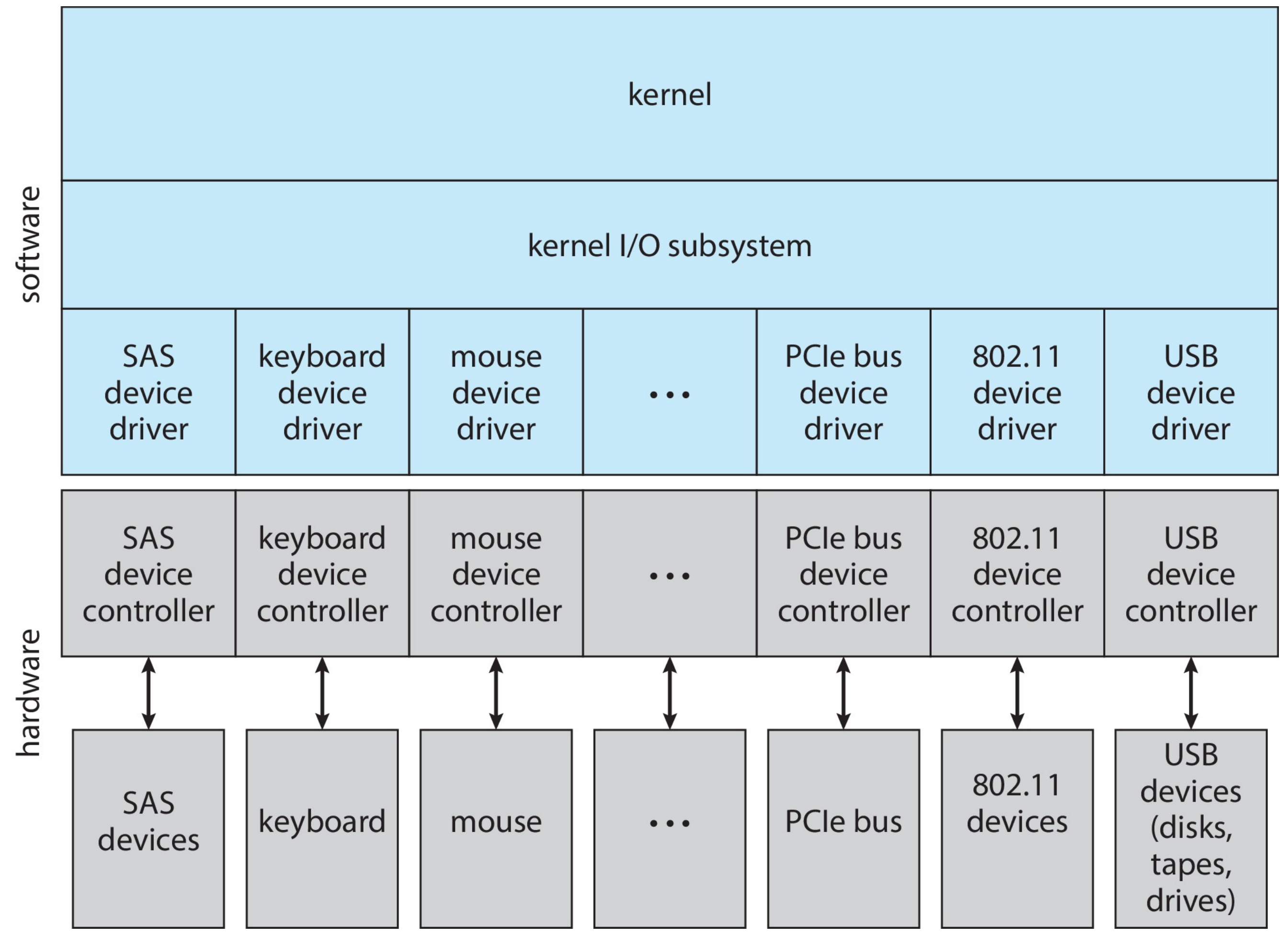
I/O Systems

Read from the textbook

- Ch12.3 Application I/O Interface
- Ch12.5 Transforming I/O Requests to Hardware Operations

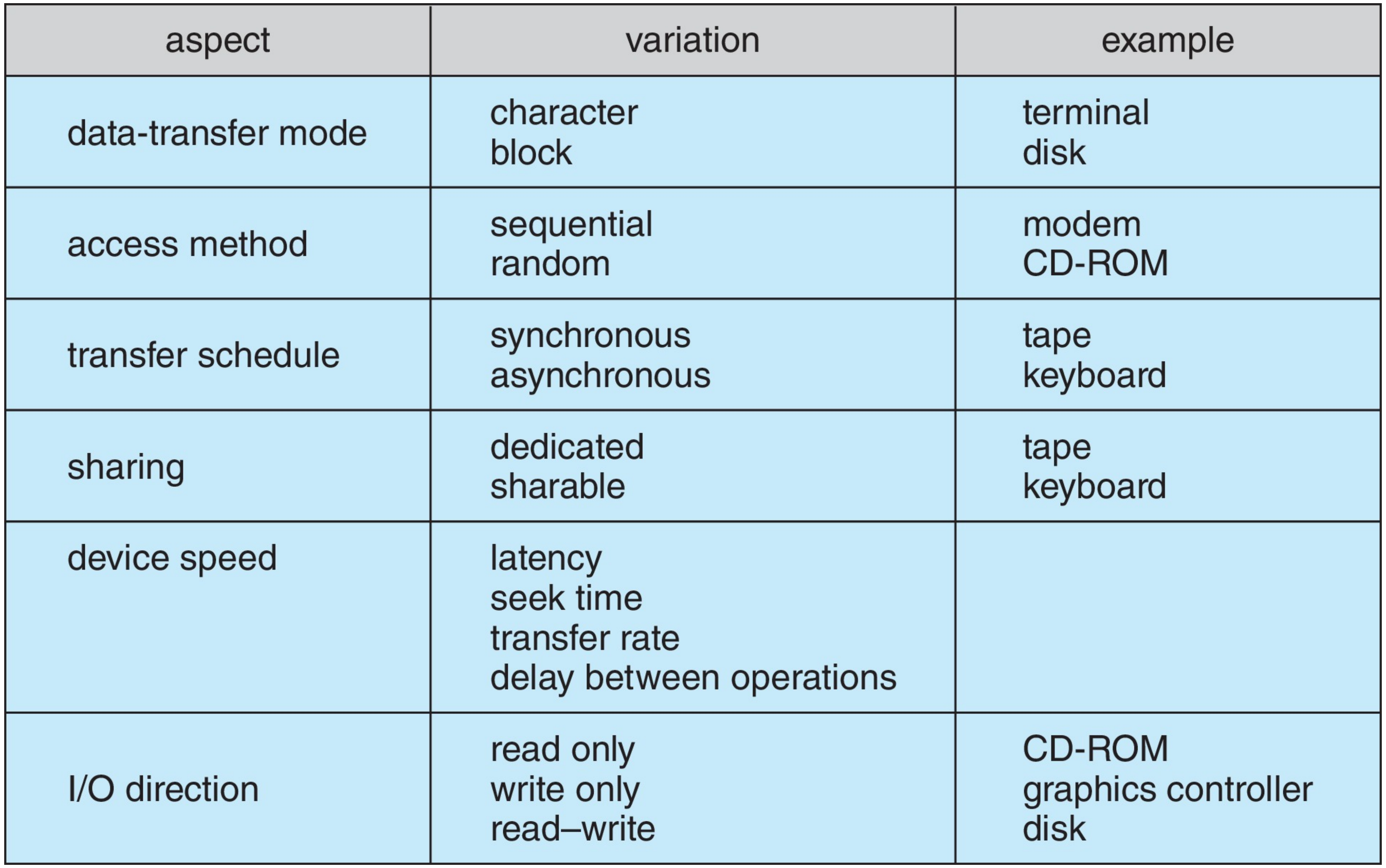
Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- A **device driver** is software which connects the OS I/O subsystem to the underlying hardware of the device



Characteristics of I/O Devices

Devices differ on many different dimensions.



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

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 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.

How does the OS know?

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

- The OS installation process could check for a variety of “standard” devices.
  - This is referred to as device **probing**.
  - 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 have only a small number of I/O ports, DMA channels and interrupt vectors.

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

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

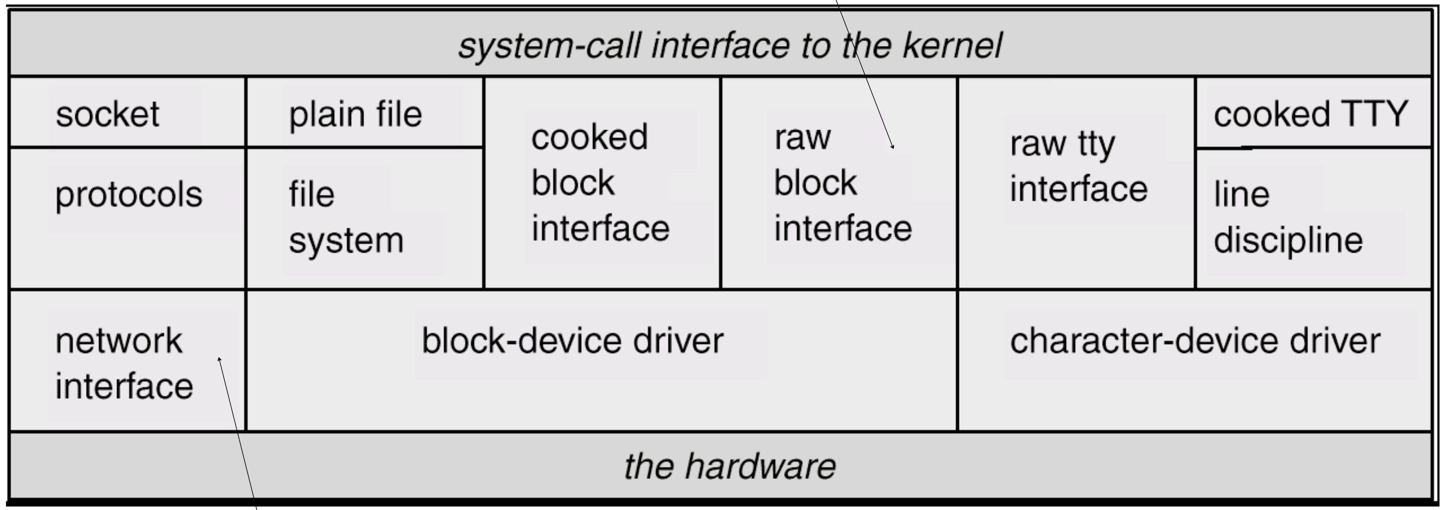
Finding the Device

- 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. This information is in the device's inode.

```
crw-rw-rw-  1 root    wheel      21,  13 17 Oct 09:00 cu.usbserial-FTAJPLHX
brw-r-----  1 root    operator    1,   0 15 Oct 22:29 disk0
brw-r-----  1 root    operator    1,   4 15 Oct 22:29 disk0s1
brw-r-----  1 root    operator    1,   5 15 Oct 22:29 disk0s2
brw-r-----  1 root    operator    1,   1 15 Oct 22:29 disk1
brw-r-----  1 root    operator    1,   3 15 Oct 22:29 disk1s1
brw-r-----  1 root    operator    1,   2 15 Oct 22:29 disk1s2
crw-rw-rw-  1 root    wheel      14,   0 15 Oct 22:29 random
crw-r-----  1 root    operator    1,   0 15 Oct 22:29 rdisk0
crw-r-----  1 root    operator    1,   4 15 Oct 22:29 rdisk0s1
crw-r-----  1 root    operator    1,   5 15 Oct 22:29 rdisk0s2
crw-r-----  1 root    operator    1,   1 15 Oct 22:29 rdisk1
crw-r-----  1 root    operator    1,   3 15 Oct 22:29 rdisk1s1
crw-r-----  1 root    operator    1,   2 15 Oct 22:29 rdisk1s2
crw-rw-rw-  1 root    wheel      21,  12 17 Oct 09:00 tty.usbserial-FTAJPLHX
crw-rw-rw-  1 root    wheel       4,   0 15 Oct 22:29 ttyp0
crw-rw-rw-  1 root    wheel       4,   1 15 Oct 22:29 ttyp1
```

UNIX I/O Structure

doesn't use the
block buffer cache,
can be used for
virtual memory



treated
separately

- **block devices**: accessed a block at a time
- **character devices**: accessed one byte at a time
- **network devices**: access is inherently different from local disk access, most systems provide a separate interface for network devices

Block Devices

- Disks, tapes. Can address a complete block (e.g. 4096 bytes)
- Block device driver may turn block numbers into tracks and cylinders etc. But many block devices work directly on block numbers.
- 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 (except network devices) 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

  - Cooked tty mode is a "line discipline".
  - Raw input is passed on exactly as it is (almost) to the program.
    - Except for Secure Attention Key (SAK) handling - like ctrl-alt-del on Windows

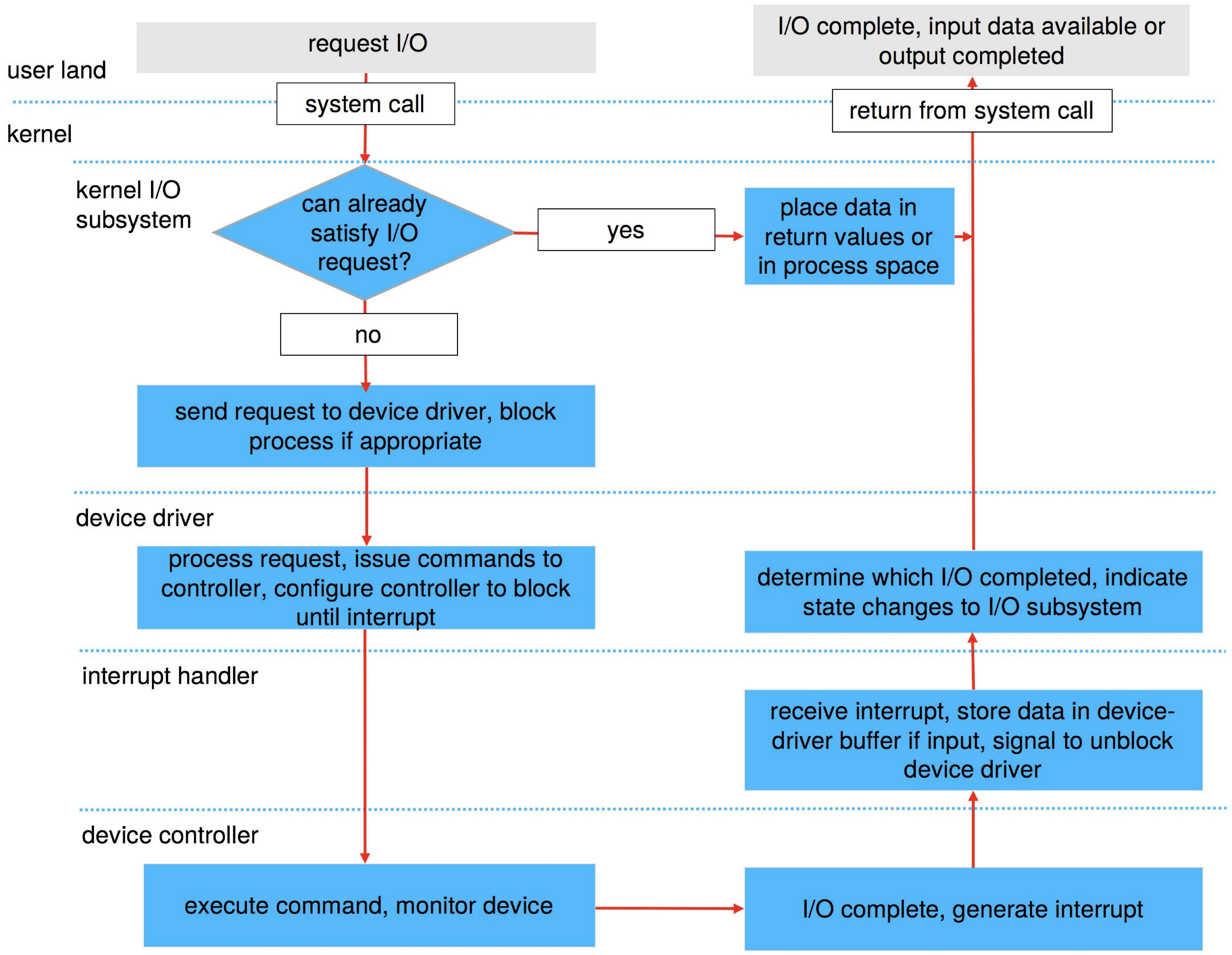
Talking to the Driver

- I/O request block - IORB
- A data structure which contains all the information needed to complete the I/O request.
- Constructed by the I/O system calls and passed to the device drivers. e.g. Read device 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.

Scheduling of Requests

- Some I/O requests get queued waiting for service. For some types of devices it is more efficient to service the requests out of order.
- We may also schedule requests according to other criteria e.g. the priority of the process making the request.
- Scheduling is commonly used when dealing with disks.

Life Cycle of An I/O Request


user land
kernel
request I/O
system call
I/O complete, input data available or output completed
return from system call
kernel I/O subsystem
can already satisfy I/O request?
yes
place data in return values or in process space
no
send request to device driver, block process if appropriate
device driver
process request, issue commands to controller, configure controller to block until interrupt
determine which I/O completed, indicate state changes to I/O subsystem
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
device controller
execute command, monitor device
I/O complete, generate interrupt

Getting a Character from the Keyboard

A program which wants every character from the keyboard uses the keyboard device in the raw mode.

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 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 top-half of the keyboard device driver) gets the letter “X” say and passes it to the bottom-half .
- The bottom-half unblocks and returns the character to the program.

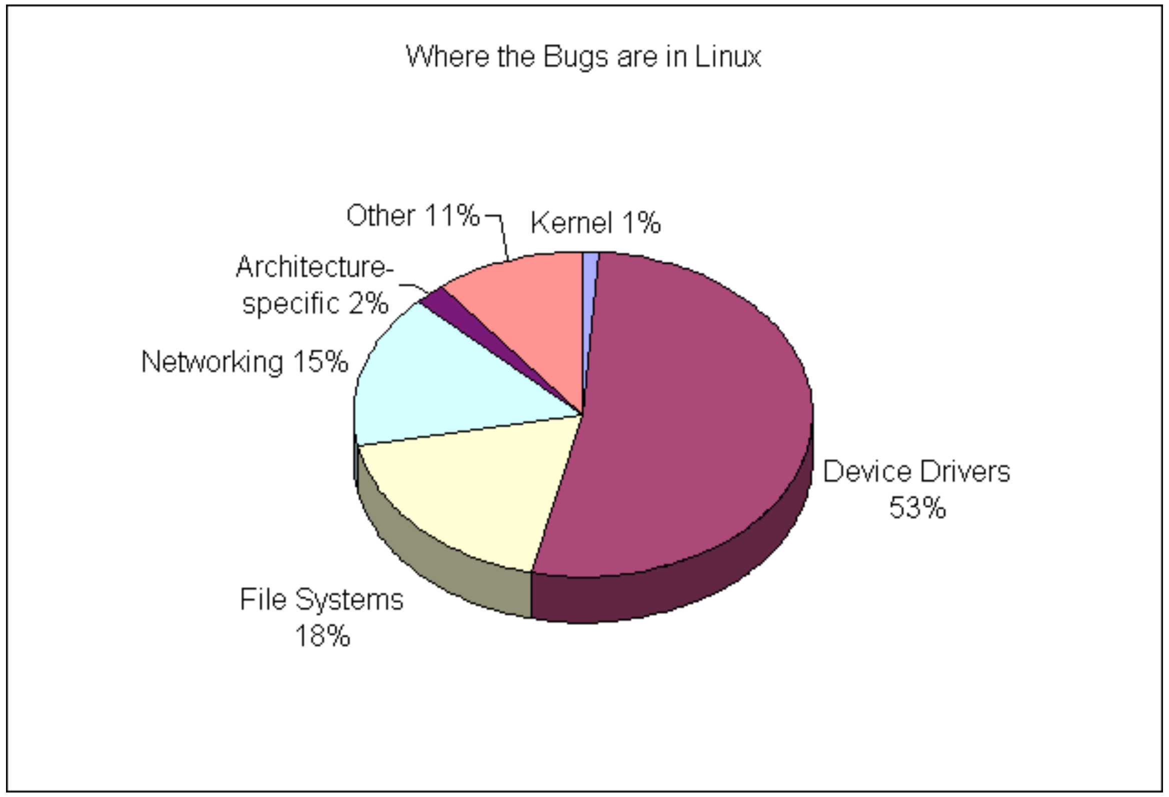
Deferred Work

- It is common for interrupts to be disabled (at least at some level) when handling an interrupt.
- So we don’t want to do a lot of work in an interrupt handler.
    - Instead the interrupt handler can set things up (e.g. put data in a data structure) for later processing.
- We can defer the work (and enable interrupts while doing it) using a variety of mechanims.
    - On Windows we can use a Deferred Procedure Call (DPC).
    - On Unix systems we call a similar concept the bottom-half of the device driver (the interrupt handler is the top-half).
    - On Linux softirqs and tasklets can be used for this job.

Where the bugs are?

- Where the bugs are in Linux (other OSs may be worse ☺).

  Reasons:
  - Device drivers are written by many people.
  - Less experience, less knowledge, less checks, kernel code is read and checked by many more people.
  - We would like a way to reduce the impact of bugs in device drivers on the rest of the system.



Windows Hardware Quality Labs (WHQL) testing is used to test hardware and software to verify it works ok.

Providing Device Drivers

Compiled as part of the kernel.

Loadable (and unloadable) sections of kernel level code.

- The way Linux and Windows do it.
- These modules are signed.

User level service providers

- There may need to be a stub driver at kernel level.
- Can be done with memory-mapped devices by allocating the corresponding real memory addresses to the driver process.
- This is safer because the interactions with the rest of the OS are constrained to specific interfaces.
- In x86 processors, processes can request permission to use particular IO ports and if granted can use them from user mode.

Linux Kernel Modules

Sections of kernel code that can be compiled, loaded, and unloaded independently of the rest of the kernel.

A kernel module may typically implement a device driver, a file system, or a networking protocol.

The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL. (Some people strongly disagree with this idea and think it is illegal.)

Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.

Three components to Linux module support:

- module management
- driver registration
- conflict resolution

Module Management

The module requestor loads modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

When the kernel is compiled some symbols (functions, variables) are exported into an internal symbol table.

- These symbols can then be referenced by modules via the linking process.

  e.g. /proc/kallsyms (to see non-zero addresses need to be root)

```
c0100000 T _text
c0100000 T startup_32
c01000b0 T startup_32_smp
c0100130 t checkCPUtype
c01001b1 t is486
c01001b8 t is386
c0100225 t check_x87
c0100260 T setup_pda
c0100282 t setup_idt
c010029f t rp_sidt
c0100322 t early_divide_err
c0100328 t early_illegal_opcode
c0100331 t early_protection_fault
c0100338 t early_page_fault
c010033f t early_fault
...
```

Module loading

The module is scanned for references to kernel symbols, these are located in the symbol table.

- If a symbol can’t be found the module is not loaded.

A continuous area of virtual kernel memory is requested.

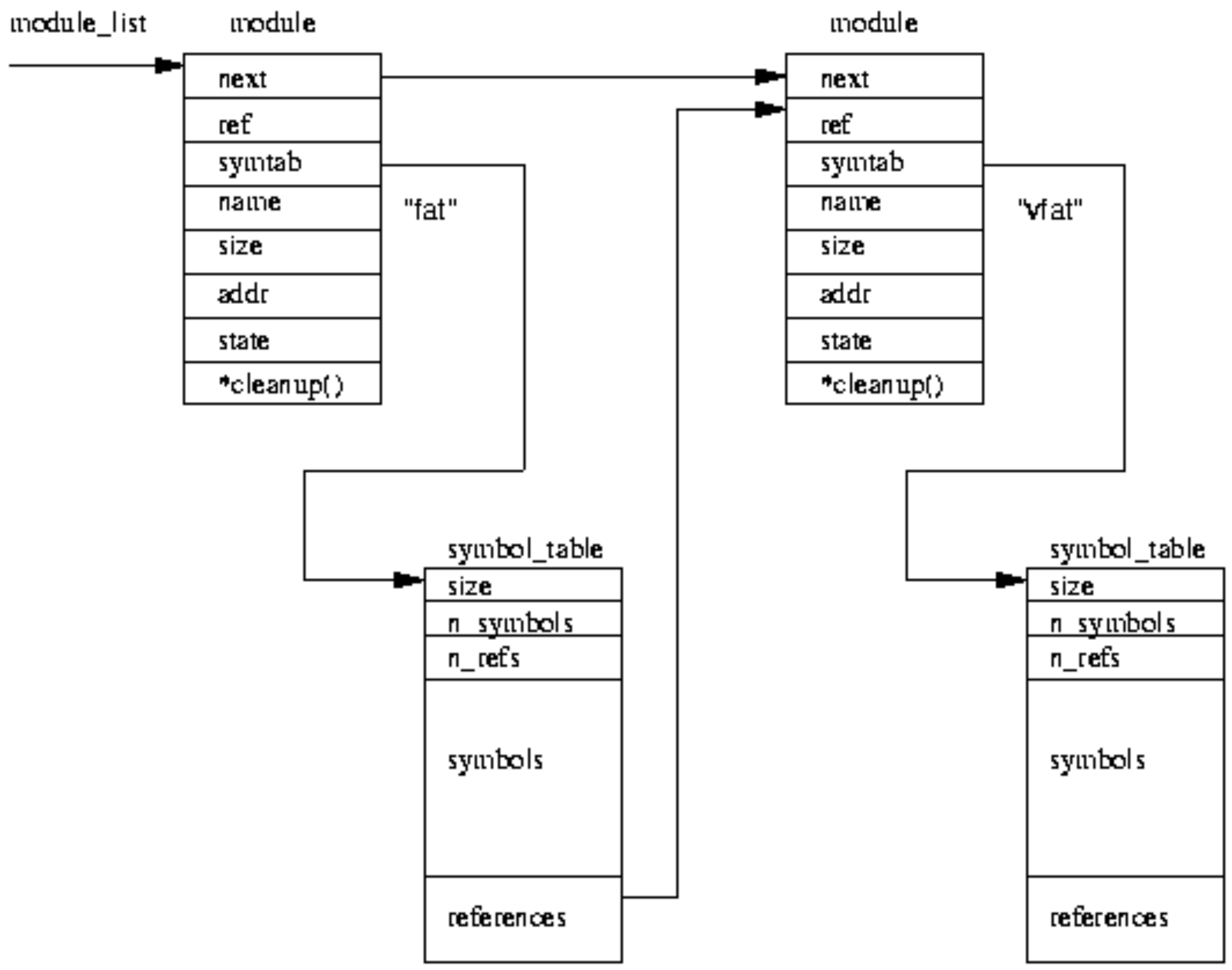
The module is moved to its location.

The module itself provides more symbol-table entries to the kernel.

- These may be used by other modules.

Module Design

•



Driver Registration

When a module is loaded the kernel calls its startup routine.

- This routine must register the module with the kernel.

Allows modules to tell the rest of the kernel that a new driver has become available.

The kernel maintains tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.

Registration tables include the following items:

- Device drivers
- File systems
- Network protocols
- Binary formats – recognise and load new types of executable files

Conflict Resolution

The conflict resolution mechanism allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver

It aims to:

- Prevent modules from clashing over access to hardware resources
- Prevent *autoprobes* from interfering with existing device drivers
- Resolve conflicts with multiple drivers trying to access the same hardware (they may be allowed to but not at the same time)

The kernel maintains lists of allocated hardware resources (interrupt lines, DMA channels and I/O address space).

- Device drivers reserve resources with the database.
- If it can’t get a resource the driver may try alternatives or simply fail and ask to be unloaded.

User Level Drivers

- User level device drivers have some of the advantages we saw with user level file systems.
    - Don't need special permissions to write them.
    - If they crash they don't affect the kernel.
    - They don't have uncontrolled access to kernel mode and other drivers.
- But device drivers commonly need low level access to the hardware.
- And if we are moving back and forward between user and kernel mode as we deal with our devices they may not be as efficient as kernel level device drivers.
    - Kernel level drivers can commonly access kernel data structures directly.

User Level Driver

- Linux allows a suid root process to access IO ports – ioperm. Then the process can drop back to the actual user's uid and exec.
- Memory mapped IO also allows user processes to access IO registers. The registers are mapped into the *ordinary* address space. This process is allowed to read and write these addresses.
- Interrupts are a problem. The driver has to somehow respond to and request interrupts. Experimental versions of Linux allow interrupts to be accessed via the /proc file system. (Currently you can read /proc/interrupts.)
- DMA uses interrupts but also requires memory buffers (allocated as contiguous pages of physical memory). There has to be a way of obtaining the bus (usually physical) addresses for the DMA controller or bus mastering device.
- Because of kernel/user transitions they can be slower than kernel level.

“The End”