Linux *Kernel Hacking* Free Course 2003

UNIVERSITA' DEGLI STUDI DI ROMA TorVergata Università

D. P. Bovet, M. Cesati

An introduction to I/O drivers

The Unix filesystem

- Unix distinguishes several kinds of files
- I/O devices are represented as device files
- A common programming model holds both for regular files and for device files
- Each supported I/O device has a corresponding device file
- Device files are usually placed in the `/dev` directory

Two kind of device files

- Block device files: disk-like devices, data can be accessed by block number, it is meaningful to have a disk cache
- Char device files: almost all other devices
- Network cards do not have a device file
- When Unix was introduced, I/O on a block device was buffered while I/O on a char device was unbuffered

Standard device file names

- `/dev/hda` - first EIDE disk
- `/dev/hdb3` - third partition of the second EIDE disk
- `/dev/ttyS0` - first serial port
- `/dev/lp0` - first parallel port
- `/dev/psaux` - mouse PS/2
- `/dev/fd0` - floppy disk

The common programming model

- Same system calls applicable both to regular files and to device files
  - `open()`
  - `close()`
  - `lseek()`
  - `read()`
  - `write()`
- For device files only:
  - `ioctl()`

Two simple examples

- Reading data from the mouse: see `readmouse.c`
- A simplified version of the `cp` command: see `copyfile.c`

Major and minor numbers (1)

- Besides its name and type (block or char), each device file has two main attributes:
- Major number (ranging from 1 to 254) to identify the device
- Minor number (ranging from 0 to 254) to identify a specific device among a group of devices that share the same major number

Major and minor numbers (2)

- As far as the kernel is concerned, the name of the device file is irrelevant, only the major and minor numbers are significant
- The same major number is used with different meanings for char and block devices
- For char devices, the same major number is sometimes used to identify widely different devices

Device file's contents

- No data blocks are allocated to device files
- All the information required is stored in the file's inode
- The major and minor numbers are stored in the device file's inode

The devfs naming scheme

- Allows I/O drivers to register devices by name rather than by major and minor numbers
- The kernel provides a default naming scheme: as an example, `/dev/hda` becomes `/dev/disc/disc0`
- Major and minor numbers are still indispensable for Unix systems: NFS server, `find` command, etc.

The Virtual File System (1)

- When a process accesses a regular file, it is accessing data blocks in a disk partition through the filesystem
- When a process accesses a device file, it is just driving a hardware device
- VFS hides the differences between device files and regular files by means of a common file model

The Virtual File System (2)

- When a process accesses a regular file, VFS uses the default file operations of the filesystem that contains the file, for instance Ext2 default file operations
- When a process accesses a device file, VFS uses the `def_blk_fops` or the `def_chr_fops` table of file operations
- The device's major number is used to index one of such table and to retrieve the proper file operations

Device driver (1)

- A software layer that makes a hardware device respond to a well-defined programming interface
- Before using a device driver, the following activities must have taken place:
  - Device registration
  - Device initialization

Device driver (2)

- I/O devices operate asynchronously and independently from the CPU(s)
- An additional task of device drivers is thus to:
  - Monitor I/O operation

Device registration (1)

- Linking the device to the corresponding device file operations
- Two main kernel data arrays of kernel structures called `blkdevs` and `chrdevs` are used to store the registered devices
- As usual, the major number is used to index the proper array element

Device registration (2)

- If a device is statically compiled in the kernel, its registration is performed during kernel initialization
- If it is compiled as a kernel module, its registration is performed while the module is loaded

Example of device registration

- In order to initialize I/O devices of the parallel printer driver class we invoke:
- `register_chrdev(LP_MAJOR, "lp", &lp_fops);`
- The three paramenters are stored in the proper structure of the `chrdevs` array
- When implementing I/O drivers for new character devices we shall make use of the `register_chrdev()` function

Device initialization (1)

- Initialize the data structures and assign the resources needed by the I/O driver
- Contrary to device registration, device initialization is often done at the last possible moment, namely when a process requires to open a device file
- A typical example is the assignment of IRQ lines

Device initialization (2)

- Device initialization is heavily dependent on the device's characteristics
- The same device file may be used by several processes at once
- A counter is usually used by device drivers to keep track of the number of processes that are currently accessing the device file
- When the counter becomes zero, the resources allocated to the I/O driver are released

I/O monitoring

- The duration of an I/O operation is often unpredictable
- When an I/O operation terminates, the status register of the I/O interface must be checked to determine whether the I/O operation was carried out succesfully
- Two main techniques to monitor an I/O operation: polling mode and interrupt mode

Polling mode (1)

- The CPU checks (polls) the device's status register repeatedly until its value signals that the I/O operation has been completed

```
for (;;) {
    if (read_status(device&
        DEVICE_END_OPERATION)
        break;
    if (--count == 0) break;
    }
```

Polling mode (2)

- If the time required to complete the I/O operation is large, it is preferable to relinquish the CPU after each unsuccesful polling:

```
for (;;) {
    if (read_status(device) &
        DEVICE_END_OPERATION)
        break;
    schedule();
    }
```

Interrupt mode (1)

- Can be used only if the device's controller is capable of signaling, via an IRQ line, the end of an I/O operation
- Three main steps:
  - Start the I/O operation
  - Suspend the process until the device issues an I/O interrupt
  - Resume the process and check the outcome of the I/O operation

Interrupt mode (2)

- While the process is sleeping, CPU cycles can be used to run other processes
- Switching a process from running to sleeping and back from sleeping to running requires thousands of instruction executions and invalidates the hardware caches
- Interrupt mode convenient only if the I/O completion time is relatively high

Time-outs

- What happens if the `DEVICE_END_OPERATION` flag is never set, or if the device never issues an I/O interrupt?
- The process will loop forever (polling mode), or it will sleep forever (interrupt mode)
- Software timers can be used to signal a time-out if the I/O operation does not terminate within a given time interval

Software timers

- A software facility that allows functions to be invoked at some future time, after a given time interval has elapsed
- Based on the Programmable Interval Timer (PIT), an I/O circuit that issues a timer interrupt at a fixed frequency (in Linux 2.4, every 10 msec)
- `add_timer()` and `del_timer()` kernel functions available to I/O driver designers

Deferrable functions (1)

- Operations to be carried by an interrupt handler, right after an interrupt has occurred, are performed with the IRQ that caused the interrupt masked
- This may cause problems if the interrupt handling requires too much time
- Classic example: the timer interrupt handler must do a lot of bookkeeping: update CPU's usage statistics, check software timers expirations, etc.

Deferrable functions (2)

- In theses cases, deferrable functions (softirqs, tasklets, or bottom halves) should be used
- These kernel functions are interruptible
- They are invoked:
  - after finishing handling an I/O interrupt
  - when a `ksoftirqd_CPUi` kernel thread is awoken (there is one such thread for each CPU)

Block device drivers

- As stated previously, block device drivers are associated with disk-like devices
- They make use of a complex cache system and of read-ahead techniques
- We won't deal with block device drivers
- We might try, however, to implement a rudimentary cacheless I/O driver for, say, a floppy disk

Character device drivers

- We shall focus our attention on device drivers that do not require a cache system, that is, on character device drivers
- These device drivers can, in turn, be divided in:
  - Unbuffered drivers: slow devices
  - Buffered drivers: fast devices that produce or consume data at a high rate

Planning the next lectures

- Start with a simple I/O device and correspondingly simple I/O driver
- Proceed with a second simple I/O device and design an interrupt based I/O driver
- Describe a third complex I/O device and design an interrupt based I/O driver that uses DMA and buffers