Linux设备驱动编程

1.引言

目前，Linux软件工程师大致可分为两个层次：

（1）Linux应用软件工程师（Application Software Engineer）：主要利用 C库函数和 Linux API进行应用软件的编写；

（2）Linux固件工程师（Firmware Engineer）：主要进行 Bootloader、Linux的移植及 Linux设备驱动程序的设计。

一般而言，固件工程师的要求要高于应用软件工程师的层次，而其中的 Linux设备驱动编程又是 Linux程序设计中比较复杂的部分，究其原因，主要包括如下几个方面：

（1）设备驱动属于 Linux内核的部分，编写 Linux设备驱动需要有一定的 Linux操作系统内核基础；

（2）编写 Linux设备驱动需要对硬件的原理有相当的了解，大多数情况下我们是针对一个特定的嵌入式硬件平台编写驱动的；

（3）Linux设备驱动中广泛涉及到多进程并发的同步、互斥等控制，容易出现 bug；

（4）由于属于内核的一部分， Linux设备驱动的调试也相当复杂。

2.Linux内核模块

Linux设备驱动属于内核的一部分， Linux内核的一个模块可以以两种方式被编译和加载：

（1）直接编译进 Linux内核，随同 Linux启动时加载；

（2）编译成一个可加载和删除的模块，使用 insmod加载（ modprobe和 insmod命令类似，但依赖于相关的配置文件），rmmod删除。这种方式控制了内核的大小而模块一旦被插入内核，它就和内核其他部分一样。

内存在 Linux内核模式下，我们不能使用用户态的 malloc()和 free()函数申请和释放内存。进行内核编程时，最常用的内存申请和释放函数为在 include/linux/kernel.h文件中声明的 kmalloc()和 kfree()，其原型为：

void \*kmalloc(unsigned int len, int priority);

void kfree(void \*\_\_ptr);

kmalloc的 priority参数通常设置为 GFP\_KERNEL，如果在中断服务程序里申请内存则要用 GFP\_ATOMIC参数，因为使用 GFP\_KERNEL参数可能会引起睡眠，不能用于非进程上下文中（在中断中是不允许睡眠的）。

模块参数

 2.4内核下，include/linux/module.h中定义的宏MODULE\_PARM(var,type) 用于向模块传递命令行参数。

var为接受参数值的变量名，

type为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。

min及max用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。

在装载内核模块时，用户可以向模块传递一些参数：

insmod modname var=value

如果用户未指定参数，var将使用模块内定义的缺省值。

 3.字符设备驱动程序

#include <linux/module.h> //所有模块都需要的头文件

#include <linux/init.h> // init&exit相关宏

#include <linux/fs.h>

#include <asm/uaccess.h>

MODULE\_LICENSE("GPL"); //声明模块的许可证

#define MAJOR\_NUM 254 //主设备号

static ssize\_t globalvar\_read(struct file \*, char \*, size\_t, loff\_t\*);

static ssize\_t globalvar\_write(struct file \*, const char \*, size\_t, loff\_t\*);

//初始化字符设备驱动的 file\_operations结构体

struct file\_operations globalvar\_fops =

{

read: globalvar\_read, write: globalvar\_write,

};

static int global\_var = 0; //“globalvar”设备的全局变量

/\*

假设一个非常简单的虚拟字符设备：

这个设备中只有一个4个字节的全局变量 int global\_var，而这个设备的名字叫做“ gobalvar”。

对“gobalvar”设备的读写等操作即是对其中全局变量 global\_var的操作。

\*/

//一个Linux内核模块需包含模块初始化和模块卸载函数，前者在insmod的时候运行，后者在rmmod的时候运行。初始化与卸载函数必须在宏module\_init和 module\_exit使用前定义，否则会出现编译错误。

static int \_\_init globalvar\_init(void)

{

int ret;

//注册设备驱动

ret = register\_chrdev(MAJOR\_NUM, "globalvar", &globalvar\_fops);

/\*

register\_chrdev函数中的参数 MAJOR\_NUM为主设备号,“gobalvar”为设备名，gobalvar\_fops为包含基本函数入口点的结构体，类型为 file\_operations。

当 gobalvar模块被加载时，gobalvar\_init被执行，它将调用内核函数 register\_chrdev，把驱动程序的基本入口点指针存放在内核的字符设备地址表中，在用户进程对该设备执行系统调用时提供入口地址。

与模块初始化函数对应的就是模块卸载函数，需要调用register\_chrdev()的“反函数”unregister\_chrdev()

\*/

if (ret)

{

printk("globalvar register failure");

}

else

{

printk("globalvar register success");

}

return ret;

}

static void \_\_exit globalvar\_exit(void)

{

int ret;

//注销设备驱动

ret = unregister\_chrdev(MAJOR\_NUM, "globalvar");

if (ret)

{

printk("globalvar unregister failure");

}

else

{

printk("globalvar unregister success");

}

/\*

每个printk都会有个优先级，内核一共有8个优先级，它们都有对应的宏定义。

如果未指定优先级，内核会选择默认的优先级DEFAULT\_MESSAGE\_LOGLEVEL。

如果优先级数字int console\_loglevel变量小的话，消息就会打印到控制台上。

如果syslogd和klogd守护进程在运行的话，则不管是否向控制台输出，消息都会被追加进/var/log/messages文件。klogd只处理内核消息，syslogd处理其他系统消息，比如应用程序。

\*/

}

static ssize\_t globalvar\_read(struct file \*filp, char \*buf, size\_t len, loff\_t \*off)

{

//将global\_var从内核空间复制到用户空间

/\*

由于内核态和用户态使用不同的内存定义，所以二者之间不能直接访问对方的内存。

而应该使用 Linux中的用户和内核态内存交互函数（这些函数在include/asm/uaccess.h中被声明）：

unsigned long copy\_from\_user(void \*to, const void \*from, unsigned long n);

unsigned long copy\_to\_user (void \* to, void \* from, unsigned long len);

copy\_from\_user、copy\_to\_user函数返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0。

\*/

if (copy\_to\_user(buf, &global\_var, sizeof(int)))

{

return -EFAULT;

}

return sizeof(int);

}

static ssize\_t globalvar\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \*off)

{

//将用户空间的数据复制到内核空间的global\_var

if (copy\_from\_user(&global\_var, buf, sizeof(int)))

{

return -EFAULT;

}

return sizeof(int);

}

module\_init(globalvar\_init);

module\_exit(globalvar\_exit);

mknod /dev/globalvar c 254 0

创建设备节点，用户进程通过/dev/globalvar这个路径就可以访问到这个全局变量虚拟设备了。我们写一个用户态的程序globalvartest.c来验证上述设备：

测试函数：

#include <sys/types.h>

#include <sys/stat.h>

#include <stdio.h>

#include <fcntl.h>

main()

{

int fd, num;

//打开“/dev/globalvar”

fd = open("/dev/globalvar", O\_RDWR, S\_IRUSR | S\_IWUSR);

if (fd != -1 )

{

//初次读globalvar

read(fd, &num, sizeof(int));

printf("The globalvar is %d\n", num);

//写globalvar

printf("Please input the num written to globalvar\n");

scanf("%d", &num);

write(fd, &num, sizeof(int));

//再次读globalvar

read(fd, &num, sizeof(int));

printf("The globalvar is %d\n", num);

//关闭“/dev/globalvar”

close(fd);

}

else

{

printf("Device open failure\n");

}

}

随着内核不断增加新的功能， file\_operations结构体已逐渐变得越来越大，但是大多数的驱动程序只是利用了其中的一部分。

对于字符设备来说，要提供的主要入口有：open ()、 release ()、read ()、write ()、ioctl ()、llseek()、poll()等。

open()函数对设备特殊文件进行 open()系统调用时，将调用驱动程序的 open () 函数： int (\*open)(struct inode \* ,struct file \*);

其中参数 inode为设备特殊文件的 inode (索引结点) 结构的指针，参数 file是指向这一设备的文件结构的指针。

open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用 MINOR(inode-> i - rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0表示成功，负数表示存在错误 ) 等；

release()函数当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release () 函数： void (\*release) (struct inode \* ,struct file \*) ;

release函数的主要任务是清理未结束的输入 /输出操作、释放资源、用户自定义排他标志的复位等。

read()函数当对设备特殊文件进行 read() 系统调用时，将调用驱动程序 read() 函数： ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);

用来从设备中读取数据。当该函数指针被赋为 NULL值时，将导致 read 系统调用出错并返回-EINVAL（“Invalid argument，非法参数”）。

函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台上的固有整数类型）。

write( ) 函数当设备特殊文件进行 write () 系统调用时，将调用驱动程序的 write () 函数：

ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);

向设备发送数据。如果没有这个函数， write 系统调用会向调用程序返回一个 -EINVAL。

如果返回值非负，则表示成功写入的字节数。

ioctl() 函数该函数是特殊的控制函数，可以通过它向设备传递控制信息或从设备取得状态信息，函数原型为：

int (\*ioctl) (struct inode \* ,struct file \* ,unsigned int ,unsigned long);

unsigned int参数为设备驱动程序要执行的命令的代码，由用户自定义，unsigned long参数为相应的命令提供参数，类型可以是整型、指针等。

如果设备不提供 ioctl 入口点，则对于任何内核未预先定义的请求， ioctl系统调用将返回错误（ -ENOTTY，“No such ioctl fordevice，该设备无此 ioctl 命令”）。

如果该设备方法返回一个非负值，那么该值会被返回给调用程序以表示调用成功。

llseek()函数该函数用来修改文件的当前读写位置，并将新位置作为（正的）返回值返回，原型为：

loff\_t (\*llseek) (struct file \*, loff\_t, int);

poll()函数 poll 方法是 poll和 select 这两个系统调用的后端实现，用来查询设备是否可读或可写，或是否处于某种特殊状态，原型为：

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);