

第 9 章

Linux 设备驱动中的异步通知与异步 I/O

本章导读

在设备驱动中使用异步通知可以使得在进行对设备的访问时，由驱动主动通知应用程序进行访问。这样，使用非阻塞 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。

除了异步通知以外，应用还可以在发起 I/O 请求后，立即返回。之后，再查询 I/O 完成情况，或者 I/O 完成后被调回。这个过程叫作异步 I/O。

9.1 节讲解了异步通知的概念与作用。

9.2 节讲解了 Linux 异步通知的编程方法。

9.3 节给出了增加异步通知的 globalfifo 驱动及其在用户空间的验证。

9.4 节则讲解了 Linux 基于 C 库的异步 I/O 和内核本身异步 I/O 的用户空间编程接口，以及驱动如何支持 AIO。

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，整套机制则更加完整了。

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用户自身可访问，之后用户再进行 I/O 处理。由此可见，这几种 I/O 方式可以相互补充。

图 9.1 呈现了阻塞 I/O，结合轮询的非阻塞 I/O 及基于 SIGIO 的异步通知在时间先后顺序上的不同。

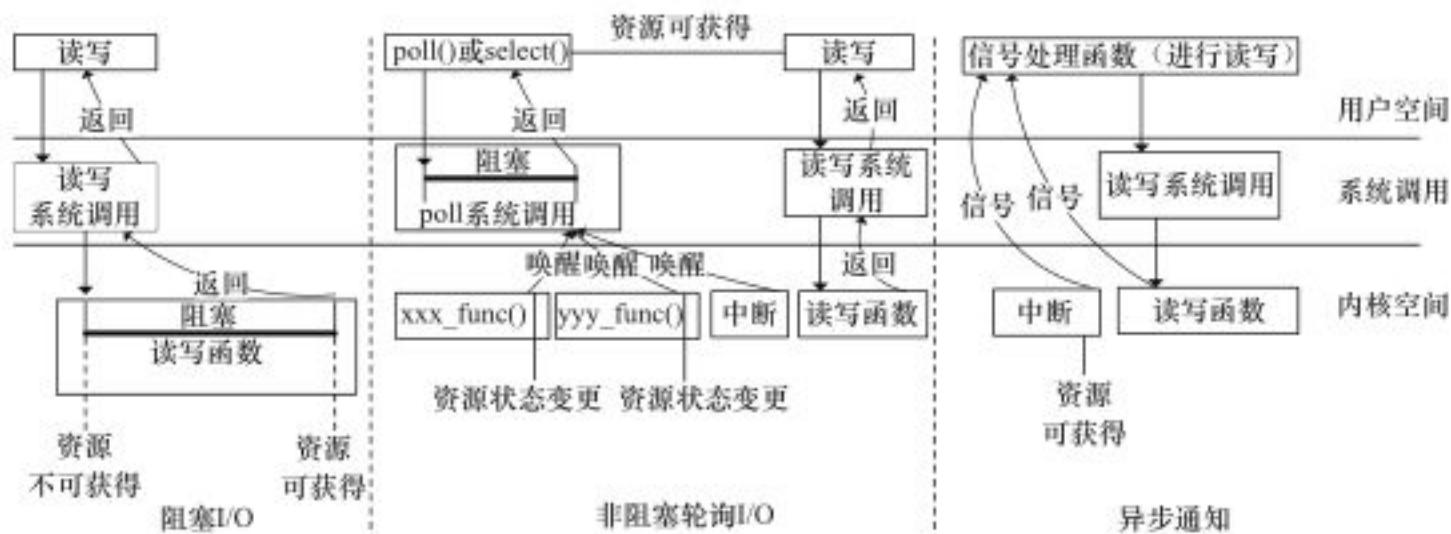


图 9.1 阻塞、结合轮询的非阻塞 I/O 和异步通知的区别

这里要强调的是：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 中的一种传统机制，Linux 也支持这种机制。在 Linux 中，异步通知使用信号来实现，Linux 中可用的信号及其定义如表 9.1 所示。

表 9.1 Linux 信号及其定义

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	半关闭管道的写操作已经发生
SIGALRM	14	计时器到期
SIGTERM	15	终止

(续)

信号	值	含义
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：

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

该函数原型较难理解，它可以分解为：

```
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 绑定的处理函数的 handler 值，失败则返回 SIG_ERR。

在进程执行时，按下“Ctrl+C”将向其发出 SIGINT 信号，正在运行 kill 的进程将向其发出 SIGTERM 信号，代码清单 9.1 的进程可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，它的原型为：

```
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，可以是除 `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，则进程会以缺省方式对信号处理；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
30     没有这个死循环会立即执行完毕 */
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，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 行、27 行代码可实现此目的。整个程序的执行效果如下：

```

[root@localhost driver_study]# ./signal_test
I am Chinese.          -> 用户输入
input available: I am Chinese.      -> signal_test 程序打印

I love Linux driver.      -> 用户输入
input available: I love Linux driver.  -> signal_test 程序打印

```

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”与驱使对应的应用程序中的 input_handler() 得以执行，并将用户输入显示出来。

由此可见，为了能在用户空间中处理一个设备释放的信号，它必须完成 3 项工作。

- 1) 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。
- 2) 通过 F_SETFL IO 控制命令设置设备文件以支持 FASYNC，即异步通知模式。
- 3) 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 3 项工作。

- 1) 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。
- 2) 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。

3) 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。

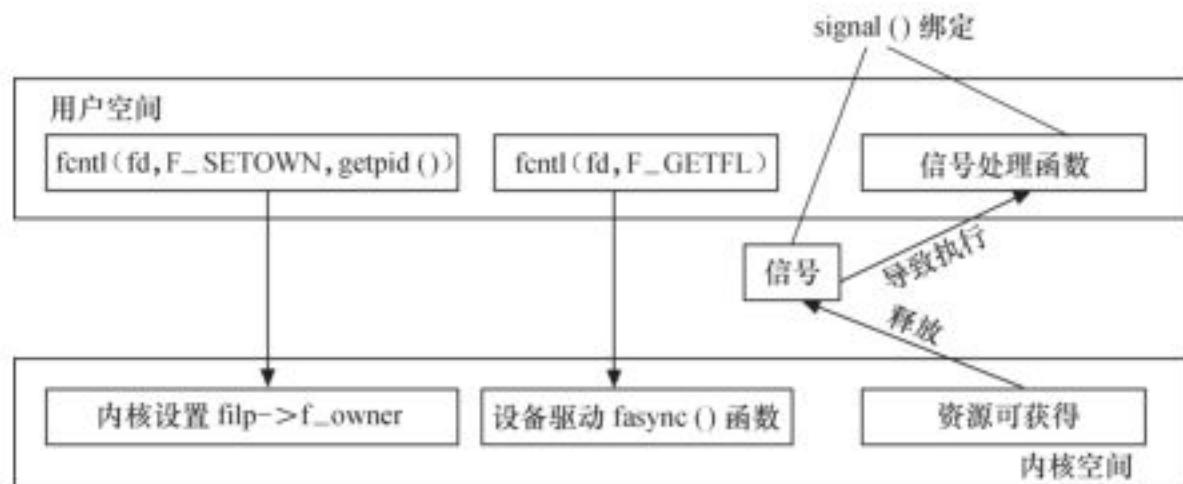


图 9.2 异步通知中设备驱动和异步通知的交互

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，两个函数分别是：

1) 处理 FASYNC 标志变更的函数。

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

2) 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     struct cdev cdev;                      /* cdev 结构体 */
3     ...
4     struct fasync_struct *async_queue;      /* 异步结构体指针 */
5 };

```

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。在可读时，第 3 个参数设置为 POLL_IN，在可写时，第 3 个参数设置为 POLL_OUT。代码清单 9.5 给出了释放信号的范例。

代码清单 9.5 支持异步通知的设备驱动信号释放范例

```

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 9.5 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() 函数模板

```

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

```

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，参考代码清单 9.3，应该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     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     struct fasync_struct *async_queue;
9 }

```

参考代码清单 9.4 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 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;
5     int ret;
6     DECLARE_WAITQUEUE(wait, current);
7
8     mutex_lock(&dev->mutex);
9     add_wait_queue(&dev->w_wait, &wait);
10
11    while (dev->current_len == GLOBALFIFO_SIZE) {
12        if (filp->f_flags & O_NONBLOCK) {
13            ret = -EAGAIN;
14            goto out;
15        }
16        __set_current_state(TASK_INTERRUPTIBLE);
17
18        mutex_unlock(&dev->mutex);
19
20        schedule();
21        if (signal_pending(current)) {
22            ret = -ERESTARTSYS;
23            goto out2;
24        }
25
26        mutex_lock(&dev->mutex);
27    }
28
29    if (count > GLOBALFIFO_SIZE - dev->current_len)
30        count = GLOBALFIFO_SIZE - dev->current_len;
31

```

```

32     if (copy_from_user(dev->mem + dev->current_len, buf, count)) {
33         ret = -EFAULT;
34         goto out;
35     } else {
36         dev->current_len += count;
37         printk(KERN_INFO "written %d bytes(s), current_len:%d\n", count,
38                dev->current_len);
39
40         wake_up_interruptible(&dev->r_wait);
41
42         if (dev->async_queue) {
43             kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
44             printk(KERN_DEBUG "%s kill SIGIO\n", __func__);
45         }
46
47         ret = count;
48     }
49
50 out:
51     mutex_unlock(&dev->mutex);
52 out2:
53     remove_wait_queue(&dev->w_wait, &wait);
54     set_current_state(TASK_RUNNING);
55     return ret;
56 }

```

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 release() 函数

```

1 static int globalfifo_release(struct inode *inode, struct file *filp)
2 {
3     globalfifo_fasync(-1, filp, 0);
4     return 0;
5 }

```

9.3.2 在用户空间中验证 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 static void signalio_handler(int signum)
2 {
3     printf("receive a signal from globalfifo,signalnum:%d\n", signum);

```

```

4  }
5
6 void main(void)
7 {
8     int fd, oflags;
9     fd = open("/dev/globalfifo", O_RDWR, S_IRUSR | S_IWUSR);
10    if (fd != -1) {
11        signal(SIGIO, signalio_handler);
12        fcntl(fd, F_SETOWN, getpid());
13        oflags = fcntl(fd, F_GETFL);
14        fcntl(fd, F_SETFL, oflags | FASYNC);
15        while (1) {
16            sleep(100);
17        }
18    } else {
19        printf("device open failure\n");
20    }
21 }

```

本书代码 /kernel/drivers/globalfifo/ch9 包含了支持异步通知的 globalfifo 驱动以及代码清单 9.11 对应的 globalfifo_test.c 测试程序，在该目录运行 make 将得到 globalfifo.ko 和 globalfifo_test。

按照与第 8 章相同的方法加载新的 globalfifo 设备驱动并创建设备文件节点，运行上述程序，每当通过 echo 向 /dev/globalfifo 写入新的数据时，signalio_handler() 将会被调用：

```

baohua@baohua-VirtualBox:~/develop/training/kernel/drivers/globalfifo/ch9$ sudo su

# ./globalfifo_test&
[1] 25251

# echo 1 > /dev/globalfifo
receive a signal from globalfifo,signalnum:29      -> globalfifo_test 程序打印

# echo hello > /dev/globalfifo
receive a signal from globalfifo,signalnum:29      -> globalfifo_test 程序打印

```

9.4 Linux 异步 I/O

9.4.1 AIO 概念与 GNU C 库 AIO

Linux 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。这是一种很好的解决方案，调用应用程序在等待 I/O 请求完成时不需要占用 CPU。但是在许多应用场景中，I/O 请求可能需要与 CPU 消耗产生交叠，以充分利用 CPU 和 I/O 提高吞吐率。

图 9.3 描绘了异步 I/O 的时序，应用程序发起 I/O 动作后，直接开始执行，并不等待 I/O

结束，它要么过一段时间来查询之前的I/O请求完成情况，要么I/O请求完成了会自动被调用与I/O完成绑定的回调函数。

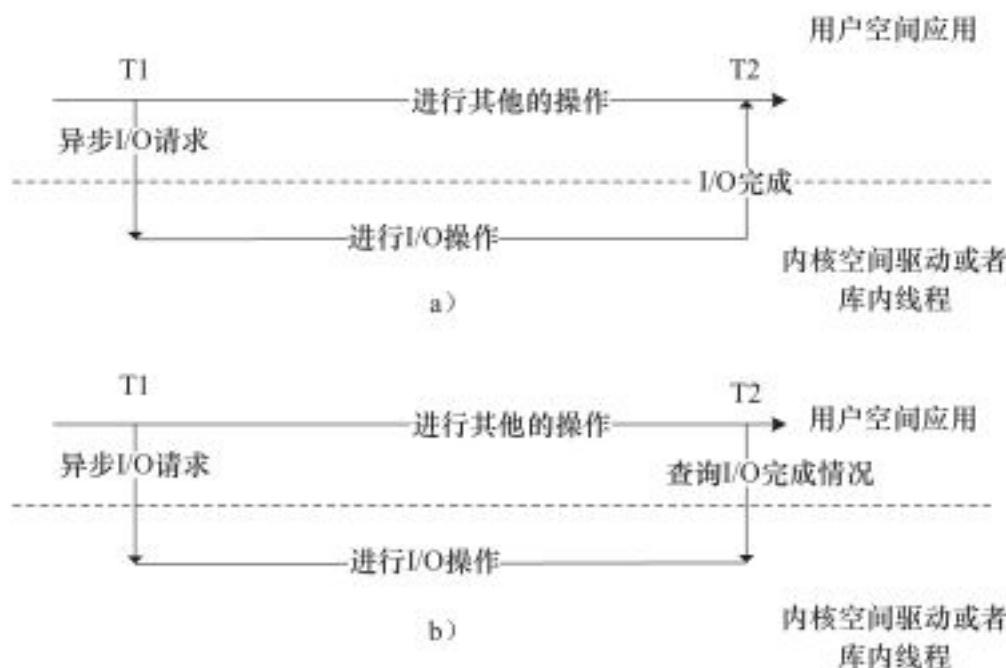


图9.3 异步I/O的时序

Linux的AIO有多种实现，其中一种实现是在用户空间的glibc库中实现的，它本质上是借用了多线程模型，用开启新的线程以同步的方法来做I/O，新的AIO辅助线程与发起AIO的线程以`pthread_cond_signal()`的形式进行线程间的同步。glibc的AIO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（尽管读操作并未完成）。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。

参数`aiocb`（AIO I/O Control Block）结构体包含了传输的所有信息，以及为AIO操作准备的用户空间缓冲区。在产生I/O完成通知时，`aiocb`结构就被用来唯一标识所完成的I/O操作。

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，并且它的请求已经被排队（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。

ECANCELED：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 和同步阻塞 I/O 方式之间的一个区别是不能立即访问这个函数的返回状态，因为异步 I/O 并没有阻塞在 read() 调用上。在标准的同步阻塞 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，返回值为负数）。

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(&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     continue;
29
30 if ((ret = aio_return(&my_aiocb)) > 0) {
31     /* 获得异步读的返回值 */
32 } else {
33     /* 读失败，分析 errno */
34 }

```

5. aio_suspend()

用户可以使用 `aio_suspend()` 函数来阻塞调用进程，直到异步请求完成为止。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 给出了用户空间进行异步读操作时使用 `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 的结构体链表中 */
5 cblist[0] = &my_aiocb;
6 ret = aio_read(&my_aiocb);
7 ret = aio_suspend(cblist, MAX_LIST, NULL );

```

当然，在 glibc 实现的 AIO 中，除了上述同步的等待方式以外，也可以使用信号或者回调机制来异步地标明 AIO 的完成。

6. aio_cancel()

`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);
```

要取消一个请求，用户需提供文件描述符和 `aiocb` 指针。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`。

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，并将 `aiocbp` 参数设置为 `NULL`。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`。

7. `lio_listio()`

`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。但是若是 `LIO_NOWAIT` 模型，在 I/O 操作进行排队之后，该函数就会返回。`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 给出了用户空间进行异步 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 aiocbl, aiocb2;
2 struct aiocb *list[MAX_LIST];
3 ...
4 /* 准备第一个 aiocb */
5 aiocbl.aio_fildes = fd;
6 aiocbl.aio_buf = malloc( BUFSIZE+1 );
7 aiocbl.aio_nbytes = BUFSIZE;
8 aiocbl.aio_offset = next_offset;
9 aiocbl.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] = &aiocbl;
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。

网页 http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html 包含了 AIO 库函数的详细信息。

9.4.2 Linux 内核 AIO 与 libaio

Linux AIO 也可以由内核空间实现，异步 I/O 是 Linux 2.6 以后版本内核的一个标准特性。对于块设备而言，AIO 可以一次性发出大量的 read/write 调用并且通过通用块层的 I/O 调度来获得更好的性能，用户程序也可以减少过多的同步负载，还可以在业务逻辑中更灵活地进行并发控制和负载均衡。相较于 glibc 的用户空间多线程同步等实现也减少了线程的负载和

上下文切换等。对于网络设备而言，在socket层面上，也可以使用AIO，让CPU和网卡的收发动作充分交叠以改善吞吐性能。选择正确的I/O模型对系统性能的影响很大，有兴趣的读者可以参阅著名的C10K问题（指的是服务器同时支持成千上万个客户端的问题），详见网址<http://www.kegel.com/c10k.html>。

在用户空间中，我们一般要结合libaio来进行内核AIO的系统调用。内核AIO提供的系统调用主要包括：

```
int io_setup(int maxevents, io_context_t *ctxp);
int io_destroy(io_context_t ctx);
int io_submit(io_context_t ctx, long nr, struct iocb *ios[]);
int io_cancel(io_context_t ctx, struct iocb *iocb, struct io_event *evt);
int io_getevents(io_context_t ctx_id, long min_nr, long nr, struct io_event *events,
    struct timespec *timeout);
void io_set_callback(struct iocb *iocb, io_callback_t cb);
void io_prep_pwrite(struct iocb *iocb, int fd, void *buf, size_t count, long long offset);
void io_prep_pread(struct iocb *iocb, int fd, void *buf, size_t count, long long offset);
void io_prep_pwritev(struct iocb *iocb, int fd, const struct iovec *iov, int iovcnt,
    long long offset);
void io_prep_preadv(struct iocb *iocb, int fd, const struct iovec *iov, int iovcnt,
    long long offset);
```

AIO的读写请求都用io_submit()下发。下发前通过io_prep_pwrite()和io_prep_pread()生成iocb的结构体，作为io_submit()的参数。这个结构体指定了读写类型、起始地址、长度和设备标志符等信息。读写请求下发之后，使用io_getevents()函数等待I/O完成事件。io_set_callback()则可设置一个AIO完成的回调函数。

代码清单9.15演示了一个简单的利用libaio向内核发起AIO请求的模版。该程序位于本书源代码的/kernel/drivers/globalfifo/ch9/aior.c下，使用命令gcc aior.c -o aior -laio编译，运行时带1个文本文件路径作为参数，该程序会打印该文本文件前4096个字节的内容。

代码清单9.15 使用libaio调用内核AIO的范例

```
1 #define _GNU_SOURCE /* O_DIRECT is not POSIX */
2 #include <stdio.h> /* for perror() */
3 #include <unistd.h> /* for syscall() */
4 #include <fcntl.h> /* O_RDWR */
5 #include <string.h> /* memset() */
6 #include <inttypes.h> /* uint64_t */
7 #include <stdlib.h>
8
9 #include <libaio.h>
10
11 #define BUF_SIZE 4096
12
13 int main(int argc, char **argv)
14 {
15     io_context_t ctx = 0;
```

```

16 struct iocb cb;
17 struct iocb *cbs[1];
18 unsigned char *buf;
19 struct io_event events[1];
20 int ret;
21 int fd;
22
23 if (argc < 2) {
24     printf("the command format: aior [FILE]\n");
25     exit(1);
26 }
27
28 fd = open(argv[1], O_RDWR | O_DIRECT);
29 if (fd < 0) {
30     perror("open error");
31     goto err;
32 }
33
34 /* Allocate aligned memory */
35 ret = posix_memalign((void **) &buf, 512, (BUF_SIZE + 1));
36 if (ret < 0) {
37     perror("posix_memalign failed");
38     goto err1;
39 }
40 memset(buf, 0, BUF_SIZE + 1);
41
42 ret = io_setup(128, &ctx);
43 if (ret < 0) {
44     printf("io_setup error:%s", strerror(-ret));
45     goto err2;
46 }
47
48 /* setup I/O control block */
49 io_prep_pread(&cb, fd, buf, BUF_SIZE, 0);
50
51 cbs[0] = &cb;
52 ret = io_submit(ctx, 1, cbs);
53 if (ret != 1) {
54     if (ret < 0) {
55         printf("io_submit error:%s", strerror(-ret));
56     } else {
57         fprintf(stderr, "could not sumbit IOs");
58     }
59     goto err3;
60 }
61
62 /* get the reply */
63 ret = io_getevents(ctx, 1, 1, events, NULL);
64 if (ret != 1) {
65     if (ret < 0) {
66         printf("io_getevents error:%s", strerror(-ret));
67     }
68 }
69
70 /* cleanup */
71 io_destroy(ctx);
72
73 /* close file */
74 close(fd);
75
76 /* free memory */
77 free(buf);
78
79 /* exit */
80 exit(0);

```

```

67         } else {
68             fprintf(stderr, "could not get Events");
69         }
70         goto err3;
71     }
72     if (events[0].res2 == 0) {
73         printf("%s\n", buf);
74     } else {
75         printf("AIO error:%s", strerror(-events[0].res));
76         goto err3;
77     }
78
79     if ((ret = io_destroy(ctx)) < 0) {
80         printf("io_destroy error:%s", strerror(-ret));
81         goto err2;
82     }
83
84     free(buf);
85     close(fd);
86     return 0;
87
88 err3:
89     if ((ret = io_destroy(ctx)) < 0)
90         printf("io_destroy error:%s", strerror(-ret));
91 err2:
92     free(buf);
93 err1:
94     close(fd);
95 err:
96     return -1;
97 }

```

9.4.3 AIO 与设备驱动

用户空间调用 `io_submit()` 后，对于用户传递的每一个 `iocb` 结构，内核会生成一个与之对应的 `kiocb` 结构。`file_operations` 包含 3 个与 AIO 相关的成员函数：

```

ssize_t (*aio_read) (struct kiocb *iocb, const struct iovec *iov, unsigned long
                     nr_segs, loff_t pos);
ssize_t (*aio_write) (struct kiocb *iocb, const struct iovec *iov, unsigned
                      long nr_segs, loff_t pos);
int (*aio_fsync) (struct kiocb *iocb, int datasync);

```

`io_submit()` 系统调用间接引起了 `file_operations` 中的 `aio_read()` 和 `aio_write()` 的调用。在早期的 Linux 内核中，`aio_read()` 和 `aio_write()` 的原型是：

```

ssize_t (*aio_read) (struct kiocb *iocb, char __user *buffer,
                     size_t size, loff_t pos);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,

```

```
size_t count, loff_t offset);
```

在这个老的原型里，只含有一个缓冲区指针，而在新的原型中，则可以传递一个向量 iovec，它含有多段缓冲区。详见位于 <https://lwn.net/Articles/170954/> 的文档《Asynchronous I/O and vectored operations》。

AIO 一般由内核空间的通用代码处理，对于块设备和网络设备而言，一般在 Linux 核心层的代码已经解决。字符设备驱动一般不需要实现 AIO 支持。Linux 内核中对字符设备驱动实现 AIO 的特例包括 drivers/char/mem.c 里实现的 null、zero 等，由于 zero 这样的虚拟设备其实也不存在在要去读的时候读不到东西的情况，所以 aio_read_zero() 本质上也不包含异步操作，不过从代码清单 9.16 我们可以一窥 iovec 的全貌。

代码清单 9.16 zero 设备的 aio_read 实现

```
1 static ssize_t aio_read_zero(struct kiocb *iocb, const struct iovec *iov,
2                               unsigned long nr_segs, loff_t pos)
3 {
4     size_t written = 0;
5     unsigned long i;
6     ssize_t ret;
7
8     for (i = 0; i < nr_segs; i++) {
9         ret = read_zero(iocb->ki_filp, iov[i].iov_base, iov[i].iov_len,
10                      &pos);
11        if (ret < 0)
12            break;
13        written += ret;
14    }
15
16    return written ? written : -EFAULT;
17 }
```

9.5 总结

本章主要讲述了 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 3 项对应的工作，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 以后的内核包含对 AIO 的支持，它为用户空间提供了统一的异步 I/O 接口。另外，glibc 也提供了一个不依赖于内核的用户空间的 AIO 支持。