Linux Kernel Hacking Free Course 2003

UNIVERSITA' DEGLI STUDI DI ROMA TorVergata Università

D. P. Bovet, M. Cesati

Driver for a sound card, part II

Writing a PCI driver in 5+3 steps

#1: initialize a table of PCI identifiers

#2: initialize a PCI device driver descriptor

#3: initialize the PCI peripheral and the device driver

#4: release the device driver

#5: load and unload the module

#6: create an interface for the User Mode programs

#7: handle the interrupts

#8: handle the DMA

Creating an interface for UM apps

The *device file* is the most common interface between a device driver and the User Mode applications

However, since the device driver handles several components of the sound card, a single device file is not enough

Furthermore, some standardization is required, otherwise existing User Mode applications could not make use of the sound card

We'll use the *soundcore* functions of the Linux kernel

The file operations

Any device file is associated with a set of file operations:

```
#include <linux/fs.h>

struct file_operations foo_fops = {
    owner: THIS_MODULE,
    open: foo_open,
    read: foo_read,
    write: foo_write,
    llseek: no_llseek,
    ioctl: foo_ioctl,
    release: foo_release,
};
```

`no_llseek()` is a small function that returns the `-ESPIPE` error code

The major number

To select a major number for the device file:

- Use numbers 42, 240-254 which are reserved for demo/sample code and for local/experimental code
- Use number 0, so as to dynamically allocate a free major number
- Request an official major number
- Use a major number owned by a whole class of devices; for instance, char major 14 is allocated to the sound cards by the *soundcore* kernel layer

Since we'll use the *soundcore* kernel layer, our device files will have major number 14

Minor numbers

Since there can be several sound cards in the system, and any sound card sports several devices, the minor number is quite overloaded

The device driver is associated with several device files having the same major number and different minor numbers

Device file	Minor number
`/dev/dsp`i	3+16*i
`/dev/audio`i	4+16*i
`/dev/mixer`i	0+16*i
`/dev/midi`i	2+16*i

Registering the device files

To register the device files, the device driver invokes:

```
dsp_minor= register_sound_dsp(&dspfops,-1);
mixer_minor= register_sound_mixer(&mixerfops,-1);
midi_minor= register_sound_midi(&midifops,-1);
```

The value -1 means " use the first free card slot *i* "

To unregister the device files:

```
unregister_sound_dsp(dsp_minor);
unregister_sound_mixer(mixer_minor);
unregister_sound_midi(midi_minor);
```

Handling DSP playback

We shall begin by writing the device driver code that handles the DSP, and specifically the playback function

In particular, we have to write the file operation methods of the `/dev/dsp` device file

- open – what has to be done when a UM process opens the `/dev/dsp` device file?
- release – what has to be done when a UM process closes the `/dev/dsp` device file?
- write – what has to be done when the UM process writes PCM data into the `/dev/dsp` device file?

The `open` and `release` methods

The `open` method of the device file's operations should:

- determine the address of the device driver's descriptor of the peripheral
- grant exclusive access to just one reader process, and to just one writer process (if the card supports full-duplex, they may coexist)
- allocate the required resources (typically, memory buffers)
- initialize the peripheral for the following read or write operations

The `release` method reverts the operations of `open`

The `write` method (*playback*)

The `write` method of the device file's operations should:

1) look for a buffer that has some free space and is not actually being played by the DSP; if there is no such buffer, put the process to sleep

2) copy the data from the User Mode address space to the buffer

3) if the sound card is not already playing, program the DSP to start the data transfer from the buffer

4) if not all data in the User Mode address space have been copied, restart from step 1)

Moving around the sound samples (1)

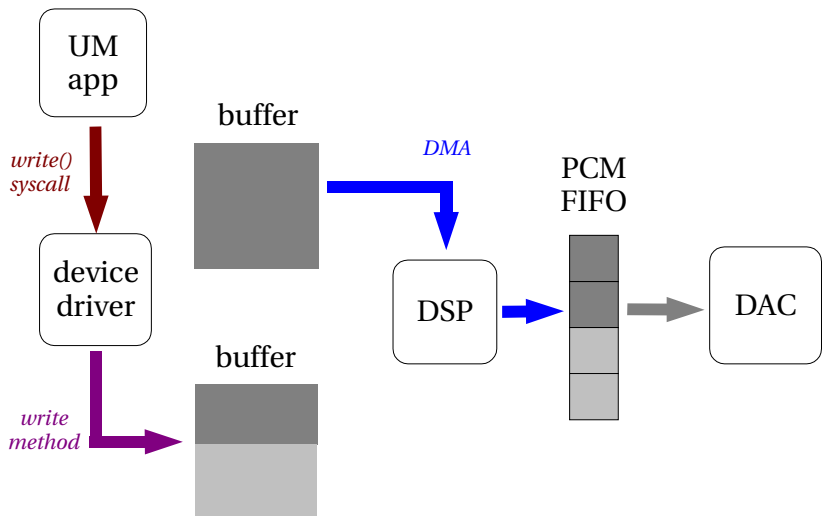
To achieve a decent sound quality, the D/A converter should get sound samples at a constant rate

The sound card includes a 16-byte PCM FIFO, which is filled by the DSP and emptied by the D/A converter

The DSP fills the PCM FIFO with data stored in a device driver buffer (DMA transfer)

While the DSP is accessing a given device driver buffer, the device driver should fill another buffer with the next data to be played

Moving around the sound samples (2)



Direct Memory Access

The address / data bus of the system has to be driven by a device

At any point in time, there could be just one such device, the so-called *bus master*

In very old buses, only the CPU can be the bus master

In old buses, other specialized devices can act as bus masters: the so-called DMAC (Direct Memory Access Controller)

In recent buses like PCI, any peripheral can act as bus master, if provided with the proper circuitry

Synchronous DMA

The usual pattern of operations involved in synchronous DMA are:

1) A process requests a data transfer from/to a peripheral

2) The device driver instructs the peripheral to transfer the data using a given memory buffer

3) If the process issued a read, it is put to sleep; in any case, the CPU continues the execution of some process

4) Meanwhile, the peripheral transfers the data into the memory buffer; once finished, it raises an interrupt

5) The device driver's interrupt handler acknowledges the interrupt (and optionally awakes the sleeping process)

Asynchronous DMA

The usual pattern of operations involved in asynchronous DMA are:

1) The peripheral recognizes that some data is available and raises an interrupt

2) The device driver's interrupt handler acknowledges the interrupt and instructs the peripheral to transfer the data into a given memory buffer

3) The CPU continues its normal execution flow

4) The peripheral transfers the data; once finished, it raises another interrupt

5) The device driver's interrupt handler acknowledges the interrupt and take care of the data

Some examples of DMA usage

- When playing or capturing data, sound cards make use of synchronous DMA
- When receiving a packet from a LAN, network cards make use of asynchronous DMA
- When transmitting a packet to a LAN, network cards make use of synchronous DMA

DMA limits (1)

DMA is very architecture-dependent, so it is hard to write portable device drivers

The main sources of troubles:

- In most architectures, the devices do not make use of the paging tables (no linear addresses!)
- In many architectures, the devices do not make use of physical addresses; rather, they use *bus addresses* (mapped to physical addresses through a limited number of bus registers)

  BUT: in PC-compatible systems, bus addresses and physical addresses coincide (hence, DMA buffers have to be physically contiguous in RAM)

DMA limits (2)

- The original PCI standard states that DMA makes use of 32-bit addresses (buffers only in "low" RAM)
- Many PCI peripherals have stronger constraints on the size of the bus addresses (like ISA devices, which use 24-bit addresses covering the lower 16 MB of RAM)

Luckily, the kernel 2.4 and 2.5 offer a rich set of functions; kernel developers can thus easily write DMA-capable and portable PCI device drivers

Initializing the PCI DMA

When initializing the peripheral, the device driver must:

- Enable the bus mastering capabilities of the peripheral
- Instruct the kernel's PCI layer about the bus address size supported by the peripheral

The operations are performed by invoking:

```
pci_set_master(pcidev);
pci_set_dma_mask(pcidev, mask)
```

and checking for a zero error code

`mask` specifies the bus address size, e.g., `0x00ffffff` for 24-bit addresses

DMA mappings

Every buffer used for DMA transfers has to be associated both with a linear address (used by the CPU) and with a bus address (used by the peripheral's DMA circuitry)

The kernel defines an architecture-independent data type to store a bus address: `dma_addr_t`

The kernel offers two kinds of mapping between linear addresses and bus addresses:

- Consistent DMA mappings
- Streaming DMA mappings

Consistent DMA mappings (1)

A *consistent DMA mapping* is used whenever a DMA buffer has to be mapped forever. Examples:

- buffers for ring descriptors of network cards
- buffers for SCSI command mailboxes

Both the CPU and the peripheral can access the buffer at the same point in time. However, any write operation of the CPU should be immediately visible to the peripheral, and vice versa

Consistent DMA mappings have to be used sparingly, because in some architectures the amount of physical addresses that can be mapped to bus addresses is limited

Consistent DMA mappings (2)

To allocate a buffer and create a new consistent DMA mapping for it:

```
dma_addr_t busadd;
void * linadd;
struct pci_dev * pcidev;
linadd = pci_alloc_consistent(pcidev, size,
                              &busadd);
```

To release a consistent DMA mapping *and* the buffer:

```
pci_free_consistent(pcidev, size,
                    linadd, busadd);
```

Streaming DMA mappings (1)

A *streaming DMA mapping* is used whenever a DMA buffer has to be mapped for a limited time interval. Examples:

- packet buffers for network cards
- filesystem buffers
- sound card buffers

In this mapping, CPU and peripheral cannot access the buffer at the same point in time

They are preferred over consistent DMA mappings because on many architectures they ensure better performances

Streaming DMA mappings (2)

The functions for streaming DMA mappings do not allocate the buffer; it has to be done by the device driver

A DMA buffer can be allocated by means of the:

- page allocator: `alloc_pages()`, `__get_free_pages()` ...
- generic memory allocator: `kmalloc()`, `kmem_cache_alloc()`

Use the `GFP_DMA` flag for allocating in the low 16 MB!

A DMA buffer *cannot* be:

- allocated at compile time
- allocated on the kernel stack

Streaming DMA mappings (3)

To create a new streaming DMA mapping for an already allocated buffer at linear address `linbuf`:

```
dma_addr_t busadd;
struct pci_dev pcidev;
busadd = pci_map_single(pcidev, linbuf,
                        size, direction);
```

`direction` is one of the following:

```
PCI_DMA_TODEVICE
PCI_DMA_FROMDEVICE
PCI_DMA_BIDIRECTIONAL
PCI_DMA_NONE
```

To release a streaming DMA mapping (*not* the buffer!):

```
pci_unmap_single(pcidev,busadd,size,direction);
```

Streaming DMA mappings (4)

Once created a streaming DMA mapping for a buffer:

- the device driver cannot access the buffer any more, until either the mapping is released or the `pci_dma_sync_single()` function is invoked
- the device driver cannot release the mapping or invoke `pci_dma_sync_single()` until any DMA data transfer on the buffer completes

Therefore, in order to transfer data *to* a peripheral, the device driver must *first* fill the buffer and *then* create the mapping