Char Drivers

We'll develop a device driver, **scull**, which treats an area of memory as a device.

There are several types of **scull** devices:

- *scull[0-3]*

  This type has four members, *scull0*, *scull1*, *scull2* and *scull3*.

  Each encapsulates a memory area that is **global** and **persistent**.

  Global means that all opens of these devices share the same data.

  Persistent means that data isn't lost across closes and reopens.

  Command such as *cp*, *cat* and *shell I/O redirection* can be used to access these devices.

- *scullpipe[0-3]*

  Four devices that act like pipes between a reader and writer process.

  Blocking and **non-blocking** reads and writes are illustrated here.

- *scullsingle, scullpriv, sculluid and scullwuid*

  Devices similar to scull0 with certain limitations.

Char Drivers

Major and Minor Numbers:

Char devices are accessed through names (or nodes) in the filesystem, usually in */dev*.

Device files are special files and are identified with a "c" for character and a "b" for block in the first column of the output of *ls -l*:

```
crw-rw----   1 root  daemon  6,   0 May  5  2002 lp0
crw-rw----   1 root  daemon  6,   1 May  5  2002 lp1
brw-rw----   1 root  disk    3,   1 May  5  2002 hda1
```

The major and minor numbers are given by integers before the date.

The **major** number identifies **the driver** associated with the device.

The **minor** number is used ONLY by the device driver, and allow the driver to manage **more than one device**.

Major and Minor Numbers

You must assign a new number to a new driver at driver (module) initialization time using the function defined in <*linux/fs.h*>:

```
int register_chrdev(unsigned int major,
   const char *name, struct file_operations *fops);
```

A negative return value indicates an error, 0 or positive indicates success.

- *major*: the major number being requested (a number < 128 or 256).
- *name*: the name of the device (which appears in */proc/devices*).
- *fops*: a pointer to a global jump table used to invoke driver functions.

How do you give programs a name by which they can request the driver?

Through a device node in */dev* or course.

To create a char device node with major 127 and minor 0, use:

```
mknod /dev/scull0 c 254 0
```

Minor numbers should be in the range of 0 to 255.

Dynamic Allocation of Major Numbers

Some major numbers are statically assigned to the most common devices (see *Documentation/devices.txt* in the linux source tree).

You can choose a major number dynamically by setting the *major* argument to **0** in the call to *register_chrdev*

The problem with this method is that you can't create the device nodes in advance, since the major number may change each time.

This prevents you from using a feature called "loading-on-demand".

For dynamic allocation without loading-on-demand, the number can be obtained from */proc/devices* as a means of creating the device node:

```
#!/bin/sh
module="scull"
device="scull"
group="wheel"
mode="664"
/ * call insmod -- module dynamically obtains # */
/sbin/insmod -f $module $* || exit 1
```

Dynamic Allocation of Major Numbers

```
#Remove old nodes.
rm -f /dev/${device}[0-3]

major=‘awk “\\$2==\”$module\” {print \\$1}” /proc/
devices‘

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

A sample from my /proc/devices looks like:

```
Character devices:
  1 mem
  2 pty
Block devices:
  2 fd
```

Dynamic Allocation of Major Numbers

This script can be called at boot time from */etc/rc.d/rc.local* or invoked manually.

You MUST release the major number when the module is unloaded in *cleanup_module* using:

```
int unregister_chrdev(unsigned int major,
   const char *name);
```

Here, the kernel compares *name* with the registered name for the major number, and if they differ, it returns -EINVAL.

Failing to unregister the device when you unload the driver results in an unrecoverable (reboot) problem !

You should also remove the nodes created by the script given earlier, when you unload the device.

Otherwise, another device may be loaded using the same major number!

Minor Numbers

When the kernel calls the driver, it tells the driver what device is being acted upon using a combined major/minor number pairing.

This number is saved in the *inode* field **i_rdev**, that every driver function receives a pointer to.

The data type is *dev_t*, declared in <*sys/types.h*>

The kernel uses a different type internally called *kdev_t* in <*linux/kdev_t.h*>

The following macros can be used to extract/convert the numbers:

```
MAJOR(kdev_t dev);
MINOR(kdev_t dev);
MKDEV(int major, int minor);
kdev_t_to_nr(kdev_t dev); /* Convert to dev_t */
to_kdev_t(int dev);       /* And back again */
```

Minor Numbers

File Operations

A device is identified internal to the kernel through a *file* structure.

The *file* structure contains a *file_operations* structure (table of function pointers defined in <*linux/fs.h*>) to allow the kernel to call the driver's functions.

The *fops* pointer, passed as an arg to **register_chrdev**, is a pointer to the table. It contains function pointers to *open*, *read*, etc. and NULL pointers for operations that are not supported.

The functions in `struct file_operations`: (2.4 kernel)

```
loff_t (*llseek) (struct file *, loff_t, int);
```

A method used to change the current read/write position in a file.

```
ssize_t (*read) (struct file *, char *, size_t,
   loff_t *);
```

A method used to retrieve data from the device.

A non-negative return value indicates the # of bytes read.

File Operations

```
ssize_t (*write) (struct file *, const char *,
   size_t, loff_t *);
```

Sends data to the device, otherwise the same as read.

```
int (*readdir) (struct file *, void *, filldir_t);
```

NULL for device nodes -- used by filesystems for reading directories.

```
unsigned int (*poll) (struct file *,
   struct poll_table_struct *);
```

(see text)

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

A method that allows device-specific commands to be issued (e.g., formatting a track of a floppy, which is neither reading nor writing).

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

Used to request a mapping of device memory to a process's memory.

File Operations

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

Always the first operation performed on the device node.

If NULL, opening always succeeds.

```
(see text for other operations)

struct module *owner
```

Not a method but rather a pointer to the module that "owns" this structure (used by kernel to maintain usage count).

The following methods are defined for scull:

```
struct file_operations scull_fops = {
   llseek: scull_llseek,
   read: scull_read,
   write: scull_write,
   ioctl: scull_ioctl,
   open: scull_open,
   release: scull_release,
   };
```

This declaration uses the tagged structure initialization syntax.

File Operations

The *file* structure:

struct file appears in <*linux/fs.h*>

It represents an "open file" which is created by the kernel on *open*() and is passed to any function that operates on file.

Note that this is different from *FILE* defined in the C library and used in user programs and a disk file represented by an inode.

Some of the more important fields with `struct file:`

```
mode_t                          f_mode;
```

The mode bits FMODE_READ and FMODE_WRITE indicate whether or not the device can be read or written to.

They may be consulted in the *ioctl*() function (since calls to the *read* and *write* functions are checked by the kernel.)

The file structure

The *file* struct (cont):

`loff_t f_pos;`

loff_t is a 64-bit value (long long) that gives the current reading or writing position.

Drivers can read this but should never update it.

Read and write should update the position using the last argument they receive and not *flp->f_pos* directly.

`unsigned int f_flags;`

These are the file flags, such as O_RDONLY, O_NONBLOCK and O_SYNC.

A driver needs to check the flag for nonblocking operation.

The read/write permission should be checked through *f_mode* field instead.

These constants are defined in <*linux/fcntl.h*>

The file structure

The *fle* struct (cont):

`struct file_operations *f_op;`

A pointer to the operations as discussed above.

The kernel assigns the pointer at *open* time and reads it when it needs to dispatch any operations.

Note that the kernel never saves the pointer *flp->f_op* so you are free to change it at any time and the effect is immediate.

`void *private_data;`

The open system call sets this pointer to NULL before calling the open method of the driver.

The driver is free to define it, e.g. point it to allocated data for use in preserving data across system calls.

The allocated data must be freed in the *release* method.

`struct dentry *f_dentry;`

Provides a means of getting at the inode structure via *flp->f_dentry->d_inode.*

The Open Method

- Checks for device-specific error, e.g. device not ready.
- Initializes the device (if opened for the first time).
- Identifies the minor number and updates the *f_op* pointer in *struct fle* , if necessary.
- Allocates and fills any data structure to be put in *flp->private_data* .
- Increments the usage count for the device.

For scull, most of these tasks depend on the minor number.

The minor number is retrieved using:

```
unsigned int minor = MINOR(inode->i_rdev);
```

Note that different minor numbers can be used to access different physical devices, OR to open the same device in a different way.

For example, */dev/st0* and */dev/st1* refer to different SCSI tape drives (minor #s 0 and 1).

In contrast, */dev/nst0* (minor # 128) and */dev/st0* refer to the same device but behave differently -- nst0 doesn't rewind the tape on a close.

The Open Method

Since device names are not used by the driver (only the number), aliases can be created (symbolic links in this case) for the same device:

```
/dev/mouse -> /dev/psaux
```

The scull driver uses the minor number like this:

- The most significant nibble (4 bits) identifies the type of the devices.
- The least significant nibble distinguishes between devices of the same type.

For example, *scull0* is different from *scullpipe0* in the top nibble, while *scull0* and *scull1* differ in the bottom nibble.

```
#define TYPE(dev) (MINOR(dev) >> 4) /* High nibble */
#define NUM(dev) (MINOR(dev) & 0xF)  /* Low nibble */
```

Each device defines its own *file_operations* structure, which is substituted into *filp->f_op* in the *open* method.

The Open Method

```
struct file_operations *scull_fop_array[] = {
   &scull_fops,         /* Type 0 */
   &scull_priv_fops,    /* Type 1 */
   &scull_pipe_fops,    /* Type 2 */
   &scull_sngl_fops,    /* Type 3 */
   &scull_user_fops,    /* Type 4 */
   &scull_wusr_fops     /* Type 5 */
   };

#define SCULL_MAX_TYPE 5

/* ============================ */
int scull_open(struct inode *inode, struct file
*filp)
   {
   int type = TYPE(inode->i_rdev);
   int num = NUM(inode->i_rdev);

   Scull_Dev *dev;
```

The Open Method

```
/* For device types 1 through 5 */
    if (type)
        {
        if (type > SCULL_MAX_TYPE) return -ENODEV;
/* Set filp->f_op to point to the appropriate list of methods and
   call the open method. */
        filp->f_op = scull_fop_array[type];
        return filp->f_op->open(inode, filp);
        }

/* Type 0 devices make it this far.  Check num against global var for
    number of type 0 devices. */
    if ( num >= scull_nr_devs)
        return -ENODEV;

/* Scull_Dev is a data struct used to hold a region of memory. */
    dev = &scull_devices[num];
    filp->private_data = dev;
```

The Open Method

```
/* Increment the usage count before we maybe sleep. */
    MOD_INC_USE_COUNT;

/* Trim to 0 the length of the device if open was write-only. */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY ) {

        if ( down_interruptible(&dev->sem)) {
            MOD_DEC_USE_COUNT;
            return -ERESTARTSYS;
            }

/* Eliminate any existing data. */
            scull_trim(dev);
            up(&dev->sem);
            }

    return 0; /* Success */
    }
```

The Release (Close) Method

Release is responsible for:

- Decrementing the usage count.
- Deallocating memory allocated in *open* pointed to by *flp->private_data* .
- Shut down the device on last close.

```
void scull_release( struct inode *inode, struct file
*filp)
   {
   MOD_DEC_USE_COUNT;
   return 0;
   }
```