Slide 1

I/O MANAGEMENT

- ➜ Categories of I/O devices and their integration with processor and bus
- ➜ Design of I/O subsystems
- ➜ I/O buffering and disk scheduling
- ➜ RAID

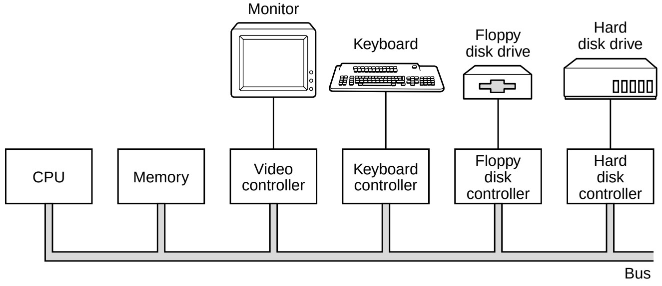
Slide 2

CATEGORIES OF I/O DEVICES

There exists a large variety of I/O devices:

- ➜ Many of them have different properties
- ➜ They seem to require a range of different interfaces
  - We don't want a new device interface for every new device
  - Diverse, but similar interfaces lead to duplication of code
- ➜ Challenge: Uniform and efficient approach to I/O

Slide 3



Controller:

- ➜ can often handle more than one identical devices
- ➜ low level interface to actual device
- ➜ for disks: perform error check, assemble data in buffer

Slide 4

Three classes of devices (by usage):

- ➜ Human readable:
  - For communication with user
  - Keyboard, mouse, video display, printer
- ➜ Machine readable:
  - For communication with electronic equipment
  - Disks, tapes, sensors, controllers, actuators
- ➜ Remote communication:
  - For communication with remote devices
  - Modems, Ethernet, wireless

Slide 5

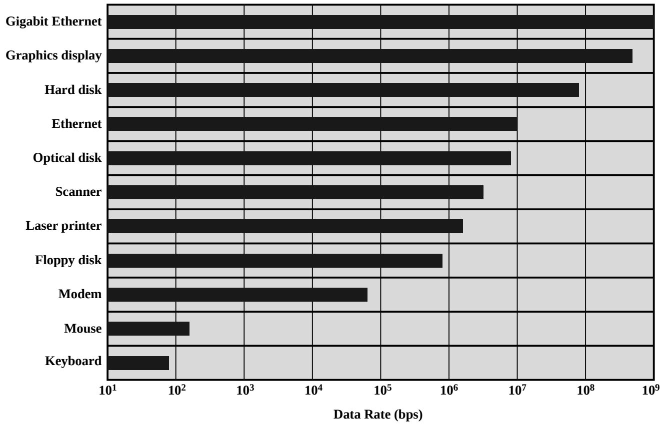
Which differences impact device handling?

- ➜ Data rate
- ➜ Complexity of control
  - e.g., line printer versus high-speed graphics
- ➜ Unit of transfer
  - stream-oriented: e.g. terminal I/O
  - block-oriented: e.g. disk I/O
- ➜ Data representation (encoding)
- ➜ Error conditions (types, severity, recovery, etc.)

Hopefully similarity within a class, but there are exceptions!

Slide 6

Typical data rates of I/O devices::



Slide 7

Accessing I/O Devices — Hardware

Interfacing alternatives:

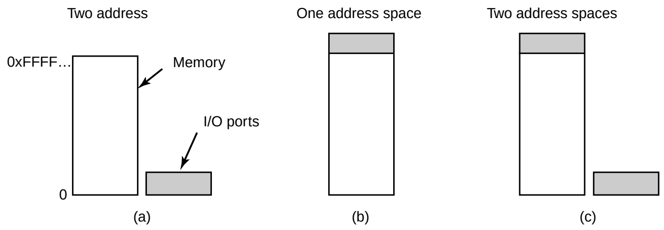
- ➜ Port mapped I/O
- ➜ Memory mapped I/O
- ➜ Direct memory access (DMA)

Slide 8

Port mapped versus memory mapped I/O:

- ➜ Memory-mapped I/O:
  - I/O device registers and memory are mapped into the normal memory address range
  - Standard memory access is used
    - any memory access function can be used to manipulate I/O device
- ➜ Port-mapped I/O:
  - I/O devices are accessed using special I/O port instructions
  - Only part of the address lines are needed
    - standard PC architecture: 16 bit port address space

Slide 9



(a) Port mapped
(b) Memory mapped
(c) Hybrid

Slide 10

Memory mapped I/O

- ✔ directly accessable in high-level languages
- ✔ no need for special protection mechanism
- ✔ not necessary to load contents into main memory/registers
- ✘ interference with caching
- ✘ memory modules and I/O devices must inspect all memory references
- ✘ complex problem if multiple buses are present

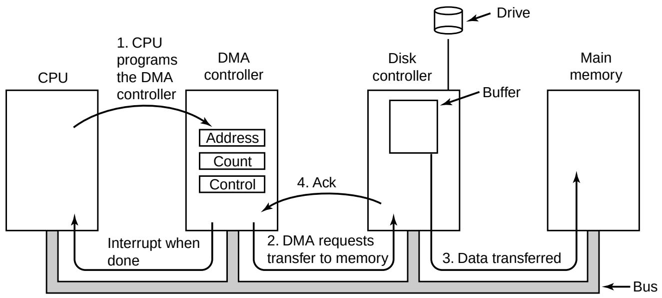
Slide 11

DIRECT MEMORY ACCESS (DMA)

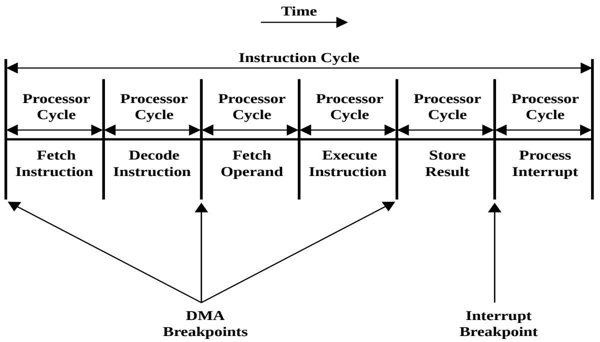
Basics:

- ➜ Takes control of the system form the CPU to transfer data to and from memory over the system bus
- ➜ Cycle stealing is used to transfer data on the system bus
- ➜ The instruction cycle is suspended so data can be transferred
- ➜ The CPU pauses one bus cycle
- ➜ No interrupts occur (i.e., no expensive context switches)

Slide 12



Slide 13



- ➜ most buses as well as DMA controllers can operate in word-by-word or block mode
  - word-by-word mode: use cycle stealing to transfer data
  - block mode: DMA controller aquires bus to transfer data
- ➜ typically, devices like disks, sound or graphic cards use DMA

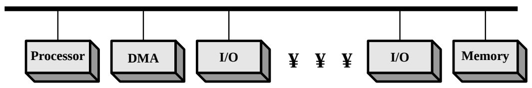
Slide 14

Processor transfers data vs DMA:

- ➜ Processor transfers data:
  - Processor copies data from main memory into processor registers or memory
  - Large data volume ⇒ CPU load can be very high
- ➜ Direct memory access (DMA):
  - Processor forwards address range of data to a DMA controller
  - The DMA controller performs the data transfer without processor intervention (but locks the memory bus)
  - Slows down processor, but overhead much less

Slide 15

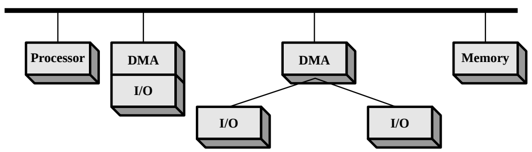
Configurations: Single bus, detached DMA::



- ➜ Cycle stealing causes the CPU to execute more slowly

Slide 16

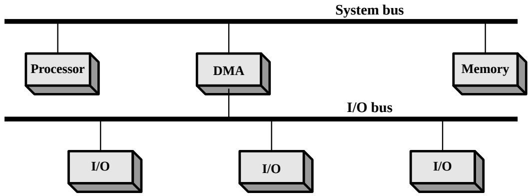
Single bus, integrated DMA I/O:



- ➜ Reduce busy cycles by integrating the DMA and I/O devices

Slide 17

Separate I/O bus:



➜ Path between DMA module and I/O module that does not include the system bus

Slide 18

Accessing I/O Devices

Three general approaches on software level:

① Programmed I/O
   - poll on device
② Interrupt-driven I/O
   - suspend when device not ready
③ I/O using direct memory access (DMA)
   - use extra hardware component

Let's have a look at each of the three methods.

Slide 19

Programmed and interrupt driven I/O

Example: what happens when the CPU reads from disk?

① Disk controller
   - reads block bit by bit into buffer
   - compute checksum to detect read errors
   - causes interrupt
② CPU copies block byte by byte into main memory

Slide 20

Programmed I/O

Read:

① poll on status of device
② issue read command to device
③ wait for read to complete, poll on status of device
④ copy data from device register into main memory
⑤ jump to (1) until all data read

Write:

① poll on status of device
② copy data to device register
③ issue write command to device
④ jump to (1) until all data written

Slide 21

Properties:

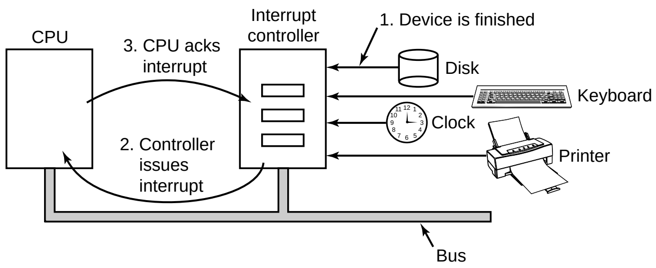
➜ programmed I/O suitable in some situations (e.g., single threaded, embedded system)

➜ usually inefficent, waste of CPU cycles

Slide 22

INTERRUPT-DRIVEN I/O

➜ avoid polling on device!



Slide 23

INTERRUPT-DRIVEN I/O

Steps involved:

① issue read/write command to device

② wait for corresponding interrupt, suspend

③ device ready: acknowledge I/O interrupt

④ handle interrupt:

- read data from device register
- write data to device register

Properties:

➜ high overhead due to frequent context switching

➜ more efficient use of CPU than programmed I/O

➜ but, still waste of CPU cycles as CPU does all the work

Alternative: use extra hardware (direct memory access controller) to offload some of the work from CPU

Slide 24

DIRECT MEMORY ACCESS (DMA)

How DMA works:

① CPU tells DMA controller what it should copy, and where it should copy to

② DMA controller issues read request to disk controller

③ disk controller

- transfers word into main memory
- signals DMA controller

④ DMA controller decrements counter

- if all words are transferred, signal CPU
- otherwise, continue with step (2)

Slide 25

THE QUEST FOR GENERALITY/UNIFORMITY

Ideal state:

➜ handle all I/O devices in the same way (both in the OS and in user processes)

Problem:

➜ Diversity of I/O devices

➜ Especially, different access methods (random access versus stream-based) as well as vastly different data rates

➜ Generality often compromises efficiency!

Slide 26

THE EVOLUTION OF THE IO-FUNCTION

Hardware changes trigger changes in handling of IO devices:

① Processor directly controls a peripheral device

② Controller or IO module is added

- Programmed IO without interrupts
- Example: Universal Asynchroneous Receiver Transmitter (UART)
- CPU reads or writes bytes to IO controller
- CPU does not need to handle details of external device

③ Controller or IO module with interrupts

- CPU does not spend time waiting on completion of operation

Slide 27

④ DMA

- CPU involved at beginning an end only

⑤ IO module has separate processor

- Example: SCSI controller)
- Controller CPU executes SCSI code out of main memory

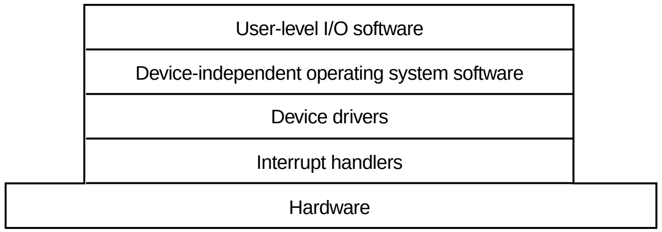
⑥ IO module is a computer in its own right

- Myrinet multi-gigabit network controller

Slide 28

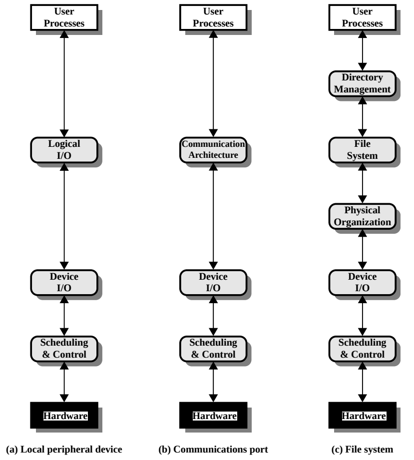
I/O SOFTWARE LAYERS

I/O software is divided into a number of layers, to provide adequate abstraction and modularisation:



Slide 29

I/O Organisation:



Slide 30

I/O Interrupt Handler

- ➜ Interrupt handlers are best "hidden"
- ➜ Can execute almost any time, raises complex concurrency issues
- ➜ Generally, drivers starting an IO operation block until interrupt notifies them of completion (`dev_read()`)
- ➜ Interrupt handler does its task, then unblocks driver

Slide 31

I/O Interrupt Handler

Main steps involved:

① save state that has not already been saved by hardware
② set up context (address space) for specific interrupt handler (stack, TLB, MMU, page table)
③ set up stack for interrupt service procedure (usually runs in kernel stack of current process)
④ acknowledge interrupt controller, re-enable other interrupts
⑤ run specific interrupt handler
   - find out what caused interrupt (received network packet, disk read finished, etc)
   - get information from device controller
⑥ re-enable interrupt
⑦ invoke scheduler

Slide 32

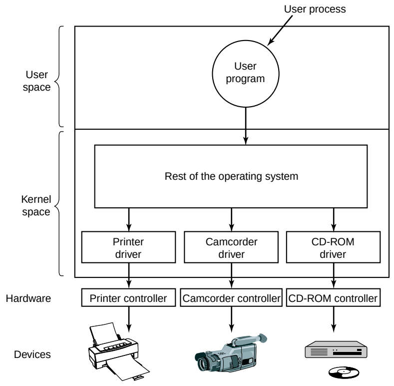
Device Driver

Code to control specific device:

- ➜ usually differs from device to device
- ➜ sometimes suitable for class of devices adhering to a standard (e.g., SCSI)
- ➜ common interface to rest of OS, depending on type (block, character, network) of device

Slide 33

Device Driver



Slide 34

Main steps involved:

① check input parameters

② translate request (`open`, `close`, `read`, ...) into appropriate sequence of commands for part. hardware

③ convert into device specific format e.g. disk

- linear block number into head, track, sector, cylinder number

④ check if device is available, queue request if necessary

⑤ program device controller (may block, depending on device)

Device drivers also initialise hardware at boot time, shut it down cleanly.

Slide 35

Device Independent I/O Software

➜ Commonality between drivers of different classes

➜ Split software into device dependent and independent part

Device independent software is responsible for:

➜ Uniform interfacing to device drivers

➜ Buffering

➜ Error reporting

➜ Allocating/releasing

➜ Providing device independent block sizes

We will look into each of these tasks separately

Slide 36

Uniform Interfacing

Design goal:

➜ interface to all device driver should be the same

➜ may not be possible, but few classes of different devices, high similarity between classes

➜ provides an abstraction layer over concrete device

➜ uniform kernel interface for device code

- `kmalloc`, installing IRQ handler
- allows kernel to evolve without breaking exisiting drivers

Naming of devices:

➜ map symbolic device names to driver

➜ Unix, Windows 2000:

- devices appear in the file system
- usual file protection rule applies to device drivers

Slide 37

Unix device files:

- ➜ Uniform device interface: devices as files
  - `read()`
  - `write()`
  - `seek()`
  - `ioctl()` etc,
- ➜ Main attributes of a device file:
  - Device type: block versus character (stream) devices
  - Major number (1–255) identifies driver (device group)
  - Minor number (8 bit) identifies a specific device in a group

Slide 38

Examples:

Name	Type	Major	Minor	Description
`/dev/fd0`	block	2	0	floppy disk
`/dev/hda`	block	3	0	first IDE disk
`/dev/hda2`	block	3	0	2nd primary partition of IDE disk
`/dev/ttyp0`	char	3	0	terminal
`/dev/ttyp0`	char	5	1	console
`/dev/null`	char	1	3	Null device

Slide 39

Some I/O devices have no device file:

- ➜ network interfaces are handled differently
- ➜ However, symbolic name for network interfaces (`eth0`)
- ➜ Device name associated with network address
- ➜ User-level interface: sockets (a BSD invention)
- ➜ Sockets are also a file in the Unix file system, but offer a different interface
  - `socket()`, `bind()`, `receive()`, `send()` instead of `open()`, `read()` and `write()`

Slide 40

Implementation of device files:

- ➜ Virtual File System (VFS)
  - Re-directs file operations to device driver
  - driver determined by the major number

Device drivers:

- ➜ Device-dependent low-level code
  - convert between device operations and standard file ops (`read()`, `write()`, `ioctl()`, etc.)
- ➜ Device drivers in Linux are:
  - statically linked into the kernel, or
  - dynamically loaded as kernel modules
- ➜ The kernel provides standardised access to DMA etc.

Slide 41

Association of device drivers with device files:

- ➜ VFS maintains tables of device file class descriptors
  - `chrdevs` for character devices
  - `blkdevs` for block devices
  - indexed by major device number
  - each contains file operation table
  - device driver registers itself with VFS by providing an entry

```
struct file_operations {
  ssize_t (*read)  (struct file *, char *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
  int     (*ioctl) (struct inode *, struct file *,
                    unsigned int, unsigned long);
  int     (*mmap)  (struct file *, struct vm_area_struct *);
  int     (*open)  (struct inode *, struct file *);
  int     (*lock)  (struct file *, int, struct file_lock *);
  ...
};
```

Slide 42

Device access via device files:

- ➜ On file `open()`, VFS does the following:
  - retrieves file type as well as major and minor number
  - indexes either `chrdevs` or `blkdevs` with major number
  - places file operations table into file descriptor
  - invokes the `open()` member of the file operations table
- ➜ On other file operations:
  - VFS performs indirect jump to handler function via file operations table
- ➜ The file operation table provides a small and well-defined interface to most devices
- ➜ However, the `ioctl()` entry is a kind of "back door":
  - implements functionality not covered by the other routines
  - ugly!

Slide 43

ALLOCATING AND RELEASING DEVICES

- ➜ some devices can only be used by single process
  - e.g., CD burner
- ➜ OS must manage allocation of devices
- ➜ if device is mapped to special file
  - `open` and `close` to aquire and release device
  - may be blocking or non-blocking (fail)

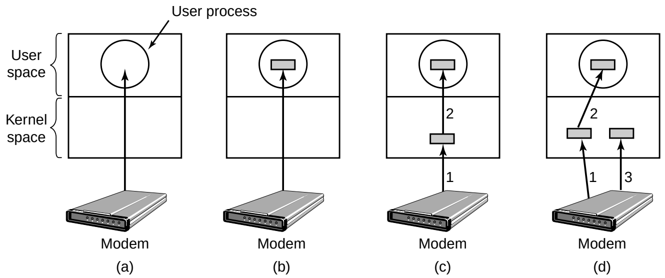
Slide 44

I/O BUFFERING

Why do we need buffering?

- ➜ Performance:
  - Put output data in buffer and write asynchronously
  - Batch several small writes into one large
  - Some devices can only read/write largish blocks
  - Read ahead
- ➜ Locking of memory pages; deadlock avoidance:
  - Cannot swap pending I/O data (especially with DMA)
  - Cannot deliver data to process that is swapped out
  - ➨ lock pages when I/O operation is queued

Slide 45



(a) no buffering

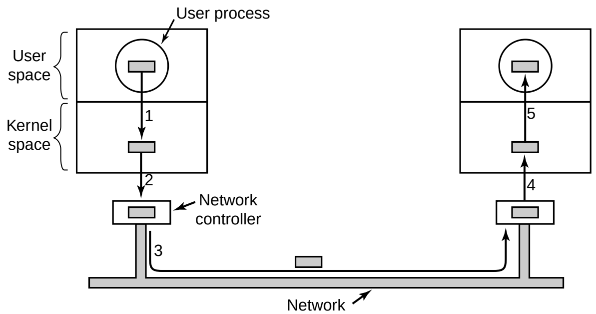
(b) buffer in user space

- what happens if page is swapped to disk?

(c) buffer in kernel space

(d) double buffering

Slide 46



① data copied to kernel space, user process does not block

② driver copies it into controller register

③ copy to network, receiver's buffer

④ send acknowledgement

⑤ copy to kernel space, then user space

Slide 47

ALLOCATING AND RELEASING DEVICES

➜ some devices can only be used by single process
  - e.g., CD burner

➜ OS must manage allocation of devices

➜ if device is mapped to special file
  - `open` and `clode` to aquire and release device
  - may be blocking or non-blocking (fail)

Slide 48

ERROR REPORTING

Errors in context of I/O are common and can occur on different levels:

➜ programming errors:
  - write to read-only device
  - invalid buffer address

  report error code to caller

➜ I/O error
  - device not present
  - storage medium defect

➜ critical errors
  - critical data structure destroyed

Slide 49

User space I/O software

Library functions of different complexity

- `write`
- `fprintf`
- graphics libraries

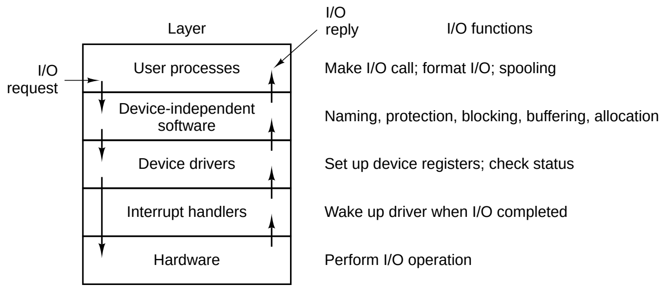
Spooling:

Special OS processes (daemons) control device

- Linux: check `ls /var/spool` to see what type of I/O operations use spooling

Slide 50

Summary of I/O system layers



Slide 51

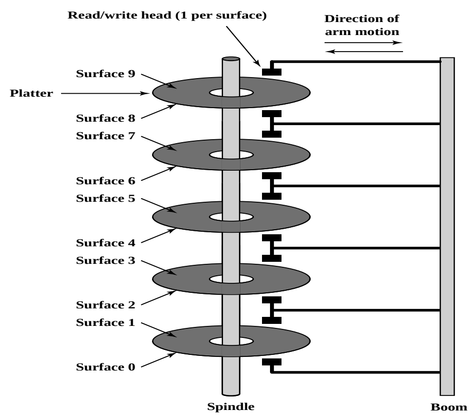
Hard Disks

After general discussion of I/O, let's look at hard disks in detail

- Disk hardware
  - architecture influences lower level I/O software
- Disk formatting
- Disk Scheduling
  - dealing with disk requests
- RAID

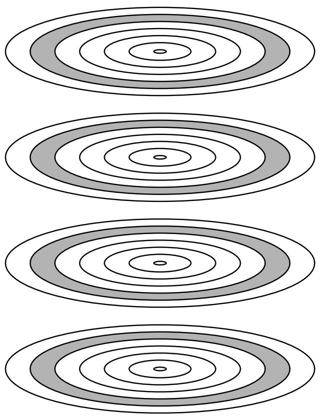
Slide 52

Components of a Disk Drive



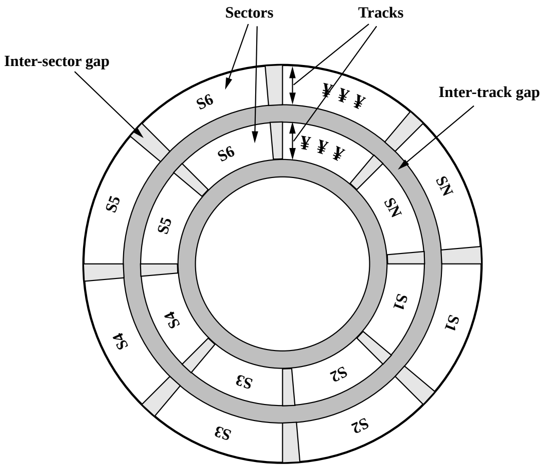
Slide 53

Tracks per Cylinder



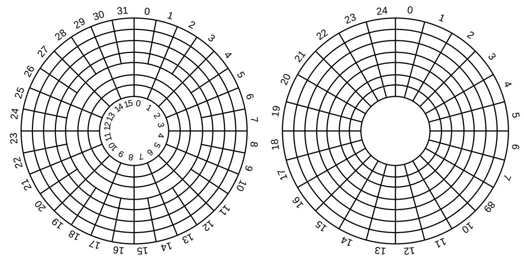
Slide 54

Disk geometry:



Slide 55

Disk geometry:



- ➜ modern disks are devided into zones
- ➜ different number of sectors per track for each zone
- ➜ present virtual geometry to OS
  - pretends to have equal amount of sectors per track
  - maps virtual (cylinder, head,sector) coordinate to real location

Slide 56

Disk Formatting

Low-level formatting:

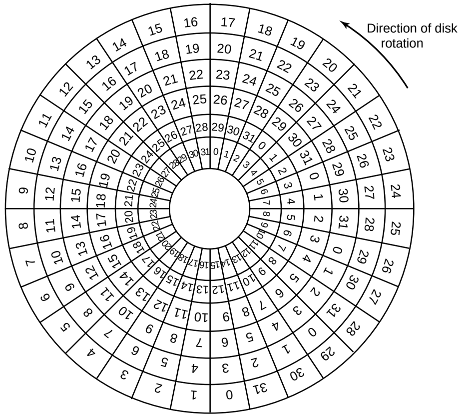
Preamble	Data	ECC

Layout of sector:

- Preamble: marks start of a sector, cylinder and sector number
- Data: usually 512 byte section
- Error correction code (ECC): used to detect, correct errors

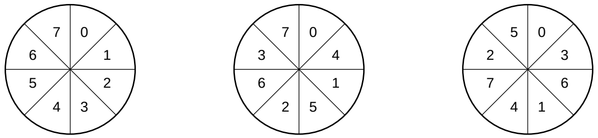
Slide 57

Disk formatting — Cylinder skew



Slide 58

Disk formatting — Interleaving sectors



(a) (b) (c)

- ➜ system may not be able to keep up with rotation speed
- ➜ to avoid interleaving sectors, modern controllers able to buffer entire track

Slide 59

Partitioning:

- ➜ disk is divided into different partitions, each treated as a separate logical disk.
- ➜ Partition table gives starting sector and size of each partition.
- ➜ master boot record: boot code and partition table

High-level formatting: each partition contains

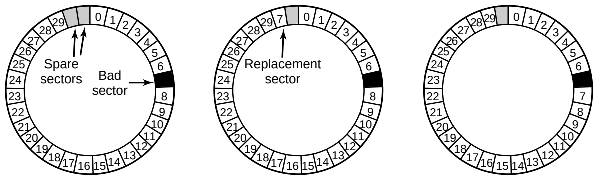
- ➜ boot block
- ➜ free storage admin info
- ➜ root directory
- ➜ type of file system

Slide 60

Disk Error Handling

- ➜ due to high density, most disks have bad sectors
- ➜ small defects can be masked using ECC
- ➜ major defects handled by remapping to spare sectors

Substituting bad sectors:



(a) (b) (c)

Slide 61

Can be done by

- ➜ disk controller
  - before disk is shipped
  - dynamically when repeated errors occur
  - remapping by maintaining internal table or rewriting preamble
- ➜ operating system: tricky (eg. backups)

Slide 62

Disk Scheduling

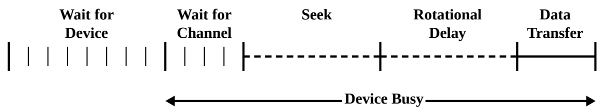
- ➜ Disk performance is critical for system performance
- ➜ Management and ordering of disk access requests have strong influence on
  - access time
  - bandwidth
- ➜ Important to optimise because:
  - huge speed gap between memory and disk
  - disk throughput extremely sensitive to
    - request order ⇒ disk scheduling
    - placement of data on disk ⇒ file system design
- ➜ Request scheduler must be aware of disk geometry

Slide 63

Disk performance parameters:

- ➜ Disk is moving device ⇒ must position correctly for I/O
- ➜ Execution of a disk operation involves:
  - Wait time: the process waits to be granted device access
    - Wait for device: time the request spends in a wait queue
    - Wait for channel: time until a shared I/O channel is available
  - Access time: time the hardware needs to position the head
    - Seek time: position the head at the desired track
    - Rotational delay (latency): spin disk to the desired sector
  - Transfer time: sectors to be read/written rotate below the head

Slide 64



Slide 65

Performance Parameters

- Seek time T_s: Moving the head to the required track
  - not linear in the number of tracks to traverse:
    - startup and settling time
  - Typical average seek time: a few milliseconds
- Rotational delay:
  - rotational speed, r, of 5,000 to 10,000rpm
  - At 10,000rpm, one revolution per 6ms $\Rightarrow$ average delay 3ms
- Transfer time:
  - to transfer b bytes, with N bytes per track:

$$T = \frac{b}{rN}$$

  - Total average access time:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

Slide 66

A Timing Comparison:

- $T_s = 2$ ms, $r = 10,000$ rpm, 512B sect, 320 sect/track
- Read a file with 2560 sectors (= 1.3MB)
- File stored compactly (8 adjacent tracks):

Read first track

Average seek	2ms
Rot. delay	3ms
Read 320 sectors	6ms
	11ms

$\Rightarrow$ All sectors: $11 + 7 * 9 = 74ms$

- Sectors distributed randomly over the disk:

Read any sector

Average seek	2ms
Rot. delay	3ms
Read 1 sector	0.01875ms
	5.01875ms

$\Rightarrow$ All: $2560 * 5.01875 = 20,328ms$

Slide 67

Disk Scheduling Policy

Observation from the calculation:

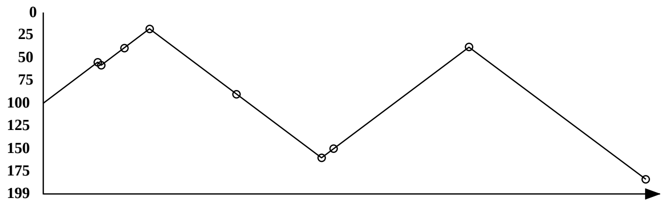
- Seek time is the reason for differences in performance
- For a single disk there will be a number of I/O requests
- Processing in random order leads to worst possible performance
- We need better strategies

Slide 68

First-in, first-out (FIFO):

- Process requests as they come in

Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



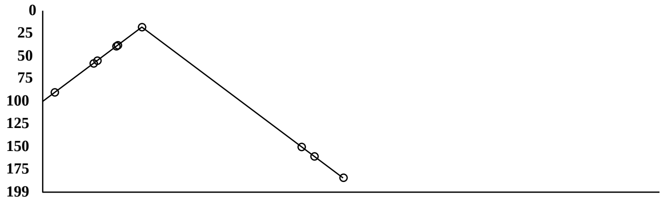
- Fair (no starvation!)
- Good for few processes with clustered requests
- deteriorates to random if there are many processes

Slide 69

Shortest Service Time First (SSTF):

- ➜ Select the request that minimises seek time

Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



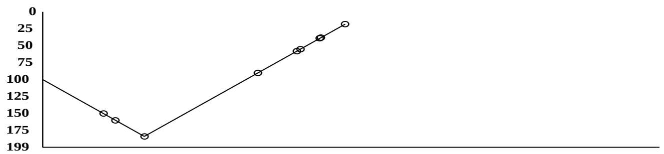
- ➜ service order: 90, 58, 55,39,18, 150,160,184
- ➜ Minimising locally may not lead to overall minimum!
- ➜ Can lead to starvation

Slide 70

SCAN (Elevator): Move head in one direction

- ➜ services requests in track order until it reaches last track, then reverse direction

Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



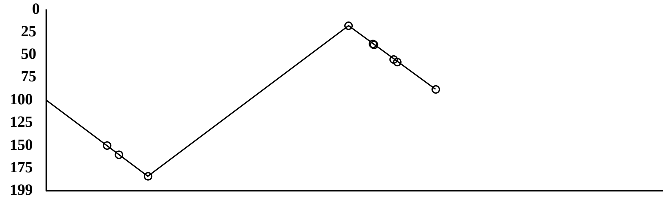
- ➜ service order: 150,160,184, (200), 90, 58, 55,39,18, (0)
- ➜ Similar to SSTF, but avoids starvation
- ➜ LOOK: variant of SCAN, moves head only to last request of one direction: 150,160,184, 90, 58, 55,39,18
- ➜ SCAN/LOOK are biased against region most recently traversed
- ➜ Favour innermost and outermost tracks
- ➜ Makes poor use of sequential reads (on down-scan)

Slide 71

Circular SCAN (C-SCAN):

- ➜ Like SCAN, but scanning to one direction only
  - When reaching last track, go back to first non-stop

Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



- ➜ Better use of locality (sequential reads)
- ➜ Better use of disk controller's read-ahead cache
- ➜ Reduces the maximum delay compared to SCAN

Slide 72

N-step-SCAN:

- ➜ SSTF, SCAN & C-SCAN allow device monopolisation
  - process issues many requests to same track
- ➜ N-step-SCAN segments request queue:
  - subqueues of length N
  - process one queue at a time, using SCAN
  - added new requests to other queue

Slide 73

FSCAN:

➜ Two queues

- one being presently processed
- other to hold new incoming requests

Slide 74

Disk scheduling algorithms:

Name	Description	Remarks
	Selection according to requestor	
RSS	Random scheduling	For analysis and simulation
FIFO	First in, first out	Fairest
PRI	By process priority	Control outside disk magmt
LIFO	Last in, first out	Maximise locality & utilisation
	Selection according to requested item	
SSTF	Shortest seek time first	High utilisation, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One-way with fast return	Better worst-case time
N-SCAN	SCAN of N recs at once	Service guarantee
FSCAN	N-SCAN (N=init. queue)	Load sensitive

Slide 75

DISK SCHEDULING

➜ Modern disks:

- seek and rotational delay dominate performance
- not efficient to read only few sectors
- cache contains substantial part of currently read track

➜ assume real disk geometry is same as virtual geometry

➜ if not, controller can use scheduling algorithm internally

So, does OS disk scheduling make any difference at all?

Slide 76

LINUX 2.4.

➜ Used a version of C-SCAN

➜ no real-time support

➜ Write and read handled in the same way — read requests have to be prioritised

Slide 77

LINUX 2.6.

Deadline I/O scheduler:

- ➜ two additional queues: FIFO read queue with deadline of 5ms, FIFO write with deadline of 500ms
- ➜ request submitted to both queues
- ➜ if request expires,scheduler dispatches from FIFO queue
- ➜ Performance:
  - ✔ seeks minimised
  - ✔ requests not starved
  - ✔ read requests handled faster
  - ✘ can result in seek storm, everything read from FIFO queues

Slide 78

Anticipatory Scheduling:

- ➜ Same, but anticipates dependent read requests
- ➜ After read request: waits for a few ms
- ➜ Performance
  - ✔ can dramatically reduce the number of seek operations
  - ✘ if no requests follow, time is wasted

Slide 79

PERFORMANCE

- ➜ Writes
  - similar for writes
  - deadline scheduler slightly better than AS
- ➜ Reads
  - deadline: about 10 times faster for reads
  - as: 100 times faster for streaming reads

Slide 80

RAID

- ➜ CPU performace has improved exponentially
- ➜ disk performance only by a factor of 5 to 10
- ➜ huge gap between CPU and disk performance

Parallel processing used to improve CPU performance.

Question: can parallel I/O be used to speed up and improve reliability of I/O?

Slide 81

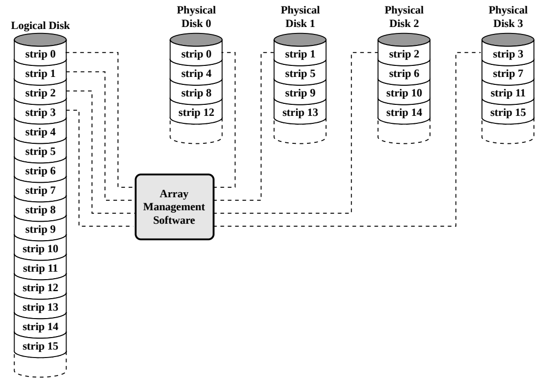
RAID: Redundant Array of Inexpensive/Independent Disks

Multiple disks for improved performance or reliability:

- ➜ Set of physical disks
- ➜ Treated as a single logical drive by OS
- ➜ Data is distributed over a number of physical disks
- ➜ Redundancy used to recover from disk failure (exception: RAID 0)
- ➜ There is a range of standard configurations
  - numbered 0 to 6
  - various redundancy and distribution arrangements

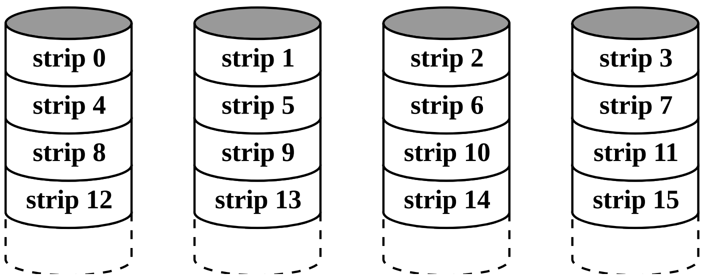
Slide 83

Data mapping for RAID 0:



Slide 82

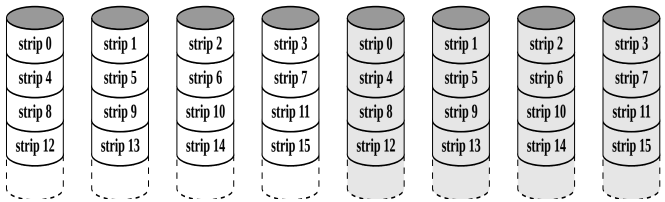
RAID 0 (striped, non-redundant):



- ➜ controller translates single request into separate requests to single disks
- ➜ requests can be processed in parallel
- ➜ simple, works well for large requests
- ➜ does not improve on reliability, no redundancy

Slide 84

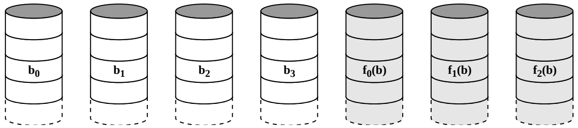
RAID 1 (mirrored, $2\times$ redundancy):



- ➜ duplicates all disks
- ➜ write: each request is written twice
- ➜ read: can be read from either disk

Slide 85

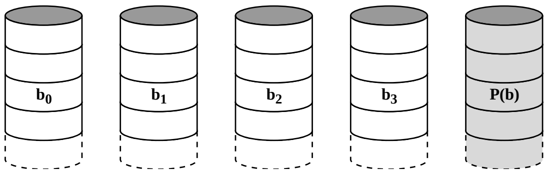
RAID 2 (redundancy through Hamming code):



- ➜ strips are very small (single byte or word)
- ➜ error correction code across corresponding bit positions
- ➜ for n disks, $log_2 n$ redundancy
- ➜ expensive
- ➜ high data rate, but only single request

Slide 86

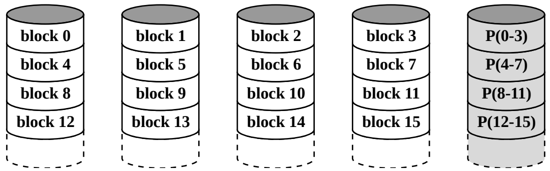
RAID 3 (bit-interleaved parity):



- ➜ strips are very small (single byte or word)
- ➜ simple parity bit based redundancy
- ➜ error detection
- ➜ partial error correction (if offender is known)

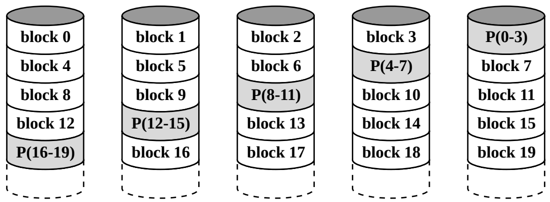
Slide 87

RAID 4 (block-level parity):



Slide 88

RAID 5 (block-level distributed parity):



Slide 89

RAID 6 (dual redundancy):