Linux文件I/O编程

Version 1.0

西安電子科技大學

需要掌握的要点

- 理解系统调用与用户编程接口（API）的关系
- 了解glibc库的组成：
  - ISO C函数集合
  - POSIX函数集合
  - System V函数集合
- 掌握底层文件I/O操作（POSIX中定义的文件函数）
  - 理解文件描述符的概念
  - 掌握如何利用fcntl函数为文件加锁
  - 重点掌握如何利用select、poll等函数实现多路复用I/O编程
- 掌握标准I/O编程（ISO C中定义的文件函数）
  - 与底层文件I/O区别是增加了缓冲管理
  - 可移植性更好

西安電子科技大學

从宏观角度看虚拟内存管理

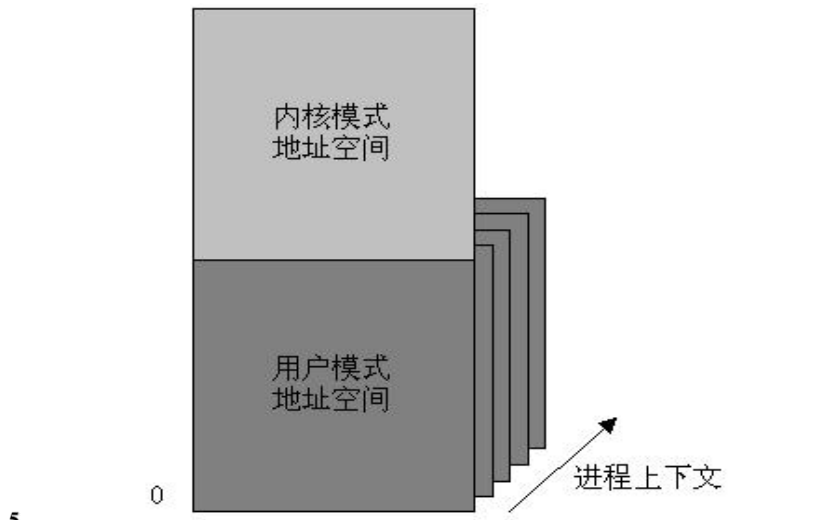
Linux的虚拟内存管理机制为应用程序和驱动程序提供了两种服务：

- ✔ 使每个进程都拥有自己独立的内存地址空间；对于32位Linux而言，每个任务可寻址的内存地址空间都为0x00000000 ~ 0xFFFFFFFF(2^{32}, 4GB)
- ✔ 当物理内存不够4GB时，虚拟内存管理模块会用外存空间模拟内存空间，并且该模拟过程对应用程序是透明的。

用户地址空间与内核地址空间

1. Linux将每个进程的4GB的独立地址空间又划分为用户地址空间(0x00000000 ~ 0xBFFFFFFF)和 内核地址空间(0xC0000000 ~ 0xFFFFFFFF)两部分。
2. 操作系统内核代码和数据存放在内核地址空间；每个进程自己私有的代码和数据存放在用户地址空间
3. 虽然Linux的内核代码和数据被映射到了每个进程的地址空间中（所有进程看到的内容是相同的），但在实际的物理内存中，只有内核代码和数据的一份拷贝。

用户地址空间与内核地址空间

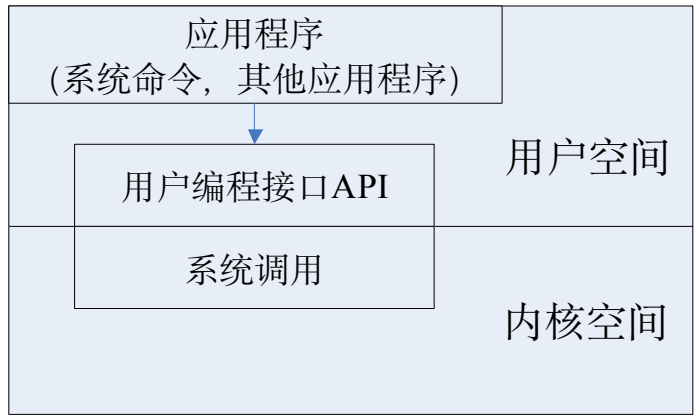


用户态与核心态

- 一般现代CPU都有几种不同的指令执行级别
- 在高执行级别下，代码可以执行特权指令，访问任意的物理地址，这种CPU执行级别就对应着内核态
- 用户态指相应的低级别执行状态，代码的掌控范围会受到限制，只能执行CPU指令集的一个子集
- 举例：intel x86 CPU有四种不同的执行级别0-3，Linux只使用了其中的0级和3级分别来表示内核态和用户态
- 0xc0000000以上的内核地址空间只能在内核态下访问，0x00000000－0xbfffffff的用户地址空间在两种状态下都可以访问
- 应用程序可以通过Linux系统调用由用户态进入内核态

系统调用与API

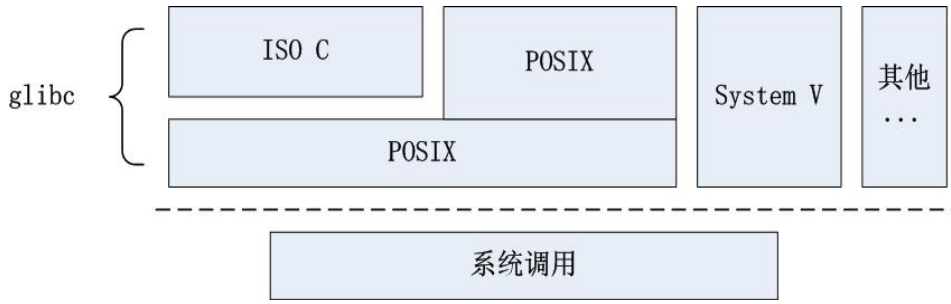
系统调用与API的区别?



Linux用户编程接口（API）

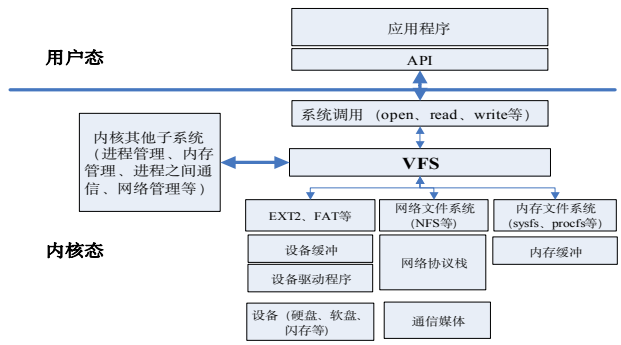
- 在Linux中，用户编程接口（API）遵循了在Unix中最流行的应用编程界面标准——POSIX标准。POSIX标准是由IEEE和ISO/IEC共同开发的标准系统。用于保证应用程序可以在源代码一级上在多种操作系统上移植运行。
- 这些系统调用编程接口主要是通过系统C库（libc）实现的。
- 实际中程序员直接使用的是用户编程接口，用户编程接口的实现可能用到了系统调用
  - 一个API函数对应一个系统调用
  - 一个API函数调用了多个系统调用
  - 一个API函数没有调用系统调用

GNU LIB C（glibc）库的构成



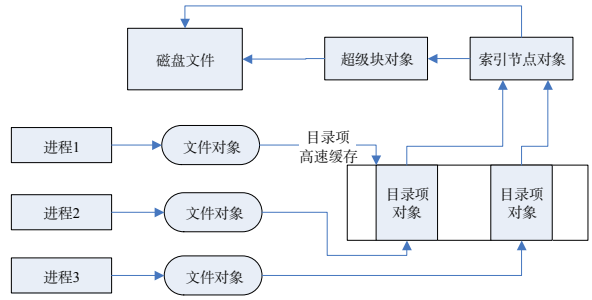
Linux文件I/O系统概述--虚拟文件系统

- 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屏蔽了底层具体文件系统的实现细节与差异。
- VFS在linux系统中的位置如图：
-

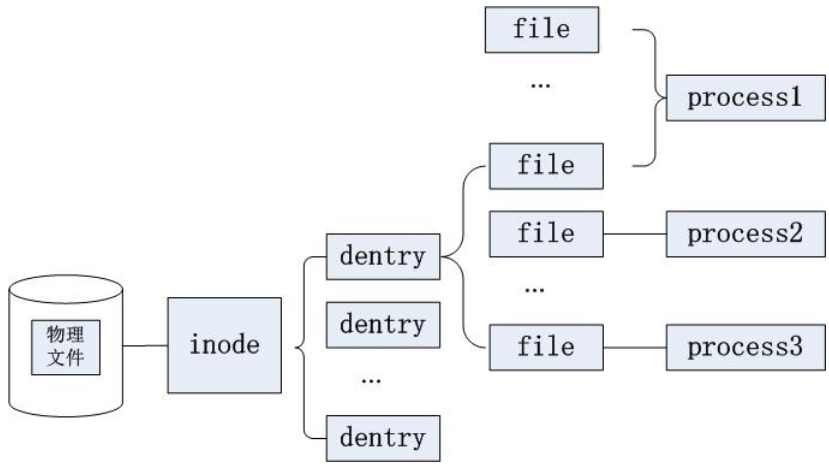


Linux文件I/O系统概述——通用文件系统

通用的文件模型（common file model），这个模型的核心是4个对象类型，即超级块对象（superblock object）、索引节点对象（inode object）、目录项对象（dentry object）和文件对象（file object）



关于linux文件的几个重要的结构体

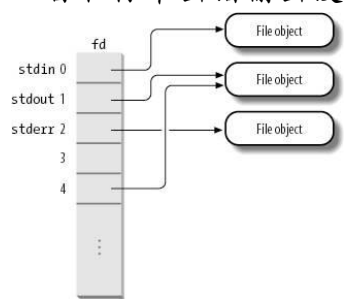


关于linux文件的几个重要的结构体

- Inode结构体主要存储文件的元数据（metadata）信息，包括文件大小、访问权限、修改时间、实体数据存储位置等，但node中不存储文件名
- dentry结构主要存储文件名及该文件在某个目录树中的位置信息。
- file结构体用于记录一个打开的文件的相关信息，如访问方式（可读、可写）、当前读写位置等。
- Inode与物理文件一一对应；inode和dentry之间是一对多的关系，因为同一个物理文件可以被“挂”到多个目录树上（学习文件硬链接的概念）；dentry与file之间是一对多的关系。

文件描述符

- 一个linux进程会同时打开多个文件，因此会在自己的进程地址空间中创建多个file结构体，这些结构体形成一个file数组。
- 某个打开文件的文件描述符就是该文件对应的file结构体在进程file数组中的索引。
- file数组大小限制了进程能够同时打开的文件数目（1024）.
- 每个进程一般会默认打开3个文件：标准输入设备、标准输出设备和标准出错输出设备，对应的文件描述符分别是0,1,2



	文件描述符	宏
标准输入	0	STDIN_FILENO
标准输出	1	STDOUT_FILENO
标准出错	2	STDERR_FILENO

Linux底层文件I/O函数

int open(const char * pathname, int flags, mode_t mode);

int close(int fd);

ssize_t read(int fd, void * buf , size_t count);

ssize_t write (int fd, const void * buf, size_t count);

off_t lseek(int fd, off_t offset , int whence);

int creat(const char * pathname, mode_tmode);

int fcntl(int fd, int cmd, struct flock *lock);

特点：不带缓存直接对文件进行读写操作。不是ANSI C的组成部分，但是是POSIX的组成部分。

Linux底层文件I/O函数

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。

所需头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

原型：

int open(const char *pathname, int flags, int perms)

Linux底层文件I/O函数

- **open函数语法要点**

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：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。此时 open 是原子操作，防止多个进程同时创建同一个文件。
		O_NOCTTY：使用本参数时，若文件为终端，那么该终端不会成为调用 open()的那个进程的控制终端
		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，在打开文件的同时，文件指针指向文件的末尾，即将写入的数据添加到文件的末尾。
	Perms	被打开文件的存取权限。 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	

Linux底层文件I/O函数

- **close()函数是用于关闭一个被打开的文件。**
  - 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

Linux底层文件I/O函数

- **read()函数是用于将从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。**
  - 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

Linux底层文件I/O函数

- **write()函数是用于向打开的文件写数据，写操作从文件的当前指针位置开始。**
  - 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

Linux底层文件I/O函数

- **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。**
  - 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，可正可负（向前移，向后移）	
	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	

文件锁

- **在Linux中，多个并发执行的进程可能会访问同一文件，访问包括读/写操作，系统如何控制其访问?**
- **文件锁包括建议性锁和强制性锁。**
  - 建议性锁只是用来查询文件是否有上锁标记，并不能阻止真正的读写访问操作。
  - 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。强制性锁对系统全局有效，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。
- **文件锁又可分为读取锁和写入锁**

文件锁

- 在Linux中，实现文件上锁的函数有fcntl()与lockf()，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，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。

利用fcntl函数加锁

int fcntl(int fd, int cmd, struct flock *lock);

所需头文件	#include <sys/types.h> #include <unistd.h> #include <fcntl.h>	
函数原型	int fcntl(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，该标志由参数 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_GETLK：根据 lock 描述，决定是否上文件锁
		F_SETLK：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））。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。
	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	

利用fcntl函数加锁

- **fcntl()函数第三个参数lock说明**

lock为结构体flock它的定义为：

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

flock成员取值含义如表：

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，新位置为文件的大小加上偏移量的大小
l_len	加锁区域的长度

加锁整个文件 (l_start=0, l_whence=SEEK_SET, l_len=0)

利用fcntl函数加锁

- 对文件区域加锁之后，必须使用底层的read与write调用来访问文件中的数据，而不要去使用高层的fread与fwrite（对读写的数据进行缓存）

- 开启Linux中的强制锁功能支持
  - 挂载文件系统时使用“-o mand”参数

    mount -oremount,mand /
- 对于要打开强制锁功能的文件lock_file，必须打开set-group-ID位，关闭group-execute

  chmod g+s,g-x mandatory.txt

阻塞式I/O与非阻塞式I/O

int open(const char * pathname, int flags, mode_t mode);

ssize_t read(int fd, void * buf , size_t count);

ssize_t write (int fd, const void * buf, size_t count);

1. 调用open函数打开文件时，如果flags设置了O_NONBLOCK标志位，则针对该文件描述符的read和write调用都是非阻塞式的；否则read和write调用都是阻塞式的。
2. 思考：如何实现同时对多个文件（设备）的读写？？
3. 同时对多个文件（设备）读写时，涉及到一个编程模型问题，我们称为“I/O处理模型问题”

I/O处理模型

- **I/O处理模型**
  - 阻塞I/O模型
    - 一个进程或线程调用阻塞I/O，串行处理文件描述符
  - 非阻塞模型
    - 循环测试
  - I/O多路复用模型
    - 一个进程或线程同时监听和处理多个文件描述符，实现了并发的效果
  - 信号驱动I/O模型
    - Signal机制
  - 异步I/O模型
    - aio
- **select和poll的I/O转接模型是处理I/O复用的一个高效的方法**

Linux的I/O机制

- 1. 同步阻塞I/O: 用户进程进行I/O操作，一直阻塞到I/O操作完成为止。
  2. 同步非阻塞I/O: 用户程序可以通过设置文件描述符的属性O_NONBLOCK，I/O操作可以立即返回，但是并不保证I/O操作成功。
  3. 异步事件阻塞I/O: 用户进程可以对I/O事件进行阻塞，但是I/O操作并不阻塞。通过select/poll/epoll等函数调用来达到此目的。
  4. 异步事件非阻塞I/O: 也叫做异步I/O(AIO)，用户程序可以通过向内核发出I/O请求命令，不用等带I/O事件真正发生，可以继续做另外的事情，等I/O操作完成，内核会通过函数回调或者信号机制通知用户进程。这样很大程度提高了系统吞吐量。

select函数

select函数是实现多路复用I/O模型的关键。

select函数可以用来同时监测多个文件（设备）的可读写状态。如果select关心的文件（设备）都不可读写，则select阻塞；当某个（某些）文件可读写时，select立即返回。

```
#include <sys/select.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset,
                        fd_set *exceptset, struct timeval *timeout);
struct timeval{
    long tv_sec;  /*秒 */
    long tv_usec; /*微秒 */
}
```

select函数

- **select()函数的语法格式：**

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：该参数值为需要监视的文件描述符的最大值加 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，若等待了 timeout 时间还没有检测到任何文件描符准备好，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 0：超时；-1：出错	

select函数

fd_set：文件描述符集合类型。该类型的变量是一个结构体，我们不必关心该结构体的构造，只需知道它代表一个文件描述符集合

int FD_ZERO(fd_set *fdset);　/*判断集合是否为空*/
int FD_CLR(int fd, fd_set *fdset);　/*从集合中删除一个元素*/
int FD_SET(int fd, fd_set *fd_set); /*在集合中增加一个元素*/
int FD_ISSET(int fd, fd_set *fdset); /*判断集合是否含有某个元素*/

select函数

- 一般来说，在每次使用select()函数之前，首先使用FD_ZERO()和FD_SET()来初始化文件描述符集（在需要重复调用select()函数的时候，先把一次初始化好的文件描述符集备份下来，每次读取它即可）。在select()函数返回之后，可循环使用FD_ISSET()来测试描述符集，在执行完对相关文件描述符的操作之后，使用FD_CLR()来清除描述符集。
- 当使用select()函数时，存在一系列的问题，例如：内核必须检查多余的文件描述符，每次调用select()之后必须重置被监听的文件描述符集，而且可监听的文件个数受限制（使用FD_SETSIZE宏来表示fd_set结构能够容纳的文件描述符的最大数目）等
- 实际上，poll机制与select机制相比效率更高，使用范围更广

poll函数

- **poll函数语法格式：**

所需头文件	#include <sys/types.h> #include <poll.h>
函数原型	int poll(struct pollfd *fds, int numfds, int timeout)

函数传入值	fds：struct pollfd 结构的指针，用于描述需要对哪些文件的哪种类型的操作进行监控。
	struct pollfd
	{
	int fd; /* 需要监听的文件描述符 */
	short events; /* 需要监听的事件 */
	short revents; /* 已发生的事件 */
	}
	events 成员描述需要监听哪些类型的事件，可以用以下几种标志来描述：
	POLLIN：文件中有数据可读，下面实例中使用到了这个标志。
	POLLPRI：文件中有紧急数据可读。
	POLLOUT：可以向文件写入数据。
	POLLERR：文件中出现错误，只限于输出。
	POLLHUP：与文件的连接被断开了，只限于输出。
	POLLNVAL：文件描述符是不合法的，即它并没有指向一个成功打开的文件。

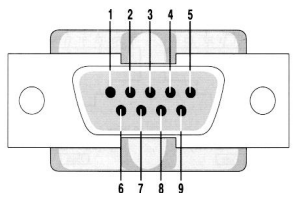
	numfds：需要监听的文件个数，即第一个参数所指向的数组中的元素数目。
	timeout：表示 poll 阻塞的超时时间（毫秒）。如果该值小于等于 0，则表示无限等待。
函数返回值	成功：返回大于 0 的值，表示事件发生的 pollfd 结构的个数。 0：超时；-1：出错

poll函数

- 一般来说，在每次使用select()函数之前，首先使用FD_ZERO()和FD_SET()来初始化文件描述符集（在需要重复调用select()函数的时候，先把一次初始化好的文件描述符集备份下来，每次读取它即可）。在select()函数返回之后，可循环使用FD_ISSET()来测试描述符集，在执行完对相关文件描述符的操作之后，使用FD_CLR()来清除描述符集。
- 当使用select()函数时，存在一系列的问题，例如：内核必须检查多余的文件描述符，每次调用select()之后必须重置被监听的文件描述符集，而且可监听的文件个数受限制（使用FD_SETSIZE宏来表示fd_set结构能够容纳的文件描述符的最大数目）等
- 例子
- http://blog.csdn.net/wenhuiqiao/article/details/7066267

嵌入式Linux串口应用编程

- **串口是计算机一种常用的接口，常用的串口有RS-232-C接口。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”。**



1	载波检测 DCD	6	数据就绪 DSR
2	接收数据 RXD	7	发送请求 RTS
3	发送数据 TXD	8	清除发送 CTS
4	数据终端就绪 DTR	9	振铃提示 RI
5	信号地 SG		

嵌入式Linux串口应用编程

- **Linux下的设备标识**
  - /dev/ttyS0，/dev/ttyS1
  - USB转串口，/dev/ttyUSB0，/dev/ttyUSB1
- **Linux下对设备的操作方法与对文件的操作保持一致**
  - Open，read，write，close
  - 需要对串口进行参数设置，波特率（115200），起始位比特数（1bit），数据位比特数（8bit），停止位比特数（1bit），流控模式（无）

嵌入式Linux串口应用编程

- **串口设置主要是设置struct termios结构体的各个成员**

```
#include <termios.h>
struct termios
{
    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;           /* 线路规程 */
    unsigned char   c_cc[NCC];    /* 控制特性 */
    speed_t         c_ispeed;     /* 输入速度 */
    speed_t         c_ospeed;     /* 输出速度 */
};
```

嵌入式Linux串口应用编程

- 终端是指用户与计算机进行交互的接口，如键盘、显示器、串口设备等物理设备和虚拟终端（文本式/X-WINDOWS)
- 终端三种工作模式
  - 规范模式，所有的输入是基于行进行处理，终端回显，在用户输入行结束符之前，系统调用read()读不到数据。支持行编辑，一次read()调用最多只读取一行数据
  - 非规范模式，所有输入即时有效，终端回显，不支持行编辑。设置MIN与TIME控制读操作
  - 原始模式，输入数据以字节为单位看待，终端不回显

嵌入式Linux串口应用编程

- 保存原先串口设置
  - 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, &old_cfg)。该函数得到由fd指向的终端的配置参数，并将它们保存于termios结构变量old_cfg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
  - 示例：

```
if (tcgetattr(fd, &old_cfg) != 0)
{
perror("tcgetattr");
return -1;
}
```

嵌入式Linux串口应用编程

- 激活选项
  - CLOCAL和CREAD分别用于本地连接和接收使能，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;
  - 调用cfmakeraw()函数可以将终端设置为原始模式，在后面的实例中，采用原始模式进行串口数据通信。

    cfmakeraw(&new_cfg);

嵌入式Linux串口应用编程

- 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()。
  - 示例：

    cfsetispeed(&new_cfg, B115200);
    cfsetospeed(&new_cfg, B115200);

嵌入式Linux串口应用编程

- 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
  - 示例：

```
new_cfg.c_cflag &= ~CSIZE; /* 用数据位掩码清空数据位设置 */
new_cfg.c_cflag |= CS8;
```


嵌入式Linux串口应用编程

- 设置奇偶校验位
  - 设置奇偶校验位需要用到termios中的两个成员：c_cflag和c_iflag。首先要激活c_cflag中的校验位使能标志PARENB和是否要进行校验，这样会对输出数据产生校验位，而输入数据进行校验检查。同时还要激活c_iflag中的对于输入数据的奇偶校验使能（INPCK）。
  - 示例：
    - 奇校验

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

    - 偶校验

```
new_cfg.c_cflag |= PARENB;
new_cfg.c_cflag &= ~PARODD;
new_cfg.c_iflag |= INPCK;
```

嵌入式Linux串口应用编程

- 设置停止位
  - 设置停止位是通过激活c_cflag中的CSTOPB而实现的。若停止位为一个，则清除CSTOPB，若停止位为两个，则激活CSTOPB。
  - 示例：

```
new_cfg.c_cflag &=  ~CSTOPB;/* 将停止位设置为一个比特 */
new_cfg.c_cflag |=  CSTOPB;  /* 将停止位设置为两个比特 */
```


嵌入式Linux串口应用编程

- 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，则在任何情况下read()函数立即返回，此时串口操作会设置为非阻塞方式。
  - 示例：

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

嵌入式Linux串口应用编程

- 清除串口缓冲
  - 由于串口在重新设置之后，需要对当前的串口设备进行适当的处理，这时就可调用在<termios.h>中声明的tcdrain()、tcflow()、tcflush()等函数来处理目前串口缓冲中的数据。
  - 原型：

    int tcflush(int fd, int queue_selector); /* 用于清空输入/输出缓冲区*/
  - tcflush()函数，对于在缓冲区中的尚未传输的数据，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，它可能的取值有以下几种。
    - TCIFLUSH：对接收到而未被读取的数据进行清空处理。
    - TCOFLUSH：对尚未传送成功的输出数据进行清空处理。
    - TCIOFLUSH：包括前两种功能，即对尚未处理的输入输出数据进行清空处理。
  - 示例：tcflush(fd, TCIFLUSH);

嵌入式Linux串口应用编程

- 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(int fd, int optional_actions, const struct termios *termios_p);

    其中参数termios_p是termios类型的新配置变量。

    参数optional_actions可能的取值有以下三种：
    - TCSANOW：配置的修改立即生效。
    - TCSADRAIN：配置的修改在所有写入fd的输出都传输完毕之后生效。
    - TCSAFLUSH：所有已接受但未读入的输入都将在修改生效之前被丢弃。
  - 该函数若调用成功则返回0，若失败则返回–1

嵌入式Linux串口应用编程

- 串口参数设置处理封装为set_com_config

嵌入式Linux串口应用编程

- 打开串口
  - 使用open函数打开串口
    - **fd = open("/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);**
    - **O_NOCTTY，不会使打开的文件（终端）成为该进程的的控制终端**
    - **O_NDELAY，非阻塞方式，不关心DCD信号线（另一端的激活状态）**
  - 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的文件描述符是否连接到一个终端设备，以进一步确认串口是否正确打开，如下所示：
    - isatty(fd);
  - 该函数调用成功则返回0，若失败则返回-1。
- 封装为open_port()

嵌入式Linux串口应用编程

- 读写串口
  - 使用read/write函数读写串口

```
write(fd, buff, strlen(buff));
read(fd, buff, BUFFER_SIZE);
```

- 例子

标准I/O编程

- 系统调用的开销，中断处理，用户态/内核态切换
- 标准I/O提供流缓冲的目的是尽可能减少使用read()和write()等系统调用的数量，降低开销，提升性能

标准I/O编程

- 三种类型的缓冲存储
  - 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。对于存放在磁盘上的文件通常是由标准I/O库实施全缓冲的。标准I/O尽量多读写文件到缓冲区，当缓冲区已满或手动flush时才会进行磁盘操作。
  - 行缓冲：在这种情况下，当在输入和输出中遇到行结束符时，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。标准输入和标准输出就是使用行缓冲的典型例子。
  - 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，则相当于用系统调用write()函数将这些字符全写到被打开的文件上。标准出错stderr通常是不带缓存的，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行结束符。

基本操作

- 打开文件

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，该指针指向对应的I/O流。

fopen()可以指定打开文件的路径和模式

fopen函数格式：

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
函数返回值	成功：指向 FILE 的指针 失败：NULL

基本操作

- 打开文件

mode类似于open()函数中的flag，可以定义打开文件的访问权限等，下面为mode的各种取值：

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，即会擦写文件以前的内容。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，即会擦写文件以前的内容。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。

基本操作

- **字符b用来指示打开的文件为二进制文件，而非纯文本文件**
  - Linux忽略该选项，为什么？

5.5.1 基本操作

- 打开文件

fdopen()可以指定打开的文件描述符和模式。

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
函数返回值	成功：指向 FILE 的指针 失败：NULL

mode取值同fopen

基本操作

- 打开文件

freopen()除可指定打开的文件、模式外，还可指定特定的I/O流。

freopen函数格式：

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
	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

mode取值同fopen

基本操作

- 实现重定向
  - 将stream指代的文件重定向到path指代的文件
  - ~/src/ch5/freopen

基本操作

- 关闭文件

关闭标准流文件的函数为fclose()，该函数将缓冲区内的数据全部写入到文件中，并释放系统所提供的文件资源。

fclose()函数格式：

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

基本操作

- 读文件

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

fread()函数格式:

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

基本操作

- 写文件

fwrite()函数是用于对指定的文件流进行写操作。

fwrite()函数格式:

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

基本操作

- **字符输入/输出**

字符输入函数：

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

字符输出函数：

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

基本操作

- **行输入/输出**

行输入函数：

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

行输出函数：

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

基本操作

– 格式化输入/输出

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

基本操作

- **格式化输入/输出**

格式化输出函数：

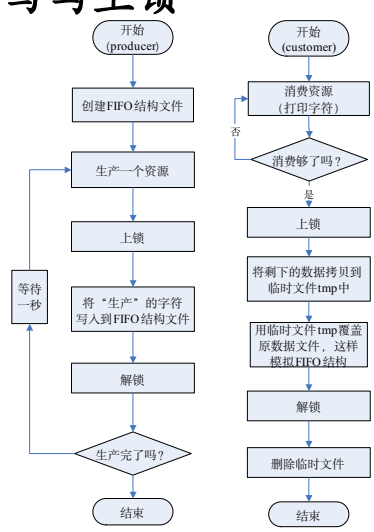
所需头文件	#include <stdio.h>
函数原型	int printf(const char *format,…) int fprintf(FILE *fp, const char *format,…) int sprintf(char *buf, const char *format,…)
函数传入值	format：记录输出格式 fp：文件描述符 buf：记录输出缓冲区
函数返回值	成功：输出字符数（sprintf 返回存入数组中的字符数） 失败：NULL

课后实验内容

- 文件的读写与上锁
  - 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。
  - 实验内容
    - 在Linux中FIFO是一种进程之间的管道通信机制。Linux支持完整的FIFO通信机制。
    - 本实验内容比较有趣，我们通过使用文件操作，仿真FIFO（先进先出）结构以及生产者-消费者运行模型。
    - 本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）。此时两个进程同时对同一个文件进行读写操作。因为这个文件是临界资源，所以可以使用文件锁机制来保证两个进程对文件的访问都是原子操作。
    - 先启动生产者进程，它负责创建仿真FIFO结构的文件（其实是一个普通文件）并投入生产，就是按照给定的时间间隔，向FIFO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进程（默认生产周期为1秒，要生产的资源总数为10个字符，显然默认生产总时间为10秒钟）。
    - 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。为了仿真FIFO结构，此时需要使用两次拷贝来实现文件内容的偏移。每次消费的资源数通过参数传递给进程，默认值为10个字符。

课后实验内容

- 文件的读写与上锁



西安電子科技大學

课后实验内容

- 多路复用式串口操作
  - 实验目的
    - 通过编写多路复用串口读写，进一步理解多路复用函数的用法，同时更加熟练掌握Linux设备文件的读写方法。
  - 实验内容
    - 本实验中，实现两台机器（宿主机和目标板）之间的串口通信，而且每台机器均可以发送和接收数据。除了串口设备名称不同（宿主机上使用串口1：/dev/ttyS0,而在目标板上使用串口5：/dev/ttyS1），两台机器上的程序基本相同。
    - 首先程序打开串口设备文件并进行相关配置。调用select()函数，使它等待从标准输入（终端）文件中的输入以及从串口设备的输入。如果有标准输入上的数据，则写入到串口，使对方读取。如果有串口设备上的输入，则将数据写入到普通文件中。

课后实验内容

- 多路复用式串口操作

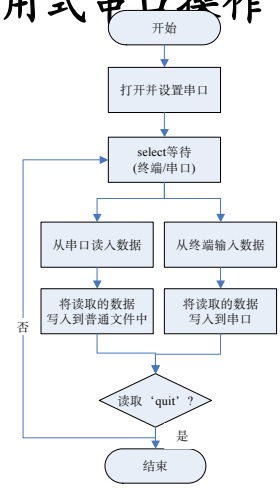


图5.5 宿主机/目标板程序的流程图

小结

1. 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。
2. 接着，本章重点讲解了嵌入式Linux中文件I/O开发相关的内容，在这里主要讲解了不带缓存的I/O系统调用函数的使用，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。其中讲解了基本文件操作、文件锁和多路复用等操作，这些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。
3. 接下来，本章讲解了嵌入式Linux串口编程。这也是嵌入式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。
4. 在本章的后面，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。
5. 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操作。希望读者认真完成。

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. 底层文件操作和标准文件操作之间有哪些区别?