);//设置进程状态为 TASK_RUNNING
```

```
39   return ret;
40 }
```

8.1.2 支持阻塞操作的 globalfifo 设备驱动

现在我们给 globalfifo 增加这样的约束：把 globalfifo 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 globalfifo 的全局内存中被拿掉；只有当 FIFO 非满时（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 globalfifo 重命名为“globalfifo”，在 globalfifo 中，读 FIFO 将唤醒写 FIFO，而写 FIFO 也将唤醒读 FIFO。首先，需要修改设备结构体，在其中增加两个等待队列头，分别对应于读和写，如代码清单 8.7 所示。

代码清单 8.7 globalfifo 设备结构体

```
1 struct globalfifo_dev
2 {
3    struct cdev cdev; /*cdev 结构体*/
4    unsigned int current_len;   /*fifo 有效数据长度*/
5    unsigned char mem[GLOBALFIFO_SIZE]; /*全局内存*/
6    struct semaphore sem; /*并发控制用的信号量*/
7    wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
8    wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
9 };
```

与 globalfifo 设备结构体的另一个不同是增加了 current_len 成员用于表征目前 FIFO 中有效数据的长度。

这个等待队列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 8.8 所示。

代码清单 8.8 globalfifo 设备驱动模块加载函数

```
1  int globalfifo_init(void)
2  {
3    int ret;
4    dev_t devno = MKDEV(globalfifo_major, 0);
5
6    /* 申请设备号*/
7    if (globalfifo_major)
8      ret = register_chrdev_region(devno, 1, "globalfifo");
9    else  /* 动态申请设备号 */
10   {
11     ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");
12     globalfifo_major = MAJOR(devno);
```

```
    13  }
    14  if (ret < 0)
    15    return ret;
    16  /* 动态申请设备结构体的内存*/
    17      globalfifo_devp  =  kmalloc(sizeof(struct  globalfifo_dev),
GFP_KERNEL);
    18  if (!globalfifo_devp)    /*申请失败*/
    19  {
    20    ret =  - ENOMEM;
    21    goto fail_malloc;
    22  }
    23
    24  memset(globalfifo_devp, 0, sizeof(struct globalfifo_dev));
    25
    26  globalfifo_setup_cdev(globalfifo_devp, 0);
    27
    28  init_MUTEX(&globalfifo_devp->sem);   /*初始化信号量*/
    29   init_waitqueue_head(&globalfifo_devp->r_wait); /*初始化读等待队列
头*/
    30   init_waitqueue_head(&globalfifo_devp->w_wait); /*初始化写等待队列
头*/
    31
    32  return 0;
    33
    34  fail_malloc: unregister_chrdev_region(devno, 1);
    35  return ret;
    36 }
```

设备驱动读写操作需要被修改，在读函数中需增加等待 globalfifo_devp->w_wait 被唤醒的语句，而在写操作中唤醒 globalfifo_devp->r_wait，如代码清单 8.9 所示。

代码清单 8.9　增加等待队列后的 globalfifo 读写函数

```
    1  /*globalfifo 读函数*/
    2  static ssize_t globalfifo_read(struct file *filp, char __user *buf,
size_t
    3      count, loff_t *ppos)
    4  {
    5      int ret;
    6      struct globalfifo_dev *dev = filp->private_data; //获得设备结构
体指针
    7      DECLARE_WAITQUEUE(wait, current); //定义等待队列
    8
```

```
    down(&dev->sem); //获得信号量
    add_wait_queue(&dev->r_wait, &wait); //进入读等待队列头

    /* 等待 FIFO 非空 */
    if (dev->current_len == 0)
    {
        if (filp->f_flags &O_NONBLOCK)
        {
            ret =  - EAGAIN;
            goto out;
        }
        __set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡眠
        up(&dev->sem);

        schedule(); //调度其他进程执行
        if (signal_pending(current))
        //如果是因为信号唤醒
        {
            ret =  - ERESTARTSYS;
            goto out2;
        }

        down(&dev->sem);
    }

    /* 复制到用户空间 */
    if (count > dev->current_len)
        count = dev->current_len;

    if (copy_to_user(buf, dev->mem, count))
    {
        ret =  - EFAULT;
        goto out;
    }
    else
    {
       memcpy(dev->mem, dev->mem + count, dev->current_len - count);
//fifo 数据前移
        dev->current_len -= count; //有效数据长度减少
```

```
    47          printk(KERN_INFO "read %d bytes(s),current_len:%d\n", count,
dev
    48              ->current_len);
    49
    50          wake_up_interruptible(&dev->w_wait); //唤醒写等待队列
    51
    52          ret = count;
    53      }
    54      out: up(&dev->sem); //释放信号量
    55      out2: remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列
头移除
    56      set_current_state(TASK_RUNNING);
    57      return ret;
    58  }
    59
    60
    61  /*globalfifo 写操作*/
    62   static ssize_t globalfifo_write(struct file *filp, const char _
_user *buf,
    63      size_t count, loff_t *ppos)
    64  {
    65      struct globalfifo_dev *dev = filp->private_data; //获得设备结构
体指针
    66      int ret;
    67      DECLARE_WAITQUEUE(wait, current); //定义等待队列
    68
    69      down(&dev->sem); //获取信号量
    70      add_wait_queue(&dev->w_wait, &wait); //进入写等待队列头
    71
    72      /* 等待 FIFO 非满 */
    73      if (dev->current_len == GLOBALFIFO_SIZE)
    74      {
    75          if (filp->f_flags &O_NONBLOCK)
    76          //如果是非阻塞访问
    77          {
    78              ret =  - EAGAIN;
    79              goto out;
    80          }
    81          __set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡
眠
    82          up(&dev->sem);
    83
    84          schedule(); //调度其他进程执行
    85          if (signal_pending(current))
    86          //如果是因为信号唤醒
    87          {
    88              ret =  - ERESTARTSYS;
    89              goto out2;
    90          }
    91
    92          down(&dev->sem); //获得信号量
    93      }
    94
    95      /*从用户空间复制到内核空间*/
```

```
    96      if (count > GLOBALFIFO_SIZE - dev->current_len)
    97          count = GLOBALFIFO_SIZE - dev->current_len;
    98
    99      if (copy_from_user(dev->mem + dev->current_len, buf, count))
    100     {
    101         ret =  - EFAULT;
    102         goto out;
    103     }
    104     else
    105     {
    106         dev->current_len += count;
    107          printk(KERN_INFO "written %d bytes(s),current_len:%d\n",
count, dev
    108             ->current_len);
    109
    110         wake_up_interruptible(&dev->r_wait); //唤醒读等待队列
    111
    112         ret = count;
    113     }
    114
    115     out: up(&dev->sem); //释放信号量
    116     out2: remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列
头移除
    117     set_current_state(TASK_RUNNING);
    118     return ret;
    119 }
```

代码清单 8.9 处理了等待队列进出和进程切换的过程。是否可以把读函数中一大段用于等待 dev->current_len != 0 的内容直接用 wait_event_interruptible（dev->r_wait, dev->current_len != 0）替换，把写函数中一大段用于等待 dev->current_len != GLOBALFIFO_SIZE 的代码用 wait_event_ interruptible （dev->w_wait, dev->current_len != 0）替换呢？

实际上，就控制等待队列非空和非满的角度而言，wait_event_interruptible（dev->r_wait, dev-> current_len != 0）和第 13～32 行代码的功能完全一样，wait_event_interruptible（dev->w_wait, dev-> current_len != 0）和第 73～93 行代码的功能完全一样。细微的区别体现在第 13～32 行代码和第 73～93 行代码在进行 schedule()即切换进程前，通过 up（&dev->sem）释放了信号量。这一细微的动作意义重大，非如此，则死锁将不可避免。

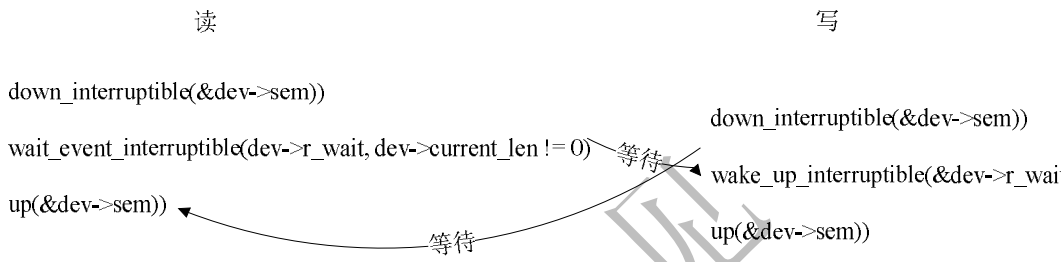
如图 8.1（a）所示，假设目前的 FIFO 为空，即 dev->current_len 为 0，此时如果有一个读进程，它会先获得信号量，因为条件不满足，它将因为 wait_event_interruptible(dev->r_wait, dev-> current_len != 0) 而阻塞，而释放 dev->r_wait 等待队列及让 dev->current_len != 0 的操作又需要在写进程中进行，写进程在执行写操作前又必须等待读进程释放信号量，造成互相等待对方资源的矛盾局面，从而死锁。

如图 8.1(b)所示，假设目前的 FIFO 为满即 dev->current_len 为 GLOBALFIFO_SIZE，此时如果有一个写进程，它先获得信号量，因为条件不满足，它将因为 wait_event_interruptible(dev->w_wait, dev->current_len != GLOBALFIFO_SIZE)而阻塞，

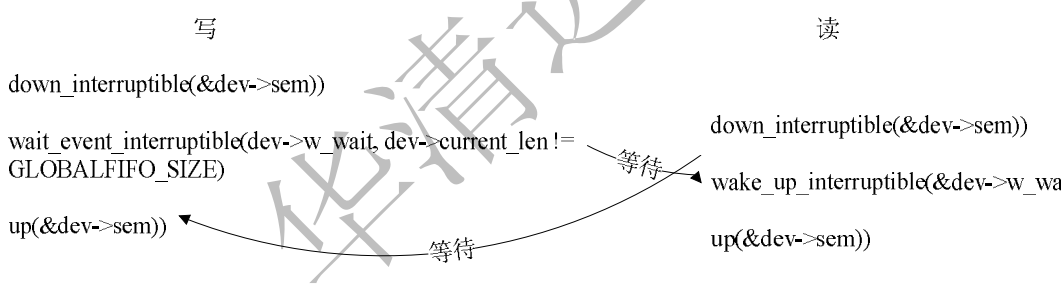
而释放 dev->w_wait 等待队列及让 dev->current_ len != GLOBALFIFO_SIZE 的操作又需要在读进程中进行，读进程在执行读操作前又必须等待写进程释放信号量，造成互相等待对方资源的矛盾局面，从而死锁。

所谓死锁，就是多个进程循环等待它方占有的资源而无限期地僵持下去的局面。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、信号量等机制同时出现时，谨防死锁。

现在回过来看一下代码清单 8.9 的第 15 行和第 75 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。



（a）队列为空时的死锁



（b）队列为满时的死锁

图 8.1　等待队列、信号量等引起的死锁

8.1.3　在用户空间验证 globalfifo 的读写

编译 globalfifo.c 并 insmod 模块即创建设备文件节点“/dev/globalfifo”后，启动两个进程，一个进程“cat /dev/globalfifo&”在后台执行，另一个进程“echo 字符串 /dev/globalfifo”在前台执行，如下所示：

```
    [root@localhost root]# cat /dev/globalfifo&
    [1] 4059
    [root@localhost root]# echo 'I want to be' > /dev/globalfifo
    [root@localhost root]# I want to be

    [root@localhost root]# echo 'a great Chinese Linux driver Engineer' >
/dev/globalfifo
```

```
[root@localhost root]# a great Chinese Linux driver Engineer
```

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。

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()系统调用最终会引发设备驱动中的 poll()函数被执行，在 2.5.45 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 中引入的，后者在 System V 中引入的。

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，其原型如下：

```
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文件描述符加 1。timeout 参数是一个指向 struct timeval 类型的指针，它可以使 select()在等待 timeout 时间后若没有文件描述符准备好则返回。struct timeval 数据结构的定义如代码清单 8.10 所示。

代码清单 8.10 timeval 结构体定义

```
struct timeval
{
    int tv_sec; /* 秒 */
    int tv_usec; /* 微妙 */
};
```

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。

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

将一个文件描述符加入文件描述符集中。

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

将一个文件描述符从文件描述符集中清除。

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

判断文件描述符是否被置位。

8.2.3 设备驱动中的轮询编程

设备驱动中 poll()函数的原型如下：

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

第一个参数为 file 结构体指针，第二个参数为轮询表指针。这个函数应该进行以下两项工作。

- 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添加到 poll_table。
- 返回表示是否能对设备进行无阻塞读、写访问的掩码。

关键的用于向 poll_table 注册等待队列的 poll_wait()函数的原型如下：

```
void poll_wait(struct file *filp, wait_queue_heat_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。

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 8.11 所示。

代码清单 8.11 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);//加读等待队列头
8      poll_wait(filp, &dev->w_wait, wait);//加写等待队列头
9
10     if (...)//可读
11     {
12         mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
13     }
14
15     if (...)//可写
16     {
17         mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
18     }
19
20     ...
21     return mask;
22 }
```

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 等待队列头添加到等待列表，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 8.12 所示。

代码清单 8.12 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; /*获得设备结构
体指针*/
    5
    6     down(&dev->sem);
    7
    8     poll_wait(filp, &dev->r_wait, wait);
    9     poll_wait(filp, &dev->w_wait, wait);
    10    /*fifo 非空*/
    11    if (dev->current_len != 0)
    12    {
    13        mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
    14    }
    15    /*fifo 非满*/
    16    if (dev->current_len != GLOBALFIFO_SIZE)
    17    {
    18        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
    19    }
    20
    21    up(&dev->sem);
    22    return mask;
    23 }
```

注意要把 globalfifo_poll 赋给 globalfifo_fops 的 poll 成员，如下所示：

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

8.3.2 在用户空间验证 globalfifo 设备的轮询

编写一个应用程序 pollmonitor.c 用于监控 globalfifo 的可读写状态，这个程序如代码清单 8.13 所示。

代码清单 8.13 监控 globalfifo 是否可非阻塞读写的应用程序

```
1  #include ...
2
3  #define FIFO_CLEAR 0x1
4  #define BUFFER_LEN 20
5  main()
6  {
7    int fd, num;
```

```
8   char rd_ch[BUFFER_LEN];
9   fd_set rfds,wfds; //读/写文件描述符集
10
11  /*以非阻塞方式打开/dev/globalfifo 设备文件*/
12  fd = open("/dev/globalfifo", O_RDONLY | O_NONBLOCK);
13  if (fd !=  - 1)
14  {
15    /*FIFO 清零*/
16    if (ioctl(fd, FIFO_CLEAR, 0) < 0)
17    {
18      printf("ioctl command failed\n");
19    }
20    while (1)
21    {
22      FD_ZERO(&rfds);
23      FD_ZERO(&wfds);
24      FD_SET(fd, &rfds);
25      FD_SET(fd, &wfds);
26
27      select(fd + 1, &rfds, &wfds, NULL, NULL);
28      /*数据可获得*/
29      if (FD_ISSET(fd, &rfds))
30      {
31          printf("Poll monitor:can be read\n");
32      }
33      /*数据可写入*/
34      if (FD_ISSET(fd, &wfds))
35      {
36          printf("Poll monitor:can be written\n");
37      }
38    }
39  }
40  else
41  {
42    printf("Device open failure\n");
43  }
44 }
```

运行时看到，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”；如果不断地通过 echo 向/dev/globalfifo 写

入数据直至写满 FIFO，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。

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 I/O 操作暂时不可进行时会让进程睡眠。

在设备驱动中阻塞 I/O 一般基于等待队列来实现，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()和 poll()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 poll()和 select()系统调用则会阻塞地等待文件描述符集合中的至少一个可访问或超时。