Device Management

Tim Robinson · timothy.robinson@ic.ac.uk · http://www.themoebius.org.uk/

Introduction

I'm writing this device management tutorial assuming that you've got a basic working protected-mode kernel. You don't need to have read my memory management tutorials (numbers one and two) but, if not, I recommend that you do. I'll assume that your kernel has a basic operating environment: a working memory manager and basic run-time library facilities.

I'm going to concentrate on designing your device manager instead of telling you things like what order to program the ATA controller registers. You can get hardware information from manufacturers' specifications or from reading other drivers' source code; however, there are very few tutorials which teach you the overall picture, and there are a lot of badly-designed device interfaces around.

Architectures

This is where the various different kernel architectures become apparent. There are two main categories:

Microkernel
Examples: Minix, QNX
Device drivers are isolated from the kernel, and are usually in their own user-mode address spaces. They communicate with the main kernel and with each other by means of messaging. Passing a message from, say, the kernel to a driver usually involves a process switch (switching address spaces) and a context switch (switching threads) which incurs a relatively high speed penalty. Microkernels' main advantage is stability: if one buggy device driver crashes, all that is affected is access to that device (until the driver is restarted).

Monolithic Kernel
Examples: Linux, Windows NT
Device drivers run as part of the kernel, either compiled in or as run-time loadable modules. Monolithic kernels have the advantage of speed and efficiency: calls to driver functions are simple local calls instead of whole address space switches. However, because it is running in kernel mode, a buggy device driver has the capability to crash the entire system.

The monolithic architecture is probably the most common in today's operating systems because it is usually more efficient. Monolithic kernels can have access to all the kernel's code and data, making it possible to share internal functions and data with device drivers: Windows NT use this to allow drivers access to the kernel run-time library and Linux allows drivers to access pretty much any non-static kernel symbol. By contrast, microkernel drivers are fully self-contained applications, which makes it a lot harder for them to share common functions. Note that the microkernel separation between the kernel and the rest of the OS components isn't restricted to device drivers: for example, Minix has its file system driver and memory manager as separate tasks too. I'll be concentrating on monolithic architectures in this tutorial.

Specifications

Your device manager will be the interface between the device drivers and the both the rest of the kernel and user applications. It needs to do two things:

- Isolate devices drivers from the kernel so that driver writers can worry about interfacing to the hardware and not about interfacing to the kernel
- Isolate user applications from the hardware so that applications can work on the majority of devices the user might connect to their system

In most operating systems, the device manager is the only part of the kernel that programmers really see. Writing a good interface will make the difference between an efficient and reliable OS which works with a variety of devices and an OS which you spend all your own time writing and debugger drivers for.

Here's the capabilities our device manager will have:

- Asynchronous I/O: that is, applications will be able to start an I/O operation and continue to run until it terminates. This is instead of blocking I/O, whereby applications are stalled while I/O operations execute. Blocking I/O can be easily implemented as a special case of asynchronous I/O.
- Plug and Play: drivers will be able to be loaded and unloaded as devices are added to and removed from the system. Devices will be detected automatically on system startup, if possible.

Drivers

Because we want our kernel to be plug-and-play capable, it isn't enough for drivers to be added to the kernel at compile time, as Minix and old Linux do. We must be able to load and unload them at run time. This isn't difficult: it just means we have to extend the executable file interface to kernel mode.

You can implement kernel driver modules any way you like. Linux uses object files (.o) which are linked to the kernel at load time: references and relocations are patched as if the module was being linked by a linker. Windows NT (and WDM, the Windows Driver Model used by Windows since Windows 98/2000) uses full PE dynamic link libraries with the .sys extension. Drivers are loaded into kernel space just like DLLs are loaded into user space. The kernel needs some kind of entry point into the driver; once it has been loaded, the kernel will be able to see all of the driver's code and data (and vice versa). It needs some way of invoking the driver's code and starting the ball rolling.

This is where device enumeration comes in. How does the kernel know which drivers to load? The same driver could be used for several devices: how does the driver know which devices to implement? Linux, Minix and pre-WDM Windows NT avoid this problem by requiring drivers to know how to detect their own hardware. This makes sense, particularly for the pre-PnP systems which were around when these operating systems were designed. The kernel loads all the drivers which it expects to be needed (often via a user-configurable script or Registry) and the drivers take care of notifying the kernel of which devices actually are present.

Today's PCs have more sophisticated hardware:

- A PCI bus makes hardware detection easily: each type of device responds to a unique combination of a 16-bit vendor ID and a device ID.
- An ISA PnP chipset, together with a PnP BIOS, allows a list of installed ISA PnP devices to be compiled.
- An ACPI chipset provides consistent access to all the motherboard devices.
- A USB hub allows the devices connected to it to be queried for their types and capabilities.

So it is possible to write bus drivers for each type, each of which is able to detect the devices connected to that bus. It may even be possible to write an über-bus driver which can detect the buses installed in a machine, or is might be sufficient for the user to specify the buses installed (or even load each bus driver blindly and get it to run its detection routines regardless). Old-fashioned jumpered devices can even have their own virtual bus driver which stores configuration information on disk instead of getting it from the hardware.

If you write bus drivers for each type you need to put an interface in device drivers which defines a callback to be called for each device supported: an “add-device” routine. If not, you need a “main” routine in each driver which is called by the kernel when the driver is loaded and which detects the devices supported.

Interfaces

Once we’ve detected the devices installed in the system we need to keep a record of them somewhere. The standard Unix model, employed by Minix and Linux, is to keep a `/dev` directory somewhere in the file system. This directory is filled with special directory entries – directory entries which don’t point to any data – each of which refers to a specific device via major and minor device numbers. The major device number specifies the device type or driver to use, and the minor number specifies a particular device implemented by that driver. For example, all IDE devices in the system might share one major device number, and each of the minor numbers would refer to one of the machine’s hard disks, CD-ROMs or tape drives. Because entries in the `/dev` directory are persistent between reboots (since they are stored on disk), major device numbers never change and minor device numbers are always allocated the same way. The advantage of keeping device links in a directory on disk is that devices appear as files, making it easy to, say, record waveform data from the microphone (`/dev/dsp`) , pipe it through a filter, and play it back through the speakers. All of this could be accomplished on one command line. BeOS also uses a format similar to this – it has a `/dev` file system – although devices are stored dynamically and are organized in a tree of directories. Although Linux’s use of a static `/dev` directory doesn’t encourage the ability to dynamically add and remove devices, the BeOS `devfs` is fully dynamic and reflects the current configuration of the system without having to tweak the special directory entries.

On the face of it, Windows NT doesn’t expose its devices to the user in the same way as Unix does; a user browsing through Explorer would only see the old MS-DOS-style C:, D: etc. volumes. Internally, however, Windows NT makes its devices available in a way similar to Unix. There exists a kernel namespace separate to the file system through which the system’s devices can be accessed like ordinary files (actually, the normal file system can be accessed via the kernel namespace – the everyday C: drive names are simply links to the physical disk device names) . The Unix and Windows NT systems have one thing in common: once you know the name, a device can be opened and manipulated from user applications: not only with the stream-orientated `read()` and `write()` (or `ReadFile()` and `WriteFile()` under Win32) but as a device in its own right, with `ioctl()` (or `DeviceIoControl()`).

This leads us to the interface between user applications and device drivers. Typically a user app will issue commands to a specific device via the APIs I mentioned just now. It traps into the kernel via a syscall and the kernel picks up the device requested via the file handle given (the kernel will have recorded the device requested somehow when the file was opened). It is then ready to pass the request onto the driver responsible for that device.

The interface between the kernel and the driver is usually that of a table of function pointers. Linux and Windows NT both use a structure representing a device which contains a series of pointers to functions for handling requests to open, close, read, write, etc. On Linux, each pointer generally corresponds to an individual device syscall, right down to the function prototype; on Windows NT, the pointers each point to

generic device control routines. Different routines can be used for each function, but usually similar tasks (e.g. reading and writing) go to the same routine: the driver can tell the requests apart by the information in the I/O Request Packet (IRP) passed to the function.

Asynchronous I/O

Here's the Linux and NT models start to differ. As I mentioned, on receipt of, say, a read syscall, Linux passes control to the driver's read routine. This routine can handle the request immediately and return (as a ramdisk would), or it can start processing the request and put the current application to sleep (as an IDE driver would).

On receipt of a read request, a ramdisk driver would typically:

- Copy memory from the ramdisk to user space
- Return to user mode

An IDE driver would typically:

- Receive the read request and put it on a queue
- If the queue was empty, issue the Read Sectors command to the controller
- Put the task to sleep
- When the IDE interrupt is received, transfer the data from the controller and awaken the task
- When the task awakes control is returned to user mode

The point to note here is that, as far as user code is concerned, the `read()` call doesn't return until the hard disk has finished reading – it doesn't know that other tasks are being scheduled in the meantime. It *definitely* doesn't regain control until it has received the data and an error/success code.

Windows NT is different because it uses an asynchronous I/O model internally. Although the synchronous `ReadFile()` and `WriteFile()` routines are most commonly used, the kernel will return control to the application just after the request is started. The overlapped `ReadFileEx()` and `WriteFileEx()` routines reflect more accurately the way Windows NT works internally (although on Windows 95 these two functions don't work in the majority of cases).

Although Windows NT's asynchronous I/O model is more complex than Linux's it suits NT's multi-threaded operating environment better. An IDE read request on Windows NT might involve the following:

App	Call overlapped `ReadFileEx()`.
Driver	Receive the read IRP from the kernel and put it on a queue. Lock the user buffer in memory and obtain its associated physical addresses. If the queue was empty, issue the Read Sectors command to the controller. Return to user mode.
App	Wait for I/O completion: either go to sleep or do other things.
Driver	When the IDE interrupt is received, transfer the data from the controller to the physical addresses recorded earlier. Notify the I/O manager that the IRP has completed.
App	Completes wait.

An apparently synchronous I/O request still goes through the same procedure, because for blocking I/O the requesting thread is put to sleep until the operation is finished.

Notice that the driver must lock the buffer the user app provided before it queues the request. This highlights an important point with any multi-tasking architecture: the interrupt which reads the data into the buffer can execute under any context. This means that the kernel could start an operation on behalf of an application and switch away from it before the operation is completed (imagine if an IDE hard drive had been put to sleep before a read occurred: to spin it up could take several seconds, during which time the scheduler might be called thousands of times). So once the controller triggers its interrupt the processor could be executing anything, which means that the current address space is not necessarily the same one from where the request originated. The original buffer is inaccessible, at least in the virtual address space. What we must do is save the physical memory addresses associated with the buffer when the request starts (and lock it in memory, to prevent it from being paged out). Then, when the interrupt handler executes, it can map in the physical pages temporarily and place the data there. Note that as the buffer is locked the kernel must make sure that it is all committed: it's no use getting the physical addresses for a block of memory if it doesn't exist in physical memory yet. Note also that a physical address must be obtained for every virtual page in the buffer, because the pages aren't necessarily contiguous in RAM.

The physical addresses obtained can also be useful for devices that use DMA, such as floppy and hard drive controllers and sound cards, because such devices could potentially transfer data straight into the user buffer. However most DMA drivers prefer to use their own static buffers, and copy the data between the user's and their own buffers as necessary. This is because various devices place restrictions on DMA transfer buffers. The buffer must:

- Be contiguous in memory, unless a scatter-gather DMA device is used (which can be programmed with a list of addresses)
- Be located completely in the first 16MB of memory
- Not span a 64KB boundary

The first one applies to PCI (all of these apply to ISA); if you want to use DMA in your drivers you need to write a physical memory allocator which satisfies these conditions. I wrote about this in my first memory tutorial.

Overview

That's by no means everything you'll need by way of device management, but it should be enough to give you a good start. You should now be able to design a good device driver interface for your kernel: the best designs are not the ones which are immediately brilliant, but which leave the most scope for future enhancement. As your kernel grows you'll be glad of a device interface which doesn't need to be redesigned every 6 months.

Created 27/12/01; last updated 01/01/02.
Updated by K.J. 3/30/02.