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 9 章 Linux 设备驱动中的异步通知与异步 I/O

在设备驱动中使用异步通知可以使得对设备的访问可进行时，由驱动主动通知应用程序进行访问。这样，使用无阻塞 I/O 的应用程序无须轮询设备是否可访问，而阻塞访问也可以被类似“中断”的异步通知所取代。

9.1 节讲解了异步通知的概念和作用，9.2 节讲解了 Linux 异步通知的编程方法，9.3 节讲解了增加异步通知的 globalfifo 驱动及其在用户空间的验证。

9.1 异步通知的概念与作用

阻塞与非阻塞访问、poll()函数提供了较好的解决设备访问的机制，但是如果有了异步通知整套机制就更加完整了。

异步通知的意思是：一旦设备就绪，则主动通知应用程序，这样应用程序根本就不需要查询设备状态，这一点非常类似于硬件上“中断”的概念，比较准确的称谓是“信号驱动的异步 I/O”。信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。

阻塞 I/O 意味着一直等待设备可访问后再访问，非阻塞 I/O 中使用 poll()意味着查询设备是否可访问，而异步通知则意味着设备通知自身可访问，实现了异步 I/O。由此可见，这几种方式 I/O 可以互为补充。

图 9.1 呈现了阻塞 I/O，结合 poll()的非阻塞 I/O 及异步通知在时间先后顺序上的不同。

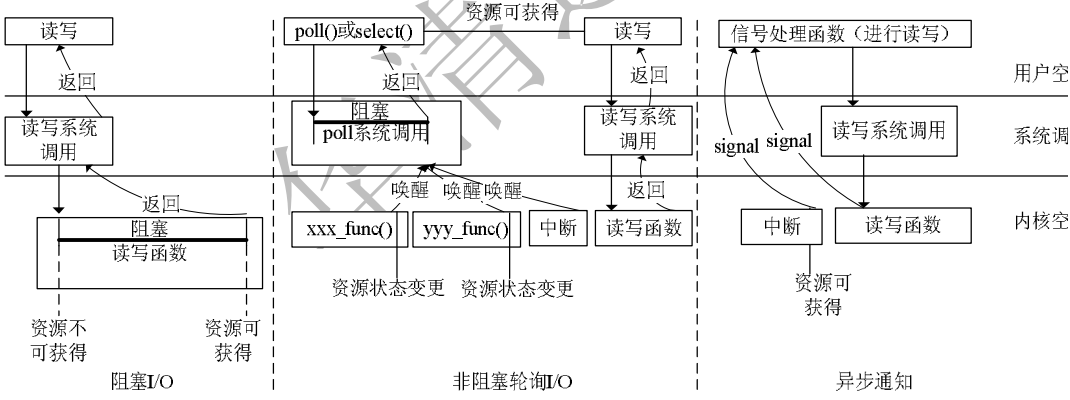

图 9.1　阻塞、非阻塞 I/O、异步通知的区别

阻塞、非阻塞 I/O、异步通知本身没有优劣，应该根据不同的应用场景合理选择。

9.2 Linux 异步通知编程

9.2.1　Linux 信号

使用信号进行进程间通信（IPC）是 UNIX 系统中的一种传统机制，Linux 系统也支持这种机制。在 Linux 系统中，异步通知使用信号来实现，Linux 系统中可用的信号及其定义如表 9-1 所示。

表 9-1 Linux 信号

信　　号	值	含　　义
SIGHUP	1	挂起
SIGINT	2	终端中断
SIGQUIT	3	终端退出
SIGILL	4	无效命令
SIGTRAP	5	跟踪陷阱
SIGIOT	6	IOT 陷阱
SIGBUS	7	BUS 错误
SIGFPE	8	浮点异常
SIGKILL	9	强行终止（不能被捕获或忽略）
SIGUSR1	10	用户定义的信号 1
SIGSEGV	11	无效的内存段处理
SIGUSR2	12	用户定义的信号 2
SIGPIPE	13	半关闭管道发生写操作
SIGALRM	14	计时器到期
SIGTERM	15	终止
SIGSTKFLT	16	堆栈错误
SIGCHLD	17	子进程已经停止或退出
SIGCONT	18	如果停止了，继续执行
SIGSTOP	19	停止执行（不能被捕获或忽略）
SIGTSTP	20	终端停止信号
SIGTTIN	21	后台进程需要从终端读取输入
SIGTTOU	22	后台进程需要向从终端写出
SIGURG	23	紧急的套接字事件
SIGXCPU	24	超额使用 CPU 分配的时间
SIGXFSZ	25	文件尺寸超额
SIGVTALRM	26	虚拟时钟信号
SIGPROF	27	时钟信号描述
SIGWINCH	28	窗口尺寸变化
SIGIO	29	I/O
SIGPWR	30	断电重启

除了 SIGSTOP 和 SIGKILL 两个信号外，进程能够忽略或捕获其他的全部信号。一个信号被捕获的意思是当一个信号到达时有相应的代码处理它。如果一个信号没有被这个进程所捕获，内核将采用默认行为处理。

9.2.2　信号的接收

在用户程序中，为了捕获信号，可以使用 signal()函数来设置对应信号的处理函数，如下所示：

```
void (*signal(int signum, void (*handler))(int))(int);
```

该函数原型较难理解，它可以分解如下：

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

第一个参数指定信号的值，第二个参数指定针对前面信号值的处理函数，若为 SIG_IGN，表示忽略该信号；若为 SIG_DFL，表示采用系统默认方式处理信号；若为用户自定义的函数，则信号被捕获到后，该函数将被执行。

如果 signal()调用成功，它返回最后一次为信号 signum 绑定的处理函数 handler 值，失败则返回 SIG_ERR。

在进程执行时，按下［Ctrl+c］组合键将向其发出 SIGINT 信号，kill 正在运行的进程将向其发出 SIGTERM 信号，代码清单 9.1 的进程捕获这两个信号并输出信号值。

代码清单 9.1　signal()捕获信号范例

```
1  void sigterm_handler(int signo)
2  {
3    printf("Have caught sig N.O. %d\n", signo);
4    exit(0);
5  }
6
7  int main(void)
8  {
9    signal(SIGINT, sigterm_handler);
10   signal(SIGTERM, sigterm_handler);
11   while(1);
12
13   return 0;
14 }
```

除了 signal()函数外，sigaction()函数可用于改变进程接收到特定信号后的行为，它的原型如下：

```
int sigaction(int signum,const struct sigaction *act,struct sigaction
*oldact);
```

该函数的第一个参数为信号的值，可以为除 SIGKILL 及 SIGSTOP 外的任何一个特定有效的信号。第二个参数是指向结构体 sigaction 的一个实例的指针，在结构体 sigaction 的实例中，指定了对特定信号的处理函数，若为空，则进程会以默认方式对信号处理。第三个参数 oldact 指向的对象用来保存原来对相应信号的处理函数，可指定 oldact 为 NULL。如果把第二、第三个参数都设为 NULL，那么该函数可用于检查信号的有效性。

先来看一个使用信号实现异步通知的例子，它通过 signal（SIGIO, input_handler）对标准输入文件描述符 STDIN_FILENO 启动信号机制。用户输入后，应用程序将接

收到 SIGIO 信号，其处理函数 input_handler()将被调用，如代码清单 9.2 所示。

代码清单 9.2 使用信号实现异步通知的应用程序实例

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <stdio.h>
4  #include <fcntl.h>
5  #include <signal.h>
6  #include <unistd.h>
7  #define MAX_LEN 100
8  void input_handler(int num)
9  {
10   char data[MAX_LEN];
11   int len;
12
13   //读取并输出 STDIN_FILENO 上的输入
14   len = read(STDIN_FILENO, &data, MAX_LEN);
15   data[len] = 0;
16   printf("input available:%s\n", data);
17 }
18
19 main()
20 {
21   int oflags;
22
23   //启动信号驱动机制
24   signal(SIGIO, input_handler);
25   fcntl(STDIN_FILENO, F_SETOWN, getpid());
26   oflags = fcntl(STDIN_FILENO, F_GETFL);
27   fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
28
29   //最后进入一个死循环，仅为保持进程不终止，如果程序中没有这个死循环
30   //会立即执行完毕
31   while (1);
32 }
```

上述代码 24 行为 SIGIO 信号安装 input_handler()作为处理函数，第 25 行设置本进程为 STDIN_FILENO 文件的拥有者（owner），没有这一步内核不会知道应该将信号发给哪个进程。而为了启用异步通知机制，还需对设备设置 FASYNC 标志，第 26～27 行代码实现此目的。整个程序的执行效果如下：

```
[root@localhost driver_study]# ./signal_test
I am Chinese.
input available: I am Chinese.

I love Linux driver.
input available: I love Linux driver.
```

从中可以看出，当用户输入一串字符后，标准输入设备释放 SIGIO 信号，这个信号“中断”驱使对应的应用程序中的 input_handler()得以执行，将用户输入显示出来。

由此可见，为了在用户空间中能处理一个设备释放的信号，它必须完成以下 3 项工作。

- 通过 F_SETOWN IO 控制命令设置设备文件的拥有者为本进程，这样从设备驱动发出的信号才能被本进程接收到。
- 通过 F_SETFL IO 控制命令设置设备文件支持 FASYNC，即异步通知模式。

- 通过 signal()函数连接信号和信号处理函数。

9.2.3　信号的释放

在设备驱动和应用程序的异步通知交互中，仅仅在应用程序端捕获信号是不够的，因为信号没有的源头在设备驱动端。因此，应该在合适的时机让设备驱动释放信号，在设备驱动程序中增加信号释放的相关代码。

为了使设备支持异步通知机制，驱动程序中涉及以下 3 项工作。

- 支持 F_SETOWN 命令，能在这个控制命令处理中设置 filp->f_owner 为对应进程 ID。不过此项工作已由内核完成，设备驱动无须处理。
- 支持 F_SETFL 命令的处理，每当 FASYNC 标志改变时，驱动程序中的 fasync()函数将得以执行。因此，驱动中应该实现 fasync()函数。
- 在设备资源可获得时，调用 kill_fasync()函数激发相应的信号。

驱动中的上述 3 项工作和应用程序中的 3 项工作是一一对应的，图 9.2 所示为异步通知处理过程中用户空间和设备驱动的交互。

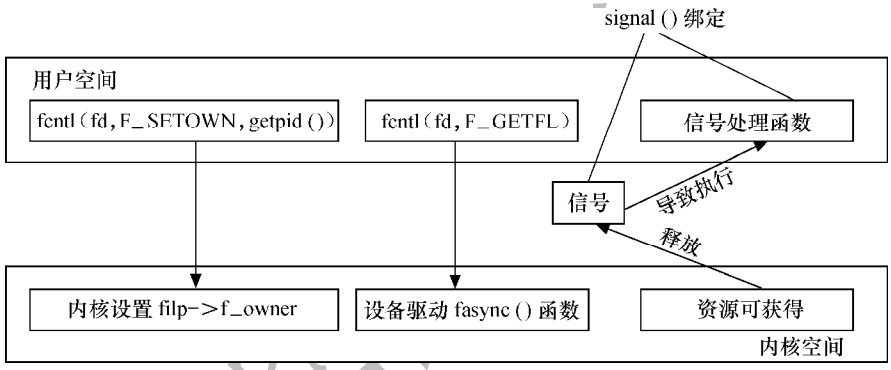

图 9.2　异步通知中设备驱动和异步通知的交互

设备驱动中异步通知编程比较简单，主要用到一项数据结构和两个函数。数据结构是 fasync_struct 结构体，两个函数分别如下。

处理 FASYNC 标志变更的函数。

```
    int fasync_helper(int fd, struct file *filp, int mode, struct
fasync_struct **fa);
```

释放信号用的函数。

```
    void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

和其他的设备驱动一样，将 fasync_struct 结构体指针放在设备结构体中仍然是最佳选择，代码清单 9.3 给出了支持异步通知的设备结构体模板。

代码清单 9.3　支持异步通知的设备结构体模板

```
1  struct xxx_dev
2  {
3      struct cdev cdev; /*cdev 结构体*/
4      ...
5      struct fasync_struct *async_queue; /* 异步结构体指针 */
6  };
```

在设备驱动的 fasync()函数中，只需要简单地将该函数的 3 个参数以及

fasync_struct 结构体指针的指针作为第 4 个参数传入 fasync_helper()函数即可。代码清单 9.4 给出了支持异步通知的设备驱动程序 fasync()函数的模板。

代码清单 9.4　支持异步通知的设备驱动 fasync()函数的模板

```
1 static int xxx_fasync(int fd, struct file *filp, int mode)
2 {
3    struct xxx_dev *dev = filp->private_data;
4    return fasync_helper(fd, filp, mode, &dev->async_queue);
5 }
```

在设备资源可以获得时，应该调用 kill_fasync()释放 SIGIO 信号，可读时第 3 个参数设置为 POLL_IN，可写时第 3 个参数设置为 POLL_OUT。代码清单 9.5 为释放信号的范例。

代码清单 9.5　支持异步通知的设备驱动信号释放的模板

```
1  static ssize_t xxx_write(struct file *filp, const char __user *buf,
size_t count,
2                          loff_t *f_pos)
3  {
4   struct xxx_dev *dev = filp->private_data;
5   ...
6   /* 产生异步读信号 */
7   if (dev->async_queue)
8       kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
9   ...
10 }
```

最后，在文件关闭时，即在设备驱动的 release()函数中，应调用设备驱动的 fasync()函数将文件从异步通知的列表中删除。代码清单 9.6 给出了支持异步通知的设备驱动 release()函数的模板。

代码清单 9.6　支持异步通知的设备驱动 release()函数的模板

```
1 static int xxx_release(struct inode *inode, struct file *filp)
2 {
3    struct xxx_dev *dev = filp->private_data;
4    /* 将文件从异步通知列表中删除 */
5    xxx_fasync(-1, filp, 0);
6    ...
7    return 0;
8 }
```

9.3 支持异步通知的 globalfifo 驱动

9.3.1 在 globalfifo 驱动中增加异步通知

首先，参照代码清单 9.2 将异步结构体指针添加到 globalfifo_dev 设备结构体内，如代码清单 9.7 所示。

代码清单 9.7 增加异步通知后的 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      struct fasync_struct *async_queue; /* 异步结构体指针，用于读 */
10 };
```

参考代码清单 9.3 的 fasync()函数模板，globalfifo 的这个函数如代码清单 9.8 所示。

代码清单 9.8 支持异步通知的 globalfifo 设备驱动的 fasync()函数

```
1 static int globalfifo_fasync(int fd, struct file *filp, int mode)
2 {
3    struct globalfifo_dev *dev = filp->private_data;
4    return fasync_helper(fd, filp, mode, &dev->async_queue);
5 }
```

在 globalfifo 设备被正确写入之后，它变得可读，这个时候驱动应释放 SIGIO 信号以便应用程序捕获，代码清单 9.9 给出了支持异步通知的 globalfifo 设备驱动的写函数。

代码清单 9.9 支持异步通知的 globalfifo 设备驱动的写函数

```
1  static ssize_t globalfifo_write(struct file *filp, const char __user
*buf,
2    size_t count, loff_t *ppos)
3  {
4    struct globalfifo_dev *dev = filp->private_data; //获得设备结构体
指针
5    int ret;
6    DECLARE_WAITQUEUE(wait, current); //定义等待队列
7
8    down(&dev->sem); //获取信号量
9    add_wait_queue(&dev->w_wait, &wait); //进入写等待队列头
10
11   /* 等待 FIFO 非满 */
```

```
12  if (dev->current_len == GLOBALFIFO_SIZE)
13  {
14    if (filp->f_flags &O_NONBLOCK)
15    //如果是非阻塞访问
16    {
17      ret =  - EAGAIN;
18      goto out;
19    }
20    __set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡眠
21    up(&dev->sem);
22
23    schedule(); //调度其他进程执行
24    if (signal_pending(current))
25    //如果是因为信号唤醒
26    {
27      ret =  - ERESTARTSYS;
28      goto out2;
29    }
30
31    down(&dev->sem); //获得信号量
32  }
33
34  /*从用户空间复制到内核空间*/
35  if (count > GLOBALFIFO_SIZE - dev->current_len)
36    count = GLOBALFIFO_SIZE - dev->current_len;
37
38  if (copy_from_user(dev->mem + dev->current_len, buf, count))
39  {
40    ret =  - EFAULT;
41    goto out;
42  }
43  else
44  {
45    dev->current_len += count;
46    printk(KERN_INFO "written %d bytes(s),current_len:%d\n", count,
47    dev->current_len);
48
49    wake_up_interruptible(&dev->r_wait); //唤醒读等待队列
50    /* 产生异步读信号 */
51    if (dev->async_queue)
```

```
    52        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
    53
    54      ret = count;
    55    }
    56
    57    out: up(&dev->sem); //释放信号量
    58    out2:remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列头移
除
    59    set_current_state(TASK_RUNNING);
    60    return ret;
    61 }
```

参考代码清单 9.6，增加异步通知后的 globalfifo 设备驱动的 release()函数中需调用 globalfifo_ fasync()函数将文件从异步通知列表中删除，代码清单 9.10 给出了支持异步通知的 globalfifo_ release()函数。

代码清单 9.10　增加异步通知后的 globalfifo 设备驱动的 release()函数

```
1 int globalfifo_release(struct inode *inode, struct file *filp)
2 {
3     struct globalfifo_dev *dev = filp->private_data;
4     /* 将文件从异步通知列表中删除 */
5     globalfifo_fasync( - 1, filp, 0);
6     return 0;
7 }
```

9.3.2　在用户空间验证 globalfifo 的异步通知

现在，我们可以采用与代码清单 9.2 类似的方法，编写一个在用户空间验证 globalfifo 异步通知的程序，这个程序在接收到由 globalfifo 发出的信号后将输出信号值，如代码清单 9.11 所示。

代码清单 9.11　监控 globalfifo 异步通知信号的应用程序

```
1  #include ...
2
3  /*接收到异步读信号后的动作*/
4  void input_handler(int signum)
5  {
6    printf("receive a signal from globalfifo,signalnum:%d\n",signum);
7  }
8
9  main()
10 {
11   int fd, oflags;
12   fd = open("/dev/globalfifo", O_RDWR, S_IRUSR | S_IWUSR);
```

```
13  if (fd !=  - 1)
14  {
15    //启动信号驱动机制
16    signal(SIGIO, input_handler); //让input_handler()处理SIGIO信号
17    fcntl(fd, F_SETOWN, getpid());
18    oflags = fcntl(fd, F_GETFL);
19    fcntl(fd, F_SETFL, oflags | FASYNC);
20    while(1)
21    {
22      sleep(100);
23    }
24  }
25  else
26  {
27    printf("device open failure\n");
28  }
29 }
```

加载新的 globalfifo 设备驱动并创建设备文件节点，运行上述程序后，每当通过 echo 向/dev/globalfifo 写入新的数据时，input_handler()将会被调用，如下所示：

```
    [root@localhost driver_study]# echo 0 > /dev/globalfifo
    [root@localhost    driver_study]#    receive    a    signal    from
globalfifo,signalnum:29

    [root@localhost driver_study]# echo 0 > /dev/globalfifo
    [root@localhost    driver_study]#    receive    a    signal    from
globalfifo,signalnum:29

    [root@localhost driver_study]# echo 0 > /dev/globalfifo
    [root@localhost    driver_study]#    receive    a    signal    from
globalfifo,signalnum:29
```

9.4 Linux 2.6 异步 I/O

9.4.1　AIO 概念与 GNU C 库函数

Linux 系统中最常用的输入/输出（I/O）模型是同步 I/O。在这个模型中，当请求发出之后，应用程序就会阻塞，直到请求满足为止。这是很好的一种解决方案，因为调用应用程序在等待 I/O 请求完成时不需要使用任何中央处理单元（CPU）。但是在某些情况下，I/O 请求可能需要与其他进程产生交叠。可移植操作系统接口（POSIX）异步 I/O（AIO）应用程序接口（API）就提供了这种功能。

Linux 异步 I/O 是 2.6 版本内核的一个标准特性，但是我们在 2.4 版本内核的补丁中也可以找到它。AIO 基本思想是允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。

select()函数所提供的功能（异步阻塞 I/O）与 AIO 类似，它对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

在异步非阻塞 I/O 中，我们可以同时发起多个传输操作。这需要每个传输操作都

有惟一的上下文，这样才能在它们完成时区分到底是哪个传输操作完成了。在 AIO 中，通过 aiocb（AIO I/O Control Block）结构体进行区分。这个结构体包含了有关传输的所有信息，包括为数据准备的用户缓冲区。在产生 I/O（称为完成）通知时，aiocb 结构就被用来惟一标识所完成的 I/O 操作。

AIO 系列 API 被 GNU C 库函数所包含，它被 POSIX.1b 所要求，主要包括如下函数。

1．aio_read

aio_read()函数请求对一个有效的文件描述符进行异步读操作。这个文件描述符可以表示一个文件、套接字甚至管道。aio_read 函数的原型如下：

```
int aio_read( struct aiocb *aiocbp );
```

aio_read()函数在请求进行排队之后会立即返回。如果执行成功，返回值就为 0；如果出现错误，返回值就为−1，并设置 errno 的值。

2．aio_write

aio_write()函数用来请求一个异步写操作，其函数原型如下：

```
int aio_write( struct aiocb *aiocbp );
```

aio_write()函数会立即返回，说明请求已经进行排队（成功时返回值为 0，失败时返回值为−1，并相应地设置 errno。

3．aio_error

aio_error 函数被用来确定请求的状态，其原型如下：

```
int aio_error( struct aiocb *aiocbp );
```

这个函数可以返回以下内容。

EINPROGRESS：说明请求尚未完成。

ECANCELLED：说明请求被应用程序取消了。

-1：说明发生了错误，具体错误原因由 errno 记录。

4．aio_return

异步 I/O 和标准块 I/O 之间的另外一个区别是不能立即访问这个函数的返回状态，因为并没有阻塞在 read()调用上。在标准的 read()调用中，返回状态是在该函数返回时提供的。但是在异步 I/O 中，我们要使用 aio_return()函数。这个函数的原型如下：

```
ssize_t aio_return( struct aiocb *aiocbp );
```

只有在 aio_error()调用确定请求已经完成（可能成功，也可能发生了错误）之后，才会调用这个函数。aio_return()的返回值就等价于同步情况中 read 或 write 系统调用的返回值（所传输的字节数，如果发生错误，返回值就为−1）。

代码清单 9.12 给出了用户空间应用程序进行异步读操作的一个例程，它首先打开文件，然后准备 aiocb 结构体，之后调用 aio_read（&my_aiocb）进行提出异步读请求，当 aio_error(&my_aiocb) == EINPROGRESS 即操作还在进行中时，一直等待，结束后

通过 aio_return（&my_aiocb）获得返回值。

代码清单 9.12　用户空间异步读例程

```
1  #include <aio.h>
2  ...
3  int fd, ret;
4  struct aiocb my_aiocb;
5
6  fd = open("file.txt", O_RDONLY);
7  if (fd < 0)
8    perror("open");
9
10 /* 清零 aiocb 结构体 */
11 bzero((char*) &my_aiocb, sizeof(struct aiocb));
12
13 /* 为 aiocb 请求分配数据缓冲区 */
14 my_aiocb.aio_buf = malloc(BUFSIZE + 1);
15 if (!my_aiocb.aio_buf)
16   perror("malloc");
17
18 /* 初始化 aiocb 的成员 */
19 my_aiocb.aio_fildes = fd;
20 my_aiocb.aio_nbytes = BUFSIZE;
21 my_aiocb.aio_offset = 0;
22
23 ret = aio_read(&my_aiocb);
24 if (ret < 0)
25   perror("aio_read");
26
27 while (aio_error(&my_aiocb) == EINPROGRESS)
28   ;
29
30 if ((ret = aio_return(&my_iocb)) > 0)
31 {
32   /* 获得异步读的返回值 */
33 }
34 else
35 {
36   /* 读失败，分析 errorno */
37 }
```

用户可以使用 aio_suspend()函数来挂起（或阻塞）调用进程，直到异步请求完成为止，此时会产生一个信号，或者发生其他超时操作。调用者提供了一个 aiocb 引用列表，其中任何一个完成都会导致 aio_suspend()返回。aio_suspend 的函数原型如下：

```
int aio_suspend( const struct aiocb *const cblist[],
                 int n, const struct timespec *timeout );
```

代码清单 9.13 给出了用户空间异步读操作时使用 aio_suspend()函数的例子。

代码清单 9.13　用户空间异步 I/O aio_suspend()函数使用例程

```
1 struct aioct *cblist[MAX_LIST]
2 /* 清零 aioct 结构体链表 */
3 bzero((char *)cblist, sizeof(cblist));
4 /* 将一个或更多的 aiocb 放入 aioct 结构体链表 */
```

```
5 cblist[0] = &my_aiocb;
6 ret = aio_read( &my_aiocb );
7 ret = aio_suspend( cblist, MAX_LIST, NULL );
```

aio_cancel()函数允许用户取消对某个文件描述符执行的一个或所有 I/O 请求。其原型如下：

```
int aio_cancel( int fd, struct aiocb *aiocbp );
```

如果要取消一个请求，用户需提供文件描述符和 aiocb 引用。如果这个请求被成功取消了，那么这个函数就会返回 AIO_CANCELED。如果请求完成了，这个函数就会返回 AIO_NOTCANCELED。

如果要取消对某个给定文件描述符的所有请求，用户需要提供这个文件的描述符以及一个对 aiocbp 的 NULL 引用。如果所有的请求都取消了，这个函数就会返回 AIO_CANCELED；如果至少有一个请求没有被取消，那么这个函数就会返回 AIO_NOT_CANCELED；如果没有一个请求可以被取消，那么这个函数就会返回 AIO_ALLDONE。然后，可以使用 aio_error()来验证每个 AIO 请求，如果某请求已经被取消了，那么 aio_error()就会返回−1，并且 errno 会被设置为 ECANCELED。

lio_listio()函数可用于同时发起多个传输。这个函数非常重要，它使得用户可以在一个系统调用（一次内核上下文切换）中启动大量的 I/O 操作。lio_listio API 函数的原型如下：

```
int lio_listio( int mode, struct aiocb *list[], int nent, struct sigevent
*sig );
```

mode 参数可以是 LIO_WAIT 或 LIO_NOWAIT。LIO_WAIT 会阻塞这个调用，直到所有的 I/O 都完成为止。在操作进行排队之后，LIO_NOWAIT 就会返回。list 是一个 aiocb 引用的列表，最大元素的个数是由 nent 定义的。如果 list 的元素为 NULL，lio_listio()会将其忽略。

代码清单 9.14 给出了用户空间异步 I/O 操作时使用 lio_listio()函数的例子。

代码清单 9.14 用户空间异步 I/O lio_listio()函数使用例程

```
1  struct aiocb aiocb1, aiocb2;
2  struct aiocb *list[MAX_LIST];
3  ...
4  /* 准备第一个 aiocb */
5  aiocb1.aio_fildes = fd;
6  aiocb1.aio_buf = malloc( BUFSIZE+1 );
7  aiocb1.aio_nbytes = BUFSIZE;
8  aiocb1.aio_offset = next_offset;
9  aiocb1.aio_lio_opcode = LIO_READ; /*异步读操作*/
10 ...  /*准备多个 aiocb */
11 bzero( (char *)list, sizeof(list) );
12
13 /*将 aiocb 填入链表*/
14 list[0] = &aiocb1;
```

```
    15 list[1] = &aiocb2;
    16 ...
    17 ret = lio_listio( LIO_WAIT, list, MAX_LIST, NULL );/*发起大量 I/O 操
作*/
```

上述代码第 9 行中，因为是进行异步读操作，所以操作码为 LIO_READ，对于写操作来说，应该使用 LIO_WRITE 作为操作码，而 LIO_NOP 意味着空操作。

9.4.2　使用信号作为 AIO 的通知

9.1～9.3 节讲述的信号作为异步通知的机制在 AIO 中仍然是适用的，为使用信号，使用 AIO 的应用程序同样需要定义信号处理程序，在指定的信号被产生时会触发调用这个处理程序。作为信号上下文的一部分，特定的 aiocb 请求被提供给信号处理函数用来区分 AIO 请求。

代码清单 9.15 给出了使用信号作为 AIO 异步 I/O 通知机制的例子。

代码清单 9.15　使用信号作为 AIO 异步 I/O 通知机制例程

```
    1  /*设置异步 I/O 请求*/
    2  void setup_io(...)
    3  {
    4    int fd;
    5    struct sigaction sig_act;
    6    struct aiocb my_aiocb;
    7    ...
    8    /* 设置信号处理函数 */
    9    sigemptyset(&sig_act.sa_mask);
   10   sig_act.sa_flags = SA_SIGINFO;
   11   sig_act.sa_sigaction = aio_completion_handler;
   12
   13   /* 设置 AIO 请求 */
   14   bzero((char*) &my_aiocb, sizeof(struct aiocb));
   15   my_aiocb.aio_fildes = fd;
   16   my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
   17   my_aiocb.aio_nbytes = BUF_SIZE;
   18   my_aiocb.aio_offset = next_offset;
   19
   20   /* 连接 AIO 请求和信号处理函数 */
   21   my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
   22   my_aiocb.aio_sigevent.sigev_signo = SIGIO;
   23   my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
   24
   25   /* 将信号与信号处理函数绑定 */
   26   ret = sigaction(SIGIO, &sig_act, NULL);
   27   ...
   28   ret = aio_read(&my_aiocb); /*发出异步读请求*/
   29 }
   30
   31 /*信号处理函数*/
   32  void  aio_completion_handler(int  signo,  siginfo_t  *info,  void
*context)
```

```
33 {
34   struct aiocb *req;
35
36   /* 确定是我们需要的信号*/
37   if (info->si_signo == SIGIO)
38   {
39     req = (struct aiocb*)info->si_value.sival_ptr; /*获得 aiocb*/
40
41     /* 请求的操作完成了吗? */
42     if (aio_error(req) == 0)
43     {
44       /* 请求的操作完成，获取返回值 */
45       ret = aio_return(req);
46     }
47   }
48   return ;
49 }
```

特别要注意上述代码的第 39 行通过(struct aiocb*)info->si_value.sival_ptr 获得了信号对应的 aiocb。

9.4.3 使用回调函数作为 AIO 的通知

除了信号之外，应用程序还可提供一个回调（Callback）函数给内核，以便 AIO 的请求完成后内核调用这个函数。

一般来说，下层对上层（如内核对应用）的调用都称为“回调”，而上层对下层（如进行 Linux 系统调用）的调用称为“调用”，如图 9.3 所示。

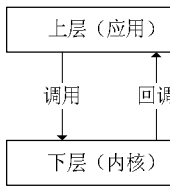

图 9.3 回调与调

代码清单 9.16 给出了使用回调函数作为 AIO 异步 I/O 请求完成的通知机制的例子。

代码清单 9.16　使用回调函数作为 AIO 异步 I/O 通知机制例程

```
1  /*设置异步 I/O 请求*/
2  void setup_io(...)
3  {
4    int fd;
5    struct aiocb my_aiocb;
6    ...
7    /* 设置 AIO 请求 */
8    bzero((char*) &my_aiocb, sizeof(struct aiocb));
9    my_aiocb.aio_fildes = fd;
10   my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
11   my_aiocb.aio_nbytes = BUF_SIZE;
12   my_aiocb.aio_offset = next_offset;
13
14   /* 连接 AIO 请求和线程回调函数 */
15   my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
16   my_aiocb.aio_sigevent.notify_function = aio_completion_handler;
17     /*设置回调函数*/
18   my_aiocb.aio_sigevent.notify_attributes = NULL;
19   my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
20   ... ret = aio_read(&my_aiocb); //发起 AIO 请求
21 }
22
23 /* 异步 I/O 完成回调函数 */
24 void aio_completion_handler(sigval_t sigval)
25 {
26   struct aiocb *req;
27   req = (struct aiocb*)sigval.sival_ptr;
28
29   /* AIO 请求完成? */
30   if (aio_error(req) == 0)
31   {
32     /* 请求完成，获得返回值 */
33     ret = aio_return(req);
34   }
35
36   return ;
37 }
```

上述程序在创建 aiocb 请求之后，使用 SIGEV_THREAD 请求了一个线程回调函数来作为通知方法。在回调函数中，通过(struct aiocb*)sigval.sival_ptr 可以获得对应的

aiocb 指针，使用 AIO 函数可验证请求是否已经完成。

proc 文件系统包含了两个虚拟文件，它们可以用来对异步 I/O 的性能进行优化。

- /proc/sys/fs/aio-nr 文件提供了系统范围异步 I/O 请求的数目。
- /proc/sys/fs/aio-max-nr 文件是所允许的并发请求的最大个数，最大个数通常是 64KB，这对于大部分应用程序来说都已经足够了。

9.4.4 AIO 与设备驱动

在内核中，每个 I/O 请求都对应于一个 kiocb 结构体，其 ki_filp 成员指向对应的 file 指针，通过 is_sync_kiocb()可以判断某 kiocb 是否为同步 I/O 请求，如果返回非真，表示为异步 I/O 请求。

块设备和网络设备本身是异步的，只有字符设备必须明确表明应支持 AIO。AIO 对于大多数字符设备而言都不是必须的，只有极少数设备需要。比如，对于磁带机，由于 I/O 操作很慢，这时候使用异步 I/O 将改善性能。

字符设备驱动程序中，file_operations 包含 3 个与 AIO 相关的成员函数，如下所示：

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

aio_read()和 aio_write()与 file_operations 中的 read()和 write()中的 offset 参数不同，它直接传递值，而后者传递的是指针，这是因为 AIO 从来不需要改变文件的位置。

aio_read()和 aio_write()函数本身不一定完成了读和写操作，它只是发起、初始化读和写操作，代码清单 9.17 给出了驱动程序中 aio_read()和 aio_write()函数的实现例子。

代码清单 9.17 设备驱动中的异步 I/O 函数

```
1  /*异步读*/
2  static ssize_t xxx_aio_read(struct kiocb *iocb, char *buf, size_t
count, loff_t
3    pos)
4  {
5    return xxx_defer_op(0, iocb, buf, count, pos);
6  }
7
8  /*异步写*/
9  static ssize_t xxx_aio_write(struct kiocb *iocb, const char *buf,
size_t count,
10   loff_t pos)
11 {
12   return xxx_defer_op(1, iocb, (char*)buf, count, pos);
```

```
13 }
14
15 /*初始化异步 I/O*/
16 static int xxx_defer_op(int write, struct kiocb *iocb, char *buf,
size_t count,
17   loff_t pos)
18 {
19   struct async_work *async_wk;
20   int result;
21   /* 当可以访问 buffer 时进行复制*/
22   if (write)
23     result = xxx_write(iocb->ki_filp, buf, count, &pos);
24   else
25     result = xxx_read(iocb->ki_filp, buf, count, &pos);
26   /* 如果是同步 IOCB，立即返回状态 */
27   if (is_sync_kiocb(iocb))
28     return result;
29
30   /* 否则，推后几μs 执行 */
31   async_wk = kmalloc(sizeof(*async_wk), GFP_KERNEL);
32   if (async_wk == NULL)
33     return result;
34   /*调度延迟的工作*/
35   async_wk->iocb = iocb;
36   async_wk->result = result;
37   INIT_WORK(&async_wk->work, xxx_do_deferred_op, async_wk);
38   schedule_delayed_work(&async_wk->work, HZ / 100);
39   return  - EIOCBQUEUED; /*控制权返回用户空间*/
40 }
41
42 /*延迟后执行*/
43 static void xxx_do_deferred_op(void *p)
44 {
45   struct async_work *async_wk = (struct async_work*)p;
46   aio_complete(async_wk->iocb, async_wk->result, 0);
47   kfree(async_wk);
48 }
```

上述代码中最核心的是使用 aync_work（异步工作）结构体将操作延后执行，aync_work 结构体定义如代码清单 9.18 所示，通过 schedule_delayed_work()函数可以调度其执行。第 46 行对 aio_complete()的调用用于通知内核驱动程序已经完成了操作。

代码清单 9.18 async_work 结构体

```
1 struct async_work
2 {
3     struct kiocb *iocb;  //kiocb 结构体指针
4     int result;  //执行结果
5     struct work_struct work; //工作结构体
6 };
```

9.5 总结

本章主要讲解了 Linux 中的异步 I/O，异步 I/O 可以使得应用程序在等待 I/O 操作的同时进行其他操作。

使用信号可以实现设备驱动与用户程序之间的异步通知，总体而言，设备驱动和用户空间要分别完成以下工作：用户空间设置文件的拥有者、FASYNC 标志及捕获信号，内核空间响应对文件的拥有者、FASYNC 标志的设置，并在资源可获得时释放信号。

Linux 2.6 内核包含对 AIO 的支持为用户空间提供统一的异步 I/O 接口。在 AIO 中，信号和回调函数是实现内核空间对用户空间应用程序通知的两种机制。

- 嵌入式 Linux 系统开发班：

  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

- 嵌入式 Linux 驱动开发班：

  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm