“黑色经典”系列之《嵌入式 Linux 应用程序开发详解》

第 6 章 文件 I/O 编程

本章目标

在搭建起嵌入式开发环境之后，从本章开始，读者将真正开始学习嵌入式 Linux 的应用开发。由于嵌入式 Linux 是经 Linux 裁减而来的，它的系统调用及用户编程接口 API 与 Linux 基本是一致的，因此，在以后的章节中，笔者将首先介绍 Linux 中相关内容的基本编程开发，主要讲解与嵌入式 Linux 中一致的部分，然后再将程序移植到嵌入式的开发板上运行。因此，没有开发板的读者也可以先在 Linux 上开发相关应用程序，这对以后进入嵌入式 Linux 的实际开发是十分有帮助的。本章主要讲解文件 I/O 相关开发，经过本章的学习，读者将会掌握以下内容。

- 掌握 Linux 中系统调用的基本概念
- 掌握 Linux 中用户编程接口（API）及系统命令的相互关系
- 掌握文件描述符的概念
- 掌握 Linux 下文件相关的不带缓存 I/O 函数的使用
- 掌握 Linux 下设备文件读写方法
- 掌握 Linux 中对串口的操作
- 熟悉 Linux 中标准文件 I/O 函数的使用

6.1 Linux 系统调用及用户编程接口（API）

由于本章是讲解 Linux 编程开发的第 1 章，因此希望读者更加明确 Linux 系统调用和用户编程接口（API）的概念。在了解了这些之后，会对 Linux 以及 Linux 的应用编程有更深入地理解。

6.1.1 系统调用

所谓系统调用是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。例如用户可以通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

在这里，为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在 Linux 中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。

Linux 系统调用部分是非常精简的系统调用（只有 250 个左右），它继承了 UNIX 系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、系统控制、存储管理、网络管理、socket 控制、用户管理等几类。

6.1.2 用户编程接口（API）

前面讲到的系统调用并不是直接与程序员进行交互的，它仅仅是一个通过软中断机制向内核提交请求，以获取内核服务的接口。在实际使用中程序员调用的通常是用户编程接口——API，也就是本书后面要讲到的 API 函数。但并不是所有的函数都一一对应一个系统调用，有时，一个 API 函数会需要几个系统调用来共同完成函数的功能，甚至还有一些 API 函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

在 Linux 中，用户编程接口（API）遵循了在 UNIX 中最流行的应用编程界面标准——POSIX 标准。POSIX 标准是由 IEEE 和 ISO/IEC 共同开发的标准系统。该标准基于当时现有的 UNIX 实践和经验，描述了操作系统的系统调用编程接口（实际上就是 API），用于保证应用程序可以在源代码一级上在多种操作系统上移植运行。这些系统调用编程接口主要是通过 C 库（libc）实现的。

6.1.3 系统命令

以上讲解了系统调用、用户编程接口（API）的概念，分析了它们之间的相互关系，那么，读者在第 2 章中学到的那么多的 Shell 系统命令与它们之间又是怎样的关系呢？

系统命令相对 API 更高了一层，它实际上一个可执行程序，它的内部引用了用户编程接口（API）来实现相应的功能。它们之间的关系如下图 6.1 所示。

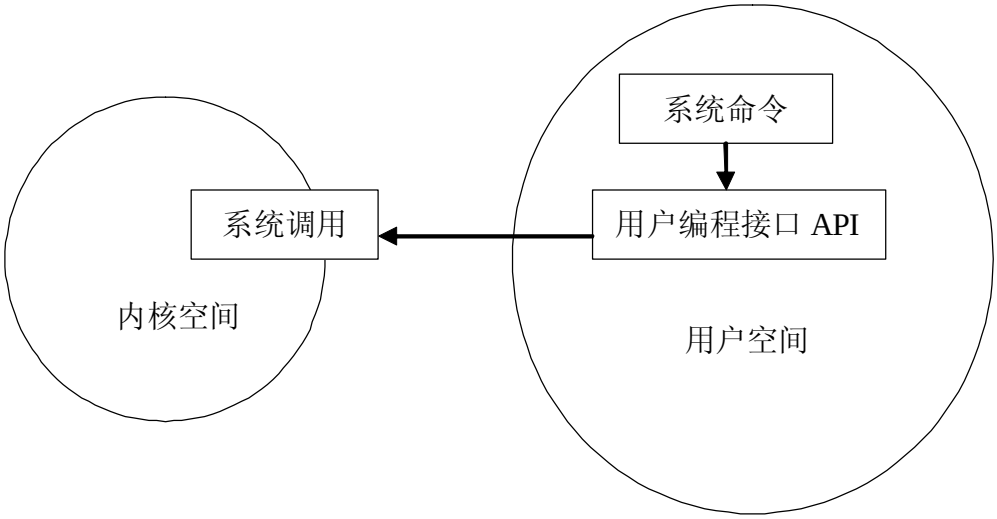


图 6.1　系统调用、API 及系统命令之间的关系

6.2　Linux 中文件及文件描述符概述

正如第 1 章中所述，在 Linux 中对目录和设备的操作都等同于文件的操作，因此，大大简化了系统对不同设备的处理，提高了效率。Linux 中的文件主要分为 4 种：普通文件、目录文件、链接文件和设备文件。

那么，内核如何区分和引用特定的文件呢？这里用到的就是一个重要的概念——文件描述符。对于 Linux 而言，所有对设备和文件的操作都使用文件描述符来进行的。文件描述符是一个非负的整数，它是一个索引值，并指向内核中每个进程打开文件的记录表。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

通常，一个进程启动时，都会打开 3 个文件：标准输入、标准输出和标准出错处理。这 3 个文件分别对应文件描述符为 0、1 和 2（也就是宏替换 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO，鼓励读者使用这些宏替换）。

基于文件描述符的 I/O 操作虽然不能移植到类 Linux 以外的系统上去（如 Windows），但它往往是实现某些 I/O 操作的惟一途径，如 Linux 中低级文件操作函数、多路 I/O、TCP/IP 套接字编程接口等。同时，它们也很好地兼容 POSIX 标准，因此，可以很方便地移植到任何 POSIX 平台上。基于文件描述符的 I/O 操作是 Linux 中最常用的操作之一，希望读者能够很好地掌握。

6.3　不带缓存的文件 I/O 操作

本节主要介绍不带缓存的文件 I/O 操作，主要用到 5 个函数：open、read、write、lseek 和 close。这里的不带缓存是指每一个函数都只调用系统中的一个函数。这些函数虽然不是

ANSI C 的组成部分，但是是 POSIX 的组成部分。

6.3.1 open 和 close

（1）open 和 close 函数说明

open 函数是用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

close 函数是用于关闭一个打开文件。当一个进程终止时，它所有已打开的文件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。

（2）open 和 close 函数格式

open 函数的语法格式如表 6.1 所示。

表 6.1 open 函数语法要点

所需头文件	#include <sys/types.h> // 提供类型 pid_t 的定义 #include <sys/stat.h> #include <fcntl.h>	
续表		
函数原型	int open(const char *pathname，flags，int perms)	
函数传入值	pathname	被打开的文件名（可包括路径名）
	flag：文件打开的方式	O_RDONLY：只读方式打开文件
		O_WRONLY：可写方式打开文件
		O_RDWR：读写方式打开文件
		O_CREAT：如果该文件不存在，就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL：如果使用 O_CREAT 时文件存在，则可返回错误消息。这一参数可测试文件是否存在
		O_NOCTTY：使用本参数时，如文件为终端，那么终端不可以作为调用 open()系统调用的那个进程的控制终端
		O_TRUNC：如文件已经存在，并且以只读或只写成功打开，那么会先全部删除文件中原有数据
		O+APPEND：以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾
	perms	被打开文件的存取权限，为 8 进制表示法
函数返回值	成功：返回文件描述符 失败：–1	

在 open 函数中，flag 参数可通过“|”组合构成，但前 3 个函数不能相互组合。perms 是文件的存取权限，采用 8 进制表示法，相关内容读者可参见第 2 章。

close 函数的语法格式如下表 6.2 所示。

表 6.2 close 函数语法要点

所需头文件	#include <unistd.h>
函数原型	int close(int fd)
函数输入值	fd：文件描述符
函数返回值	0：成功 –1：出错

（3）open 和 close 函数使用实例

下面实例中的 open 函数带有 3 个 flag 参数：O_CREAT、O_TRUNC 和 O_WRONLY，这样就可以对不同的情况指定相应的处理方法。另外，这里对该文件的权限设置为 0600。其源码如下所示：

```
/*open.c*/
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
      int fd;
/*调用 open 函数，以可读写的方式打开，注意选项可以用“|”符号连接*/
      if((fd = open("/tmp/hello.c", O_CREAT | O_TRUNC | O_WRONLY , 0600 ))<0){
            perror("open:");
            exit(1);
      }
      else{
            printf("Open file: hello.c %d\n",fd);
      }
      if( close(fd) < 0 ){
            perror("close:");
            exit(1);
      }
      else
            printf("Close hello.c\n");
      exit(0);
}
[root@(none) 1]# ./open
```

```
Open file: hello.c 3
Close hello.c
[root@(none) tmp]# ls -l |grep hello.c
-rw-------    1 root     root           0 Dec  4 00:59 hello.c
```

经过交叉编译后，将文件下载到目标板，则该可执行文件运行后就能在目录/tmp 下新建一个 hello.c 的文件，其权限为 0600。

注意 open 函数返回的文件描述符一定是最小的未用文件描述符。由于一个进程在启动时自动打开了 0、1、2 三个文件描述符，因此，该文件运行结果中返回的文件描述符为 3。读者可以尝试在调用 open 函数之前，加依据 close(0)，则此后在 open 函数时返回的文件描述符为 0（若关闭文件描述符 1，则在执行时会由于没有标准输出文件而无法输出）。

6.3.2 read、write 和 lseek

（1）read、write 和 lseek 函数作用

read 函数是用于将指定的文件描述符中读出数据。当从终端设备文件中读出数据时，通常一次最多读一行。

write 函数是用于向打开的文件写数据，写操作从文件的当前位移量处开始。若磁盘已满或超出该文件的长度，则 write 函数返回失败。

lseek 函数是用于在指定的文件描述符中将文件指针定位到相应的位置。

（2）read 和 write 函数格式

read 函数的语法格式如下表 6.3 所示。

表 6.3 read 函数语法要点

所需头文件	#include <unistd.h>
函数原型	ssize_t read(int fd,void *buf,size_t count)
函数传入值	fd：文件描述符
	buf：指定存储器读出数据的缓冲区
	count：指定读出的字节数
函数返回值	成功：读到的字节数 0：已到达文件尾 −1：出错

在读普通文件时，若读到要求的字节数之前已到达文件的尾部，则返回的字节数会小于希望读出的字节数。

write 函数的语法格式如下表 6.4 所示。

表 6.4 write 函数语法要点

所需头文件	#include <unistd.h>
函数原型	ssize_t write(int fd,void *buf,size_t count)

函数传入值	fd：文件描述符
	buf：指定存储器写入数据的缓冲区
	count：指定读出的字节数
函数返回值	成功：已写的字节数 –1：出错

在写普通文件时，写操作从文件的当前位移处开始。

lseek 函数的语法格式如下表 6.5 所示。

表 6.5　　lseek 函数语法要点

所需头文件	#include <unistd.h> #include <sys/types.h>	
函数原型	off_t lseek(int fd,off_t offset,int whence)	
函数传入值	fd：文件描述符	
	offset：偏移量，每一读写操作所需要移动的距离，单位是字节的数量，可正可负（向前移，向后移）	
续表		
	whence：当前位置的基点	SEEK_SET：当前位置为文件的开头，新位置为偏移量的大小
		SEEK_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量
		SEEK_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小
函数返回值	成功：文件的当前位移 –1：出错	

（3）函数使用实例

该示例程序首先打开上一节中创建的文件，然后对此文件进行读写操作（记得要将文件打开属性改为可读写，将文件权限也做相应更改）。接着，写入“Hello! I'm writing to this file!”，此时文件指针位于文件尾部。接着在使用 lseek 函数将文件指针移到文件开始处，并读出 10 个字节并将其打印出来。程序源代码如下所示：

```
/*write.c*/
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MAXSIZE
```

```
int main(void)
{
     int i,fd,size,len;
     char *buf="Hello! I'm writing to this file!";
     char buf_r[10];
     len = strlen(buf);
/*首先调用 open 函数，并指定相应的权限*/
     if((fd = open("/tmp/hello.c", O_CREAT | O_TRUNC | O_RDWR,0666 ))<0){
            perror("open:");
            exit(1);
     }
     else
            printf("open file:hello.c %d\n",fd);
/*调用 write 函数，将 buf 中的内容写入到打开的文件中*/
     if((size = write( fd, buf, len)) < 0){
            perror("write:");
            exit(1);
     }
     else
            printf("Write:%s\n",buf);
/*调用 lsseek 函数将文件指针移到文件起始，并读出文件中的 10 个字节*/
     lseek( fd, 0, SEEK_SET );
     if((size = read( fd, buf_r, 10))<0){
            perror("read:");
            exit(1);
     }
     else
            printf("read form file:%s\n",buf_r);
     if( close(fd) < 0 ){
            perror("close:");
            exit(1);
     }
     else
            printf("Close hello.c\n");
     exit(0);
}
[root@(none) 1]# ./write
open file:hello.c 3
```

```
Write:Hello! I'm writing to this file!
read form file:Hello! I'm
Close hello.c
[root@(none) 1]# cat /tmp/hello.c
Hello! I'm writing to this file!
```

6.3.3 fcntl

（1）fcntl 函数说明

前面的这 5 个基本函数实现了文件的打开、读写等基本操作，这一节将讨论的是，在文件已经共享的情况下如何操作，也就是当多个用户共同使用、操作一个文件的情况，这时，Linux 通常采用的方法是给文件上锁，来避免共享的资源产生竞争的状态。

文件锁包括建议性锁和强制性锁。建议性锁要求每个上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁。强制性锁是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何文件对其进行读写操作。采用强制性锁对性能的影响很大，每次读写操作都必须检查是否有锁存在。

在 Linux 中，实现文件上锁的函数有 lock 和 fcntl，其中 flock 用于对文件施加建议性锁，而 fcntl 不仅可以施加建议性锁，还可以施加强制锁。同时，fcntl 还能对文件的某一记录进行上锁，也就是记录锁。

记录锁又可分为读取锁和写入锁，其中读取锁又称为共享锁，它能够使多个进程都能在文件的同一部分建立读取锁。而写入锁又称为排斥锁，在任何时刻只能有一个进程在文件的某个部分上建立写入锁。当然，在文件的同一部分不能同时建立读取锁和写入锁。

注意 fcntl 是一个非常通用的函数，它还可以改变文件进程各方面的属性，在本节中，主要介绍它建立记录锁的方法，关于它其他用户感兴趣的读者可以参看 fcntl 手册。

（2）fcntl 函数格式

用于建立记录锁的 fcntl 函数格式如表 6.6 所示。

表 6.6 fcntl 函数语法要点

所需头文件	#include <sys/types.h> #include <unistd.h> #include <fcntl.h>	
函数原型	int fcnt1(int fd,int cmd,struct flock *lock)	
函数传入值	fd：文件描述符	
	cmd	F_DUPFD：复制文件描述符
		F_GETFD：获得 fd 的 close-on-exec 标志，若标志未设置，则文件经过 exec 函数之后仍保持打开状态
		F_SETFD：设置 close-on-exec 标志，该标志以参数 arg 的 FD_CLOEXEC 位决定
		F_GETFL：得到 open 设置的标志
		F_SETFL：改变 open 设置的标志

		F_GETFK：根据 lock 描述，决定是否上文件锁
		F_SETFK：设置 lock 描述的文件锁
		F_SETLKW：这是 F_SETLK 的阻塞版本（命令名中的 W 表示等待（wait））。如果存在其他锁，则调用进程睡眠；如果捕捉到信号则睡眠中断
		F_GETOWN：检索将收到 SIGIO 和 SIGURG 信号的进程号或进程组号
		F_SETOWN：设置进程号或进程组号
	Lock：结构为 flock，设置记录锁的具体状态，后面会详细说明	
函数返回值	成功：0 –1：出错	

这里，lock 的结构如下所示：

```
Struct flock{
short l_type;
off_t l_start;
short l_whence;
off_t l_len;
pid_t l_pid;
}
```

lock 结构中每个变量的取值含义如表 6.7 所示。

表 6.7 lock 结构变量取值

l_type	F_RDLCK：读取锁（共享锁）
	F_WRLCK：写入锁（排斥锁）
	F_UNLCK：解锁
l_stat	相对位移量（字节）
l_whence：相对位移量的起点（同 lseek 的 whence）。	SEEK_SET：当前位置为文件的开头，新位置为偏移量的大小
	SEEK_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量
	SEEK_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小
l_len	加锁区域的长度

小技巧 为加锁整个文件，通常的方法是将 l_start 说明为 0，l_whence 说明为 SEEK_SET，l_len 说明为 0。

（3）fcntl 使用实例

下面首先给出了使用 fcntl 函数的文件记录锁函数。在该函数中，首先给 flock 结构体的对应位赋予相应的值。接着使用两次 fcntl 函数分别用于给相关文件上锁和判断文件是否可以上锁，这里用到的 cmd 值分别为 F_SETLK 和 F_GETLK。

这个函数的源代码如下所示：

```
/*lock_set 函数*/
void lock_set(int fd, int type)
{
      struct flock lock;
      lock.l_whence = SEEK_SET;//赋值 lock 结构体
      lock.l_start = 0;
      lock.l_len =0;
      while(1){
            lock.l_type = type;
/*根据不同的 type 值给文件上锁或解锁*/
            if((fcntl(fd, F_SETLK, &lock)) == 0){
            if( lock.l_type == F_RDLCK )
                  printf("read lock set by %d\n",getpid());
            else if( lock.l_type == F_WRLCK )
                  printf("write lock set by %d\n",getpid());
            else if( lock.l_type == F_UNLCK )
                  printf("release lock by %d\n",getpid());
            return;
            }
/*判断文件是否可以上锁*/
            fcntl(fd, F_GETLK,&lock);
/*判断文件不能上锁的原因*/
            if(lock.l_type != F_UNLCK){
/*/该文件已有写入锁*/
                  if( lock.l_type == F_RDLCK )
                        printf("read lock already set by %d\n",lock.l_pid);
/*该文件已有读取锁*/
                  else if( lock.l_type == F_WRLCK )
                        printf("write lock already set by %d\n",lock.l_pid);
                  getchar();
            }
      }
}
```

下面的实例是测试文件的写入锁，这里首先创建了一个 hello 文件，之后对其上写入锁，最后释放写入锁。代码如下所示：

```
/*fcntl_write.c 测试文件写入锁主函数部分*/
#include <unistd.h>
#include <sys/file.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
     int fd;
/*首先打开文件*/
     fd=open("hello",O_RDWR | O_CREAT, 0666);
     if(fd < 0){
           perror("open");
           exit(1);
     }
/*给文件上写入锁*/
     lock_set(fd, F_WRLCK);
     getchar();
/*给文件接锁*/
     lock_set(fd, F_UNLCK);
     getchar();
     close(fd);
     exit(0);
}
```

为了能够使用多个终端，更好地显示写入锁的作用，本实例主要在 PC 机上测试，读者可将其交叉编译，下载到目标板上运行。下面是在 PC 机上的运行结果。为了使程序有较大的灵活性，笔者采用文件上锁后由用户键入一任意键使程序继续运行。建议读者开启两个终端，并且在两个终端上同时运行该程序，以达到多个进程操作一个文件的效果。在这里，笔者首先运行终端一，请读者注意终端二中的第一句。

终端一：

```
[root@localhost file]# ./fcntl_write
write lock set by 4994

release lock by 4994
```

终端二：

```
[root@localhost file]# ./fcntl_write
write lock already set by 4994
```

```
write lock set by 4997

release lock by 4997
```

由此可见，写入锁为互斥锁，一个时刻只能有一个写入锁存在。
接下来的程序是测试文件的读取锁，原理同上面的程序一样。

```
/*fcntl_read.c 测试文件读取锁主函数部分*/
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
     int fd;
     fd=open("hello",O_RDWR | O_CREAT, 0666);
     if(fd < 0){
           perror("open");
           exit(1);
     }
/*给文件上读取锁*/
     lock_set(fd, F_RDLCK);
     getchar();
/*给文件接锁*/
     lock_set(fd, F_UNLCK);
     getchar();
     close(fd);
     exit(0);
}
```

同样开启两个终端，并首先启动终端一上的程序，其运行结果如下所示：
终端一：

```
[root@localhost file]# ./fcntl2
read lock set by 5009

release lock by 5009
```

终端二：

```
[root@localhost file]# ./fcntl2
read lock set by 5010

release lock by 5010
```

读者可以将此结果与写入锁的运行结果相比较，可以看出，读取锁为共享锁，当进程 5009 已设定读取锁后，进程 5010 还可以设置读取锁。

思考 如果在一个终端上运行设置读取锁，则在另一个终端上运行设置写入锁，会有什么结果呢？

6.3.4 select

（1）select 函数说明

前面的 fcntl 函数解决了文件的共享问题，接下来该处理 I/O 复用的情况了。

总的来说，I/O 处理的模型有 5 种。

- **阻塞 I/O 模型：**在这种模型下，若所调用的 I/O 函数没有完成相关的功能就会使进程挂起，直到相关数据到才会出错返回。如常见对管道设备、终端设备和网络设备进行读写时经常会出现这种情况。
- **非阻塞模型：**在这种模型下，当请求的 I/O 操作不能完成时，则不让进程睡眠，而且返回一个错误。非阻塞 I/O 使用户可以调用不会永远阻塞的 I/O 操作，如 open、write 和 read。如果该操作不能完成，则会立即出错返回，且表示该 I/O 如果该操作继续执行就会阻塞。
- **I/O 多路转接模型：**在这种模型下，如果请求的 I/O 操作阻塞，且它不是真正阻塞 I/O，而是让其中的一个函数等待，在这期间，I/O 还能进行其他操作。如本节要介绍的 select 函数和 poll 函数，就是属于这种模型。
- **信号驱动 I/O 模型：**在这种模型下，通过安装一个信号处理程序，系统可以自动捕获特定信号的到来，从而启动 I/O。这是由内核通知用户何时可以启动一个 I/O 操作决定的。
- **异步 I/O 模型：**在这种模型下，当一个描述符已准备好，可以启动 I/O 时，进程会通知内核。现在，并不是所有的系统都支持这种模型。

可以看到，select 的 I/O 多路转接模型是处理 I/O 复用的一个高效的方法。它可以具体设置每一个所关心的文件描述符的条件、希望等待的时间等，从 select 函数返回时，内核会通知用户已准备好的文件描述符的数量、已准备好的条件等。通过使用 select 返回值，就可以调用相应的 I/O 处理函数了。

（2）select 函数格式

Select 函数的语法格式如表 6.8 所示。

表 6.8 fcntl 函数语法要点

所需头文件	#include <sys/types.h> #include <sys/time.h>

	#include <unistd.h>	
函数原型	int select(int numfds,fd_set *readfds,fd_set *writefds，fd_set *exeptfds,struct timeval *timeout)	
函数传入值	numfds：需要检查的号码最高的文件描述符加 1	
	readfds：由 select()监视的读文件描述符集合	
	writefds：由 select()监视的写文件描述符集合	
	exeptfds：由 select()监视的异常处理文件描述符集合	
	timeout	NULL：永远等待，直到捕捉到信号或文件描述符已准备好为止
		具体值：struct timeval 类型的指针，若等待为 timeout 时间还没有文件描符准备好，就立即返回
		0：从不等待，测试所有指定的描述符并立即返回
函数返回值	成功：准备好的文件描述符 –1：出错	

思考 请读者考虑一下如何确定最高的文件描述符？

可以看到，select 函数根据希望进行的文件操作对文件描述符进行了分类处理，这里，对文件描述符的处理主要涉及到 4 个宏函数，如表 6.9 所示。

表 6.9 select 文件描述符处理函数

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)	测试该集中的一个给定位是否有变化

一般来说，在使用 select 函数之前，首先使用 FD_ZERO 和 FD_SET 来初始化文件描述符集，在使用了 select 函数时，可循环使用 FD_ISSET 测试描述符集，在执行完对相关后文件描述符后，使用 FD_CLR 来清楚描述符集。

另外，select 函数中的 timeout 是一个 struct timeval 类型的指针，该结构体如下所示：

```
struct timeval {
       long tv_sec; /* second */
       long tv_unsec; /* and microseconds*/
}
```

可以看到，这个时间结构体的精确度可以设置到微秒级，这对于大多数的应用而言已经足够了。

（3）使用实例

由于 Select 函数多用于 I/O 操作可能会阻塞的情况下，而对于可能会有阻塞 I/O 的管道、

网络编程，本书到现在为止还没有涉及。因此，本例主要表现了如何使用 select 函数，而其中的 I/O 操作是不会阻塞的。

本实例中主要实现将文件 hello1 里的内容读出，并将此内容每隔 10s 写入 hello2 中去。在这里建立了两个描述符集，其中一个描述符集 inset1 是用于读取文件内容，另一个描述符集 inset2 是用于写入文件的。两个文件描述符 fds[0]和 fds[1]分别指向这一文件描述符。在首先初始化完各文件描述符集之后，就开始了循环测试这两个文件描述符是否可读写，由于在这里没有阻塞，所以文件描述符处于准备就绪的状态。这时，就分别对文件描述符 fds[0]和 fsd[1]进行读写操作。该程序的流程图如图 6.2 所示。

```
/*select.c*/
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
     int fds[2];
     char buf[7];
     int i,rc,maxfd;
     fd_set inset1,inset2;
     struct timeval tv;
/*首先按一定的权限打开 hello1 文件*/
     if((fds[0] = open ("hello1", O_RDWR|O_CREAT,0666))<0)
           perror("open hello1");
/*再按一定的权限打开 hello2 文件*/
     if((fds[1] = open ("hello2", O_RDWR|O_CREAT,0666))<0)
           perror("open hello2");
     if((rc = write(fds[0],"Hello!\n",7)))
           printf("rc=%d\n",rc);
     lseek(fds[0],0,SEEK_SET);
/*取出两个文件描述符中的较大者*/
     maxfd = fds[0]>fds[1] ? fds[0] : fds[1];
/*初始化读集合 inset1，并在读集合中加入相应的描述集*/
     FD_ZERO(&inset1);
     FD_SET(fds[0],&inset1);
/*初始化写集合 inset2，并在写集合中加入相应的描述集*/
     FD_ZERO(&inset2);
```

```
        FD_SET(fds[1],&inset2);
        tv.tv_sec=2;
        tv.tv_usec=0;
    /*循环测试该文件描述符是否准备就绪，并调用 select 函数对相关文件描述符做对应操作*/
        while(FD_ISSET(fds[0],&inset1)||FD_ISSET(fds[1],&inset2)){
            if(select(maxfd+1,&inset1,&inset2,NULL,&tv)<0)
                perror("select");
            else{
                if(FD_ISSET(fds[0],&inset1)){
                    rc = read(fds[0],buf,7);
                    if(rc>0){
                        buf[rc]='\0';
                        printf("read: %s\n",buf);
                    }else
                        perror("read");
                }
                if(FD_ISSET(fds[1],&inset2)){
                    rc = write(fds[1],buf,7);
                    if(rc>0){
                        buf[rc]='\0';
                        printf("rc=%d,write: %s\n",rc,buf);
                    }else
                        perror("write");
                sleep(10);
                }
            }
        }
        exit(0);
    }
```

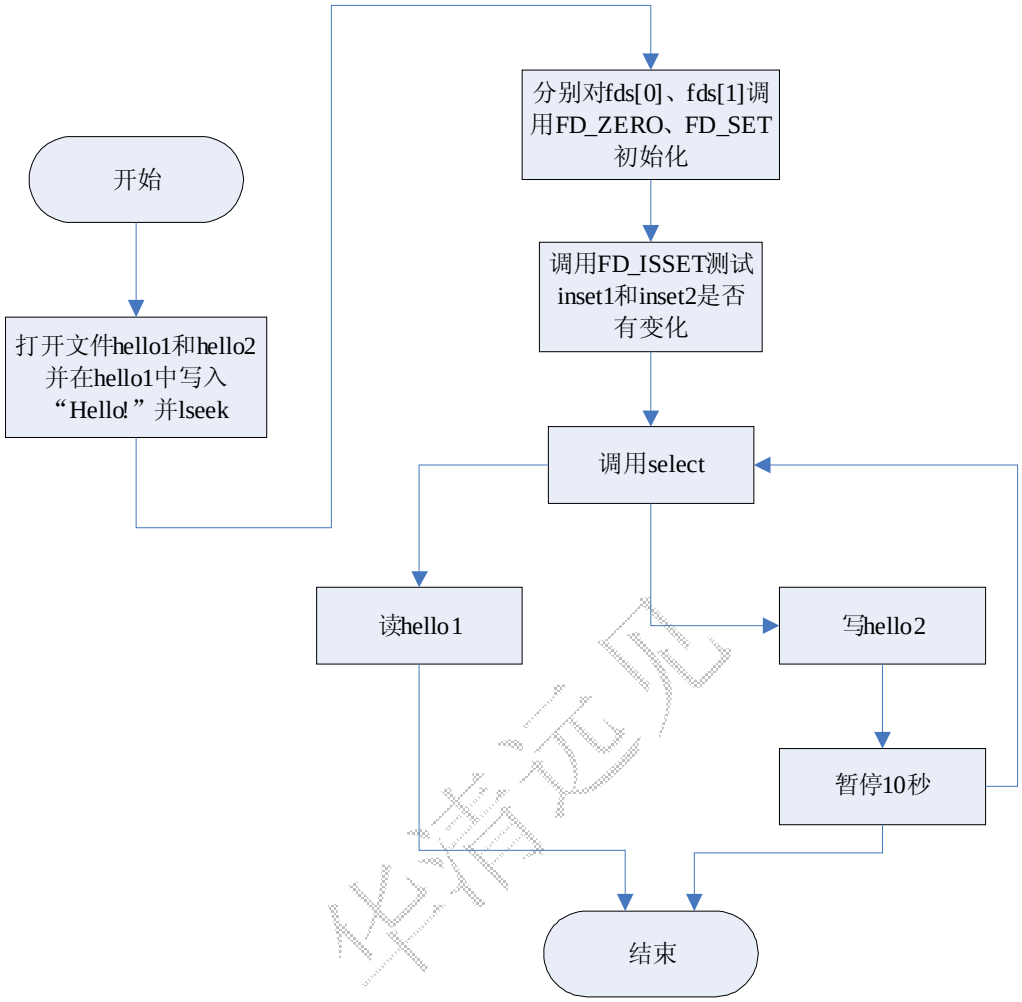


图 6.2 select 实例流程图

读者可以将以上程序交叉编译，并下载到开发板上运行。以下是运行结果：

```
[root@(none) 1]# ./select
rc=7
read: Hello!

rc=7,write: Hello!

rc=7,write:Hello!

rc=7,write:Hello!

...
```

```
[root@(none) 1]# cat hello1
Hello!
[root@(none) 1]# cat hello2
Hello!
Hello!
…
```

可以看到，使用 select 可以很好地实现 I/O 多路复用，在有阻塞的情况下更能够显示出它的作用。

6.4 嵌入式 Linux 串口应用开发

6.4.1 串口概述

用户常见的数据通信的基本方式可分为并行通信与串行通信两种。

- 并行通信是指利用多条数据传输线将一个资料的各位同时传送。它的特点是传输速度快，适用于短距离通信，但要求传输速度较高的应用场合。
- 串行通信是指利用一条传输线将资料一位位地顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信，降低成本，适用于远距离通信，但传输速度慢的应用场合。

串口是计算机一种常用的接口，常用的串口有 RS-232-C 接口。它是于 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通讯的标准，它的全称是“数据终端设备（DTE）和数据通讯设备（DCE）之间串行二进制数据交换接口技术标准”。该标准规定采用一个 DB25 芯引脚的连接器或 9 芯引脚的连接器，其中 25 芯引脚的连接器如图 6.3 所示。

S3C2410X 内部具有 2 个独立的 UART 控制器，每个控制器都可以工作在 Interrupt（中断）模式或者 DMA（直接内存访问）模式。同时，每个 UART 均具有 16 字节的 FIFO（先入先出寄存器），支持的最高波特率可达到 230.4Kbps。UART 的操作主要可分为以下几个部分：资料发送、资料接收、产生中断、产生波特率、Loopback 模式、红外模式以及自动流控模式。

串口参数的配置读者在配置超级终端和 minicom 时也已经接触到过，一般包括波特率、起始位数量、数据位数量、停止位数量和流控协议。在此，可以将其配置为波特率 115200、起始位 1b、数据位 8b、停止位 1b 和无流控协议。

在 Linux 中，所有的设备文件一般都位于“/dev”下，其中串口一、串口二对应的设备名依次为“/dev/ttyS0”、“/dev/ttyS1”，可以查看在“/dev”下的文件以确认。在本章中已经提到过，在 Linux 下对设备的操作方法与对文件的操作方法是一样的，因此，对串口的读写就可以使用简单的“read”，“write”函数来完成，所不同的是只是需要对串口的其他参数另做配置，下面就来详细讲解串口应用开发的步骤。

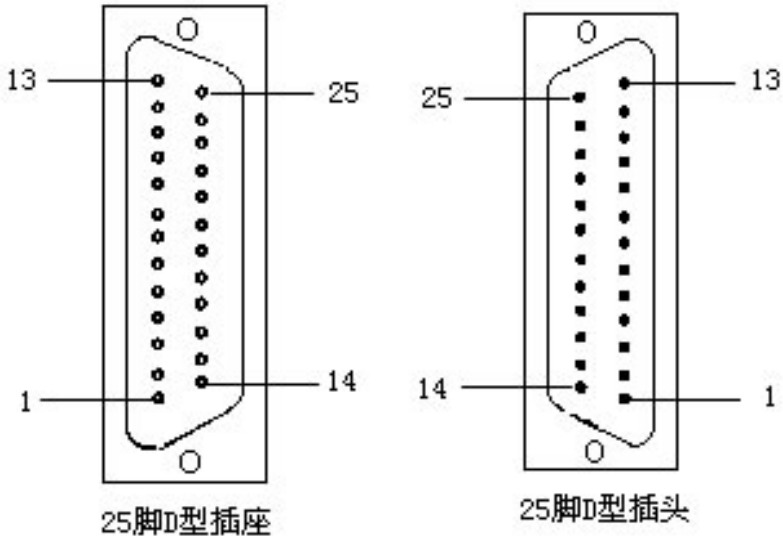


图 6.3　25 引脚串行接口图

6.4.2　串口设置详解

本节主要讲解设置串口的主要方法。

如前所述，设置串口中最基本的包括波特率设置，校验位和停止位设置。串口的设置主要是设置 struct termios 结构体的各成员值，如下所示：

```
#include<termios.h>
struct termio
{      unsigned short  c_iflag; /* 输入模式标志 */
       unsigned short  c_oflag;     /* 输出模式标志 */
       unsigned short  c_cflag;     /* 控制模式标志*/
       unsigned short  c_lflag;     /*本地模式标志 */
       unsigned char  c_line;       /* line discipline */
       unsigned char  c_cc[NCC];    /* control characters */
};
```

在这个结构中最为重要的是 c_cflag，通过对它的赋值，用户可以设置波特率、字符大小、数据位、停止位、奇偶校验位和硬件流控等。另外 c_iflag 和 c_cc 也是比较常用的标志。在此主要对这 3 个成员进行详细说明。

c_cflag 支持的常量名称如表 6.10 所示。其中设置波特率为相应的波特率前加上‘B’，由于数值较多，本表没有全部列出。

表 6.10　　c_cflag 支持的常量名称

CBAUD	波特率的位掩码
B0	0 波特率（放弃 DTR）

…	…

续表

B1800	1800 波特率
B2400	2400 波特率
B4800	4800 波特率
B9600	9600 波特率
B19200	19200 波特率
B38400	38400 波特率
B57600	57600 波特率
B115200	115200 波特率
EXTA	外部时钟率
EXTB	外部时钟率
CSIZE	数据位的位掩码
CS5	5 个数据位
CS6	6 个数据位
CS7	7 个数据位
CS8	8 个数据位
CSTOPB	2 个停止位（不设则是 1 个停止位）
CREAD	接收使能
PARENB PARODD	校验位使能 使用奇校验而不使用偶校验
HUPCL	最后关闭时挂线（放弃 DTR）
CLOCAL	本地连接（不改变端口所有者）
LOBLK	块作业控制输出
CNET_CTSRTS	硬件流控制使能

在这里，对于 c_cflag 成员不能直接对其初始化，而要将其通过“与”、“或”操作使用其中的某些选项。

输入模式 c_iflag 成员控制端口接收端的字符输入处理。c_iflag 支持的变量名称，如表 6.11 所示。

表 6.11　　c_iflag 支持的常量名称

INPCK	奇偶校验使能
IGNPAR	忽略奇偶校验错误
PARMRK	奇偶校验错误掩码

ISTRIP	除去奇偶校验位

续表

IXON	启动出口硬件流控
IXOFF	启动入口软件流控
IXANY	允许字符重新启动流控
IGNBRK	忽略中断情况
BRKINT	当发生中断时发送 SIGINT 信号
INLCR	将 NL 映射到 CR
IGNCR	忽略 CR
ICRNL	将 CR 映射到 NL
IUCLC	将高位情况映射到低位情况
IMAXBEL	当输入太长时回复 ECHO

c_cc 包含了超时参数和控制字符的定义。c_cc 所支持的常用变量名称，如表 6.12 所示。

表 6.12 c_cc 支持的常量名称

VINTR	中断控制，对应键为 CTRL+C
VQUIT	退出操作，对应键为 CRTL+Z
VERASE	删除操作，对应键为 Backspace（BS）
VKILL	删除行，对应键为 CTRL+U
VEOF	位于文件结尾，对应键为 CTRL+D
VEOL	位于行尾，对应键为 Carriage return（CR）
VEOL2	位于第二行尾，对应键为 Line feed（LF）
VMIN	指定了最少读取的字符数
VTIME	指定了读取每个字符的等待时间

下面就详细讲解设置串口属性的基本流程。

1．保存原先串口配置

首先，为了安全起见和以后调试程序方便，可以先保存原先串口的配置，在这里可以使用函数 tcgetattr（fd，&oldtio）。该函数得到与 fd 指向对象的相关参数，并将它们保存于 oldtio 引用的 termios 结构中。该函数还可以测试配置是否正确、该串口是否可用等。若调用成功，函数返回值为 0，若调用失败，函数返回值为–1，其使用如下所示：

```
if  ( tcgetattr( fd,&oldtio)  !=  0) {
      perror("SetupSerial 1");
```

```
        return -1;
    }
```

2．激活选项有 CLOCAL 和 CREAD

CLOCAL 和 CREAD 分别用于本地连接和接受使能，因此，首先要通过位掩码的方式激活这两个选项。

```
    newtio.c_cflag  |=  CLOCAL | CREAD;
```

3．设置波特率

设置波特率有专门的函数，用户不能直接通过位掩码来操作。设置波特率的主要函数有：cfsetispeed 和 cfsetospeed。这两个函数的使用很简单，如下所示：

```
    cfsetispeed(&newtio, B115200);
    cfsetospeed(&newtio, B115200);
```

一般地，用户需将输入输出函数的波特率设置成一样的。这几个函数在成功时返回 0，失败时返回–1。

4．设置字符大小

与设置波特率不同，设置字符大小并没有现成可用的函数，需要用位掩码。一般首先去除数据位中的位掩码，再重新按要求设置。如下所示：

```
    options.c_cflag &= ~CSIZE;/* mask the character size bits */
    options.c_cflag |= CS8;
```

5．设置奇偶校验位

设置奇偶校验位需要用到两个 termio 中的成员：c_cflag 和 c_iflag。首先要激活 c_cflag 中的校验位使能标志 PARENB 和是否要进行偶校验，同时还要激活 c_iflag 中的奇偶校验使能。如使能奇校验时，代码如下所示：

```
    newtio.c_cflag |= PARENB;
    newtio.c_cflag |= PARODD;
    newtio.c_iflag |= (INPCK | ISTRIP);
```

而使能偶校验时代码为：

```
    newtio.c_iflag |= (INPCK | ISTRIP);
    newtio.c_cflag |= PARENB;
    newtio.c_cflag &= ~PARODD;
```

6．设置停止位

设置停止位是通过激活 c_cflag 中的 CSTOPB 而实现的。若停止位为 1，则清除 CSTOPB，若停止位为 0，则激活 CSTOPB。下面是停止位是 1 时的代码：

```
newtio.c_cflag &=  ~CSTOPB;
```

7．设置最少字符和等待时间

在对接收字符和等待时间没有特别要求的情况下，可以将其设置为 0，如下所示：

```
newtio.c_cc[VTIME]  = 0;
newtio.c_cc[VMIN] = 0;
```

8．处理要写入的引用对象

由于串口在重新设置之后，在此之前要写入的引用对象要重新处理，这时就可调用函数 tcflush（fd，queue_selector）来处理要写入引用的对象。对于尚未传输的数据，或者收到的但是尚未读取的数据，其处理方法取决于 queue_selector 的值。

这里，queue_selector 可能的取值有以下几种。

- TCIFLUSH：刷新收到的数据但是不读。
- TCOFLUSH：刷新写入的数据但是不传送。
- TCIOFLUSH：同时刷新收到的数据但是不读，并且刷新写入的数据但是不传送。

如在本例中所采用的是第一种方法：

```
tcflush(fd,TCIFLUSH);
```

9．激活配置

在完成全部串口配置之后，要激活刚才的配置并使配置生效。这里用到的函数是 tcsetattr，它的函数原型是：

tcsetattr(fd,OPTION,&newtio);

这里的 newtio 就是 termios 类型的变量，OPTION 可能的取值有以下三种：

- TCSANOW：改变的配置立即生效。
- TCSADRAIN：改变的配置在所有写入 fd 的输出都结束后生效。
- TCSAFLUSH：改变的配置在所有写入 fd 引用对象的输出都被结束后生效，所有已接受但未读入的输入都在改变发生前丢弃。

该函数若调用成功则返回 0，若失败则返回–1。

如下所示：

```
if((tcsetattr(fd,TCSANOW,&newtio))!=0)
{
      perror("com set error");
```

```
        return -1;
    }
```

下面给出了串口配置的完整的函数。通常，为了函数的通用性，通常将常用的选项都在函数中列出，这样可以大大方便以后用户的调试使用。该设置函数如下所示：

```
int set_opt(int fd,int nSpeed, int nBits, char nEvent, int nStop)
{
    struct termios newtio,oldtio;
/*保存测试现有串口参数设置，在这里如果串口号等出错，会有相关的出错信息*/
    if  ( tcgetattr( fd,&oldtio)  !=  0) {
        perror("SetupSerial 1");
        return -1;
    }
    bzero( &newtio, sizeof( newtio ) );
/*步骤一，设置字符大小*/
    newtio.c_cflag  |=  CLOCAL | CREAD;
    newtio.c_cflag &= ~CSIZE;
/*设置停止位*/
    switch( nBits )
    {
    case 7:
        newtio.c_cflag |= CS7;
        break;
    case 8:
        newtio.c_cflag |= CS8;
        break;
    }
/*设置奇偶校验位*/
    switch( nEvent )
    {
    case 'O': //奇数
        newtio.c_cflag |= PARENB;
        newtio.c_cflag |= PARODD;
        newtio.c_iflag |= (INPCK | ISTRIP);
        break;
    case 'E': //偶数
        newtio.c_iflag |= (INPCK | ISTRIP);
        newtio.c_cflag |= PARENB;
        newtio.c_cflag &= ~PARODD;
```

```
                break;
        case 'N':  //无奇偶校验位
                newtio.c_cflag &= ~PARENB;
                break;
        }
    /*设置波特率*/
    switch( nSpeed )
        {
        case 2400:
                cfsetispeed(&newtio, B2400);
                cfsetospeed(&newtio, B2400);
                break;
        case 4800:
                cfsetispeed(&newtio, B4800);
                cfsetospeed(&newtio, B4800);
                break;
        case 9600:
                cfsetispeed(&newtio, B9600);
                cfsetospeed(&newtio, B9600);
                break;
        case 115200:
                cfsetispeed(&newtio, B115200);
                cfsetospeed(&newtio, B115200);
                break;
        case 460800:
                cfsetispeed(&newtio, B460800);
                cfsetospeed(&newtio, B460800);
                break;
        default:
                cfsetispeed(&newtio, B9600);
                cfsetospeed(&newtio, B9600);
               break;
        }
    /*设置停止位*/
        if( nStop == 1 )
                newtio.c_cflag &=  ~CSTOPB;
        else if ( nStop == 2 )
                newtio.c_cflag |=  CSTOPB;
    /*设置等待时间和最小接收字符*/
```

```
        newtio.c_cc[VTIME]  = 0;
        newtio.c_cc[VMIN] = 0;
    /*处理未接收字符*/
        tcflush(fd,TCIFLUSH);
    /*激活新配置*/
        if((tcsetattr(fd,TCSANOW,&newtio))!=0)
        {
               perror("com set error");
               return -1;
        }
        printf("set done!\n");
        return 0;
    }
```

6.4.3 串口使用详解

在配置完串口的相关属性后，就可以对串口进行打开、读写操作了。它所使用的函数和普通文件读写的函数一样，都是 open、write 和 read。它们相区别的只是串口是一个终端设备，因此在函数的具体参数的选择时会有一些区别。另外，这里会用到一些附加的函数，用于测试终端设备的连接情况等。下面将对其进行具体讲解。

1．打开串口

打开串口和打开普通文件一样，使用的函数同打开普通文件一样，都是 open 函数。如下所示：

```
    fd = open( "/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
```

可以看到，这里除了普通的读写参数外，还有两个参数 O_NOCTTY 和 O_NDELAY。

- O_NOCTTY 标志用于通知 Linux 系统，这个程序不会成为对应这个端口的控制终端。如果没有指定这个标志，那么任何一个输入（诸如键盘中止信号等）都将会影响用户的进程。
- O_NDELAY 标志通知 Linux 系统，这个程序不关心 DCD 信号线所处的状态（端口的另一端是否激活或者停止）。如果用户指定了这个标志，则进程将会一直处在睡眠状态，直到 DCD 信号线被激活。

接下来可恢复串口的状态为阻塞状态，用于等待串口数据的读入。可用 fcntl 函数实现，如下所示：

```
    fcntl(fd, F_SETFL, 0);
```

再接着可以测试打开文件描述符是否引用一个终端设备，以进一步确认串口是否正确打开，如下所示：

```
    isatty(STDIN_FILENO);
```

该函数调用成功则返回 0，若失败则返回-1。

这时，一个串口就已经成功打开了。接下来就可以对这个串口进行读、写操作。

下面给出了一个完整的打开串口的函数，同样写考虑到了各种不同的情况。程序如下所示：

```
/*打开串口函数*/
int open_port(int fd,int comport)
{
	char *dev[]={"/dev/ttyS0","/dev/ttyS1","/dev/ttyS2"};
	long  vdisable;
	if (comport==1)//串口 1
	{    fd = open( "/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
		if (-1 == fd){
			perror("Can't Open Serial Port");
			return(-1);
		}
	}
	else if(comport==2)//串口 2
	{    fd = open( "/dev/ttyS1", O_RDWR|O_NOCTTY|O_NDELAY);
		if (-1 == fd){
			perror("Can't Open Serial Port");
			return(-1);
		}
	}
	else if (comport==3)//串口 3
	{
		fd = open( "/dev/ttyS2", O_RDWR|O_NOCTTY|O_NDELAY);
		if (-1 == fd){
			perror("Can't Open Serial Port");
			return(-1);
		}
	}
/*恢复串口为阻塞状态*/
	if(fcntl(fd, F_SETFL, 0)<0)
		printf("fcntl failed!\n");
	else
		printf("fcntl=%d\n",fcntl(fd, F_SETFL,0));
/*测试是否为终端设备*/
	if(isatty(STDIN_FILENO)==0)
```

```
            printf("standard input is not a terminal device\n");
        else
            printf("isatty success!\n");
        printf("fd-open=%d\n",fd);
        return fd;
}
```

2．读写串口

读写串口操作和读写普通文件一样，使用 read、write 函数即可。如下所示：

```
write(fd,buff,8);
read(fd,buff,8);
```

下面两个实例给出了串口读和写的两个程序的 main 函数部分，这里用到的函数有前面讲述到的 open_port 和 set_opt 函数。

```
/*写串口程序*/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <stdlib.h>

/*读串口程序*/
int main(void)
{
    int fd;
    int nread,i;
    char buff[]="Hello\n";

    if((fd=open_port(fd,1))<0){//打开串口
        perror("open_port error");
        return;
    }
    if((i=set_opt(fd,115200,8,'N',1))<0){//设置串口
        perror("set_opt error");
        return;
```

```
        }
        printf("fd=%d\n",fd);
        fd=3;
        nread=read(fd,buff,8);//读串口
        printf("nread=%d,%s\n",nread,buff);
        close(fd);
        return;
    }
```

读者可以将该程序在宿主机上运行，然后用串口线将目标板和宿主机连接起来，之后将目标板上电，这样就可以看到宿主机上有目标板的串口输出。

```
 [root@localhost file]# ./receive
fcntl=0
isatty success!
fd-open=3
set done
fd=3
nread=8,…
```

另外，读者还可以考虑一下如何使用 select 函数实现串口的非阻塞读写，具体实例会在后面的实验中给出。

6.5 标准 I/O 开发

本章前面几节所述的文件及 I/O 读写都是基于文件描述符的。这些都是基本的 I/O 控制，是不带缓存的。而本节所要讨论的 I/O 操作都是基于流缓冲的，它是符合 ANSI C 的标准 I/O 处理，这里有很多函数读者已经非常熟悉了（如 printf、scantf 函数等），因此本节中仅简要介绍最主要的函数。

标准 I/O 提供流缓冲的目的是尽可能减少使用 read 和 write 调用的数量。标准 I/O 提供了 3 种类型的缓冲存储。

- 全缓冲。在这种情况下，当填满标准 I/O 缓存后才进行实际 I/O 操作。对于驻在磁盘上的文件通常是由标准 I/O 库实施全缓冲的。在一个流上执行第一次 I/O 操作时，通常调用 malloc 就是使用全缓冲。
- 行缓冲。在这种情况下，当在输入和输出中遇到新行符时，标准 I/O 库执行 I/O 操作。这允许我们一次输出一个字符（如 fputc 函数），但只有写了一行之后才进行实际 I/O 操作。当流涉及一个终端时（例如标准输入和标准输出），典型地使用行缓冲。
- 不带缓冲。标准 I/O 库不对字符进行缓冲。如果用标准 I/O 函数写若干字符到不带缓冲的流中，则相当于用 write 系统的用函数将这些字符写全相比较的打开文件上。标准出错况 stderr 通常是不带缓存后，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个新行字符。

在下面讨论具体函数时，请读者注意区分这 3 种不同的情况。

6.5.1 打开和关闭文件

1．打开文件

（1）函数说明

打开文件有三个标准函数，分别为：fopen、fdopen 和 freopen。它们可以以不同的模式打开，但都返回一个指向 FILE 的指针，该指针以将对应的 I/O 流相绑定了。此后，对文件的读写都是通过这个 FILE 指针来进行。其中 fopen 可以指定打开文件的路径和模式，fdopen 可以指定打开的文件描述符和模式，而 freopen 除可指定打开的文件、模式外，还可指定特定的 IO 流。

（2）函数格式定义

fopen 函数格式如表 6.13 所示。

表 6.13 fopen 函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * fopen(const char * path,const char * mode)
函数传入值	path：包含要打开的文件路径及文件名
	mode：文件打开状态（后面会具体说明）
函数返回值	成功：指向 FILE 的指针 失败：NULL

这里的 mode 类似于 open 中的 flag，可以定义打开文件的具体权限等，表 6.14 说明了 fopen 中 mode 的各种取值。

表 6.14 mode 取值说明

r 或 rb	打开只读文件，该文件必须存在
r＋或 r＋b	打开可读写的文件，该文件必须存在
w 或 wb	打开只写文件，若文件存在则文件长度清为 0，即会擦些文件以前内容。若文件不存在则建立该文件
w+或 w＋b	打开可读写文件，若文件存在则文件长度清为 0，即会擦些文件以前内容。若文件不存在则建立该文件
a 或 ab	以附加的方式打开只写文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留
a+或 a＋b	以附加方式打开可读写的文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留

注意在每个选项中加入 b 字符用来告诉函数库打开的文件为二进制文件，而非纯文字文件。不过在 Linux 系统中会自动识别不同类型的文件而将此符号忽略。

fdopen 函数格式如表 6.15 所示。

表 6.15 fdopen 函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * fdopen(int fd,const char * mode)
函数传入值	fd：要打开的文件描述符
	mode：文件打开状态（后面会具体说明）
函数返回值	成功：指向 FILE 的指针 失败：NULL

freopen 函数格式如表 6.16 所示。

表 6.16 freopen 函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * freopen(const char *path,const char * mode,FILE * stream)
函数传入值	path：包含要打开的文件路径及文件名
	mode：文件打开状态（后面会具体说明）
	stream：已打开的文件指针
函数返回值	成功：指向 FILE 的指针 失败：NULL

2．关闭文件

（1）函数说明

关闭标准流文件的函数为 fclose，这时缓冲区内的数据写入文件中，并释放系统所提供的文件资源。

（2）函数格式说明

freopen 函数格式如表 6.17 所示。

表 6.17 fclose 函数语法要点

所需头文件	#include <stdio.h>
函数原型	int fclose(FILE * stream)
函数传入值	stream：已打开的文件指针
函数返回值	成功：0 失败：EOF

3．使用实例

文件打开关闭的操作都比较简单，这里仅以 fopen 和 fclose 为例，代码如下所示：

```
/*fopen.c*/
#include <stdio.h>
main()
{
    FILE *fp;
```

```
    int c;
/*调用 fopen 函数*/
    if((fp=fopen("exist","w"))!=NULL){
        printf("open success!");
    }
    fclose(fp);
}
```

读者可以尝试用其他文件打开函数进行练习。

6.5.2 文件读写

1．读文件

（1）fread 函数说明

在文件流打开之后，可对文件流进行读写等操作，其中读操作的函数为 fread。

（2）fread 函数格式

fread 函数格式如表 6.18 所示。

表 6.18 fread 函数语法要点

所需头文件	#include <stdio.h>
函数原型	size_t fread(void * ptr,size_t size,size_t nmemb,FILE * stream)
函数传入值	ptr：存放读入记录的缓冲区
	size：读取的记录大小
	nmemb：读取的记录数
	stream：要读取的文件流
函数返回值	成功：返回实际读取到的 nmemb 数目 失败：EOF

2．写文件

（1）fwrite 函数说明

fwrite 函数是用于对指定的文件流进行写操作。

（2）fwrite 函数格式

fwrite 函数格式如表 6.19 所示。

表 6.19 fwrite 函数语法要点

所需头文件	#include <stdio.h>
函数原型	size_t fwrite(const void * ptr,size_t size,size_t nmemb,FILE * stream)
函数传入值	ptr：存放写入记录的缓冲区

	size：写入的记录大小
	nmemb：写入的记录数
	stream：要写入的文件流
函数返回值	成功：返回实际写入到的 nmemb 数目 失败：EOF

这里仅以 fwrite 为例简单说明：

```
/*fwrite.c*/
#include <stdio.h>
int main()
{
     FILE *stream;
     char s[3]={'a','b','c'};
/*首先使用 fopen 打开文件，之后再调用 fwrite 写入文件*/
     stream=fopen("what","w");
     i=fwrite(s,sizeof(char),nmemb,stream);
     printf("i=%d",i);
     fclose(stream);
}
```

运行结果如下所示：

```
[root@localhost file]# ./write
i=3
[root@localhost file]# cat what
abc
```

6.5.3 输入输出

文件打开之后，根据一次读写文件中字符的数目可分为字符输入输出、行输入输出和格式化输入输出，下面分别对这 3 种不同的方式进行讨论。

1．字符输入输出

字符输入输出函数一次仅读写一个字符。其中字符输入输出函数如表 6.20 和表 6.21 所示。

表 6.20 字符输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	int getc(FILE * stream) int fgetc(FILE * stream) int getchar(void)
函数传入值	stream：要输入的文件流

函数返回值	成功：下一个字符 失败：EOF

表 6.21 字符输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	int putc(int c,FILE * stream) int fputc(int c,FILE * stream) int putchar(int c)
函数返回值	成功：字符 c 失败：EOF

这几个函数功能类似，其区别仅在于 getc 和 putc 通常被实现为宏，而 fgetc 和 fputc 不能实现为宏，因此，函数的实现时间会有所差别。

下面这个实例结合 fputc 和 fgetc，将标准输入复制到标准输出中去。

```
/*fput.c*/
#include<stdio.h>
main()
{
    int c;
/*把 fgetc 的结果作为 fputc 的输入*/
    fputc(fgetc(stdin),stdout);
}
```

运行结果如下所示：

```
[root@localhost file]# ./file
w（用户输入）
w（屏幕输出）
```

2．行输入输出

行输入输出函数一次操作一行。其中行输入输出函数如表 6.22 和表 6.23 所示。

表 6.22 行输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	char * gets(char *s) char fgets(char * s,int size,FILE * stream)
函数传入值	s：要输入的字符串 size：输入的字符串长度 stream：对应的文件流
函数返回值	成功：s

	失败：NULL

表 6.23 行输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	int puts(const char *s) int fputs(const char * s,FILE * stream)
函数传入值	s：要输出的字符串 stream：对应的文件流
函数返回值	成功：s 失败：NULL

这里以 gets 和 puts 为例进行说明，本实例将标准输入复制到标准输出，如下所示：

```
/*gets.c*/
#include<stdio.h>
main()
{
    char s[80];
/*同上例，把 fgets 的结果作为 fputs 的输入*/
    fputs(fgets(s,80,stdin),stdout);
}
```

运行该程序，结果如下所示：

```
[root@www yul]# ./file2
This is stdin（用户输入）
This is stdin（屏幕输出）
```

3．格式化输入输出

格式化输入输出函数可以指定输入输出的具体格式，这里有读者已经非常熟悉的 printf、scanf 等函数，这里就简要介绍一下它们的格式。如下表 6.24～表 6.26 所示。

表 6.24 格式化输出函数 1

所需头文件	#include <stdio.h>
函数原型	int printf(const char *format,…) int fprintf(FILE *fp,const char *format,…) int sprintf(char *buf,const char *format,…)
函数传入值	format：记录输出格式 fp：文件描述符 buf：记录输出缓冲区

函数返回值	成功：输出字符数（sprintf 返回存入数组中的字符数） 失败：NULL

表 6.25　　格式化输出函数 2

所需头文件	#include <stdarg.h> #include <stdio.h>
函数原型	int vprintf(const char *format,va_list arg) int vfprintf(FILE *fp,const char *format,va_list arg) int vsprintf(char *buf,const char *format,va_list arg)
函数传入值	format：记录输出格式 fp：文件描述符 arg：相关命令参数
函数返回值	成功：存入数组的字符数 失败：NULL

表 6.26　格式化输入函数

所需头文件	#include <stdio.h>
函数原型	int scanf(const char *format,…) int fscanf(FILE *fp,const char *format,…) int sscanf(char *buf,const char *format,…)
函数传入值	format：记录输出格式 fp：文件描述符 buf：记录输入缓冲区
函数返回值	成功：输出字符数（sprintf 返回存入数组中的字符数） 失败：NULL

由于本节的函数用法比较简单，并且比较常用，因此就不再举例了，请读者需要用到时自行查找其用法。

6.6　实验内容

6.6.1　文件读写及上锁

1．实验目的

通过编写文件读写及上锁的程序，进一步熟悉 Linux 中文件 I/O 相关的应用开发，并且熟练掌握 open、read、write、fcntl 等函数的使用。

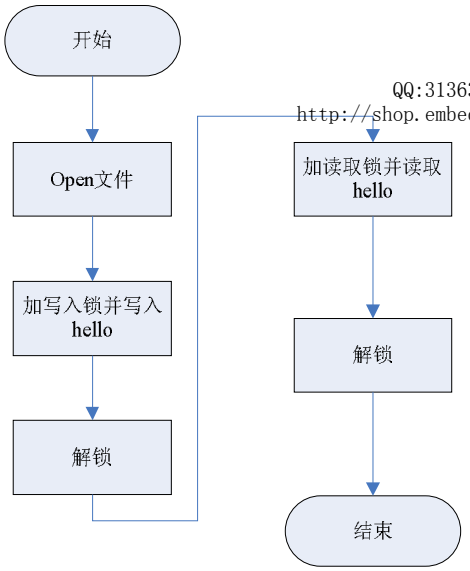
2．实验内容

该实验要求首先打开一个文件，然后将该文件上写入锁，并写入 hello 字符串。接着在解锁后再将该文件上读取锁，并读取刚才写入的内容。最后模拟多进程，同时读写一个文件时的情况。

3．实验步骤

（1）画出实验流程图

该实验流程图如图 6.4 所示。



6.4　实验 6.6.1 节流程图

（2）编写代码

该实验源代码如下所示，其中用到的 lock_set 函数可参见第 6.3.3 节。

```
/*expr1.c 实验一源码*/
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void lock_set(int fd,int type);

int main(void)
{
     int fd,nwrite,nread,len;
     char *buff="Hello\n";
     char buf_r[100];
     len=strlen(buff);
     fd=open("hello",O_RDWR | O_CREAT, 0666);
     if(fd < 0){
          perror("open");
          exit(1);
     }
/*加上写入锁*/
     lock_set(fd, F_WRLCK);
     if((nwrite=write(fd,buff,len))==len){
          printf("write success\n");
     }
     getchar();
/*解锁*/
     lock_set(fd, F_UNLCK);
     getchar();
/*加上读取锁*/
```

```
        lock_set(fd, F_RDLCK);
        lseek(fd,0,SEEK_SET);
        if((nread=read(fd,buf_r,len))==len){
              printf("read:%s\n",buf_r);
        }
        getchar();
    /*解锁*/
        lock_set(fd, F_UNLCK);
        getchar();
        close(fd);
        exit(0);
    }
```

（3）首先在宿主机上编译调试该程序，如下所示：

```
[root@localhost process]# gcc expr1.c –o expr1
```

（4）在确保没有编译错误后，使用交叉编译该程序，如下所示：

```
[root@localhost process]# arm-linux-gcc expr1.c –o expr2
```

（5）将生成的可执行程序下载到目标板上运行。

4．实验结果

此实验在目标板上的运行结果如下所示：

```
[root@(none) 1]# ./expr1
write lock set by 75
write success

release lock by 75

read lock set by 75
read:Hello


release lock by 75
```

另外，在本机上可以开启两个终端，同时运行该程序。实验结果会和这两个进程运行过程具体相关，希望读者能具体分析每种情况。下面列出其中一种情况：

终端一：

```
[root@localhost file]# ./expr1
write lock set by 3013
```

```
write success

release lock by 3013

read lock set by 3013
read:Hello


release lock by 3013
```

终端二：

```
[root@localhost file]# ./expr1
write lock already set by 3013

write lock set by 3014
write success

release lock by 3014

read lock set by 3014
read:Hello



release lock by 3014
```

6.6.2 多路复用式串口读写

1．实验目的

通过编写多路复用式串口读写，进一步理解 select 函数的作用，同时更加熟练掌握 Linux 设备文件的读写方法。

2．实验内容

完成串口读写操作，这里设定从串口读取消息时使用 select 函数，发送消息的程序不需要用 select 函数，只发送“Hello”消息由接收端接收。

3．实验步骤

（1）画出流程图

下图 6.5 是读串口的流程图，写串口的流程图与此类似。

（2）编写代码

分别编写串口读写程序，该程序中用到的 open_port 和 set_opt 函数请参照 6.4 节所述。

写串口程序的代码如下所示：

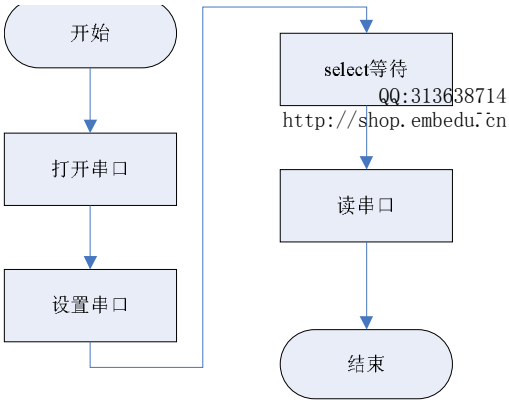


图 6.5　实验 6.6.2 节流程图

```
/*写串口*/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <stdlib.h>
int main(void)
{
     int fd;
     int nwrite,i;
     char buff[]="Hello\n";
/*打开串口*/
     if((fd=open_port(fd,1))<0){
           perror("open_port error");
           return;
     }
/*设置串口*/
     if((i=set_opt(fd,115200,8,'N',1))<0){
           perror("set_opt error");
           return;
     }
     printf("fd=%d\n",fd);
/*向串口写入字符串*/
     nwrite=write(fd,buff,8);
     printf("nwrite=%d\n",nwrite);
     close(fd);
     return;
}
```

读串口程序的代码如下所示：

```
/*读串口*/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <stdlib.h>
int main(void)
{
     int fd;
     int nread,nwrite,i;
     char buff[8];
     fd_set rd;
/*打开串口*/
     if((fd=open_port(fd,1))<0){
           perror("open_port error");
           return;
     }
/*设置串口*/
     if((i=set_opt(fd,115200,8,'N',1))<0){
           perror("set_opt error");
           return;
     }
/*利用 select 函数来实现多个串口的读写*/
     FD_ZERO(&rd);
     FD_SET(fd,&rd);
     while(FD_ISSET(fd,&rd)){
     if(select(fd+1,&rd,NULL,NULL,NULL)<0)
                perror("select");
     else{
           while((nread = read(fd, buff, 8))>0)
           {
                printf("nread=%d,%s\n",nread,buff);
           }
     }
     close(fd);
```

```
        return;
    }
```

（3）接下来把第一个写串口程序交叉编译，再把第二个读串口程序在 PC 机上编译，分别得到可执行文件 write 和 read。

（4）将写串口程序下载到开发板上，然后连接 PC 和开发板的串口 1。首先运行读串口程序，再运行写串口程序。

4．实验结果

发送端的运行结果如下所示：

```
[root@(none) 1]# ./write
fcntl=0
isatty success!
fd-open=3
set done
fd=3
nwrite=8
```

接收端的运行结果如下所示：

```
[root@localhost file]# ./read
fcntl=0
isatty success!
fd-open=3
set done
fd=3
nread=8,Hello!
```

读者还可以尝试修改 select 函数选项，例如设定一个等待时间，查看程序的运行结果。

本 章 小 结

本章首先讲解了系统调用、用户函数接口（API）和系统命令之间的关系和区别，这也是贯穿本书的一条主线，本书的讲解就是从系统命令、用户函数接口（API）到系统调用为顺序一层层深入进行讲解的，希望读者能有一个较为深刻的认识。

接着，本章主要讲解了嵌入式 Linux 中文件 I/O 相关的开发，在这里主要讲解了不带缓存 I/O 函数的使用，这也是本章的重点。因为不带缓存 I/O 函数的使用范围非常广泛，在有很多情况下必须使用它，这也是学习嵌入式 Linux 开发的基础，因此读者一定要牢牢掌握相关知识。其中主要讲解了 open、close、read、write、lseek、fcntl 和 select 等函数，这几个函数包括了不带缓存 I/O 处理的主要部分，并且也体现了它的主要思想。

接下来，本章讲解了嵌入式 Linux 串口编程。这其实是 Linux 中设备文件读写的实例，由于它能很好地体现前面所介绍的内容，而且在嵌入式开发中也较为常见，因此对它进行了比较详细地讲解。

之后，本章简单介绍了标准 I/O 的相关函数，希望读者也能对它有一个总体的认识。

最后，本章安排了两个实验，分别是文件使用及上锁和多用复用串口读写。希望读者能够认真完成。

思考与练习

使用 select 函数实现 3 个串口的通信：串口 1 接收数据，串口 2 和串口 3 向串口 1 发送数据。