

Device drivers

A device driver is software which connects the OS I/O subsystem to the underlying hardware of the device.

Devices vary in many dimensions:

- Character-stream or block

- Sequential or random-access

- Sharable or dedicated

- Speed of operation

- read-write, read only, or write only

I/O system calls encapsulate device behaviors in generic classes (device independence requires this).

Device-driver layer hides differences among I/O controllers from kernel.

What services?

The most popular OSs at the moment make devices look very much like files (or streams).

open – make the device usable for a process

read – get input from the device

write – send output to the device

control – send any sort of special command to the device

In UNIX this is `ioctl` (I/O control)

In Windows `DeviceIoControl()`.

close – sever the connection between the device and the process

Some OSs allow drivers to have extra explicit entry points (we could just use `ioctl`).

e.g. Cancel all current requests.

Clear any associated queues.

Shut the device down

Detach the device.

Attach the device.

Connections

There are close connections between device drivers and the kernel (or at least other low-level services).

Drivers need to be able to reserve memory (including non-pageable memory), set up interrupt handlers, have privileged access to I/O instructions.

Drivers are loaded and initialized at different times.

On older systems this was always done once at OS boot time.

We want to be able to add and remove devices (and hence drivers) as the system is running.

There are sometimes complicated dependencies between drivers.

Some drivers still need to be loaded at boot time.

What is an obvious example?

How does the OS know?

How does the OS know what types of devices are installed?

The OS installation process probably checks for a variety of “standard” devices.

This is referred to as *autoprobing*.

A table is compiled of all attached devices. This table is then used in system startup to check the presence of the devices.

Administrators can edit configuration files with details of attached devices.

These days, most hardware is designed to explicitly and easily identify itself to the OS.

When new hardware is detected the OS may have to request a driver for the device.

Or there may be a standard driver which will do.

Conflicts

Computers (particularly PC compatibles) have only a small number of I/O ports, DMA channels and interrupt vectors.

Thus it is easy for devices to want to use the same resources.

Bus interfaces help: SCSI, USB, Firewire (IEEE 1394)

One set of resources is shared amongst several devices (the limits of the bus controller or hub).

Plug and play

No hardware addresses to set up.

Supposedly configurable by software.

Devices can identify themselves.

Device drivers can be loaded.

Resources are automatically allocated.

Device is configured and started.

Need to be able to stop devices (and pending requests) and alter their resource requirements on the fly as new devices are added.

Finding the device

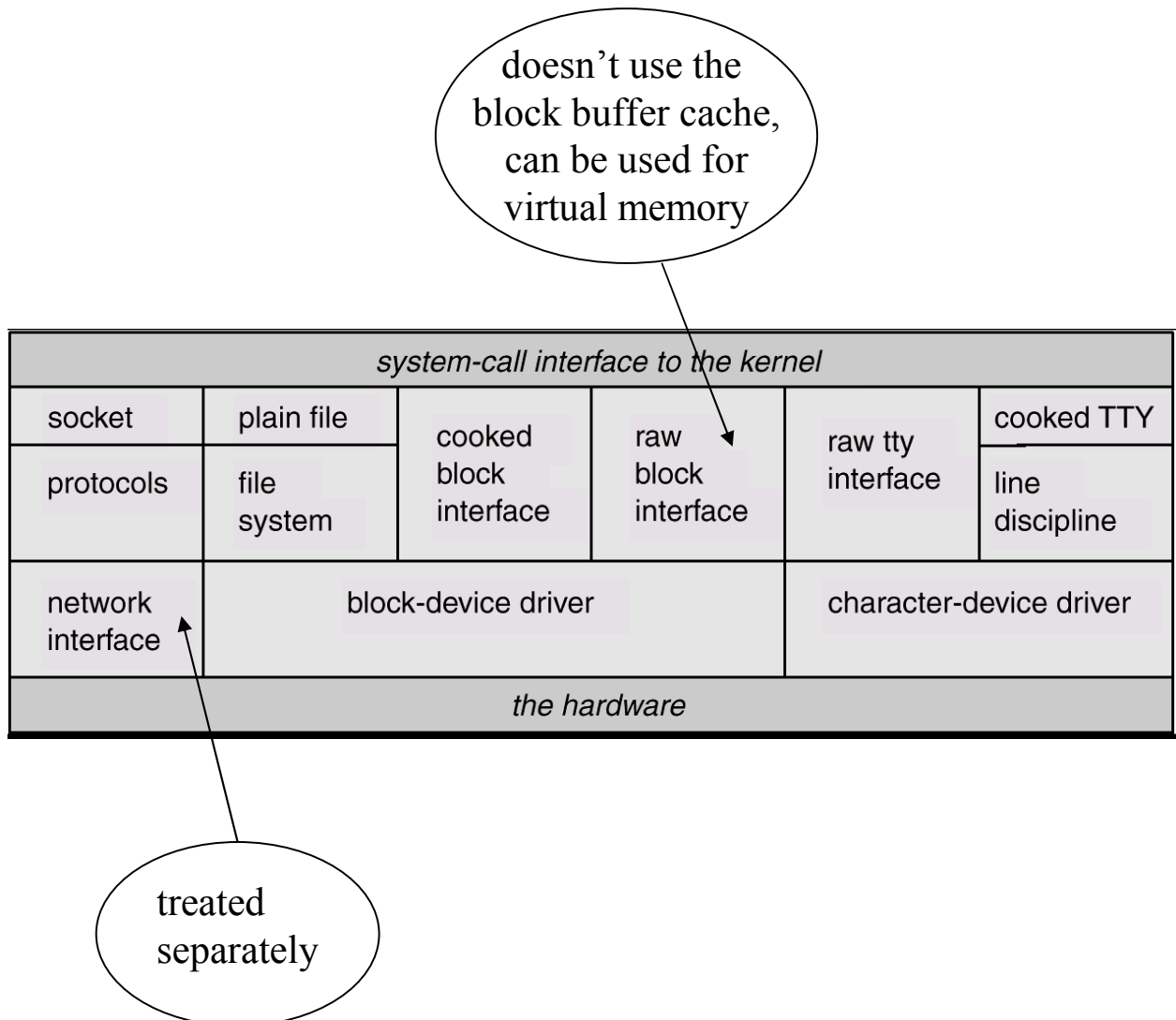
There has to be some way for programs to find the name of the device.

A device driver could publish the device's name to a system-wide table.

In UNIX special files are created in the /dev directory. They are marked as devices (c or b) and have major (which device driver) and minor (which device) device numbers.

```
crw-rw---- 1 root    uucp      5, 64 Apr 14  2001 cua0
crw-rw---- 1 root    uucp      5, 65 Apr 14  2001 cua1
brw-rw---- 1 robert  floppy    2,  0 Apr 14  2001 fd0
brw-rw---- 1 root    disk      3,  0 Apr 14  2001 had
brw-rw---- 1 root    disk      3,  1 Apr 14  2001 hda1
crw-rw---- 1 root    root     10,  0 Apr 14  2001
logimouse
crw----- 1 robert  root     195,  0 Sep 12 19:31 nvidia0
```

UNIX I/O structure



Block devices

Disks, tapes. Can address a complete block (e.g. 4096 bytes)

Block device driver turns block numbers into tracks and cylinders etc.

Transfers are buffered through the block buffer cache.

Block Buffer Cache

Blocks are cached as they are read/written.

A hashtable connects device and block numbers to corresponding buffers.

When a block is wanted from a device, the cache is searched.

If the block is found it is used, and no I/O transfer is necessary.

Also have a raw mode which bypasses the buffer cache.

Character devices

Everything that doesn't use the block buffer cache:

- keyboards

- mice

- printers

- modems

- sound cards

- video cards

- etc

Raw and cooked tty input

Cooked input makes changes to the data before it is passed to the requesting program. e.g.

- deleting characters with backspace

- clearing the whole line

Raw input is passed on exactly as it is (almost) to the program.

Talking to the driver

I/O request block - IORB

A data structure which contains all information needed to complete the I/O request.

Constructed by the I/O system calls and passed to the device drivers. e.g. Read logical block 42.

Called IRPs (I/O request packets) in Windows.

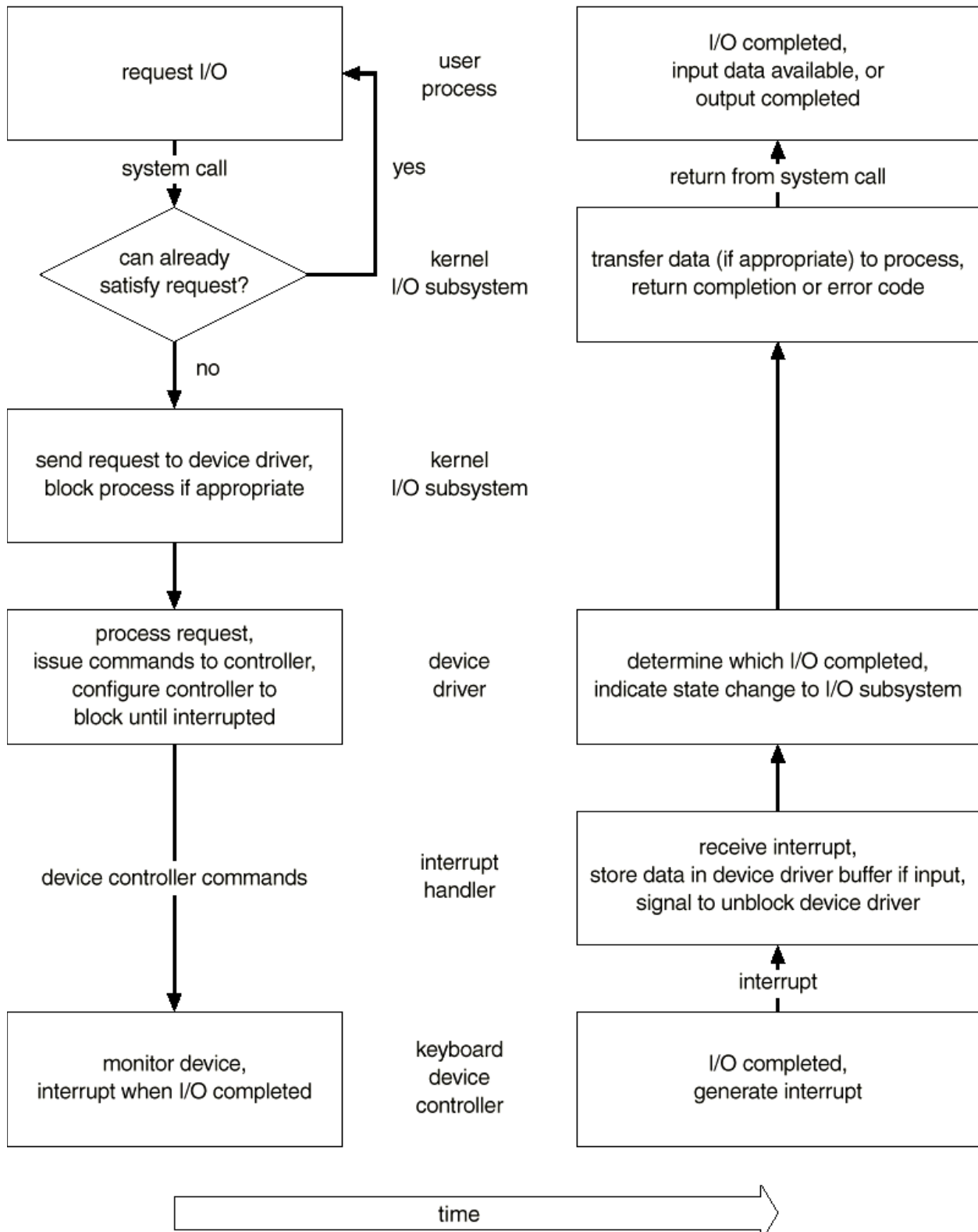
The driver might deal with IORBs immediately

Some devices don't block and completely finish before returning to the caller – e.g. display video cards

or it might queue the requests until they can be handled.

Disk drivers would do this.

Lifecycle



Getting a character from the keyboard

A program (like a word processor) which wants every character from the keyboard uses the keyboard device in the raw mode.

It doesn't want the device driver or kernel mucking around by automatically editing lines (e.g. in response to the backspace key) before passing the information on.

Almost certainly there is still a little system processing which goes on, because of the need to keep some control e.g. stopping the program in response to certain key combinations.

The program requests a character from the keyboard.

The top-half of the keyboard device driver receives the request. The program blocks because there is no data.

The user types a key. This generates an interrupt.

The interrupt handler (the bottom-half of the keyboard device driver) gets the letter "X" say and passes it to the top-half.

The top-half unblocks and returns the character to the program.

GUI programs don't wait the same way

In GUI environments the requests for keyboard and mouse input do not come from the individual application programs.

The user interface manager (in charge of managing the interface - deals with user input and control) catches the keyboard and mouse information and directs it to the correct program. In some systems this is the job of the window manager.

There is the concept of focus – which window (and hence program) receives the packaged-up event corresponding to the generated interrupt.

In Windows, Macintosh and other GUI programs there is an application event loop which waits for events (the Java event loop is an example).

Levels of device drivers

A lot of drivers depend on other device drivers.

e.g. Devices attached to a SCSI bus depend on the SCSI device driver.

So device drivers can be layered on top of each other.

A call to a SCSI hard disk driver will send requests to the disk via the SCSI driver.

In Windows these layered drivers are referred to as mini-port drivers.

Another type of low-level driver can be used to allow high-level drivers access to the hardware.

e.g. the low-level parallel port driver is used by parallel printer drivers to allocate exclusive access to the parallel port. The printer driver then talks directly to the printer via the port.

Trees of devices

So device drivers can be represented as a hierarchy (or tree) of drivers.

This means we can use the same mechanism to add functionality to our devices.

A compression or encryption device driver could be provided as a *filter* over the top of a conventional driver. The underlying driver doesn't have to be aware of the extra layer.

Before next time

Read from the textbook

13.3 Application I/O Interface

20.8 Input and Output

13.5 Transforming I/O to Hardware Operations