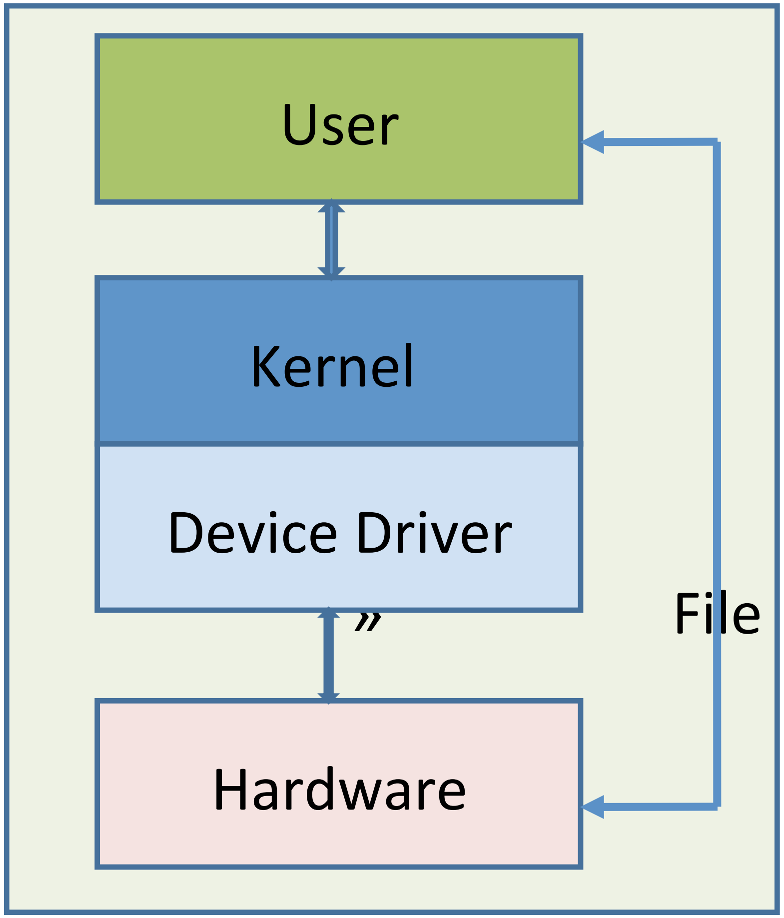
Device Driver Training
Session 2

Contents

- Anatomy of character device driver
- User interface to driver
- Device files
- Kernel interface to device driver
- File operations structure
- Major and minor number
- Registering/De-registering char device driver
- User space drivers

Anatomy of Device drivers

- A device driver has three sides
  - One side talks to the rest of the kernel
  - One talks to the hardware and
  - One talks to the user



Device Drivers

- Device drivers take on a special role in the Linux kernel
- They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works
- The driver translates between the hardware commands understood by the device and the stylized programming interface used by the kernel
- The existence of the driver layer helps to keep UNIX reasonably device-independent
- User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver

Device Drivers cont..

- This programming interface is such that drivers can be built separately from the rest of the kernel and “plugged in” at runtime when needed
- This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available
- Device drivers are part of the kernel; they are not user processes
- However, a driver can be accessed both from within the kernel and from user space
- User-level access to devices is provided through special device file that live in the /dev directory

Device files

- By convention, device files are kept in the **/dev** directory
- Device files are created with the mknod command, which has the syntax:
  - mknod /dev/</device> type[c|b] major minor
    - Filename is the device file to be created.
    - Type is c for a character device or b for a block device.
    - Major and minor are the major and minor device numbers
  - Needs root privileges
  - Coherency between device files and devices handled by the kernel is left to the system developer
- A very important Unix design decision was to represent most of the “system objects” as “file

Device files cont..

- It allows applications to manipulate all “system objects” with the normal file API (open, read, write, close, etc.)
- So, devices had to be represented as “files” to the applications
- This is done through a special artefact called a **device file**
- It a special type of file, that associates a file name visible to user space applications to the triplet (type, major, minor) that the kernel understands All device files are by convention stored in the /dev directory

Major and Minor number

- Device files are mapped to devices via their "major and minor device numbers", values that are stored in the file's inode structure
- The major device number identifies the driver that the file is associated with
- The minor device number identifies which particular device of a given type is to be addressed
- It is often called the unit number or "instance" of the device
- There are two types of device files:
  - Block device files: it is read or written a block (a group of bytes, usually multiple of 512) at a time
  - Character device files: it can be read or written one byte at a time
- Internally the kernel identifies each device by a triplet of information
  - Type (character or block)
  - Major number (typically the category of devices)
  - Minor number (typically the identifier of the device)

Major and Minor number cont…

- Some devices support access via both block and character device files
- Each driver has routines for performing some of the common functions like:
  - Open
  - Close
  - Read
  - Write
  - Ioctl
- Inside the kernel, the addresses of these functions for each driver are stored in a structure called a jump table
- List of Major numbers and associated devices are documented in Linux source code at location Documentation/devices.txt

Major and Minor number cont…

- There are actually two tables:
  - One for character device and
  - One for block devices
- The jump tables are indexed by major device numbers.
- When a program performs an operation on a device file, the kernel automatically catches the reference, looks up the appropriate function name in the jump table, and transfers control to it
- To perform an unusual operation that doesn't have a direct analog in filesystem model (for example, ejecting floppy disk), the ioctl system call is used to pass a message directly from user space into the driver

Device classes

- The Linux way of looking at devices distinguishes between three fundamental device types
- Each module usually implements one of these types, and thus is classifiable as
  - *char module*
  - *block module*
  - *network module*
- This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code
- Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability

Char devices

- A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior.
- Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls
- The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction
- Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*.
- The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially

Block devices

- Like char devices, block devices are accessed by filesystem nodes in the */dev* directory
- A block device is a device (e.g., a disk) that can host a filesystem
- In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length
- Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time
- As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface
- Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user
- Block drivers have a completely different interface to the kernel than char drivers

Network devices

- Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts
- Usually, an *interface* is a hardware device, but it might also be a pure software device, like the loopback interface
- A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transac- tions map to the actual packets being transmitted
- Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets
- A network driver knows nothing about individual connections; it only handles packets

Implementing character driver

- Four major steps
  - Implement operations corresponding to the system calls an application can apply to a file: file operations
  - Define a "file_operations" structure containing function pointers to system call functions in your driver
  - Reserve a set of major and minors for your driver
  - Tell the kernel to associate the reserved major and minor to your file operations
- This is a very common design scheme in the Linux kernel
- A common kernel infrastructure defines a set of operations to be implemented by a driver and functions to register your driver
- Your driver only needs to implement this set of well-defined operations

File operations

- Before registering character devices, you have to define file_operations (called **fops**) for the device files
- The file_operations structure is generic to all files handled by the Linux kernel
- Here are the most important operations for a character driver. All of them are optional. (include/linux/fs.h)

```
struct file_operations {
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
[...]
};
```

File operations example

```
#include <linux/fs.h>
	static const struct file_operations sample_device_fops = {
		.owner = THIS_MODULE,
		.open = sample_device_open,
		.release = sample_device_release,
		.unlocked_ioctl = sample_device_ioctl,
	};
```

- You need to fill the fops with function your device needs to be supported

File operations cont…

- open: for opening the device(allocating resources)
- release: for closing the device (releasing resources)
- write: for writing data to the device
- read : for reading data from the device
- ioctl: for query the device statistics and passing configuration parameters to device
- mmap: for potentially faster but more complex direct access to the device

Open

- int open(struct inode *i, struct file *f)
  - The *open* method is provided for a driver to do any initialization in preparation for later operations
  - Called when user-space opens the device file
  - It should check for device-specific errors (such as device-not-ready or similar hardware problems)
  - Initialize the device if it is being opened for the first time
  - inode is a structure that uniquely represent a file in the system
  - file is a structure created every time a file is opened
  - The other way to identify the device being opened is to look at the minor number stored in the inode structure

Release

- int release(struct inode *i, struct file *f)
  - Called when user-space closes the file
  - The role of release is reverse of open()
  - It performs all the operation to undo the tasks done in open() such as de-allocating the memory resources allocated at time of open()
  - Shutdown the device on last close

Read

- ssize_t read (struct file *file, __user char *buf, size_t size, loff_t *off)
  - Called when user-space uses the read() system call on the device
  - Must read data from the device
  - write at most ‘size’ bytes in the user-space buffer buf, and
  - update the current position in the file offset
  - “file “ is a pointer to the same file structure that was passed in the open() operation
  - Must return the number of bytes read
  - On UNIX/Linux, read() operations typically block when there isn't enough data to read from the device

Write

- ssize_t foo_write(struct file *file, __user const char *buf, size_t size ,loff_t *off)
  - Called when user-space uses the write() system call on the device
  - The opposite of read, must read at most ‘size’ bytes from buf
  - write it to the device
  - update offset and
  - return the number of bytes written

ioctls

- static long ioctl(struct file *file, unsigned int cmd, unsigned long arg)
  - Associated with the ioctl system call
  - Allows to extend drivers capabilities beyond read/write API.
  - Example:
    - changing the speed of a serial port,
    - setting video output format,
    - querying a device serial number

dev_t data types

- The kernel data type dev_t represent a major/ minor number pair
  - Also called a device number
  - Defined in <linux/kdev_t.h>
- Linux 2.6: 32 bit size (major: 12 bits, minor: 20 bits)
- Macro to compose the device number:
  - MKDEV(int major, int minor);
- Macro to extract the minor and major numbers:
  - MAJOR(dev_t dev);
  - MINOR(dev_t dev);

Registering device numbers

```
#include <linux/fs.h>
int register_chrdev_region(
    dev_t from,             /* Starting device number */
    unsigned count,   /* Number of device numbers */
    const char *name);          /* Registered name */
```

- Returns 0 if the allocation was successful
- If you don't have fixed device numbers assigned to your driver
  - Better not to choose arbitrary ones as there could be conflicts with other drivers
  - The kernel API offers an alloc_chrdev_region function to have the kernel allocate free ones for you
  - You can find the allocated major number in /proc/devices

Registering device numbers cont..

- Registered devices are visible in /proc/devices:
  - Character devices:
    - 1 mem
    - 4 /dev/vc/0
    - 4 tty
    - And so on..

Registering character device

- The kernel represents character drivers with a cdev structure
- Declare this structure globally (within your module):

```
#include <linux/cdev.h>
static struct cdev char_cdev;
```

- In the init function, initialize the structure

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

Registering character device cont..

- Then, now that your structure is ready, add it to the system:

```
int cdev_add(
        struct cdev *p,      /* Character device structure */
        dev_t dev,           /* Starting device major / minor number */
        unsigned count);     /* Number of devices */
If (cdev_add(&char_cdev, dev_no, device_count))
        printk("Char device registration failed\n");
```

- After this function call, the kernel knows the association between the major/ minor numbers and the file operations
- Your device is ready to be used!

Unregistering character device

- First delete your character device:

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

- Then, and only then, free the device number:

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

- Example :

```
cdev_del(&char_cdev);
unregister_chrdev_region(char_dev, count);
```

Registering and Unregistering classic way

- If you dig through much driver code in the 2.6 kernel, you may notice that quite a few char drivers do not use the cdev interface
- The classic way to register a char device driver is with:
  - int register_chrdev(unsigned int major,

    const char *name,

    struct file_operations *fops);
    - Major is the major number of interest or 0 for dynamic allocation
    - Name is the name of the driver (it appears in */proc/devices*)
    - Fops is the default file_operations structure
  - A call to *register_chrdev* registers minor numbers 0–255 for the given major, and sets up a default cdev structure for each

Registering and Unregistering classic way

- If you use *register_chrdev*, the proper function to remove your device(s) from the system is:
  - int unregister_chrdev(unsigned int major, const char *name);
    - Major and name must be the same as those passed to *register_chrdev*, or the call will fail

Linux error codes

- The kernel convention for error management is
  - Return 0 on success
  - Return a negative error code on failure
    - Convention is to return –EFAULT;
- Error codes
  - include/asm-generic/errno-base.h
  - include/asm-generic/errno.h

Character driver summary

Character driver writer **- Define the file operations callbacks for the device file: read, write, ioctl...** **- In the module init function, reserve major and minor numbers with register_chrdev()** **- In the module exit function, call unregister_chrdev() and**	**Kernel**
System administration - Load the character driver module - Create device files with matching major and minor numbers if needed The device file is ready to use!	User space
System user - Open the device file, read, write, or send ioctl's to it.	User space
Kernel - Executes the corresponding file operations	Kernel

User space drivers

- A Unix programmer who's addressing kernel issues for the first time might be nervous about writing a module
- Writing a user space driver that reads and writes directly to the device ports may be easier
- The advantages of user-space drivers are:
  - The full C library can be linked in
  - The programmer can run a conventional debugger on the driver code without having to go through contortions to debug a running kernel
  - If a user-space driver hangs, you can simply kill it
  - Problems with the driver are unlikely to hang the entire system, unless the hardware being controlled is *really* misbehaving
  - User memory is swappable, unlike kernel memory
  - A well-designed driver program can still, like kernel-space drivers, allow concurrent access to a device
- Examples are - USB drivers can be written for user space – lsusb and Xserver

User space drivers

- The dis-advantages of user-space drivers are:
  - Interrupts are not available in user space
  - Direct access to memory is possible only by *mmap*ping */dev/mem*, and only a privileged user can do that
  - Access to I/O ports is available only after calling *ioperm* or *iopl*.
  - Moreover, not all platforms support these system calls, and access to */dev/port* can be too slow to be effective
  - Both the system calls and the device file are reserved to a privi- leged user
  - Response time is slower, because a context switch is required to transfer information or actions between the client and the hardware
  - Worse yet, if the driver has been swapped to disk, response time is unacceptably long
  - Most important devices can't be handled in user space, including, but not limited to, network interfaces and block device

Thank you