华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

嵌入式应用程序设计综合教程

作者：华清远见

专业始于专注 卓识源于远见

华清远见

第 2 章 Linux 文件 I/O 编程

本章目标

在 Linux 系统中，大部分机制都会抽象成一个文件，这样对它们的操作就像对文件的操作一样。在嵌入式应用开发中，文件 I/O 编程是最常用的也是最基本的内容，希望读者好好掌握。

本章主要内容：

- Linux 文件 I/O 概述；
- 文件 I/O 操作。

2.1 Linux 文件 I/O 概述

2.1.1 POSIX 规范

POSIX （Portable Operating System Interface）表示可移植操作系统接口规范。该标准最初由 IEEE 制定，目的是为了提高 UNIX 环境下应用程序的可移植性。POSIX 已发展成一个庞大的标准族，其中的 POSIX.1 提供了源代码级别的 C 语言应用程序编程接口。通俗地讲，为一个 POSIX 兼容的操作系统编写的程序，可以在任何其他 POSIX 操作系统上编译执行。不仅仅是 UNIX，很多操作系统如 Linux 也支持 POSIX 标准。

2.1.1 虚拟文件系统

Linux 系统成功的关键因素之一就是具有与其他操作系统和谐共存的能力。Linux 的文件系统由两层结构构建：第一层是虚拟文件系统（VFS），第二层是各种不同的具体的文件系统。

VFS 就是把各种具体的文件系统的公共部分抽取出来，形成一个抽象层，是系统内核的一部分。它位于用户程序和具体的文件系统之间。它对用户程序提供了标准的文件系统调用接口，对具体的文件系统（比如：Ext2、FAT32 等），它通过一系列的对不同文件系统通用的函数指针来调用对应的文件系统函数，完成相应的操作。任何使用文件系统的程序必须通过这层接口来访问。通过这样的方式，VFS 就对用户屏蔽了底层文件系统的实现细节和差异。

VFS 不仅可以对具体文件系统的数据结构进行抽象，以一种统一的数据接口进行管理，并且还可以接受用户层的系统调用，如 open()、read()、write()、stat()、link()等。此外，它还支持多种具体文件系统之间的相互访问，接受内核其他子系统的操作请求，例如，内存管理和进程调度。VFS 在 Linux 系统中的位置如图 2.1 所示。

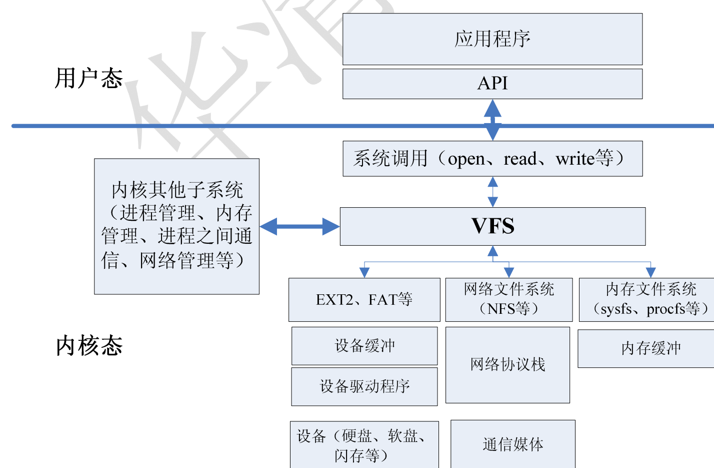


图 2.1 VFS 在 Linux 系统中的位置

通过以下命令可以查看系统中支持哪些文件系统。

```
$ cat /proc/filesystems
nodev   sysfs
nodev   rootfs
……
nodev   tmpfs
nodev   pipefs
```

```
……
        ext2
nodev   ramfs
nodev   hugetlbfs
        iso9660
nodev   mqueue
nodev   seLinuxfs
        ext3
nodev   rpc_pipefs
……
```

2.1.2 文件和文件描述符

Linux 操作系统是基于文件概念的。文件是以字符序列构成的信息载体。根据这一点，可以把 I/O 设备当作文件来处理。因此，与磁盘上的普通文件进行交互所用的同一系统调用可以直接用于 I/O 设备。这样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 6 种：普通文件、目录文件、符号链接文件、管道文件、套接字文件和设备文件。

那么，内核如何区分和引用特定的文件呢？这里用到了一个重要的概念——文件描述符。对于 Linux 而言，所有对设备和文件的操作都是通过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，并指向在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；读写文件时，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 个流：标准输入、标准输出和标准错误。这 3 个流分别对应文件描述符 0、1 和 2（对应的宏分别是 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO）。

基于文件描述符的 I/O 操作虽然不能直接移植到类 Linux 以外的系统上去（如 Windows），但它往往是实现某些 I/O 操作的唯一途径，如 Linux 中低层文件操作函数、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。

2.1.3 文件 I/O 和标准 I/O 的区别

读者可能会思考，标准 I/O 和文件 I/O 都可以用来访问文件，它们之间有什么区别呢？

文件 I/O 又称为低级磁盘 I/O，遵循 POSIX 相关标准。任何兼容 POSIX 标准的操作系统上都支持文件 I/O。标准 I/O 被称为高级磁盘 I/O，遵循 ANSI C 相关标准。只要开发环境中有标准 C 库，标准 I/O 就可以使用。（Linux 中使用的是 GLIBC，它是标准 C 库的超集。不仅包含 ANSI C 中定义的函数，还包括 POSIX 标准中定义的函数。因此，Linux 下既可以使用标准 I/O，也可以使用文件 I/O）。

通过文件 I/O 读写文件时，每次操作都会执行相关系统调用。这样处理的好处是直接读写实际文件，坏处是频繁的系统调用会增加系统开销。标准 I/O 可以看成是在文件 I/O 的基础上封装了缓冲机制。先读写缓冲区，必要时再访问实际文件，从而减少了系统调用的次数。

文件 I/O 中用文件描述符表示一个打开的文件，可以访问不同类型的文件如普通文件、设备文件和管道文件等。而标准 I/O 中用 FILE（流）表示一个打开的文件，通常只用来访问普通文件。

2.2 文件 I/O 操作

本节主要介绍文件 I/O 相关函数：open()、read()、write()、lseek()和 close()。这些函数的特点是不带缓冲，直接对文件（包括设备）进行读写操作。这些函数不是 ANSI C 的组成部分，而是由 POSIX 相关标准来定义。

2.2.1 文件打开和关闭

1．函数说明

open()函数用于创建或打开文件，在打开或创建文件时可以指定文件打开方式及文件的访问权限。

close()函数用于关闭一个被打开的文件。当一个进程终止时，所有打开的文件都由内核自动关闭。很多程序都利用这一特性而不显式地关闭一个文件。

2．函数格式

open()函数的语法格式如表 2.1 所示。

表 2.1 open()函数语法要点

所需头文件	#include <sys/stat.h> #include <fcntl.h>	
数原型	int open（const char *pathname, int flags, int perms）；	
函数传入值	pathname	被打开的文件名（可包括路径名）
	flags：文件打开的方式	O_RDONLY：以只读方式打开文件
		O_WRONLY：以只写方式打开文件
		O_RDWR：以读写方式打开文件
		O_CREAT：如果该文件不存在，就创建一个新的文件，并用第 3 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已存在
		O_NOCTTY：使用本参数时，若打开的是终端文件，那么该终端不会成为当前进程的控制终端
		O_TRUNC：若文件已经存在，那么会删除文件中的全部原有数据，并且设置文件大小为 0
		O_APPEND：以添加方式打开文件，在写文件时，文件读写位置自动指向文件的末尾，即将写入的数据添加到文件的末尾
	perms	新建文件的存取权限 可以用一组宏定义：S_I（R/W/X）（USR/GRP/OTH） 其中 R/W/X 分别表示读/写/执行权限 USR/GRP/OTH 分别表示文件所有者/文件所属组/其他用户 例如，S_IRUSR | S_IWUSR 表示设置文件所有者的可读可写属性。八进制表示法中 0600 也表示同样的权限
函数返回值	成功：返回文件描述符	

	失败：–1

在 open()函数中，flags 参数可通过“ | ”组合构成，但前 3 个标志常量（O_RDONLY、O_WRONLY 以及 O_RDWR）不能相互组合。perms 是文件的存取权限，既可以用宏定义表示法，也可以用八进制表示法。

close()函数的语法格式如表 2.2 所示。

表 2.2　　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．示例代码

```
/* sample1.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    int fd;

    if ((fd = open("test.txt", O_RDWR|O_CREAT|O_TRUNC, 0666)) <0 )
    {
        perror("fail to open");
        return -1;
    }
    close(fd);

    return 0;
}
```

2.2.2　文件读写

1．函数说明

read()函数从文件中读取数据存放到缓存区中，并返回实际读取的字节数。若返回 0，则表示没有数据可读，即已达到文件尾。读操作从文件的当前读写位置开始读取内容，当前读写位置自动往后移动。

write()函数将数据写入文件中，并返回实际写入的字节数。写操作从文件的当前读写位置开始写入。对磁盘文件进行写操作时，若磁盘已满，write()函数返回失败。

2．函数格式

read()函数的语法格式如表 2.3 所示。

表 2.3 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指定读出的字节数。

write()函数的语法格式如表 2.4 所示。

表 2.4 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

3．示例代码

```
/* sample2.c */
#include <stdio.h>
```

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#define  N  64

int main()
{
    int fd, nbyte, sum = 0;
    char buf[N];
    if ((fd = open("test.txt", O_RDONLY)) < 0)
    {
        perror("fail to open");
        return -1;
    }
    while((nbyte = read(fd, buf, N))>0)  // 循环从文件中读取数据，直到文件末尾
    {
        sum += nbyte;  // 累加每次读取的字节数
    }
    printf("the length of test.txt is %d\n", sum);
    close(fd);
    return 0;
}
```

2.2.3 文件定位

1．函数说明

lseek()函数对文件当前读写位置进行定位。它只能对可定位（可随机访问）文件操作。管道、套接字和大部分字符设备文件不支持此类操作。

2．函数格式

lseek()函数的语法格式如表 2.5 所示。

表 2.5 lseek()函数的语法要点

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：相对于基准点 whence 的偏移量。以字节为单位，正数表示向前移动，负数表示向后移动	
	whence： 当前位置的基点	SEEK_SET：文件的起始位置
		SEEK_CUR：文件当前读写位置
		SEEK_END：文件的结束位置
函数返回值	成功：文件当前读写位置	
	–1：出错	

3．示例代码

下面示例中的 open()函数带有 3 个 flags 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。

实现的功能是从一个文件（源文件）中读取最后 10KB 数据并复制到另一个文件（目标文件）。源文件以只读方式打开，目标文件是以只写方式打开，若目标文件不存在，可以创建并设置权限的初始值为 644，即文件所有者可读可写，文件所属组和其他用户只能读。

```
/* sample3.c */
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>


#define  BUFFER_SIZE  1024            /* 每次读写缓存大小，影响运行效率*/

#define SRC_FILE_NAME "src_file"      /* 源文件名 */

#define DEST_FILE_NAME    "dest_file" /* 目标文件名*/

#define OFFSET       10240            /* 复制的数据大小 */


int main()
{
    int fds, fdd;
    unsigned char buff[BUFFER_SIZE];
    int read_len;
```

```
    /* 以只读方式打开源文件 */
    if ((fds = open(SRC_FILE_NAME, O_RDONLY)) < 0)
    {
        perror("fail to open src_file");
        return -1;
    }

    /* 以只写方式打开目标文件，若此文件不存在则创建该文件, 访问权限值为 644 */
    if ((fdd = open(DEST_FILE_NAME, O_WRONLY|O_CREAT|O_TRUNC, 0644)) < 0)
    {
        perror("fail to open dest_file");
        return -1;
    }

    /* 将源文件的读写指针移到最后 10KB 的起始位置*/
    lseek(fds, -OFFSET, SEEK_END);

    /* 读取源文件的最后 10KB 数据并写到目标文件中，每次读写 1KB */
    while ((read_len = read(fds, buff, sizeof(buff))) > 0)
    {
        write(fdd, buff, read_len);
    }
    close(fds);
    close(fdd);

    return 0;
}

$ gcc -o sample3 sample3.c -Wall
$ ./sample3
$ ls -l dest_file
-rw-r--r-- 1 Linux Linux 10240 14:06 dest_file
```

2.2.4　文件锁

1．fcntl()函数说明

前面介绍的这 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，也就是当多个程序共同操作一个文件的情况。Linux 中通常采用的方法是给文件上锁，来解决对共享的资源的竞争。

文件锁包括建议性锁和强制性锁。建议性锁要求每个相关程序在访问文件之前检查是否有锁存在，并且尊重已有的锁。一般情况下，不建议使用建议性锁，因为无法保证每个程序都自动检查是否有锁。而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程序对该文件进行读写操作。采用强制性锁对性能的影响较大，每次读写操作内核都检查是否有锁存在。

在 Linux 中，实现文件上锁的函数有 lockf()和 fcntl()，其中 lockf()用于对文件施加建议性锁，而 fcntl()不仅可以施加建议性锁，还可以施加强制性锁。同时，fcntl()还能对文件的某一记录上锁，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，多个同时执行的程序允许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程序在文件的某个部分上建立写入锁。显然，在文件的同一部分不能同时建立读取锁和写入锁。

fcntl()函数具有丰富的功能，它可以对已打开的文件进行各种操作。不仅能够管理文件锁，还可以获取和设置文件相关标志位以及复制文件描述符等。在本节中，主要介绍利用它建立记录锁的方法。有兴趣的读者可以查看 fcntl 手册了解其他用法。

华清远见

2．函数格式

用于建立记录锁的 fcntl()函数的语法格式如表 2.6 所示。

表 2.6　　fcntl()函数语法要点

<table>
<tr><td>所需头文件</td><td colspan="2">#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h></td></tr>
<tr><td>函数原型</td><td colspan="2">int fcnt1（int fd, int cmd, …）；</td></tr>
<tr><td rowspan="5">函数传入值</td><td colspan="2">fd：文件描述符</td></tr>
<tr><td rowspan="4">cmd</td><td>F_GETLK：检测文件锁状态</td></tr>
<tr><td>F_SETLK：设置 lock 描述的文件锁</td></tr>
<tr><td>F_SETLKW：这是 F_SETLK 的阻塞版本（命令名中的 W 表示等待（wait））</td></tr>
<tr><td>在无法获取锁时，会进入睡眠状态；如果可以获取锁或者捕捉到信号则会返回</td></tr>
<tr><td>函数返回值</td><td colspan="2">成功：0</td></tr>
</table>

	–1：出错

如果 cmd 和锁操作相关，则第三个参数的类型为 struct *flock，其定义如下。

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

flock 结构中每个成员的取值含义如表 2.7 所示。

表 2.7　　flock 结构成员含义

l_type	F_RDLCK：读取锁（共享锁）
	F_WRLCK：写入锁（排斥锁）
	F_UNLCK：解锁
l_start	加锁区域在文件中的相对位移量（字节），与 l_whence 值一起决定加锁区域的起始位置
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，新位置为文件的长度加上偏移量的大小
l_len	加锁区域的长度
l_pid	具有阻塞当前进程的锁，其持有进程的进程号存放在 l_pid 中，仅由 F_GETLK 返回

若要加锁整个文件，可以将 l_start 设置为 0，l_whence 设置为 SEEK_SET，l_len 设置为 0。

3．fcntl()使用实例

下面给出了使用 fcntl()函数对文件加记录锁的代码。首先给 flock 结构体赋予相应的值，接着调用两次 fcntl()函数。第一次用 F_GETLK 命令判断是否可以执行 flock 结构所描述的锁操作：若成员 l_type 的值为 F_UNLCK，表示文件当前可以执行相应锁操作；否则成员 l_type 的值表示当前已有的锁类型并且成员 l_pid 被设置为拥有当前文件锁的进程号。

第二次用 F_SETLK 或 F_SETLKW 命令设置 flock 结构所描述的锁操作，后者是前者的阻塞版本。使用后者时，当不能执行相应上锁/解锁操作时，程序会被阻塞，直到能够操作为止。

文件记录锁的代码具体如下：

```
/* lock_set.c */

int lock_set(int fd, int type)

{
```

```
struct flock old_lock, lock;
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;
lock.l_type = type;
lock.l_pid = -1;

/* 判断文件是否可以上锁 */

fcntl(fd, F_GETLK, &lock);

if (lock.l_type != F_UNLCK)
{
    /* 判断文件不能上锁的原因 */

    if (lock.l_type == F_RDLCK) /* 该文件已有读取锁 */
    {
        printf("Read lock already set by %d\n", lock.l_pid);
    }
    else if (lock.l_type == F_WRLCK) /* 该文件已有写入锁 */
    {
        printf("Write lock already set by %d\n", lock.l_pid);
    }
}

/* l_type 可能已被 F_GETLK 修改过 */
lock.l_type = type;
/* 根据不同的 type 值进行阻塞式上锁或解锁 */

if ((fcntl(fd, F_SETLKW, &lock)) < 0)
{
    printf("Lock failed:type = %d\n", lock.l_type);
    return -1;
}

switch(lock.l_type)
{
```

```
            case F_RDLCK:
            {

                printf("Read lock set by %d\n", getpid());

            }
            break;
            case F_WRLCK:
            {

                printf("Write lock set by %d\n", getpid());

            }
            break;
            case F_UNLCK:
            {

                printf("Release lock by %d\n", getpid());

                return 1;
            }
            break;

        }/* end of switch  */
        return 0;
    }
```

下面的示例是文件写入锁的测试用例，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。

```
/* write_lock.c */
#include <unistd.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>


int lock_set(int fd, int type);



int main(void)

{
    int fd;


    /* 首先打开文件 */

    if ((fd = open("hello", O_RDWR)) < 0)

    {

        perror( "fail to open" );
```

```
        return -1;
    }


    /* 给文件上写入锁 */

    lock_set(fd, F_WRLCK);

    getchar(); //  等待用户键盘输入

    /* 给文件解锁 */

    lock_set(fd, F_UNLCK);

    getchar();

    close(fd);


    return 0;
}
```

运行如下命令编译程序

gcc -o write_lock write_lock.c lock_set.c

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目的。

终端一：

```
$ ./write_lock
write lock set by 4994
release lock by 4994
```

终端二：

```
$ ./write_lock
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，同一时刻只能有一个写入锁存在。

接下来的程序是文件读取锁的测试用例，原理和上面的程序一样。

```
/* fcntl_read.c */
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include "lock_set.c"


int main(void)
{
    int fd;
```

```
    fd = open("hello",O_RDWR | O_CREAT, 0644);

    if(fd < 0)
    {
        printf("Open file error\n");

        exit(1);
    }

    /* 给文件上读取锁 */

    lock_set(fd, F_RDLCK);

    getchar();

    /* 给文件解锁 */

    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。

终端一：

```
$ ./read_lock
read lock set by 5009
release lock by 5009
```

终端二：

```
$ ./read_lock
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 已设置读取锁后，进程 5010 仍然可以设置读取锁。

2.3 实验内容——生产者和消费者

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

使用文件来模拟 FIFO（先进先出）结构以及生产者—消费者运行模型。

实验中需要打开两个虚拟终端，分别运行生产者程序（producer）和消费者程序（customer）。两个程序同时对同一个文件进行读写操作。因为这个文件是共享资源，所以使用文件锁机制来保证两个程序对文件的访问都是原子操作。

先启动生产者进程，它负责创建模拟 FIFO 结构的文件（其实是一个普通文件）并投入生产，即按照给定的时间间隔，向文件写入自动生成的字符（在程序中用宏定义选择使用数字还是使用英文字符）。生产周期以及要生产的资源数通过参数传递给程序（默认生产周期为 1s，要生产的资源总数为 10 个字符，默认生产总时间为 10s）。

后启动的消费者进程按照给定的数目进行消费。首先从文件中读取相应数目的字符并在屏幕上显示，然后从文件中删除刚才消费过的数据。为了模拟 FIFO 结构，此时需要使用两次复制来实现文件内容的前移。每次消费的资源数通过参数传递给程序，默认值为 10 个字符。

3．实验步骤

（1）画出实验流程图。

文件读写及上锁实验流程图如图 2.2 所示。

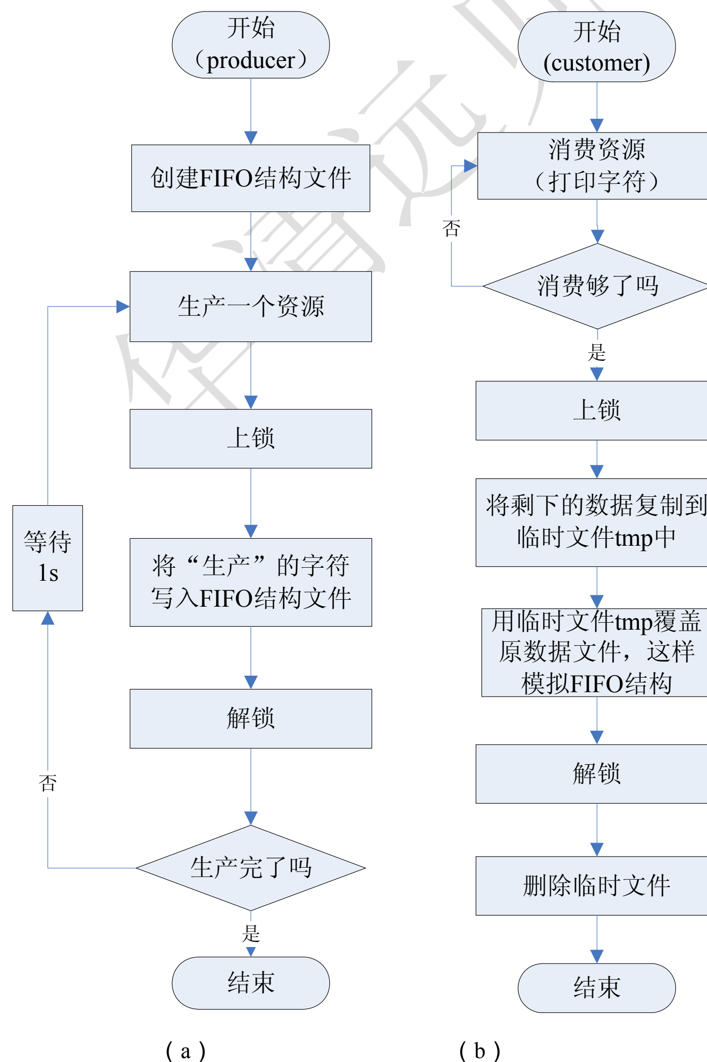


图 2.2 文件读写及上锁实验流程图

（2）编写代码。

本实验中的生产者程序的源代码如下，其中用到的 lock_set()函数可参见第 2.2.4 小节。

```
/* producer.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "mylock.h"

#define MAXLEN              10          /* 缓冲区大小最大值*/

#define ALPHABET            1           /* 表示使用英文字符 */

#define ALPHABET_START      'a'         /* 头一个字符，可以用 'A'*/

#define COUNT_OF_ALPHABET 26            /* 字母字符的个数 */

#define DIGIT               2           /* 表示使用数字字符 */

#define DIGIT_START         '0'         /* 头一个字符 */

#define COUNT_OF_DIGIT      10          /* 数字字符的个数 */

#define SIGN_TYPE ALPHABET              /* 本实例选用英文字符 */

const char *fifo_file = "./myfifo";    /* 仿真 FIFO 文件名 */

char buff[MAXLEN];                      /* 缓冲区 */

/* 功能：生产一个字符并写入仿真 FIFO 文件中 */

int product(void)

{
    int fd;
    unsigned int sign_type, sign_start, sign_count, size;
    static unsigned int counter = 0;


    /* 打开仿真 FIFO 文件 */

    if ((fd = open(fifo_file, O_CREAT|O_RDWR|O_APPEND, 0644)) < 0)

    {

        printf("Open fifo file error\n");

        exit(1);
```

```
    }

    sign_type =  SIGN_TYPE;
    switch(sign_type)
    {
        case ALPHABET:/* 英文字符 */
        {
            sign_start = ALPHABET_START;
            sign_count = COUNT_OF_ALPHABET;
        }
        break;

        case DIGIT:/* 数字字符 */
        {
            sign_start = DIGIT_START;
            sign_count = COUNT_OF_DIGIT;
        }
        break;

        default:
        {
            return -1;
        }
    }/*end of switch*/

    sprintf(buff, "%c", (sign_start + counter));
    counter = (counter + 1) % sign_count;

    lock_set(fd, F_WRLCK); /* 上写锁*/
    if ((size = write(fd, buff, strlen(buff))) < 0)
    {
        printf("Producer: write error\n");
        return -1;
    }
    lock_set(fd, F_UNLCK); /* 解锁 */

    close(fd);
```

```
    return 0;
}


int main(int argc ,char *argv[])
{
    int time_step = 1;     /* 生产周期 */
    int time_life = 10;   /* 需要生产的资源总数 */

    if (argc > 1)
    {/* 第一个参数表示生产周期 */
        sscanf(argv[1], "%d", &time_step);
    }

    if (argc > 2)
    {/* 第二个参数表示需要生产的资源数 */
        sscanf(argv[2], "%d", &time_life);
    }
    while (time_life--)
    {
        if (product() < 0)
        {
            break;
        }
        sleep(time_step);
    }

    exit(EXIT_SUCCESS);
}
```

本实验中的消费者程序的源代码如下：

```
/* customer.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <fcntl.h>

#define MAX_FILE_SIZE     100 * 1024 * 1024 /* 100MB */

const char *fifo_file = "./myfifo";            /* 仿真 FIFO 文件名 */

const char *tmp_file = "./tmp";                /* 临时文件名 */

/* 资源消费函数 */
int customing(const char *myfifo, int need)
{
    int fd;
    char buff;
    int counter = 0;

    if ((fd = open(myfifo, O_RDONLY)) < 0)
    {
        printf("Function customing error\n");
        return -1;
    }

    printf("Enjoy:");
    lseek(fd, SEEK_SET, 0);
    while (counter < need)
    {
        while ((read(fd, &buff, 1) == 1) && (counter < need))
        {
            fputc(buff, stdout); /* 消费就是在屏幕上简单地显示 */
            counter++;
        }
    }
    fputs("\n", stdout);
    close(fd);
    return 0;
```

```
}

/* 功能:从 sour_file 文件的 offset 偏移处开始

将 count 个字节数据复制到 dest_file 文件 */

int myfilecopy(const char *sour_file,

           const char *dest_file, int offset, int count, int copy_mode)

{
    int in_file, out_file;
    int counter = 0;
    char buff_unit;

    if ((in_file = open(sour_file, O_RDONLY|O_NONBLOCK)) < 0)

    {

        printf("Function myfilecopy error in source file\n");

        return -1;
    }

    if ((out_file = open(dest_file,

                O_CREAT|O_RDWR|O_TRUNC|O_NONBLOCK, 0644)) < 0)

    {

        printf("Function myfilecopy error in destination file:");

        return -1;
    }

    lseek(in_file, offset, SEEK_SET);

    while ((read(in_file, &buff_unit, 1) == 1) && (counter < count))

    {

        write(out_file, &buff_unit, 1);

        counter++;
    }

    close(in_file);

    close(out_file);
```

```
    return 0;
}

/* 功能：实现 FIFO 消费者 */

int custom(int need)

{
    int fd;

    /* 对资源进行消费，need 表示该消费的资源数目 */

    customing(fifo_file, need);

    if ((fd = open(fifo_file, O_RDWR)) < 0)

    {

        printf("Function myfilecopy error in source_file:");

        return -1;
    }

    /* 为了模拟 FIFO 结构，对整个文件内容进行平行移动 */

    lock_set(fd, F_WRLCK);

    myfilecopy(fifo_file, tmp_file, need, MAX_FILE_SIZE, 0);

    myfilecopy(tmp_file, fifo_file, 0, MAX_FILE_SIZE, 0);

    lock_set(fd, F_UNLCK);

    unlink(tmp_file);

    close(fd);

    return 0;
}

int main(int argc ,char *argv[])

{
    int customer_capacity = 10;

    if (argc > 1) /* 第一个参数指定需要消费的资源数目，默认值为 10 */
```

```
    {
        sscanf(argv[1], "%d", &customer_capacity);
    }
    if (customer_capacity > 0)
    {
        custom(customer_capacity);
    }
    exit(EXIT_SUCCESS);
}
```

（3）编译并运行。

4．实验结果

实验运行结果和程序的参数相关，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：

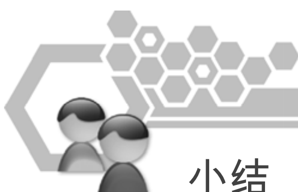
```
$ ./producer 1 20  /* 生产周期为 1s，需要生产的资源总数为 20 个 */
Write lock set by 21867
Release lock by 21867
Write lock set by 21867
Release lock by 21867
……
```

终端二：

```
$ ./customer 5              /* 需要消费的资源数为 5 个 */

Enjoy:abcde                 /* 对资源进行消费，即打印到屏幕上 */

Write lock set by 21872     /* 为了仿真 FIFO 结构，进行两次复制 */

Release lock by 21872
```

在两个程序结束之后，文件的内容如下。

```
$ cat myfifo

fghijklmnopqr   /* a 到 e 的 5 个字符已经被消费，就剩下后面 15 个字符 */
```

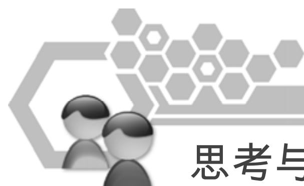


小结

本章首先讲解了系统调用、用户函数接口和系统命令之间的联系和区别，以及 Linux 的文件系统的基本知识。

接着，本章重点讲解了不带缓冲的文件 I/O 相关函数的使用。文件 I/O 函数的使用范围非常广泛，在很多应用开发中都会涉及，是学习嵌入式 Linux 应用开发的基础。

最后，本章安排了文件锁实验，希望读者认真练习。



思考与练习

1．简述虚拟文件系统在 Linux 系统中的位置和通用文件系统模型。

2．文件 I/O 和标准 I/O 之间有哪些区别？

联系方式

集团官网：www.hqyj.com　嵌入式学院：www.embedu.org　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　物联网学院：www.topsight.cn　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

全国免费咨询电话：400-706-1880

双休日及节假日请致电值班手机：15010390966

在线咨询：张老师 QQ（619366077），王老师 QQ（2814652411），杨老师 QQ（1462495461）

企业培训洽谈专线：010-82600901

院校合作洽谈专线：010-82600350，在线咨询：QQ（248856300）