

# LDD3, Ch 3

ECEN 427

**BYU** Electrical & Computer  
Engineering  
IRA A. FULTON COLLEGE OF ENGINEERING

- The goal of this chapter is to write a complete char device driver.
  
- We develop a character driver because this class is suitable for most simple hardware devices. Char drivers are also easier to understand than block drivers or network drivers (which we get to in later chapters).

The textbook walks through the creation of a driver called *scull*.

- “scull is a char driver that acts on a memory area as though it were a device.”

The scull source implements the following devices:

#### *scull0 to scull3*

Four devices, each consisting of a memory area that is both global and persistent. Global means that if the device is opened multiple times, the data contained within the device is shared by all the file descriptors that opened it. Persistent means that if the device is closed and reopened, data isn't lost. This device can be fun to work with, because it can be accessed and tested using conventional commands, such as *cp*, *cat*, and shell I/O redirection.

#### *scullpipe0 to scullpipe3*

Four FIFO (first-in-first-out) devices, which act like pipes. One process reads what another process writes. If multiple processes read the same device, they contend for data. The internals of *scullpipe* will show how blocking and non-blocking *read* and *write* can be implemented without having to resort to interrupts. Although real drivers synchronize with their devices using hardware interrupts, the topic of blocking and nonblocking operations is an important one and is separate from interrupt handling (covered in Chapter 10).

#### *scullsingle*

#### *scullpriv*

# Major and Minor Numbers

If you issue the `ls -l` command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```

crw-rw-rw-   1 root    root      1,   3 Apr 11  2002 null
crw-----   1 root    root     10,   1 Apr 11  2002 psaux
crw-----   1 root    root      4,   1 Oct 28 03:04 tty1
crw-rw-rw-   1 root    tty       4,  64 Apr 11  2002 ttys0
crw-rw----   1 root    uucp      4,  65 Apr 11  2002 ttyS1
crw--w----   1 vcsa    tty       7,   1 Apr 11  2002 vcs1
crw--w----   1 vcsa    tty      7, 129 Apr 11  2002 vcsa1
crw-rw-rw-   1 root    root      1,   5 Apr 11  2002 zero


```

- Traditionally, the major number identifies the driver associated with the device.
- The minor number is used by the kernel to determine exactly which device is being referred to.

# dev\_t

Within the kernel, the `dev_t` type (defined in `<linux/types.h>`) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, `dev_t` is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in `<linux/kdev_t.h>`. To obtain the major or minor parts of a `dev_t`, use:


```
MAJOR(dev_t dev);  
MINOR(dev_t dev);
```



**Get** major/minor  
from `dev_t`

If, instead, you have the major and minor numbers and need to turn them into a `dev_t`, use:

```
MKDEV(int major, int minor);
```



**Create** `dev_t` from  
major/minor  
numbers.



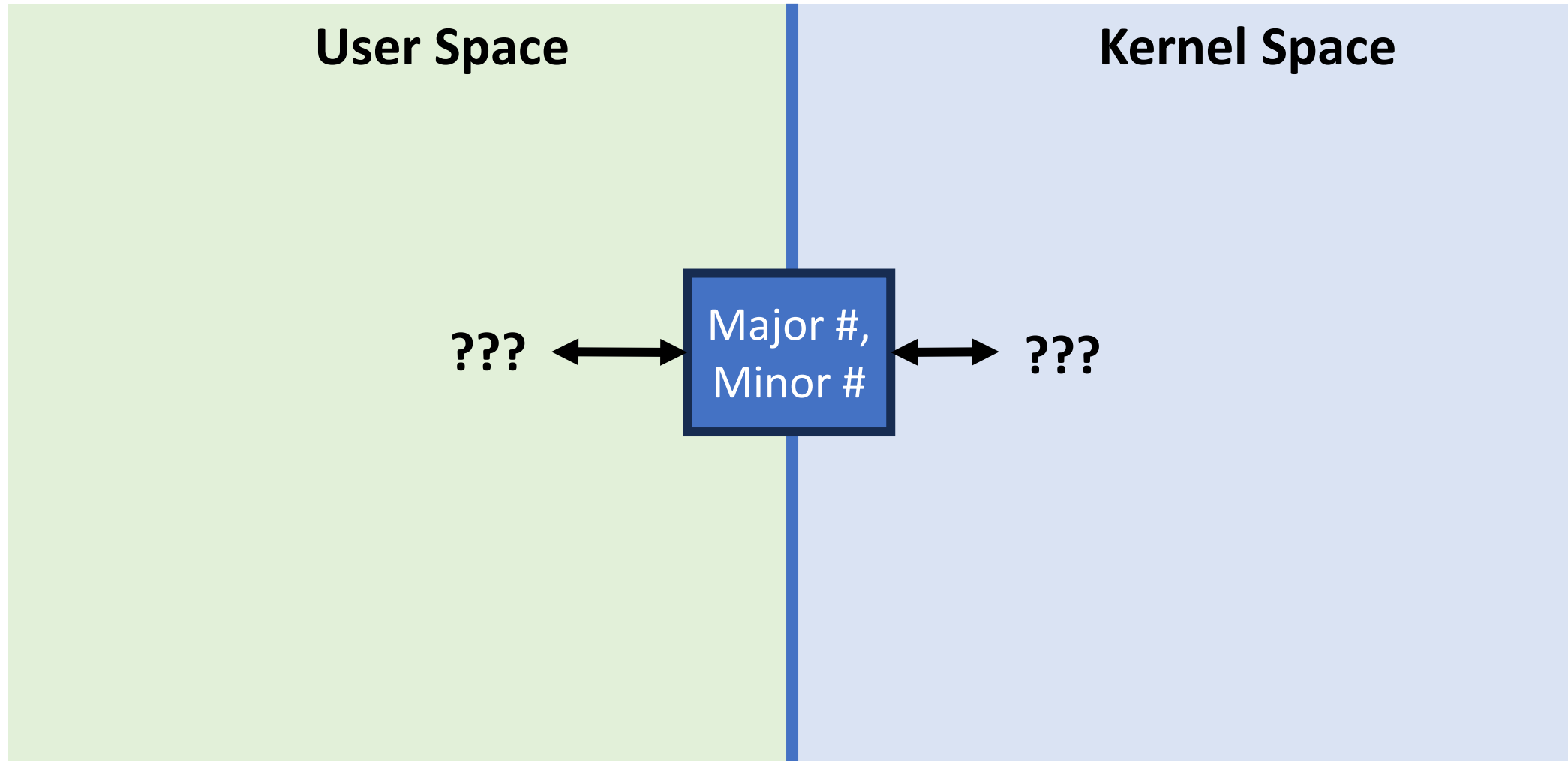
# Allocating Device Numbers

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,  
                        unsigned int count, char *name);
```

- With this function, **dev** is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
- **firstminor** should be the requested first minor number to use; it is usually 0.
- **count** is the total number of contiguous device numbers you are requesting.
- Finally, **name** is the name of the device that should be associated with this number range; it will appear in `/proc/devices` and `sysfs`.

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```



The following script, *scull\_load*, is part of the *scull* distribution. The user of a driver that is distributed in the form of a module can invoke such a script from the system's *rc.local* file or call it manually whenever the module is needed.

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod ./${module}.ko $* || exit 1

# remove stale 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

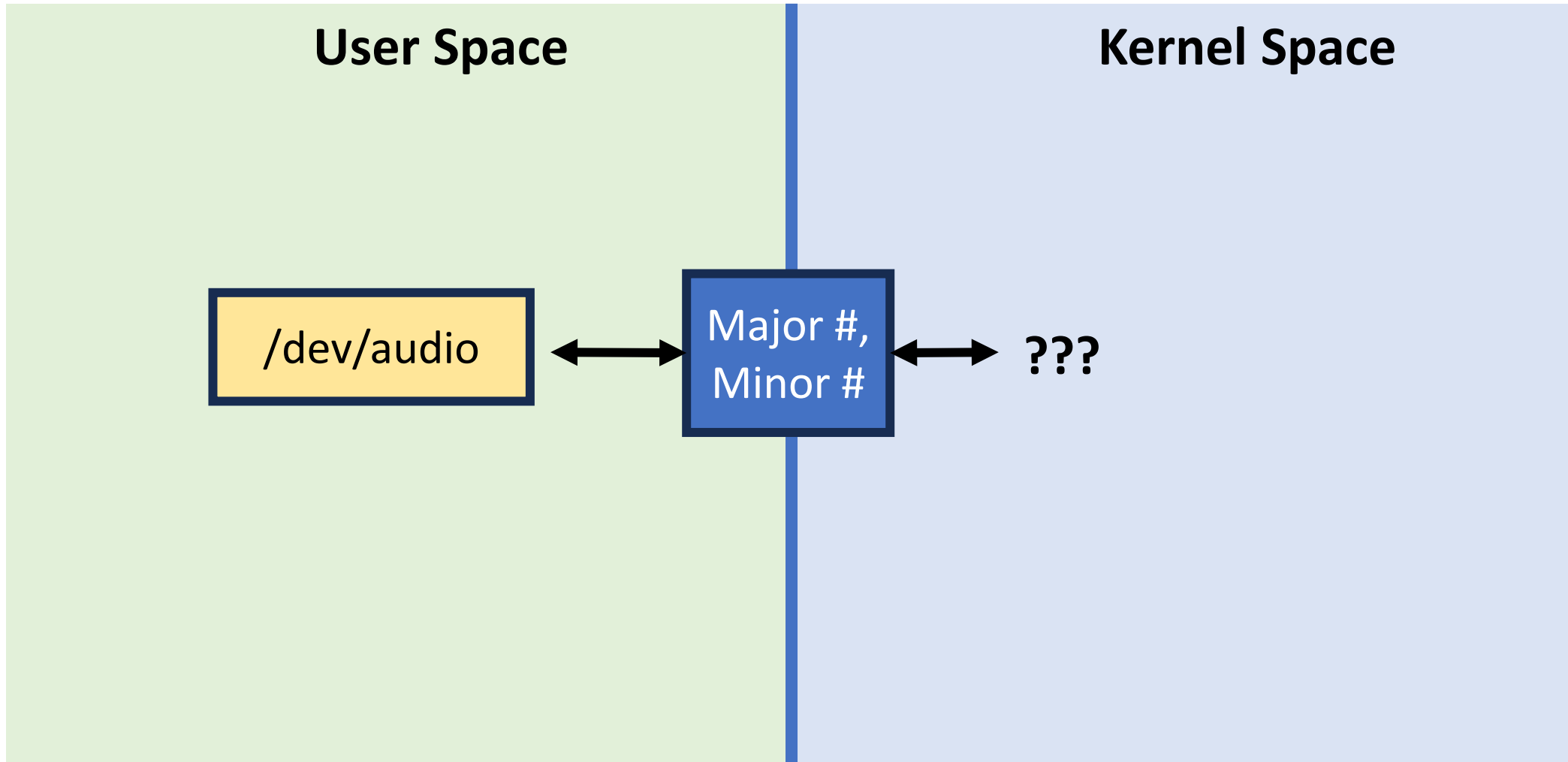
# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"

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

The script can be adapted for another driver by redefining the variables and adjusting the *mknod* lines. The script just shown creates four devices because four is the default in the *scull* sources.

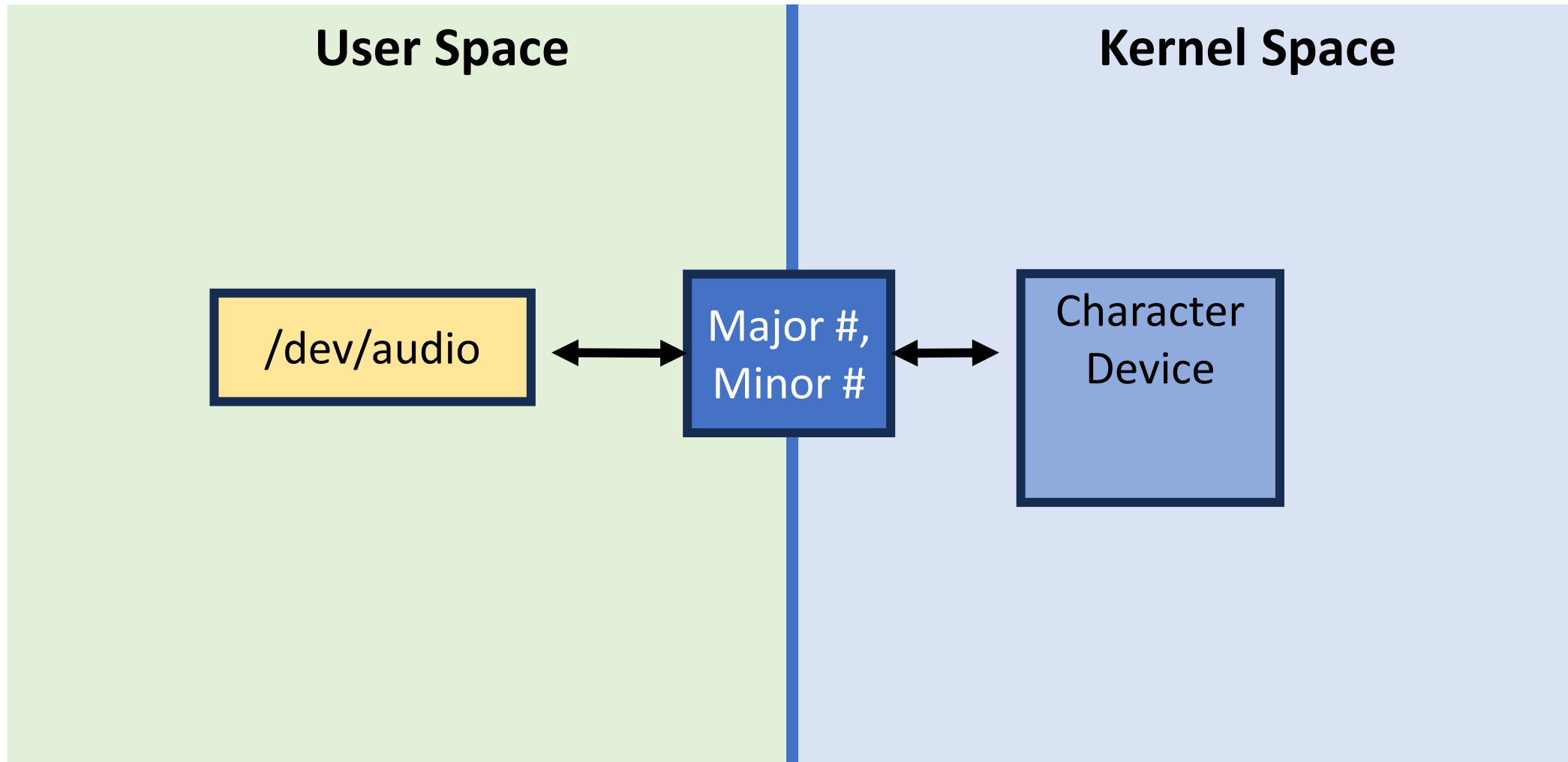
- In the textbook, the device file is created using a “*scull\_load*” shell script called from user space.
- Uses **mknod**
- In lab4, we will trigger the device file creation from within your driver





Next we need to tell Linux about our device...

We will create a new “character device” in the kernel



# Char Device Registration

There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone `cdev` structure at runtime, you may do so with code such as:

```
struct cdev *my_cdev = cdev_alloc();  
my_cdev->ops = &my_fops;
```

Chances are, however, that you will want to embed the `cdev` structure within a device-specific structure of your own; that is what *scull* does. In that case, you should initialize the structure that you have already allocated with:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

<https://elixir.bootlin.com/linux/v5.4/source/include/linux/cdev.h#L14>

The only field you need to set yourself:

```
cdev.owner = THIS_MODULE;
```

Once the `cdev` structure is set up, the final step is to tell the kernel about it with a call to:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Here, `dev` is the `cdev` structure, `num` is the first device number to which this device responds, and `count` is the number of device numbers that should be associated with the device.

To remove a char device from the system, call:

```
void cdev_del(struct cdev *dev);
```

Clearly, you should not access the `cdev` structure after passing it to `cdev_del`.

Internally, *scull* represents each device with a structure of type `struct scull_dev`. This structure is defined as:

```
struct scull_dev {
    struct scull_qset *data; /* Pointer to first quantum set */
    int quantum;           /* the current quantum size */
    int qset;              /* the current array size */
    unsigned long size;    /* amount of data stored here */
    unsigned int access_key; /* used by sculluid and scullpriv */
    struct semaphore sem;  /* mutual exclusion semaphore */
    struct cdev cdev;      /* Char device structure */
};
```

- For lab4, we will do something similar, and have our own “struct” for each device in the driver.
  - (Our driver will only have 1 device) 😊



# File Operations

- The fops\* we provide to the character device contains a struct of pointers to different functions in our driver for any file operations we want to support.
- We can leave function pointers NULL for unsupported operations. “The exact behavior of the kernel when a NULL pointer is specified is different for each function”

<https://elixir.bootlin.com/linux/v5.4/source/include/linux/fs.h#L1814>

`struct module *owner`

The first `file_operations` field is not an operation at all; it is a pointer to the module that “owns” the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to `THIS_MODULE`, a macro defined in `<linux/module.h>`.

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

The `llseek` method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The `loff_t` parameter is a “long offset” and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If this function pointer is `NULL`, seek calls will modify the position counter in the file structure (described in the section “The file Structure”) in potentially unpredictable ways.

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

Used to retrieve data from the device. A null pointer in this position causes the `read` system call to fail with `-EINVAL` (“Invalid argument”). A nonnegative return value represents the number of bytes successfully read (the return value is a “signed size” type, usually the native integer type for the target platform).

`ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);`

Initiates an asynchronous read—a read operation that might not complete before the function returns. If this method is `NULL`, all operations will be processed (synchronously) by `read` instead.

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

Sends data to the device. If NULL, -EINVAL is returned to the program calling the *write* system call. The return value, if nonnegative, represents the number of bytes successfully written.

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

The *ioctl* system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few *ioctl* commands are recognized by the kernel without referring to the fops table. If the device doesn't provide an *ioctl* method, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device").

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

*mmap* is used to request a mapping of device memory to a process's address space. If this method is NULL, the *mmap* system call returns -ENODEV.

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

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

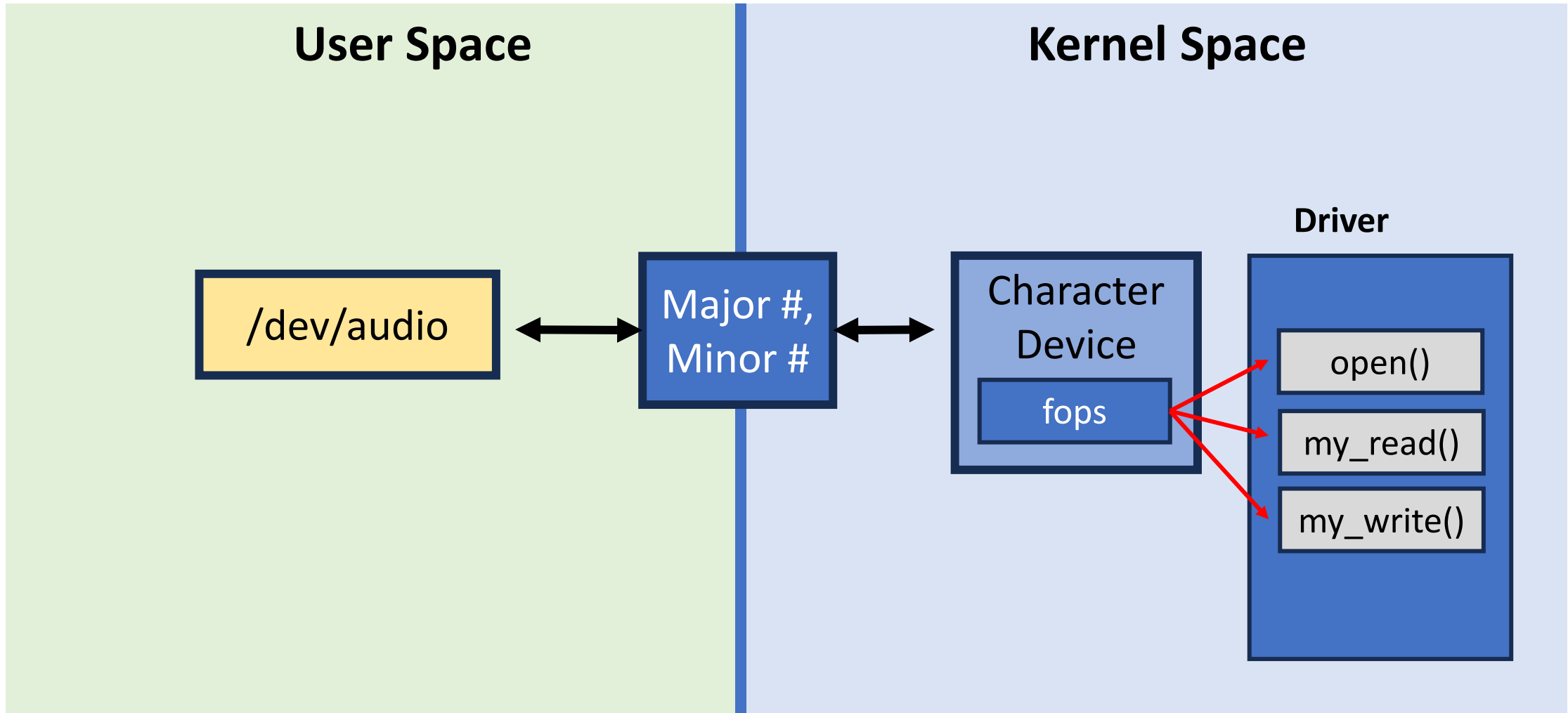
```
int (*release) (struct inode *, struct file *);
```

This operation is invoked when the file structure is being released. Like *open*, *release* can be NULL.\*

The *scull* device driver implements only the most important device methods. Its `file_operations` structure is initialized as follows:

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

- Will this initialize the other function pointers to NULL?



# struct file

`struct file`, defined in `<linux/fs.h>`, is the second most important data structure used in device drivers. Note that a `file` has nothing to do with the `FILE` pointers of user-space programs. A `FILE` is defined in the C library and never appears in kernel code. A `struct file`, on the other hand, is a kernel structure that never appears in user programs.

The `file` structure represents an *open file*. (It is not specific to device drivers; every open file in the system has an associated `struct file` in kernel space.) It is created by the kernel on *open* and is passed to any function that operates on the file, until the last *close*. After all instances of the file are closed, the kernel releases the data structure.



`mode_t f_mode;`

The file mode identifies the file as either readable or writable (or both), by means of the bits `FMODE_READ` and `FMODE_WRITE`. You might want to check this field for read/write permission in your *open* or *ioctl* function, but you don't need to check permissions for *read* and *write*, because the kernel checks before invoking your method. An attempt to read or write when the file has not been opened for that type of access is rejected without the driver even knowing about it.

`loff_t f_pos;`

The current reading or writing position. `loff_t` is a 64-bit value on all platforms (long long in *gcc* terminology). The driver can read this value if it needs to know the current position in the file but should not normally change it; *read* and *write* should update a position using the pointer they receive as the last argument instead of acting on `filp->f_pos` directly. The one exception to this rule is in the *llseek* method, the purpose of which is to change the file position.

`void *private_data;`

The *open* system call sets this pointer to `NULL` before calling the *open* method for the driver. You are free to make its own use of the field or to ignore it; you can use the field to point to allocated data, but then you must remember to free that memory in the *release* method before the file structure is destroyed by the kernel. `private_data` is a useful resource for preserving state information across system calls and is used by most of our sample modules.

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

The *inode* argument has the information we need in the form of its `i_cdev` field, which contains the `cdev` structure we set up before. The only problem is that we do not normally want the `cdev` structure itself, we want the `scull_dev` structure that contains that `cdev` structure. The C language lets programmers play all sorts of tricks to make that kind of conversion; programming such tricks is error prone, however, and leads to code that is difficult for others to read and understand. Fortunately, in this case, the kernel hackers have done the tricky stuff for us, in the form of the *container\_of* macro, defined in `<linux/kernel.h>`:

```
container_of(pointer, container_type, container_field);
```

This macro takes a pointer to a field of type `container_field`, within a structure of type `container_type`, and returns a pointer to the containing structure. In *scull\_open*, this macro is used to find the appropriate device structure:

```
struct scull_dev *dev; /* device information */

dev = container_of(inode->i_cdev, struct scull_dev, cdev);
filp->private_data = dev; /* for other methods */
```

# open / release

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

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

“You may be wondering what happens when a device file is closed more times than it is opened. How does a driver know when an open device file has really been closed?”

“The answer is simple: not every close system call causes the release method to be invoked...The kernel keeps a counter of how many times a file structure is being used. The close system call executes the release method only when the counter for the file structure drops to 0”

# read and write

The *read* and *write* methods both perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar, and it's worth introducing them at the same time:

```
ssize_t read(struct file *filp, char __user *buff,  
             size_t count, loff_t *offp);  
ssize_t write(struct file *filp, const char __user *buff,  
              size_t count, loff_t *offp);
```

For both methods, *filp* is the file pointer and *count* is the size of the requested data transfer. The *buff* argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, *offp* is a pointer to a “long offset type” object that indicates the file position the user is accessing. The return value is a “signed size type”; its use is discussed later.

Let us repeat that the *buff* argument to the *read* and *write* methods is a user-space pointer. Therefore, it cannot be directly dereferenced by kernel code. There are a few reasons for this restriction:

- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all. There may be no mapping for that address, or it could point to some other, random data.
- Even if the pointer does mean the same thing in kernel space, user-space memory is paged, and the memory in question might not be resident in RAM when the system call is made. Attempting to reference the user-space memory directly could generate a page fault, which is something that kernel code is not allowed to do. The result would be an “oops,” which would result in the death of the process that made the system call.
- The pointer in question has been supplied by a user program, which could be buggy or malicious. If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space program to access or overwrite memory anywhere in the system. If you do not wish to be responsible for compromising the security of your users’ systems, you cannot ever dereference a user-space pointer directly.

```
unsigned long copy_to_user(void __user *to,  
                           const void *from,  
                           unsigned long count);  
unsigned long copy_from_user(void *to,  
                             const void __user *from,  
                             unsigned long count);
```

“The role of the two functions is not limited to copying data to and from user-space: they also check whether the user space pointer is valid. If the pointer is invalid, no copy is performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied. In both cases, the **return value is the amount of memory still to be copied**. The scull code looks for this error return, and returns-EFAULT to the user if it’s not 0.”

Whatever the amount of data the methods transfer, they should generally update the file position at \*offp to represent the current file position after successful completion of the system call. The kernel then propagates the file position change back into the file structure when appropriate.



```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                  loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr; /* the first listitem */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* how many bytes in the listitem */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;

    /* find listitem, qset index, and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);

    if (dptr == NULL || !dptr->data || !dptr->data[s_pos])
        goto out; /* don't fill holes */

    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

out:
    up(&dev->sem);
    return retval;
}
```