# 设备号管理

2.6版本内核前使用register\_chrdev (major, \*name, \*fops)函数来分配字符设备的设备节点，但是这种方式每个主设备只能存放一种设备，也就是说内核最多只能支持255（CHRDEV\_MAJOR\_HASH\_SIZ）个字符设备。

在 2.6 的内核之后，新增如下两个函数：

* register\_chrdev\_region (from, count, \*name)

适用于给定主设备号时的设备号注册

from:需要申请的起始设备号，可通过from=MKDEV(major,minor)获取

* alloc\_chrdev\_region(\*dev, baseminor, count, \*name)

适用于不需要指定主设备号时的设备号注册

dev:设备号指针，由内核进行分配，可通过major=MAJOR(\*dev)和minor=MINOR(\*dev)获取主设备号和次设备号

baseminor：次设备号起始地址

上述两个函数支持同一个主设备号下的次设备号进行分段，每一段供给一个字符设备驱动程序使用，使得资源利用率大大提升，同时，2.6 的内核保留了原有的 register\_chrdev 方法。

内核中的每个字符设备设备号都由如下结构描述：

struct char\_device\_struct {

struct char\_device\_struct \*next;

unsigned int major;//主设备号

unsigned int baseminor;//起始次设备号

int minorct;//次设备号个数

char name[64];//设备名

struct cdev \*cdev; /\* will die \*/

} \*chrdevs[CHRDEV\_MAJOR\_HASH\_SIZE];

内核使用chrdevs管理所有字符设备设备号，数组大小还是255（CHRDEV\_MAJOR\_HASH\_SIZE），那是不是还是最多支持255个字符设备呢，并不是，注意此结构中包含的next指针，它可以指向其他字符设备。真实的chrdevs结构如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| chrdevs[254] | ---> | struct char\_device\_struct | ---> | struct char\_device\_struct |
| : |  |  |  |  |
| : |  |  |  |  |
| chrdevs[1] | ---> | struct char\_device\_struct | ---> | struct char\_device\_struct |
| chrdevs[0] | ---> | struct char\_device\_struct | ---> | struct char\_device\_struct |

接下来我们来看函数的具体实现，其实上述三个函数都是调用了\_\_register\_chrdev\_region进行实现。

register\_chrdev(major, \*name, struct file\_operations \*fops)

\_\_register\_chrdev(major, 0, 256, name, fops);

cd = \_\_register\_chrdev\_region(major, baseminor, count, name);

cdev = cdev\_alloc();

err = cdev\_add(cdev, MKDEV(cd->major, baseminor), count);

它指定了次设备号为0，并申请256个，直接将主设备号下的所有次设备申请光，但是注意其内部封装了cdev\_alloc和cdev\_add，完成了我们下一节要讲的字符设备的注册。

register\_chrdev\_region(from, count, \*name)

cd = \_\_register\_chrdev\_region(MAJOR(n), MINOR(n), next - n, name);

alloc\_chrdev\_region(\*dev, baseminor, count, \*name)

cd = \_\_register\_chrdev\_region(0, baseminor, count, name);

\*dev = MKDEV(cd->major, cd->baseminor);

可以看到上面的这两个函数的区别就是我们上面提到的是否指定主设备号，接下来我们重点来看\_\_register\_chrdev\_region的实现。

char\_device\_struct \*\_\_register\_chrdev\_region(major, baseminor,minorct, \*name)

cd = kzalloc(sizeof(struct char\_device\_struct), GFP\_KERNEL);

if (major == 0) {//判断是否为动态申请

ret = find\_dynamic\_major();

}

//整个cdrdev数组每个节点内的char\_device\_struct首先按照主设备号从小到到进行排序, 如果是主设备号相同则按照所对应的次设备号baseminor以及minorct（baseminor， baseminor+minorct）进行排序，比如（1，59)排到（60，70)之前。

for (curr = chrdevs[i]; curr; prev = curr, curr = curr->next) {

//比较主设备号，小于要注册的主设备号，则继续循环

if (curr->major < major)

continue;

//比较主设备号，大于要注册的主设备号，则跳出循环

if (curr->major > major)

break;

//主设备号相等的话，比较次设备号，次设备号小于的话重新循环

if (curr->baseminor + curr->minorct <= baseminor)

continue;

//大于的话直接跳出循环

if (curr->baseminor >= baseminor + minorct)

break;

//次设备号出现重叠，则出错

goto out;

}

//如果prev为空，则表明该新建的cd节点插入到cdrdev[i]数组中的首个位置。如果不为空则将cd节点插入到prev->next位置。

if (!prev) {

cd->next = curr;

chrdevs[i] = cd;

} else {

cd->next = prev->next;

prev->next = cd;

}

# 字符设备注册

上一节提到只有register\_chrdev在申请设备号的同时注册了字符设备，那么如果使用其他两个设备号管理函数时就需要进行自己注册。

内核使用cdev结构描述字符设备，描述如下:

struct cdev {

struct kobject kobj;

struct module \*owner;//所属模块

const struct file\_operations \*ops;//文件操作结构，单独讲

struct list\_head list;

dev\_t dev;//设备号

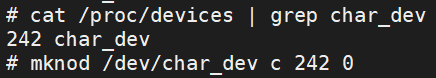
unsigned int count;

}

内核已经提供了一系列函数进行对cdev结构的操作，cdev\_init()初始化cdev，cdev\_add()/cdev\_del()则分别为字符设备的注册与注销。

# 设备节点创建

设备注册完成后发现ls /dev并没有对应设备，此时需要手动创建，首先查看设备的设备号，然后通过mknod创建。此时ls /dev就可以发现char\_dev设备。



如果想自动创建设备节点，可以按照如下步骤操作：

设备是在类下面进行创建的，首先进行类的创建，：

struct class \*\_\_class\_create(\*owner, \*name, \*key)

然后就可以进行设备节点的创建：

struct device \*device\_create(\*class, \*parent, devt, \*drvdata, fmt, ...)

class:上面创建的类，设备会在此类下创建

parent:父设备，没有的话填NULL

drvdata:设备的数据，没有的话填NULL

如果在创建节点时添加了drvdata，或者利用dev\_set\_drvdata()设置了设备数据，此后可以利用dev\_get\_drvdata()在需要此数据的地方拿到此数据进行使用。

# 文件私有数据

文件的私有数据file->private\_data可以用来传递自定义数据，我在char\_dev\_open()中将私有数据进行设置，file->private\_data = dev，在char\_dev\_write()中拿到此数据，为了验证数据传递的正确性，我通过dev\_name(dev->device)打印了私有数据结构中device的设备名称。

在节点创建小结中提到可以给设备device设置数据，然后我通过dev\_get\_drvdata(dev->device)拿到此数据，同样也是通过打印设备名称验证数据传递的正确性。

关于在char\_dev\_open()中的container\_of()，作用为通过结构体成员的指针找到对应结构体的指针。

container\_of(ptr, type, member)

ptr:结构体成员的指针

type:整个结构体的类型

member:第一个参数即结构体成员的类型

# 文件操作结构

此结构中的成员函数是字符设备驱动设计的主要部分，这些函数会在应用程序进行系统调用的时候被相应的函数调用，比如open()就会在应用程序打开设备的时候被调用。我们并不需要实现所有的函数，只需要实现我们用到的即可，下面我会分析部分成员。

struct file\_operations {

struct module \*owner;

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

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

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

ssize\_t (\*read\_iter) (struct kiocb \*, struct iov\_iter \*);

ssize\_t (\*write\_iter) (struct kiocb \*, struct iov\_iter \*);

int (\*iopoll)(struct kiocb \*kiocb, bool spin);

int (\*iterate) (struct file \*, struct dir\_context \*);

int (\*iterate\_shared) (struct file \*, struct dir\_context \*);

\_\_poll\_t (\*poll) (struct file \*, struct poll\_table\_struct \*);

long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);

long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

unsigned long mmap\_supported\_flags;

int (\*open) (struct inode \*, struct file \*);

int (\*flush) (struct file \*, fl\_owner\_t id);

int (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync);

int (\*fasync) (int, struct file \*, int);

int (\*lock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);

unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

int (\*check\_flags)(int);

int (\*flock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);

ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);

int (\*setlease)(struct file \*, long, struct file\_lock \*\*, void \*\*);

long (\*fallocate)(struct file \*file, int mode, loff\_t offset,

loff\_t len);

void (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f);

#ifndef CONFIG\_MMU

unsigned (\*mmap\_capabilities)(struct file \*);

#endif

ssize\_t (\*copy\_file\_range)(struct file \*, loff\_t, struct file \*,

loff\_t, size\_t, unsigned int);

loff\_t (\*remap\_file\_range)(struct file \*file\_in, loff\_t pos\_in,

struct file \*file\_out, loff\_t pos\_out,

loff\_t len, unsigned int remap\_flags);

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

}

.read/write:用于设备数据的读取与写入，成功后返回读取/写入的字节数，内核态和用户态的内存不能直接相互访问，所以需要使用如下函数:

内核空间向用户空间的数据拷贝：

unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n)

用户空间的数据拷贝到内核空间：

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

如果只是单值的交互，无论是char、int还是long，也可以直接使用：

put\_user(x, ptr);//向用户空间指针ptr写入值x

get\_user(x, ptr);//从用户空间指针ptr读取值x

上述函数执行成功返回0。

.llseek:用来修改文件的当前读写位置，并将新位置作为返回值。更改位置的起始点可以为文件开头SEEK\_SET、当前位置SEEK\_CUR、文件末尾SEEK\_END。

.poll:在用户空间进行poll/epoll/select时被调用，用于查询设备是否可被非阻塞地立即读写，如果此函数未实现，则被假定为不阻塞的可读可写。

.unlocked\_ioctl:用于进行I/O控制，可以用来向设备发送命名、设置参数等。

关于ioctl命令的生成采用如下方式进行命令的定义：

|  |  |  |  |
| --- | --- | --- | --- |
| 方向 | 数据 | 设备类型 | 序列号 |
| 2位 | 14位 | 8位 | 8位 |

方向表示数据传输的方向：\_ICO\_NONE(无方向)，\_ICO\_READ(读)，\_ICO\_WRITE(写)， \_ICO\_READ|\_ICO\_WRITE(双向)，方向都是从应用程序得角度来看的。内核已经提供了宏辅助命令的生成：

#define \_IO(type,nr) \_IOC(\_IOC\_NONE,(type),(nr),0)

#define \_IOR(type,nr,size) \_IOC(\_IOC\_READ,(type),(nr),(\_IOC\_TYPECHECK(size)))

#define \_IOW(type,nr,size) \_IOC(\_IOC\_WRITE,(type),(nr),(\_IOC\_TYPECHECK(size)))

#define \_IOWR(type,nr,size) \_IOC(\_IOC\_READ|\_IOC\_WRITE,(type),(nr),(\_IOC\_TYPECHECK(size)))

.mmap:将设备内存映射到进程的虚拟地址空间中，此函数可以使应用程序直接访问内核中的内存，减少了内核与应用之间的内存拷贝。

# seq\_file

sequential file即序列文件是内核中一种特殊类型的文件。当用户访问序列文件时，内核会调用序列文件的操作函数，生成需要输出的数据，并将数据输出到用户空间，这种方式使得序列文件非常灵活，可以根据需要动态生成内容。这种文件类型通常用于实现 /proc 文件系统中的文件，用于向用户空间提供内核状态信息。序列文件由如下结构表示：

struct seq\_file {

char \*buf;//输出缓冲区

size\_t size;//缓冲区的大小

size\_t from;//输出数据的起始索引

size\_t count;//缓冲区中剩余要输出的字节数

size\_t pad\_until;

loff\_t index;//当前索引

loff\_t read\_pos;

u64 version;

struct mutex lock;//用于同步对序列文件访问的互斥锁

const struct seq\_operations \*op;

int poll\_event;

const struct file \*file;

void \*private;//私有数据

};

序列文件的操作方法由如下结构表示，描述中的元素可以理解为数据单元，可以是一个变量或一个结构，也可以是一个链表，多个结构当然也可以作为一个元素直接输出，所以到底是使用1个元素还是多个元素按需而定。

struct seq\_operations {

//用于初始化序列文件并返回一个指针，该指针将在后续的 next、stop 和 show 函数中使用

void \* (\*start) (struct seq\_file \*m, loff\_t \*pos);

//用于释放由 start 分配的资源

void (\*stop) (struct seq\_file \*m, void \*v);

//用于返回序列文件中的下一个元素，并将文件指针位置更新到下一个元素，v指向上一个数据元素

void \* (\*next) (struct seq\_file \*m, void \*v, loff\_t \*pos);

//用于将序列文件中的元素输出到用户空间,v指向当前数据元素

int (\*show) (struct seq\_file \*m, void \*v);

};

使用seq\_file需要包含头文件linux/seq\_file.h，常用的操作接口如下：

打开序列文件，需提供序列文件的操作方法:

int seq\_open(struct file \*file, const struct seq\_operations \*op);

关闭序列文件:

int seq\_release(struct inode \*inode, struct file \*file);

读取序列文件中的数据，并将数据输出到用户空间:

ssize\_t seq\_read(struct file \*file, char \_\_user \*buf, size\_t size, loff\_t \*ppos);

更新文件指针位置:

loff\_t seq\_lseek(struct file \*file, loff\_t offset, int whence);

对于只包含一个数据元素的数据文件，可以使用如下函数创建序列文件：

int single\_open(struct file \*file, int (\*show)(struct seq\_file \*, void \*), void \*data);

只需提供show方法，start/stop/next由内核提供。data为私有数据，会赋值到seq\_file->private。

在show方法中需要向序列文件中写入数据：

将指定长度的数据写入序列文件的缓冲区中：

int seq\_write(struct seq\_file \*seq, const void \*data, size\_t len);

向序列文件的缓冲区中写入格式化的数据：

void seq\_printf(struct seq\_file \*m, const char \*f, ...);