网络安全

安全技巧（十五）

小未 主编

目　录

UNIX 下设备驱动程序的基本结构

在 UNIX 系统里，对用户程序而言，设备驱动程序隐藏了设备的具体细节，对各种不同设备提供了一致的接口，一般来说是把设备映射为一个特殊的设备文件，用户程序可以象对其它文件一样对此设备文件进行操作。UNIX 对硬件设备支持两个标准接口：块特别设备文件和字符特别设备文件，通过块（字符）特别设备文件存取的设备称为块（字符）设备或具有块（字符）设备接口。块设备接口仅支持面向块的 I/O 操作，所有 I/O 操作都通过在内核地址空间中的 I/O 缓冲区进行，它可以支持几乎任意长度和任意位置上的 I/O 请求，即提供随机存取的功能。

字符设备接口支持面向字符的 I/O 操作，它不经过系统的快速缓存,所以它们负责管理自己的缓冲区结构。字符设备接口只支持顺序存取的功能，一般不能进行任意长度的 I/O 请求，而是限制 I/O 请求的长度必须是设备要求的基本块长的倍数。显然，本程序所驱动的串行卡只能提供顺序存取的功能，属于是字符设备，因此后面的讨论在两种设备有所区别时都只涉及字符型设备接口。设备由一个主设备号和一个次设备号标识。主设备号唯一标识了设备类型，即设备驱动程序类型，它是块设备表或字符设备表中设备表项的索引。次设备号仅由设备驱动程序解释，一般用于识别在若干可能的硬件设备中，I/O 请求所涉及到的那个设备。

设备驱动程序可以分为三个主要组成部分：

(1)自动配置和初始化子程序，负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的、设备驱动程序需要的软件状态进行初始化。这部分驱动程序仅在初始化的时候被调用一次。

(2)服务于 I/O 请求的子程序，又称为驱动程序的上半部分。调用这部分是由于系统调用的结果。这部分程序在执行的时候，系统仍认为是和进行调用的进程属于同一个进程，只是由用户态变成了核心态，具有进行此系统调用的用户程序的运行环境，因此可以在其中调用 sleep()等与进程运行环境有关的函数。

(3)中断服务子程序，又称为驱动程序的下半部分。在 UNIX 系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由 UNIX 系统来接收硬件中断，再由系统调用中断服务子程序。中断可以产生在任何一个进程运行的时候，因此在中断服务程序被调用的时候，不能依赖于任何进程的状态，也就不能调用任何与进程运行环境有关的函数。

在系统内部，I/O 设备的存取通过一组固定的入口点来进行，这组入口点是由每个设备的设备驱动程序提供的。一般来说，字符型设备驱动程序能够提供如下几个入口点：

(1)open 入口点。打开设备准备 I/O 操作。对字符特别设备文件进行打开操作，都会调用设备的 open 入口点。open 子程序必须对将要进行的 I/O 操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的，即同一时刻只能有一个程序访问此设备，则 open 子程序必须设置一些标志以表示设备处于忙状态。

(2)close 入口点。关闭一个设备。当最后一次使用设备终结后，调用 close 子程序。独占设备必须标记设备可再次使用。

(3)read 入口点。从设备上读数据。对于有缓冲区的 I/O 操作，一般是从缓冲区里读数据。对字符特别设备文件进行读操作将调用 read 子程序。

(4)write 入口点。往设备上写数据。对于有缓冲区的 I/O 操作，一般是把数据写入缓冲区里。对字符特别设备文件进行写操作将调用 write 子程序。

(5)ioctl 入口点。执行读、写之外的操作。

(6)select 入口点。检查设备，看数据是否可读或设备是否可用于写数据。select 系统调用在检查与设备特别文件相关的文件描述符时使用 select 入口点。如果设备驱动程序没有提供上述入口点中的某一个，系统会用缺省的子程序来代替。对于不同的系统，也还有一些其它的入口点。

3.2、LINUX 系统下的设备驱动程序

具体到 LINUX 系统里，设备驱动程序所提供的这组入口点由一个结构来向系统进行说明,此结构定义为：

```
#include$#@60;linux/fs.h$#@62;
structfile_operations{
int(*lseek)(structinode*inode,structfile*filp,
off_toff,intpos);
int(*read)(structinode*inode,structfile*filp,
char*buf,intcount);
int(*write)(structinode*inode,structfile*filp,
char*buf,intcount);
int(*readdir)(structinode*inode,structfile*filp,
```

```
structdirent*dirent,intcount);
int(*select)(structinode*inode,structfile*filp,
intsel_type,select_table*wait);
int(*ioctl)(structinode*inode,structfile*filp,
unsignedintcmd,unsignedintarg);
int(*mmap)(void);
int(*open)(structinode*inode,structfile*filp);
void(*release)(structinode*inode,structfile*filp);
int(*fsync)(structinode*inode,structfile*filp);
};
```

其中，structinode **提供了关于特别设备文件**/dev/driver**（假设此设备名为** driver**）的信息，它的定义为：**

```
#include$#@60;linux/fs.h$#@62;
structinode{
dev_ti_dev;
unsignedlongi_ino;/*Inodenumber*/
umode_ti_mode;/*Modeofthefile*/
nlink_ti_nlink;
uid_ti_uid;
gid_ti_gid;
dev_ti_rdev;/*Devicemajorandminornumbers*/
off_ti_size;
time_ti_atime;
time_ti_mtime;
time_ti_ctime;
unsignedlongi_blksize;
unsignedlongi_blocks;
```

```
structinode_operations*i_op;
structsuper_block*i_sb;
structwait_queue*i_wait;
structfile_lock*i_flock;
structvm_area_struct*i_mmap;
structinode*i_next,*i_prev;
structinode*i_hash_next,*i_hash_prev;
structinode*i_bound_to,*i_bound_by;
unsignedshorti_count;
unsignedshorti_flags;/*Mountflags(seefs.h)*/
unsignedchari_lock;
unsignedchari_dirt;
unsignedchari_pipe;
unsignedchari_mount;
unsignedchari_seek;
unsignedchari_update;
union{
structpipe_inode_infopipe_i;
structminix_inode_infominix_i;
structext_inode_infoext_i;
structmsdos_inode_infomsdos_i;
structiso_inode_infoisofs_i;
structnfs_inode_infonfs_i;
}u;
};
```

structfile **主要用于与文件系统对应的设备驱动程序使用。当然，其它设备驱动程序也可以使用它。它提供关于被打开的文**

件的信息，定义为：

```
#include$#@60;linux/fs.h$#@62;
structfile{
mode_tf_mode;
dev_tf_rdev;/*neededfor/dev/tty*/
off_tf_pos;/*Curr.posninfile*/
unsignedshortf_flags;/*Theflagsargpassedtoopen*/
unsignedshortf_count;/*Numberofopensonthisfile*/
unsignedshortf_reada;
structinode*f_inode;/*pointertotheinodestruct*/
structfile_operations*f_op;/*pointertothefopsstruct*/
};
```

在结构 file_operations 里，指出了设备驱动程序所提供的入口点位置，分

别是：

(1)lseek，移动文件指针的位置，显然只能用于可以随机存取的设备。

(2)read，进行读操作，参数 buf 为存放读取结果的缓冲区，count 为所要

读取的数据长度。返回值为负表示读取操作发生错误，否则返回实际读取

的字节数。对于字符型，要求读取的字节数和返回的实际读取字节数都必

须是 inode-$#@62;i_blksize 的的倍数。

(3)write，进行写操作，与 read 类似。

(4)readdir，取得下一个目录入口点，只有与文件系统相关的设备驱动程序

才使用。

(5)selec，进行选择操作，如果驱动程序没有提供 select 入口，select 操

作将会认为设备已经准备好进行任何的 I/O 操作。

(6)ioctl，进行读、写以外的其它操作，参数 cmd 为自定义的的命令。

(7)mmap，用于把设备的内容映射到地址空间，一般只有块设备驱动程序使

用。

(8)open，打开设备准备进行 I/O 操作。返回 0 表示打开成功，返回负数表

示失败。如果驱动程序没有提供 open 入口，则只要 /dev/driver 文件存

在就认为打开成功。

(9)release，即 close 操作。

设备驱动程序所提供的入口点，在设备驱动程序初始化的时候向系统进行登

记，以便系统在适当的时候调用。LINUX 系统里，通过调用 register_chrdev

向系统注册字符型设备驱动程序。register_chrdev 定义为：

```
#include$#@60;linux/fs.h$#@62;
#include$#@60;linux/errno.h$#@62;
intregister_chrdev(unsignedintmajor,constchar*name,
structfile_operations*fops);
```

其中，major 是为设备驱动程序向系统申请的主设备号，如果为 0 则系统为此

驱动程序动态地分配一个主设备号。name 是设备名。fops 就是前面所说的对各个

调用的入口点的说明。此函数返回 0 表示成功。返回-EINVAL 表示申请的主设备号

非法，一般来说是主设备号大于系统所允许的最大设备号。返回-EBUSY 表示所申

请的主设备号正在被其它设备驱动程序使用。如果是动态分配主设备号成功，此

函数将返回所分配的主设备号。如果 register_chrdev 操作成功，设备名就会出

现在/proc/devices 文件里。

初始化部分一般还负责给设备驱动程序申请系统资源，包括内存、中断、时

钟、I/O 端口等，这些资源也可以在 open 子程序或别的地方申请。在这些资源不

用的时候，应该释放它们，以利于资源的共享。

在 UNIX 系统里，对中断的处理是属于系统核心的部分，因此如果设备与系

统之间以中断方式进行数据交换的话，就必须把该设备的驱动程序作为系统核心

的一部分。设备驱动程序通过调用 request_irq 函数来申请中断，通过 free_irq

来释放中断。它们的定义为：

```
#include$#@60;linux/sched.h$#@62;
intrequest_irq(unsignedintirq,
void(*handler)(intirq,voiddev_id,structpt_regs*regs),
unsignedlongflags,
constchar*device,
void*dev_id);
voidfree_irq(unsignedintirq,void*dev_id);
```

参数说明：

参数 irq 表示所要申请的硬件中断号。handler 为向系统登记的中断处理子

程序，中断产生时由系统来调用，调用时所带参数 irq 为中断号，dev_id 为申

请时告诉系统的设备标识，regs 为中断发生时寄存器内容。device 为设备名，

将会出现在/proc/interrupts 文件里。flag 是申请时的选项，它决定中断处理

程序的一些特性，其中最重要的是中断处理程序是快速处理程序（flag 里设置

了 SA_INTERRUPT）还是慢速处理程序（不设置 SA_INTERRUPT），快速处理程序

运行时，所有中断都被屏蔽，而慢速处理程序运行时，除了正在处理的中断外，

其它中断都没有被屏蔽。在 LINUX 系统中，中断可以被不同的中断处理程序共享，

这要求每一个共享此中断的处理程序在申请中断时在 flags 里设置 SA_SHIRQ，

这些处理程序之间以 dev_id 来区分。如果中断由某个处理程序独占，则 dev_id

可以为 NULL。request_irq 返回 0 表示成功，返回-INVAL 表示 irq$#@62;15 或

handler==NULL，返回-EBUSY 表示中断已经被占用且不能共享。

作为系统核心的一部分，设备驱动程序在申请和释放内存时不是调用 malloc

和 free，而代之以调用 kmalloc 和 kfree，它们被定

义为：

#include$#@60;linux/kernel.h$#@62;

void*kmalloc(unsignedintlen,intpriority);

voidkfree(void*obj);

参数 len 为希望申请的字节数，obj 为要释放的内存指针。priority 为分配内存操

作的优先级，即在没有足够空闲内存时如何操作，一般用 GFP_KERNEL。

与中断和内存不同，使用一个没有申请的 I/O 端口不会使 CPU 产生异常，也

就不会导致诸如“ segmentationfault"一类的错误发生。任何进程都可以访问

任何一个 I/O 端口。此时系统无法保证对 I/O 端口的操作不会发生冲突，甚至会

因此而使系统崩溃。因此，在使用 I/O 端口前，也应该检查此 I/O 端口是否已有

别的程序在使用，若没有，再把此端口标记为正在使用，在使用完以后释放它。

这样需要用到如下几个函数：

intcheck_region(unsignedintfrom,unsignedintextent);

voidrequest_region(unsignedintfrom,unsignedintexten
t,

constchar*name);

voidrelease_region(unsignedintfrom,unsignedintexten
t);

调用这些函数时问　　篅 rom 表示所申请的 I/O 端口的起始地址；

extent 为所要申请的从 from 开始的端口数；name

为设备名，将会出现在

/proc/ioports 文件里。check_region 返回 0 表示 I/O 端口空闲，否则为正在

被使用。

在申请了 I/O 端口之后，就可以如下几个函数来访问 I/O 端口：

#include$#@60;asm/io.h$#@62;

inlineunsignedintinb(unsignedshortport);

inlineunsignedintinb_p(unsignedshortport);

inlinevoidoutb(charvalue,unsignedshortport);

inlinevoidoutb_p(charvalue,unsignedshortport);

其中 inb_p 和 outb_p 插入了一定的延时以适应某些慢的 I/O 端口。

在设备驱动程序里，一般都需要用到计时机制。在 LINUX 系统中，时钟是由

系统接管，设备驱动程序可以向系统申请时钟。与时钟有关的系统调用有：

#include$#@60;asm/param.h$#@62;

#include$#@60;linux/timer.h$#@62;

voidadd_timer(structtimer_list*timer);

intdel_timer(structtimer_list*timer);

inlinevoidinit_timer(structtimer_list*timer);

structtimer_list 的定义为：

structtimer_list{

structtimer_list*next;

structtimer_list*prev;

unsignedlongexpires;

unsignedlongdata;

void(*function)(unsignedlongd);

};

其中 expires 是要执行 function 的时间。系统核心有一个全局变量 JIFFIES

表示当前时间，一般在调用 add_timer 时 jiffies=JIFFIES+num,表示在 num 个

系统最小时间间隔后执行 function。系统最小时间间隔与所用的硬件平台有关，

在核心里定义了常数 HZ 表示一秒内最小时间间隔的数目，则 num*HZ 表示 num

秒。系统计时到预定时间就调用 function，并把此子程序从定时队列里删除，

因此如果想要每隔一定时间间隔执行一次的话，就必须在 function 里再一次调

用 add_timer。function 的参数 d 即为 timer 里面的 data 项。

在设备驱动程序里，还可能会用到如下的一些系统函数：

#include$#@60;asm/system.h$#@62;

#definecli()__asm____volatile__("cli"::)

#definesti()__asm____volatile__("sti"::)

这两个函数负责打开和关闭中断允许。

#include$#@60;asm/segment.h$#@62;

voidmemcpy_fromfs(void*to,constvoid*from,unsignedlongn);

voidmemcpy_tofs(void*to,constvoid*from,unsignedlongn);

在用户程序调用 read、write 时，因为进程的运行状

态由用户态变为核心

态，地址空间也变为核心地址空间。而 read、write 中参数 buf 是指向用户程

序的私有地址空间的，所以不能直接访问，必须通过上述两个系统函数来访问用

户程序的私有地址空间。memcpy_fromfs 由用户程序地址空间往核心地址空间

复制，memcpy_tofs 则反之。参数 to 为复制的目的指针，from 为源指针，n

为要复制的字节数。

在设备驱动程序里，可以调用 printk 来打印一些调试信息，用法与 printf

类似。printk 打印的信息不仅出现在屏幕上，同时还记录在文件 syslog 里。

3.3、LINUX 系统下的具体实现

在 LINUX 里，除了直接修改系统核心的源代码，把设备驱动程序加进核心里

以外，还可以把设备驱动程序作为可加载的模块，由系统管理员动态地加载它，

使之成为核心地一部分。也可以由系统管理员把已加载地模块动态地卸载下来。

LINUX 中，模块可以用 C 语言编写，用 gcc 编译成目标文件（不进行链接，作

为*.o 文件存在），为此需要在 gcc 命令行里加上-c 的参数。在编译时，还应该在

gcc 的命令行里加上这样的参数：-D__KERNEL__-DMODULE。由于在不链接时，

gcc 只允许一个输入文件，因此一个模块的所有部

分都必须在一个文件里实现。

编译好的模块*.o 放在/lib/modules/xxxx/misc 下（xxxx 表示核心版本，如

在核心版本为 2.0.30 时应该为/lib/modules/2.0.30/misc），然后用 depmod-a

使此模块成为可加载模块。模块用 insmod 命令加载，用 rmmod 命令来卸载，并可

以用 lsmod 命令来查看所有已加载的模块的状态。

编写模块程序的时候，必须提供两个函数，一个是 intinit_module(void)，

供 insmod 在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。

init_module 返回 0 以表示初始化成功，返回负数表示失败。另一个函数是 void

cleanup_module(void)，在模块被卸载时调用，负责进行设备驱动程序的清除

工作。

在成功的向系统注册了设备驱动程序后（调用 register_chrdev 成功后），

就可以用 mknod 命令来把设备映射为一个特别文件，其它程序使用这个设备的时

候，只要对此特别文件进行操作就行了。

Windows2000 新功能:初级篇

“个性化”菜单

用户在使用计算机的过程中，常常为了某些需要而

不得不安装大量的应用程序，而这些程序大部分会自动加入“开始”菜单中，天长日久，“开始”菜单中列出的应用程序选项会混乱不堪。更要命的是，大多数程序的使用机率相当低，只有极少程序我们会经常用到。为了解决这一问题，Windows2000Professional 首次采用了“个性化”菜单。

简单地说，“个性化”菜单的作用就是不断地监视并显示经常使用的菜单项目，隐藏那些不经常使用的程序选项。当用户要想使用那些隐藏了的菜单时，只须将鼠标指针悬停在双箭头上方，菜单中就会显示出所有可以使用的应用程序条目。这样用户在使用“开始”菜单访问常用程序时将更加迅速。由于 Windows2000Professional 会根据程序的使用频率对“开始”菜单中的程序项进行动态调整，所以，如果某个程序被用户频繁使用，它将逐渐上升到顶行。

我的文档

为了增强对用户文件的管理，Windows2000Professional 把“我的文档”作为所有应用程序保存文件的缺省文件夹(除非某个程序明确要求保存在不同的文件夹中，否则 Windows2000Professional 都会截获保存路径并将其重定向到“我的文档”文件夹)。这样，用户保存和查找信息就有了统一的位置。

“我的文档”加强了用户文件的安全性，它对文件的保存过程都是基于每个用户的，这样处理后，即使是多人共享一台计算机，一个用户也不会看到另一个用户的文档。当然，计算机管理员除外。

我的图片

这是“我的文档”文件夹中新增的一个文件夹，在