filesystems 2 / HDDs and SSDs

Changelog

Changes made in this version not seen in first lecture:

6 November: sector numbers: low sector numbers probably near faster outside of disk not center

last time

device driver top/bottom halfs

- “top half” — called from program request, checks buffer and waits
- “bottom half” — caled via interrupt, fills buffer and wakes

devices as magic memory

devices talk to memory: direct memory access (DMA)

- instead of reading or writing on-controller buffer

filesystem problems: finding files, space for files, ...

disk interface: read/write whole sectors

FAT

- file allocation table: linked list of clusters
- directories = files containing list of names + start clusters

start locations?

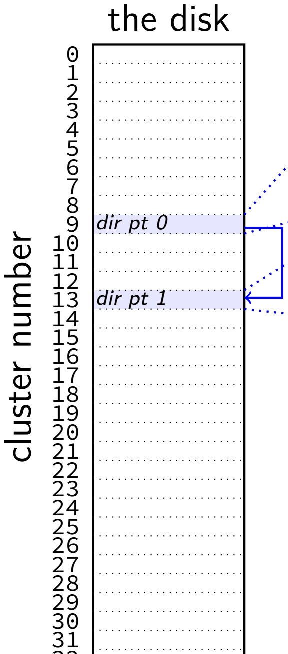
really want filenames

stored in directories!

in FAT: directory is a file, but its data is list of:

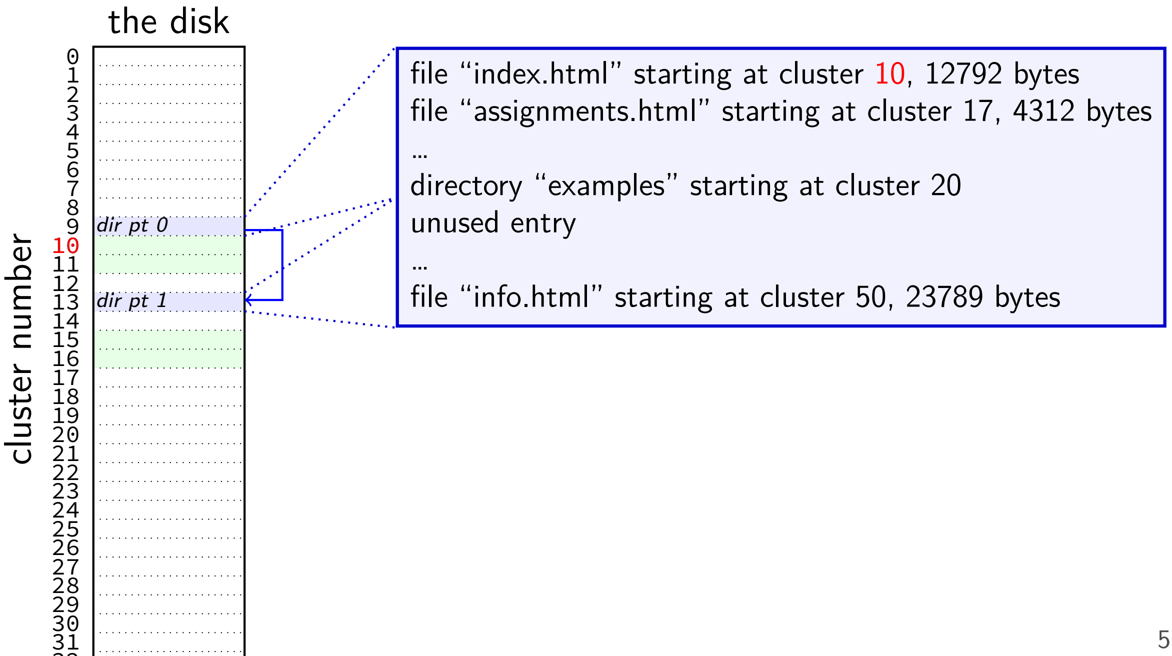
(name, starting location, other data about file)

finding files with directory

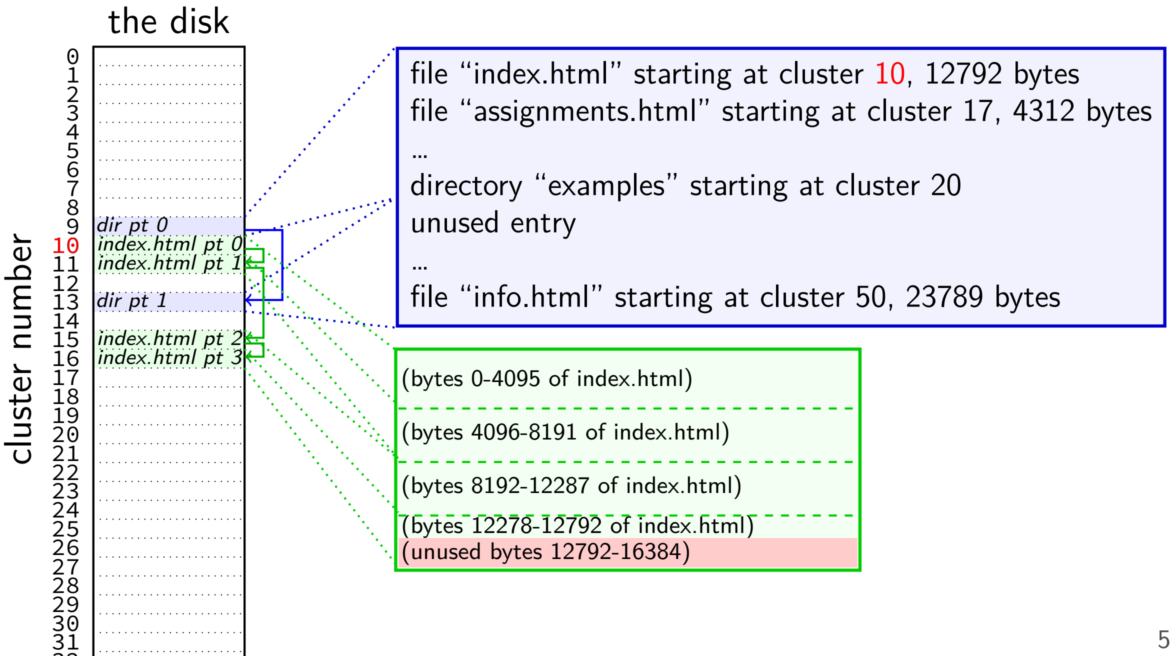


file “index.html” starting at cluster 10, 12792 bytes
file “assignments.html” starting at cluster 17, 4312 bytes
…
directory “examples” starting at cluster 20
unused entry
…
file “info.html” starting at cluster 50, 23789 bytes

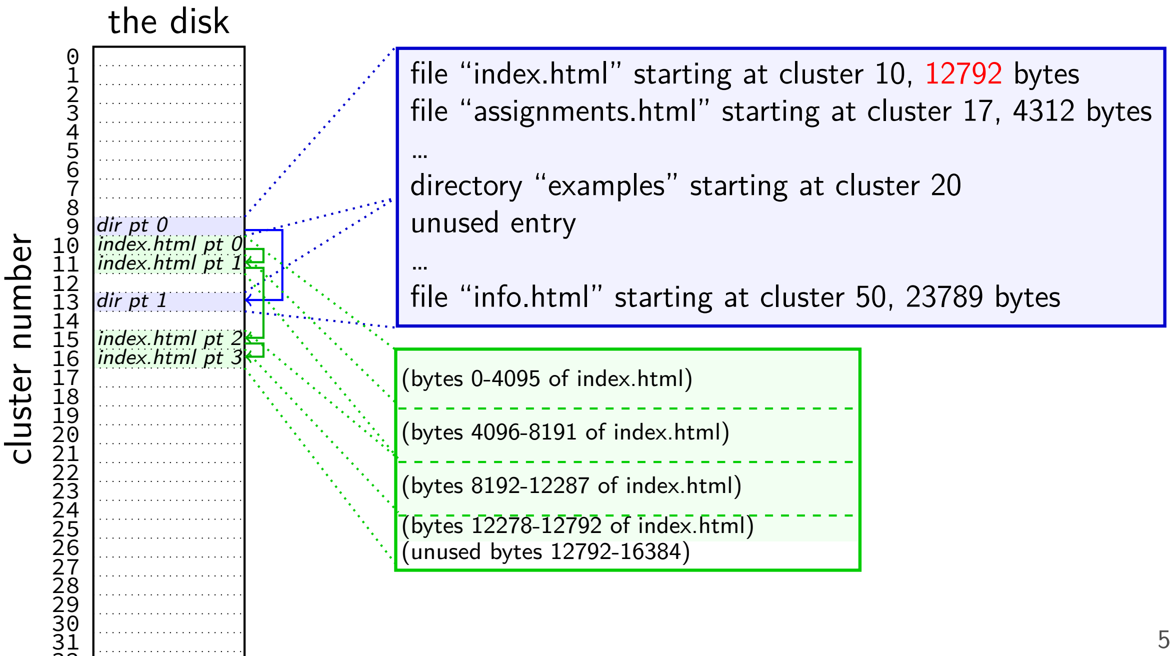
finding files with directory



finding files with directory



finding files with directory



FAT directory entry

box = 1 byte

entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4

`'R'`	`'E'`	`'A'`	`'D'`	`'M'`	`'E'`	`'␣'`	`'␣'`	`'T'`	`'X'`	`'T'`	`0x00`
filename + extension (`README.TXT`)											attrs

directory?
read-only?
hidden?
…

`0x9C`	`0xA1`	`0x20`	`0x7D`	`0x3C`	`0x7D`	`0x3C`	`0x01`	`0x00`	`0xEC`	`0x62`	`0x76`
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

`0x3C`	`0xF4`	`0x04`	`0x56`	`0x01`	`0x00`	`0x00`	`'F'`	`'O'`	`'O'`	…
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry…			

FAT directory entry

box = 1 byte

entry for README.TXT, 342 byte file, starting at cluster 0x104F4

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

32-bit first cluster number split into two parts
(history: used to only be 16-bits)

FAT directory entry

box = 1 byte

entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

8 character filename + 3 character extension
longer filenames? encoded using extra directory entries
(special `attrs` values to distinguish from normal entries)

FAT directory entry

box = 1 byte

entry for README.TXT, 342 byte file, starting at cluster 0x104F4

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

8 character filename + 3 character extension
history: used to be all that was supported

FAT directory entry

box = 1 byte

entry for README.TXT, 342 byte file, starting at cluster 0x104F4

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

attributes: is a subdirectory, read-only, ...
also marks directory entries used to hold extra filename data

FAT directory entry

box = 1 byte

entry for README.TXT, 342 byte file, starting at cluster 0x104F4

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

convention: if first character is 0x0 or 0xE5 — unused
0x00: for filling empty space at end of directory
0xE5: 'hole' — e.g. from file deletion

aside: FAT date encoding

seperate date and time fields (16 bits, little-endian integers)

bits 0-4: seconds (divided by 2), 5-10: minute, 11-15: hour

bits 0-4: day, 5-8: month, 9-15: year (minus 1980)

sometimes extra field for 100s(?) of a second

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Name[11];       // short name
  uint8_t DIR_Attr;           // File attribute
  uint8_t DIR_NTRes;          // set value to 0, never change t
  uint8_t DIR_CrtTimeTenth;   // millisecond timestamp for file
  uint16_t DIR_CrtTime;       // time file was created
  uint16_t DIR_CrtDate;       // date file was created
  uint16_t DIR_LstAccDate;    // last access date
  uint16_t DIR_FstClusHI;     // high word of this entry's firs
  uint16_t DIR_WrtTime;       // time of last write
  uint16_t DIR_WrtDate;       // dat eof last write
  uint16_t DIR_FstClusLO;     // low word of this entry's first
  uint32_t DIR_FileSize;      // file size in bytes
};
```

FAT directory entries (from C)

```c
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Name[11];           // short name
  uint8_t DIR_Attr;               // File attribute
  uint8_t DI                                                ge t
  uint8_t DI                                                file
  uint16_t D
  uint16_t D
  uint16_t DIR_LstAccDate;        // last access date
  uint16_t DIR_FstClusHI;         // high word of this entry's firs
  uint16_t DIR_WrtTime;           // time of last write
  uint16_t DIR_WrtDate;           // dat eof last write
  uint16_t DIR_FstClusLO;         // low word of this entry's first
  uint32_t DIR_FileSize;          // file size in bytes
};
```

GCC/Clang extension to disable padding

normally compilers add padding to structs

(to avoid splitting values across cache blocks or pages)

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
    uint8_t DIR_Name[11
    uint8_t DIR_Attr;
    uint8_t DIR_NTRes;                                                          ge t
    uint8_t DIR_CrtTime                                                         file
    uint16_t DIR_CrtTim
    uint16_t DIR_CrtDate;       // date file was created
    uint16_t DIR_LstAccDate;    // last access date
    uint16_t DIR_FstClusHI;     // high word of this entry's firs
    uint16_t DIR_WrtTime;       // time of last write
    uint16_t DIR_WrtDate;       // dat eof last write
    uint16_t DIR_FstClusLO;     // low word of this entry's first
    uint32_t DIR_FileSize;      // file size in bytes
};
```

8/16/32-bit unsigned integer
use exact size that's on disk
just copy byte-by-byte from disk to memory
(and everything happens to be little-endian)

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Nam
  uint8_t DIR_Att
  uint8_t DIR_NTR                                              ge t
  uint8_t DIR_CrtTimeTenth;   // millisecond timestamp for file
  uint16_t DIR_CrtTime;       // time file was created
  uint16_t DIR_CrtDate;       // date file was created
  uint16_t DIR_LstAccDate;    // last access date
  uint16_t DIR_FstClusHI;     // high word of this entry's firs
  uint16_t DIR_WrtTime;       // time of last write
  uint16_t DIR_WrtDate;       // dat eof last write
  uint16_t DIR_FstClusLO;     // low word of this entry's first
  uint32_t DIR_FileSize;      // file size in bytes
};
```

why are the names so bad ("FstClusHI", etc.)?
comes from Microsoft's documentation this way

nested directories

foo/bar/baz/file.txt

read root directory entries to find foo

read foo's directory entries to find bar

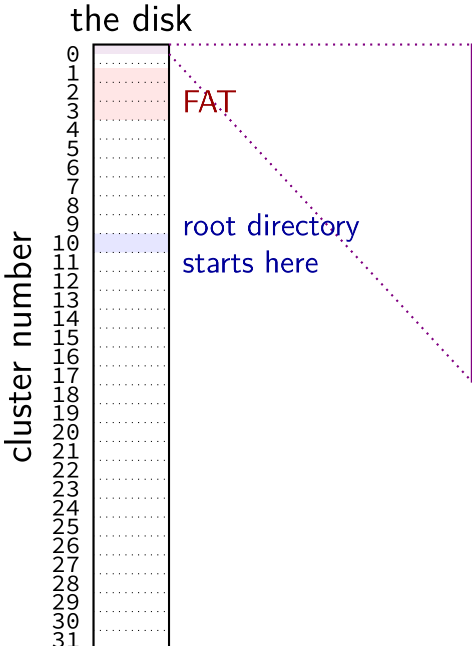
read bar's directory entries to find baz

read baz's directory entries to find file.txt

the root directory?

but where is the first directory?

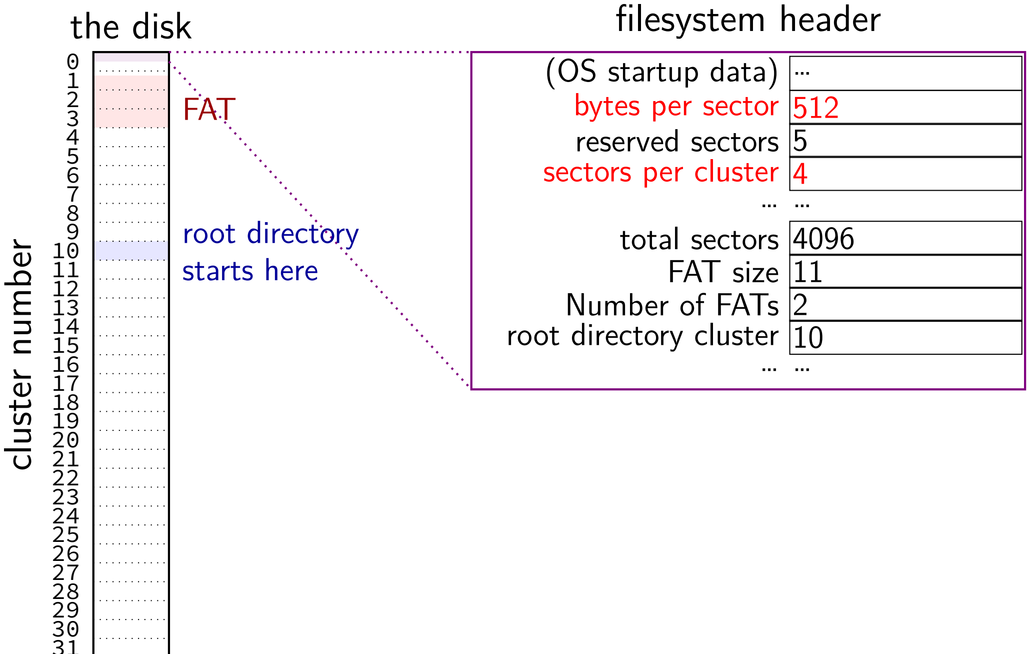
FAT disk header



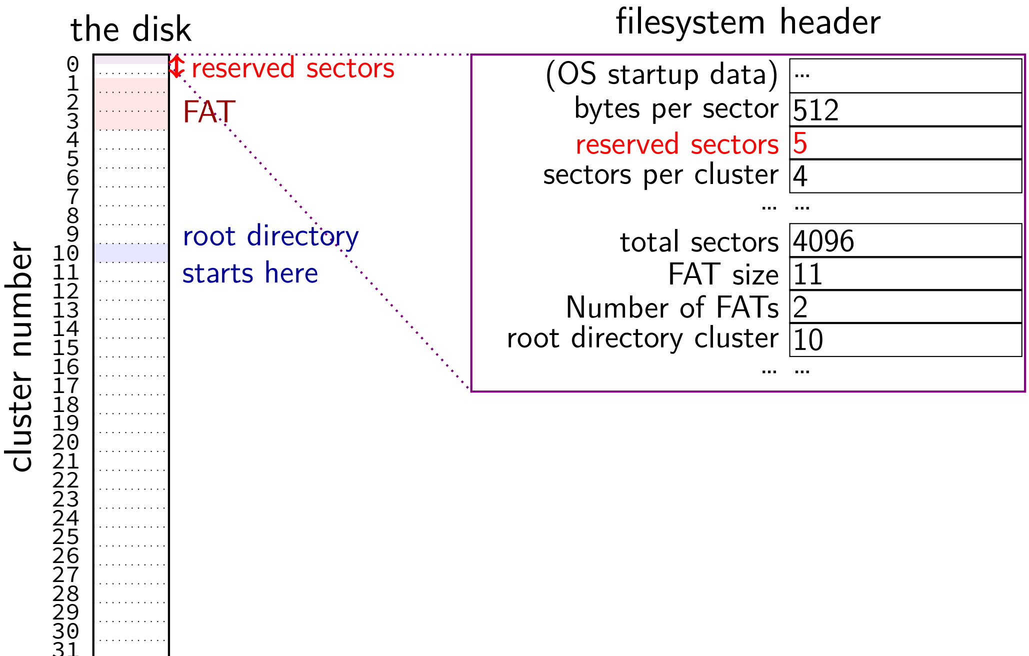
filesystem header

(OS startup data)	...
bytes per sector	512
reserved sectors	5
sectors per cluster	4
...	...
total sectors	4096
FAT size	11
Number of FATs	2
root directory cluster	10
...	...

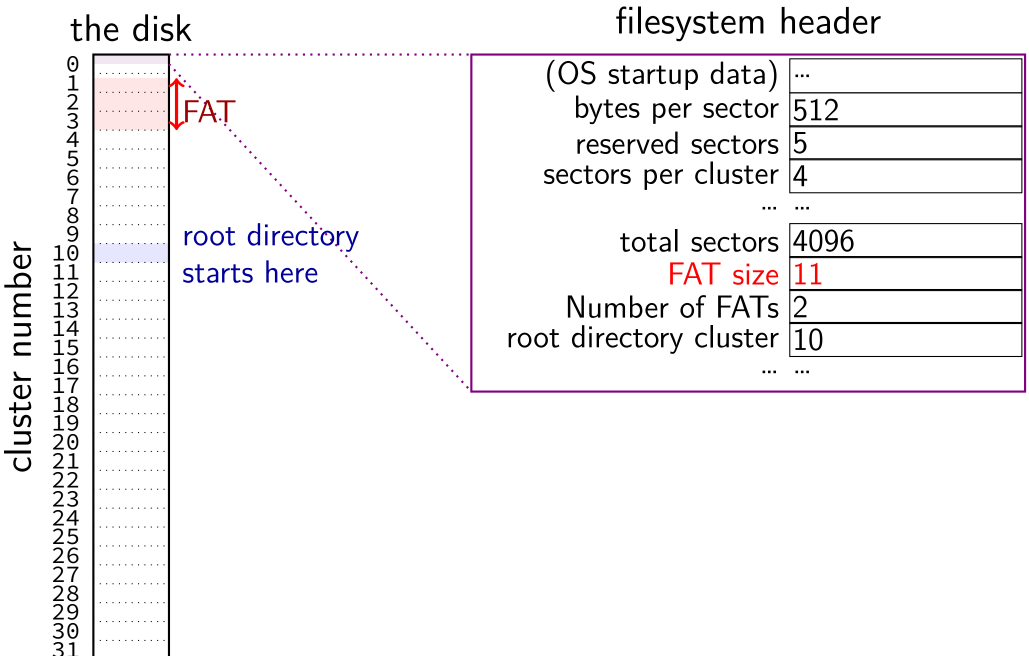
FAT disk header



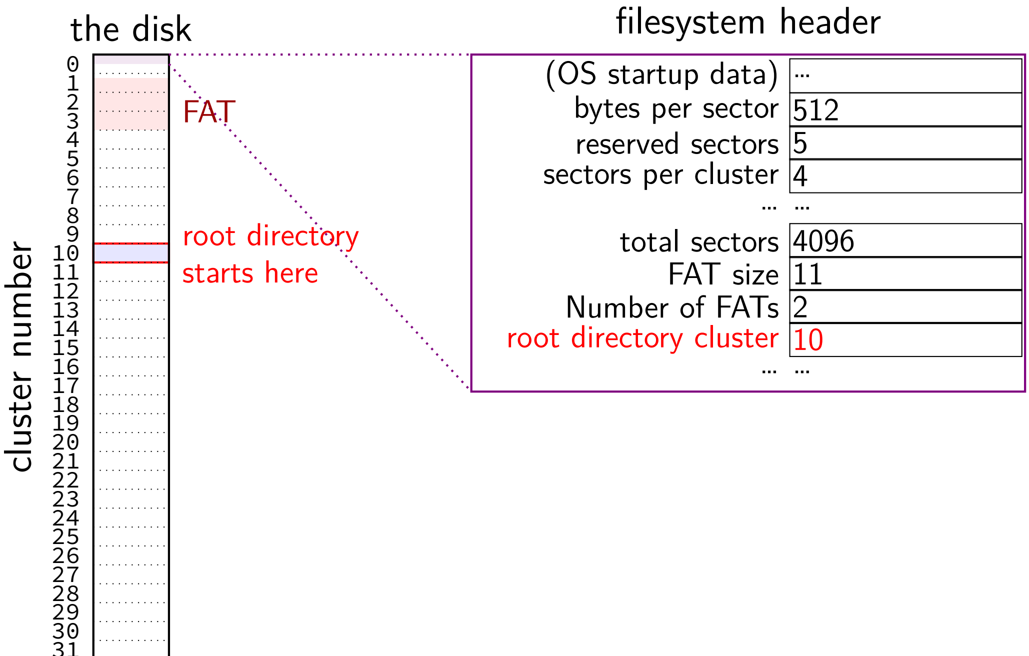
FAT disk header



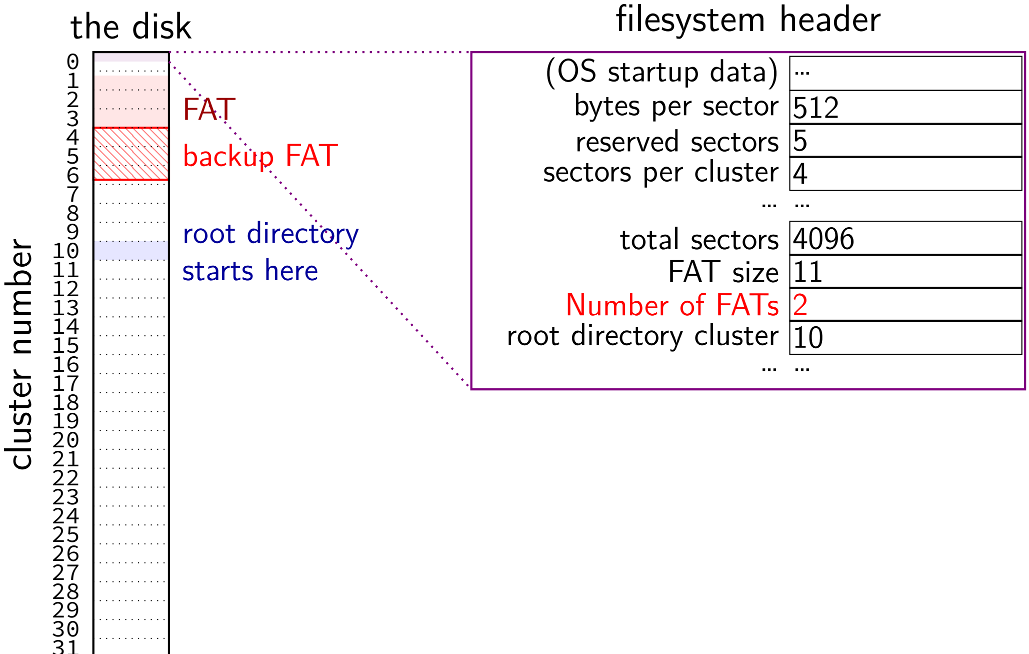
FAT disk header



FAT disk header



FAT disk header



filesystem header

fixed location near beginning of disk

determines size of clusters, etc.

tells where to find FAT, root directory, etc.

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_jmpBoot[3];    // jmp instr to boot code
  uint8_t BS_oemName[8];    // indicates what system formatted this
  uint16_t BPB_BytsPerSec;  // count of bytes per sector
  uint8_t BPB_SecPerClus;   // no.of sectors per allocation unit
  uint16_t BPB_RsvdSecCnt;  // no.of reserved sectors in the reserve
  uint8_t BPB_NumFATs;      // count of FAT datastructures on the vo
  uint16_t BPB_rootEntCnt;  // count of 32-byte entries in root dir,
  uint16_t BPB_totSec16;    // total sectors on the volume
  uint8_t BPB_media;        // value of fixed media
....
  uint16_t BPB_ExtFlags;    // flags indicating which FATs are activ
```

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_
  uint8_t BS_                                                    this
  uint16_t BPB_BytsPerSec;   // count of bytes per sector
  uint8_t BPB_SecPerClus;    // no.of sectors per allocation unit
  uint16_t BPB_RsvdSecCnt;   // no.of reserved sectors in the reserve
  uint8_t BPB_NumFATs;       // count of FAT datastructures on the vo
  uint16_t BPB_rootEntCnt;   // count of 32-byte entries in root dir,
  uint16_t BPB_totSec16;     // total sectors on the volume
  uint8_t BPB_media;         // value of fixed media
....
  uint16_t BPB_ExtFlags;     // flags indicating which FATs are activ
```

size of sector (in bytes) and size of cluster (in sectors)

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_jmpBoot[3];    // jmp instr to boot code
  uint8_t BS_oemName[8];    // indicates what system formatted this
  uint16_t BPB_BytsPerSec;  // count of bytes per sector
  uint8_t BPB_SecPerClus;   // n                              t
  uint16_t BPB_RsvdSecCnt;  // no.of reserved sectors in the reserve
  uint8_t BPB_NumFATs;      // count of FAT datastructures on the vo
  uint16_t BPB_rootEntCnt;  // count of 32-byte entries in root dir,
  uint16_t BPB_totSec16;    // total sectors on the volume
  uint8_t BPB_media;        // value of fixed media
....
  uint16_t BPB_ExtFlags;    // flags indicating which FATs are activ
```

space before file allocation table

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_jmpBoot[3];     // jmp instr to boot code
  uint8_t BS_oemName[8];     // indicates what system formatted this
  uint16_t BPB_BytsPerSec;   // count of bytes per sector
  uint8_t BPB_SecPerClus;                                     t
  uint16_t BPB_RsvdSecCnt;                                    serve
  uint8_t BPB_NumFATs;                                        he vo
  uint16_t BPB_rootEntCnt;                                     dir,
  uint16_t BPB_totSec16;     // total sectors on the volume
  uint8_t BPB_media;         // value of fixed media
....
  uint16_t BPB_ExtFlags;     // flags indicating which FATs are acti
```

number of copies of file allocation table
extra copies in case disk is damaged
typically two with writes made to both

FAT: creating a file

add a directory entry

choose clusters to store file data (how???)

update FAT to link clusters together

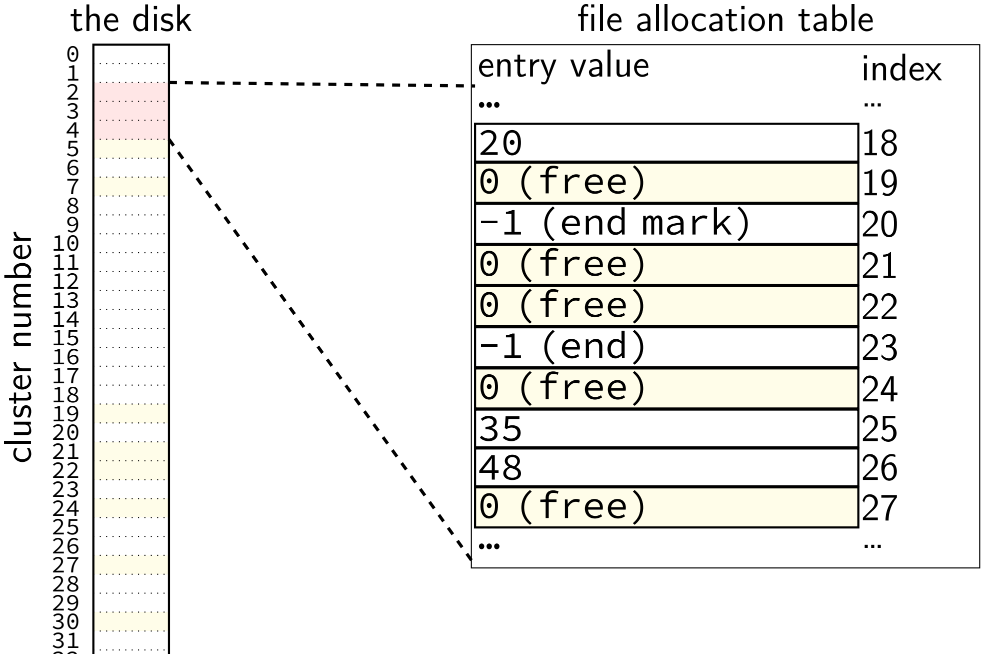
FAT: creating a file

add a directory entry

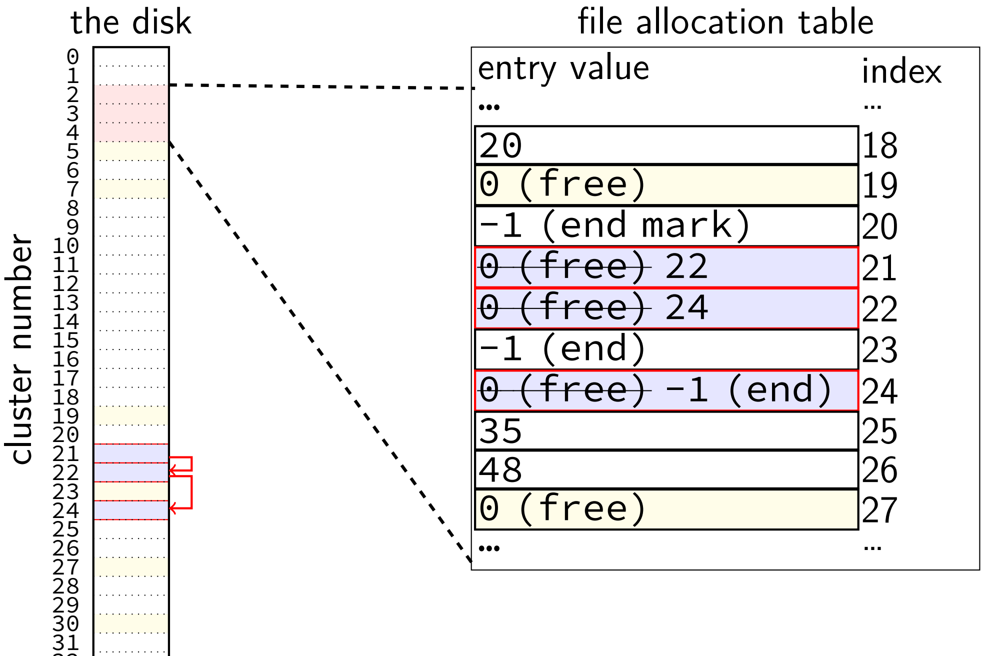
choose clusters to store file data (how???)

update FAT to link clusters together

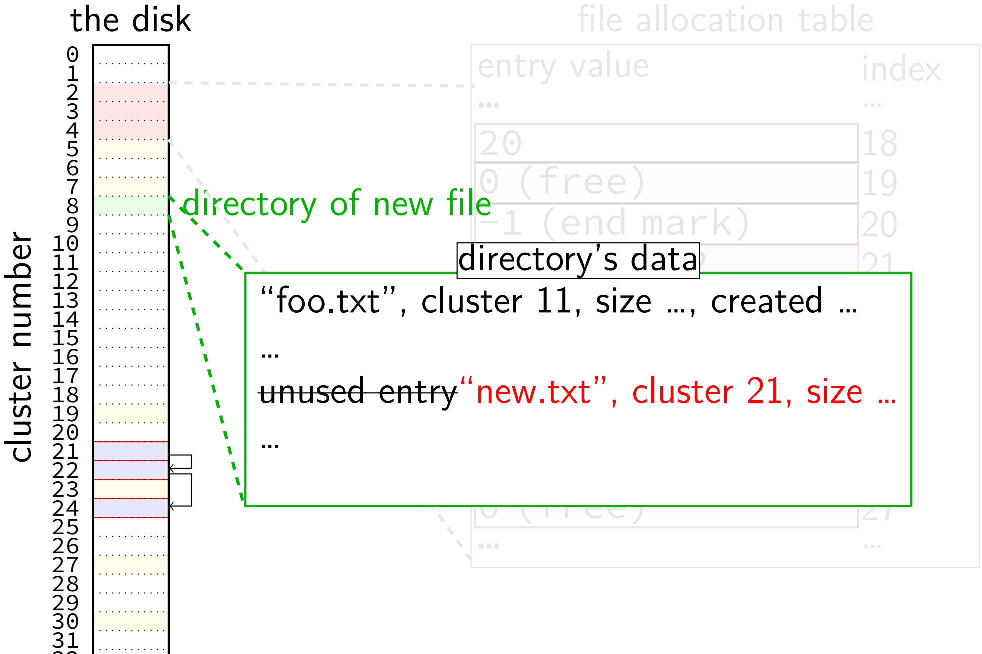
FAT: free clusters



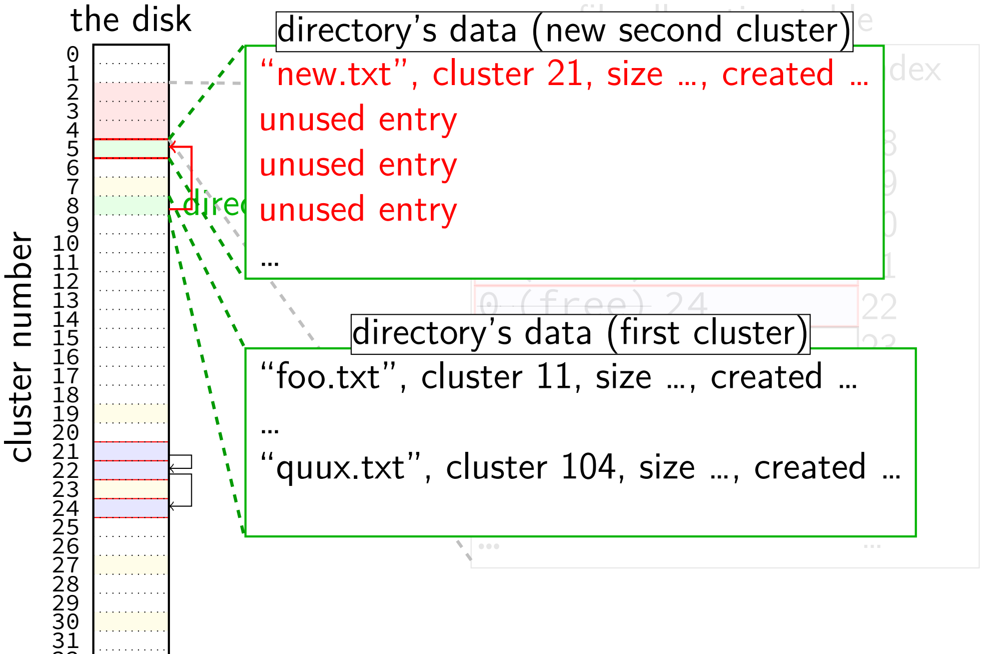
FAT: writing file data



FAT: replacing unused directory entry



FAT: extending directory



FAT: deleting files

reset FAT entries for file clusters to free (0)

write “unused” character in filename for directory entry

- maybe rewrite directory if that’ll save space?

exercise

say FAT filesystem with:

- 4-byte FAT entries
- 32-byte directory entries
- 2048-byte clusters

how many FAT entries+clusters (outside of the FAT) is used to store a directory of 200 30KB files?

- count clusters for both directory entries and the file data

how many FAT entries+clusters is used to store a directory of 2000 3KB files?

FAT pros and cons?

hard drive operation/performance

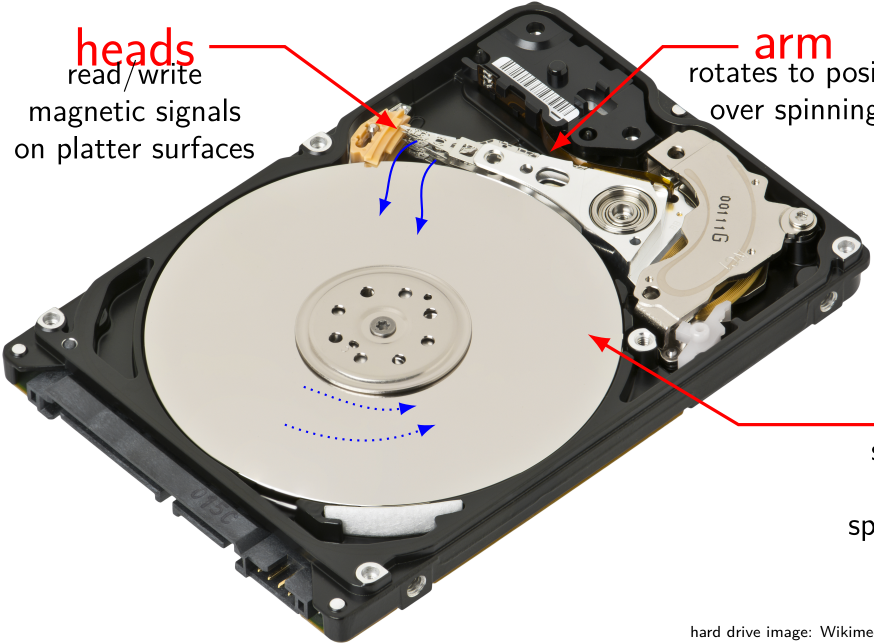
why hard drives?

what filesystems were designed for

currently most cost-effective way to have a lot of online storage

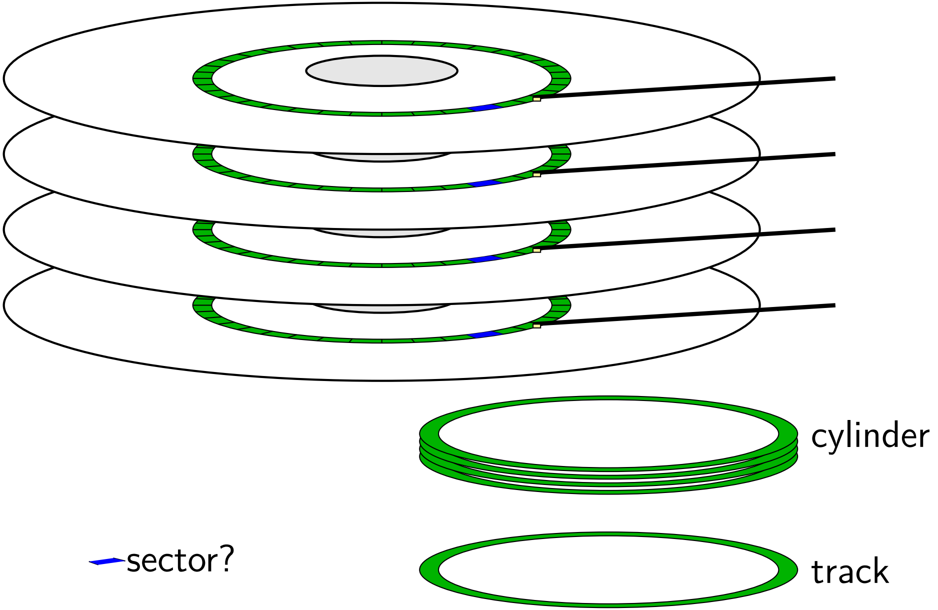
solid state drives (SSDs) imitate hard drive interfaces

hard drives



hard drive image: Wikimedia Commons / Evan-Amos

sectors/cylinders/etc.

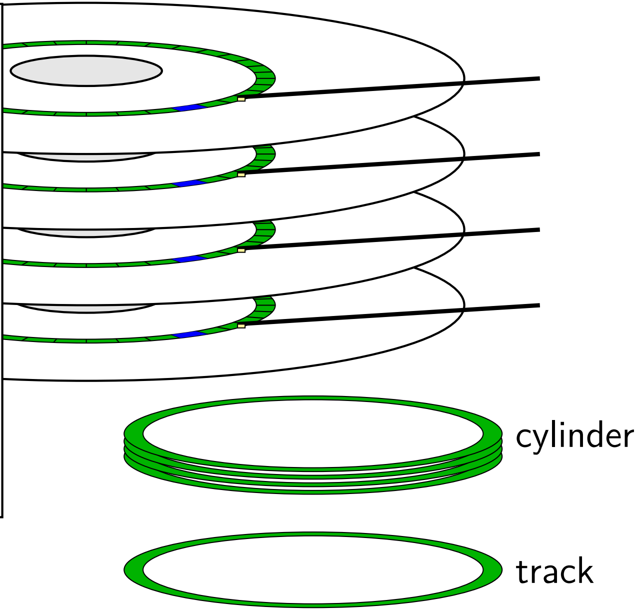


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data

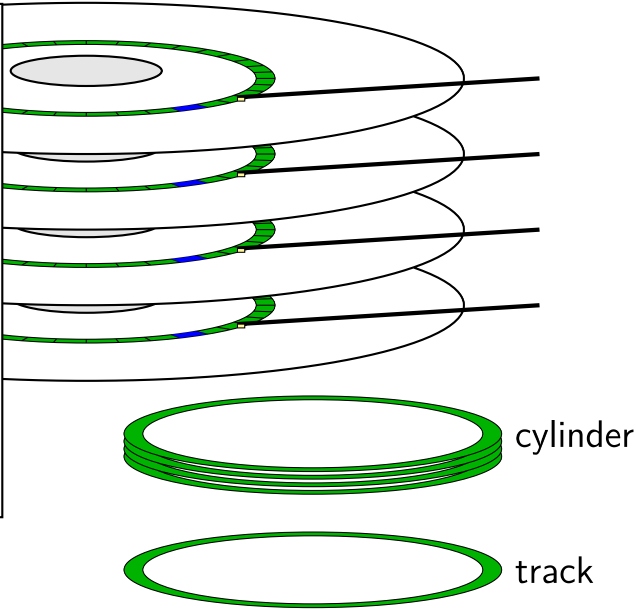


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



sectors/cylinders/etc.

seek time — 5–10ms

move heads to cylinder

faster for adjacent accesses

rotational latency — 2–8ms

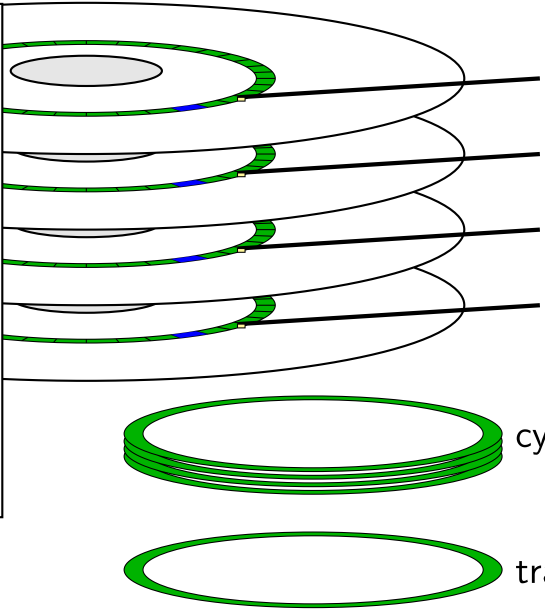
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data

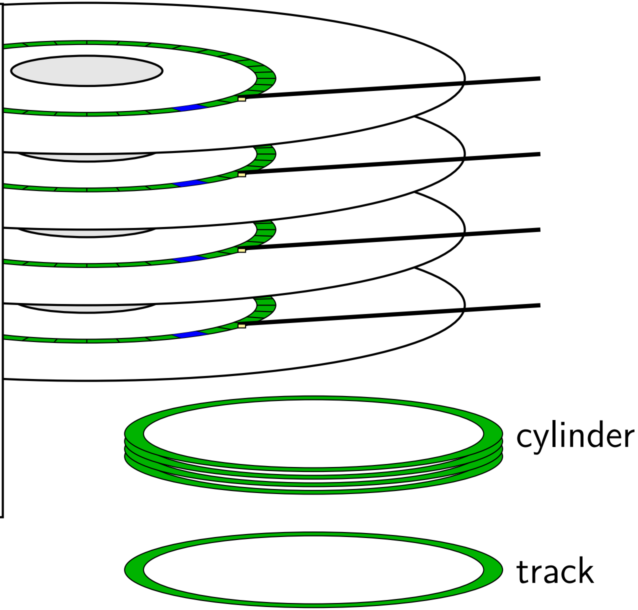


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



sector?

disk latency components

queue time — how long read waits in line?

- depends on number of reads at a time, scheduling strategy

disk controller/etc. processing time

seek time — head to cylinder

rotational latency — platter rotate to sector

transfer time

cylinders and latency

cylinders closer to edge of disk are faster (maybe)

less rotational latency

sector numbers

historically: OS knew cylinder/head/track location

now: opaque sector numbers

- more flexible for hard drive makers
- same interface for SSDs, etc.

typical pattern: low sector numbers = probably closer to edge (faster)

typical pattern: adjacent sector numbers = adjacent on disk

actual mapping: decided by disk controller

OS to disk interface

disk takes read/write requests

- sector number(s)
- location of data for sector
- modern disk controllers: typically direct memory access

can have queue of pending requests

disk processes them in some order

- OS can say "write X before Y"

hard disks are unreliable

Google study (2007), heavily utilized cheap disks

1.7% to 8.6% annualized failure rate

- varies with age
- $\approx$ chance a disk fails each year
- disk fails $=$ needs to be replaced

9% of working disks had reallocated sectors

bad sectors

modern disk controllers do sector remapping

part of physical disk becomes bad — use a different one

- disk uses error detecting code to tell data is bad
- similar idea to storing + checking hash of data

this is expected behavior

maintain mapping (special part of disk, probably)

queuing requests

recall: multiple active requests

queue of reads/writes

- in disk controller *and/or* OS

disk is faster for adjacent/close-by reads/writes

- less seek time/rotational latency

disk scheduling

schedule I/O to the disk

schedule = decide what read/write to do next

- by OS: what to request from disk next?
- by controller: which OS request to do next?

typical goals:

minimize seek time

don't starve requiests

disk scheduling

schedule I/O to the disk

schedule = decide what read/write to do next

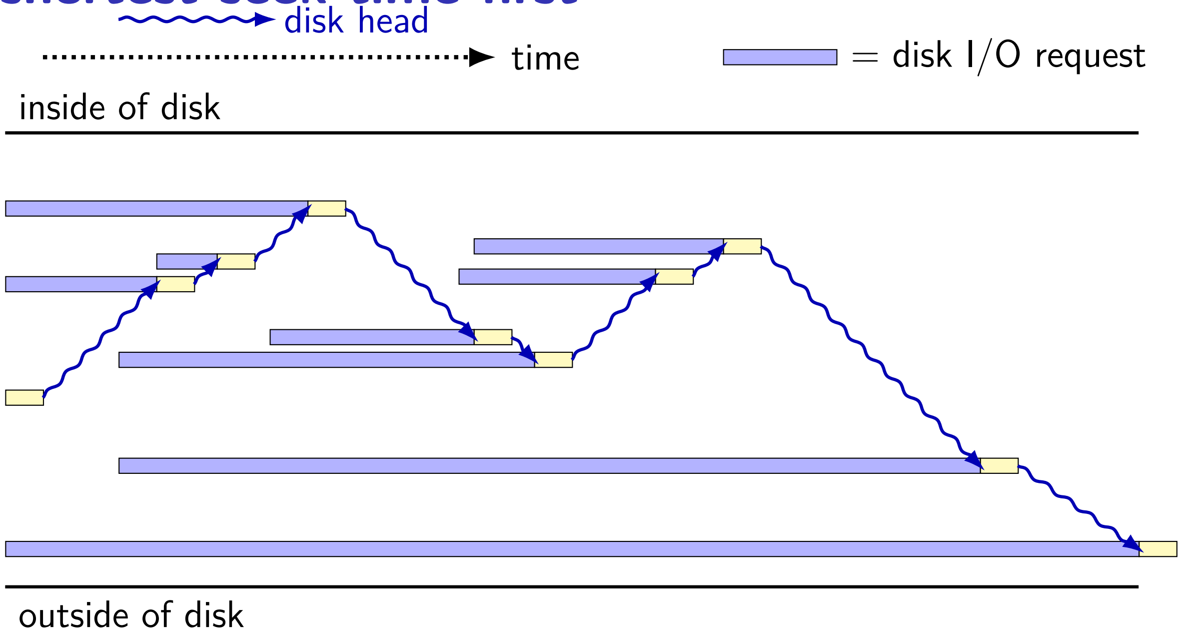
- by OS: what to request from disk next?
- by controller: which OS request to do next?

typical goals:

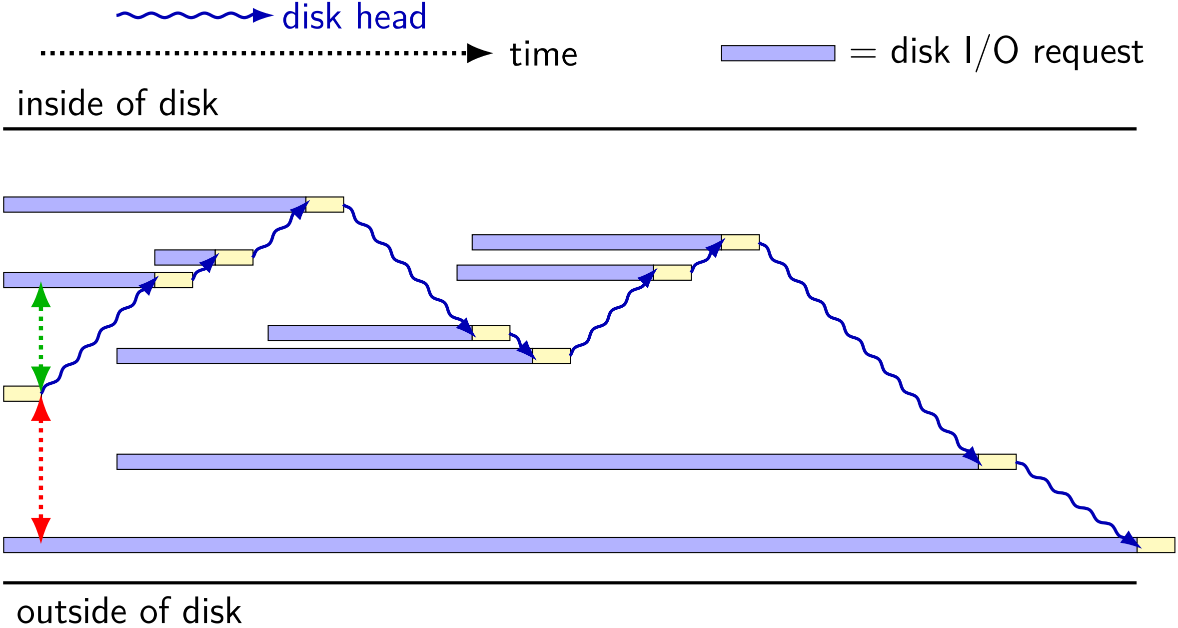
minimize seek time

don't starve requiests

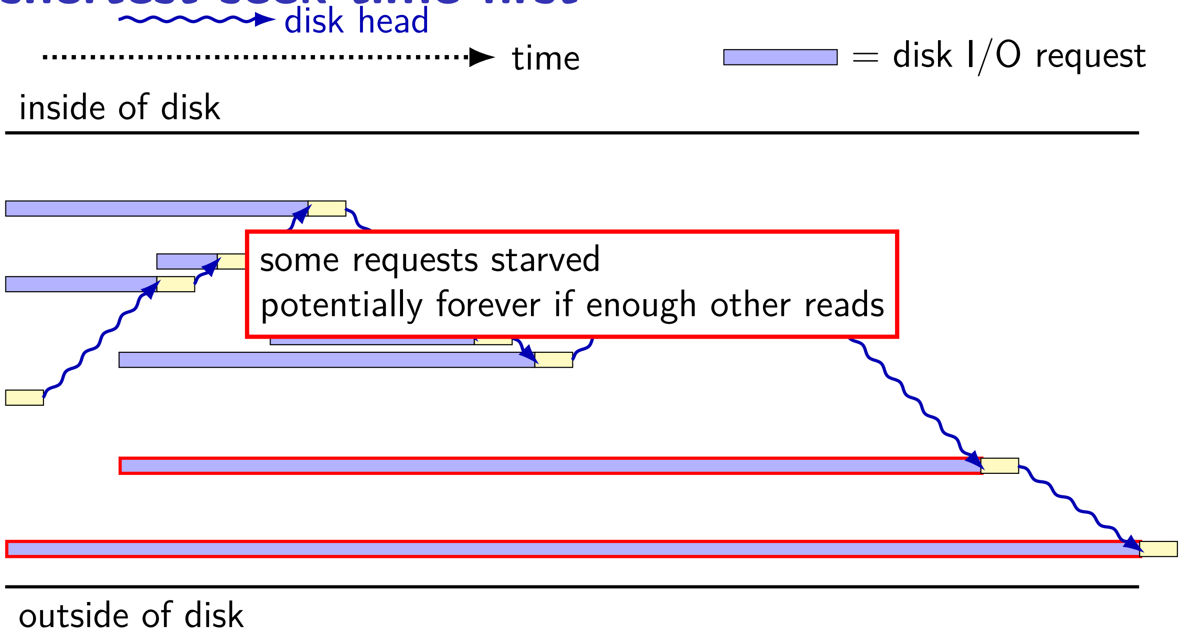
shortest seek time first



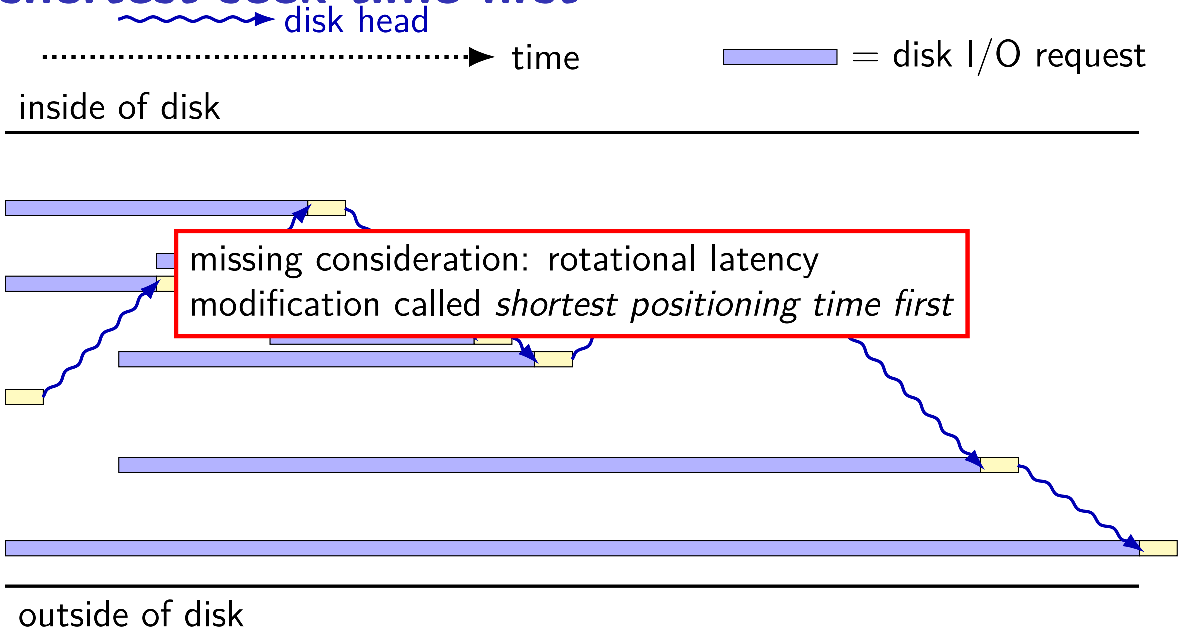
shortest seek time first



shortest seek time first



shortest seek time first



missing consideration: rotational latency
modification called *shortest positioning time first*

disk scheduling

schedule I/O to the disk

schedule = decide what read/write to do next

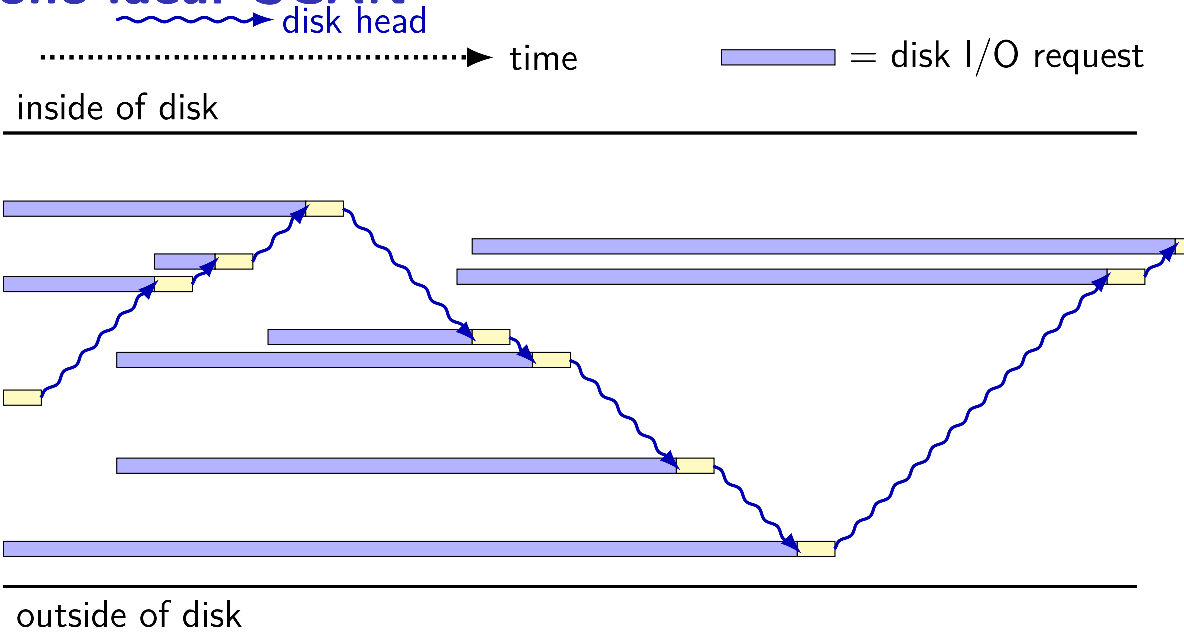
- by OS: what to request from disk next?
- by controller: which OS request to do next?

typical goals:

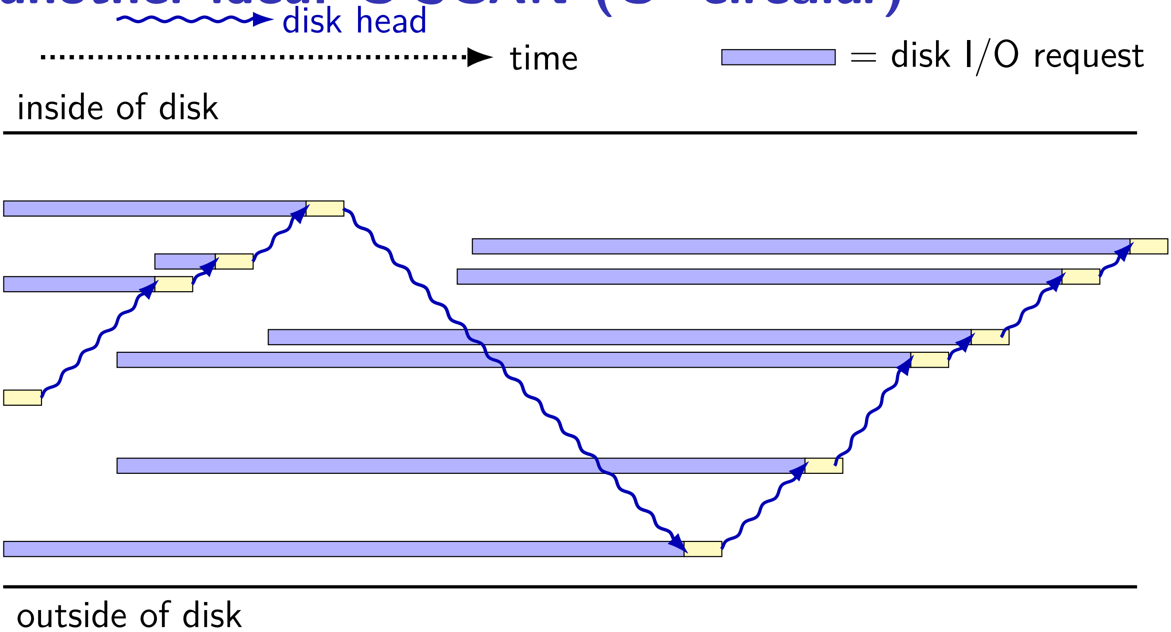
minimize seek time

don't starve requiests

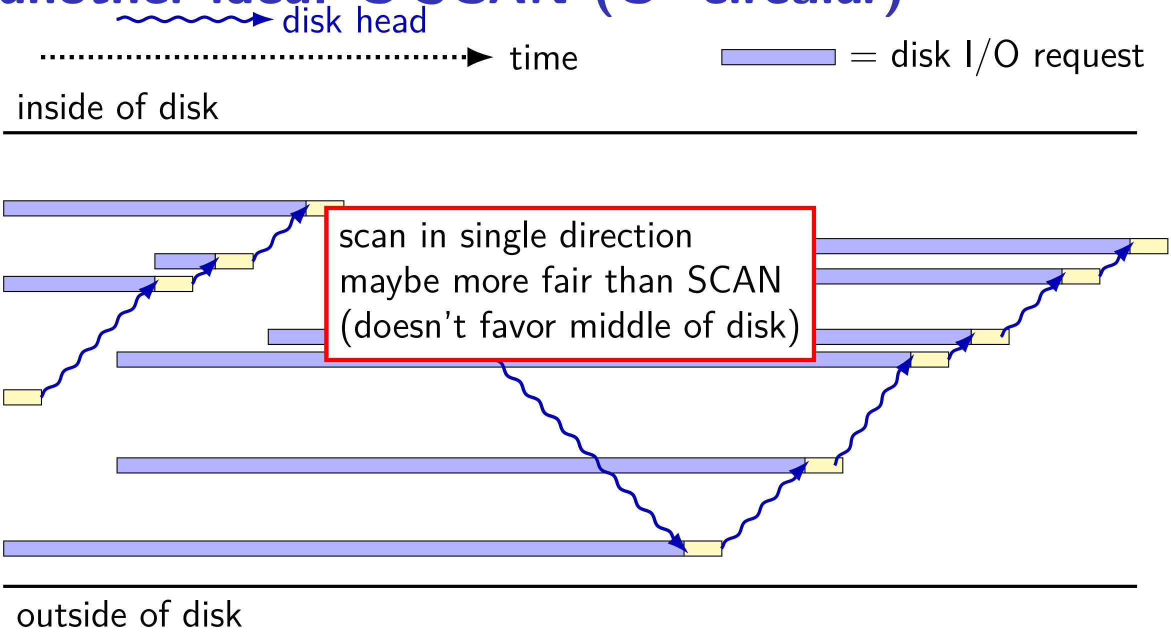
one idea: SCAN



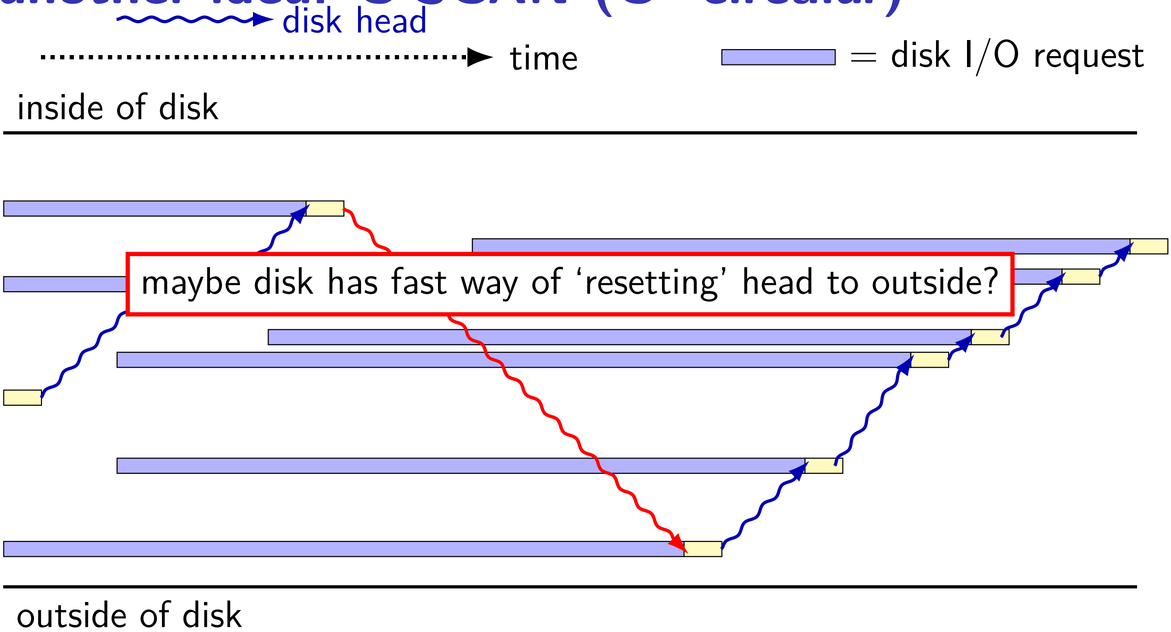
another idea: C-SCAN (C=circular)



another idea: C-SCAN (C=circular)



another idea: C-SCAN (C=circular)



some disk scheduling algorithms (text)

SSTF: take request with shortest seek time next

- subject to starvation — stuck on one side of disk
- could also take into account rotational latency — yields SPTF
  - shortest positioning time first

SCAN/elevator: move disk head towards center, then away

- let requests pile up between passes
- limits starvation; good overall throughput

C-SCAN: take next request closer to center of disk (if any)

- variant of scan that moves head in one direction
- avoids bias towards center of disk

caching in the controller

controller often has a DRAM cache

can hold things controller thinks OS might read

- e.g. sectors ‘near’ recently read sectors
- helps hide sector remapping costs?

can hold data waiting to be written

- makes writes a lot faster
- problem for reliability

disk performance and filesystems

filesystem can...

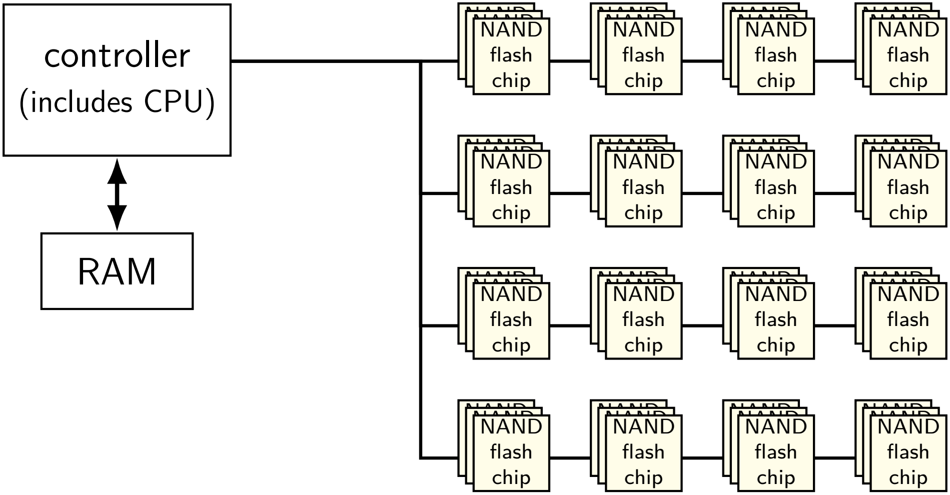
do contiguous or nearby reads/writes

- bunch of consecutive sectors much faster to read
- nearby sectors have lower seek/rotational delay

start a lot of reads/writes at once

- avoid reading something to find out what to read next
- array of sectors better than linked list

solid state disk architecture



flash

no moving parts

- no seek time, rotational latency

can read in sector-like sizes (“pages”) (e.g. 4KB or 16KB)

write once between erasures

erasure only in large *erasure blocks* (often 256KB to megabytes!)

can only rewrite blocks order tens of thousands of times

- after that, flash starts failing

SSDs: flash as disk

SSDs: implement hard disk interface for NAND flash

- read/write sectors at a time
- sectors much smaller than erasure blocks
- sectors sometimes smaller than flash ‘pages’
- read/write with use sector numbers, not addresses
- queue of read/writes

need to hide erasure blocks

- trick: block remapping — move where sectors are in flash

need to hide limit on number of erases

- trick: wear levening — spread writes out

block remapping

Flash Translation Layer

remapping table

logical	physical
0	93
1	260
…	…
31	74
32	75
…	…

OS sector numbers

flash locations

block remapping

Flash Translation Layer

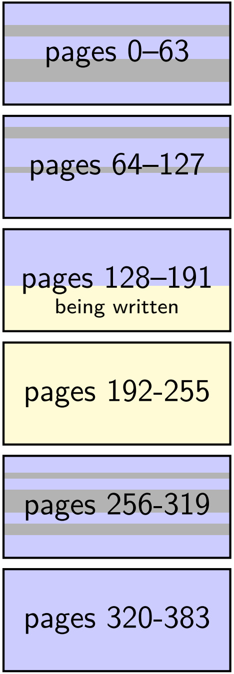
remapping table

logical	physical
0	93
1	260
...	...
31	74
32	75
...	...

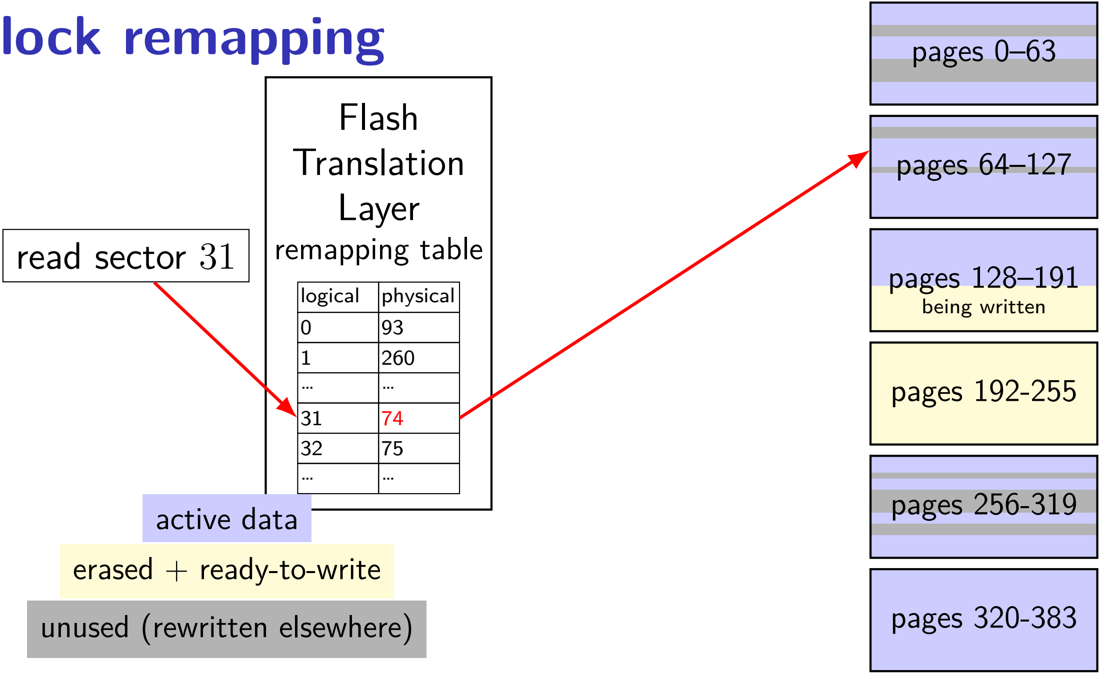
active data

erased + ready-to-write

unused (rewritten elsewhere)



block remapping



logical	physical
0	93
1	260
…	…
31	74
32	75
…	…

block remapping

write sector 32

Flash Translation Layer

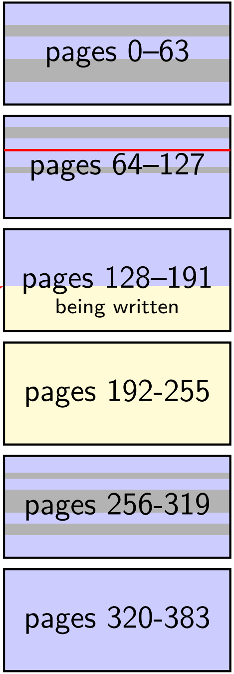
remapping table

logical	physical
0	93
1	260
...	...
31	74
32	~~75~~ 163
...	...

active data

erased + ready-to-write

unused (rewritten elsewhere)



block remapping

Flash Translation Layer

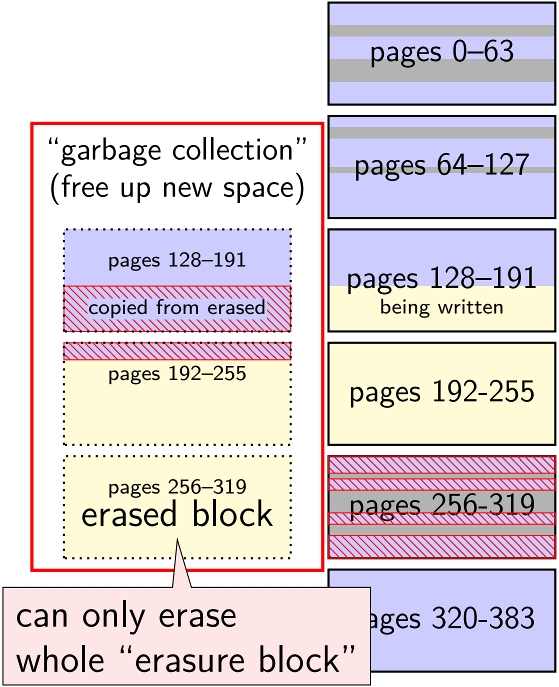
remapping table

logical	physical
0	93
1	~~260~~ 187
...	...
31	74
32	~~75~~ 163
...	...

active data

erased + ready-to-write

unused (rewritten elsewhere)



block remapping

controller contains mapping: sector $\rightarrow$ location in flash

on write: write sector to *new location*

eventually do *garbage collection* of sectors

- if erasure block contains some replaced sectors and some current sectors…
- copy current blocks to new locationt to reclaim space from replaced sectors

doing this efficiently is very complicated

SSDs sometimes have a 'real' processor for this purpose

SSD performance

reads/writes: sub-millisecond

contiguous blocks don't really matter

can depend a lot on the controller

- faster/slower ways to handle block remapping

writing can be slower, especially when almost full

- controller may need to move data around to free up erasure blocks
- erasing an erasure block is pretty slow (milliseconds?)

extra SSD operations

SSDs sometimes implement non-HDD operations

on operation: TRIM

way for OS to mark sectors as unused/erase them

SSD can remove sectors from block map

- more efficient than zeroing blocks
- frees up more space for writing new blocks

aside: future storage

emerging non-volatile memories...

slower than DRAM ("normal memory")

faster than SSDs

read/write interface like DRAM but persistent

capacities similar to/larger than DRAM

xv6 filesystem

xv6's filesystem similar to modern Unix filesytems

better at doing contiguous reads than FAT

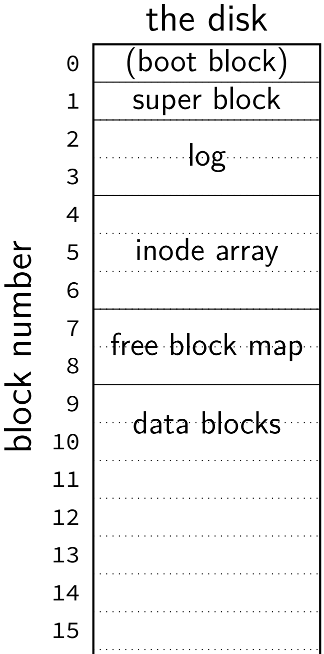
better at handling crashes

supports *hard links* (more on these later)

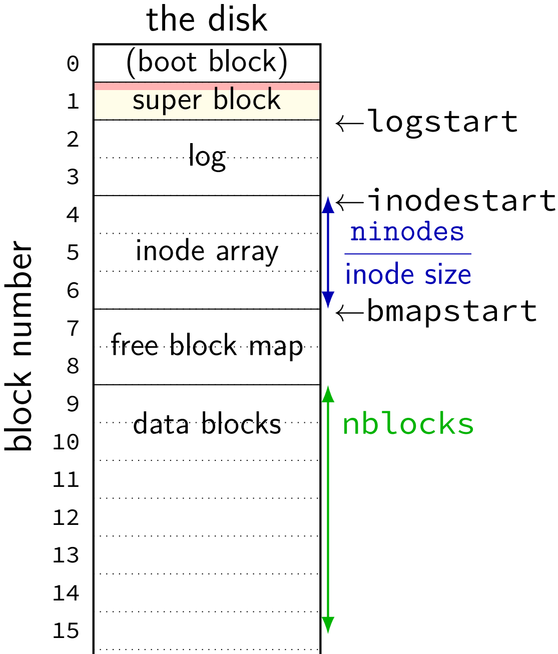
divides disk into *blocks* instead of clusters

file block numbers, free blocks, etc. in different tables

xv6 disk layout



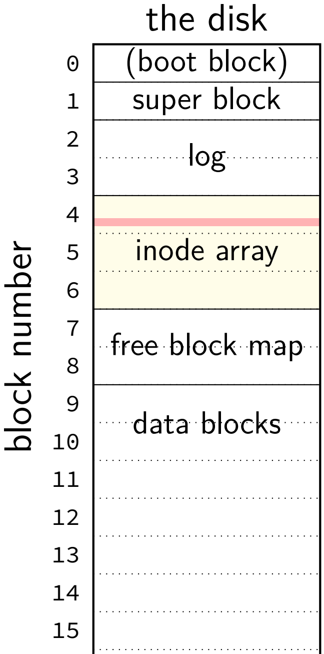
xv6 disk layout



superblock — "header"

```
struct superblock {
  uint size;
    // Size of file system image (b
  uint nblocks;
    // # of data blocks
  uint ninodes;
    // # of inodes
  uint nlog;
    // # of log blocks
  uint logstart;
    // block # of first log block
  uint inodestart;
    // block # of first inode block
  uint bmapstart;
    // block # of first free map bl
};
```

xv6 disk layout



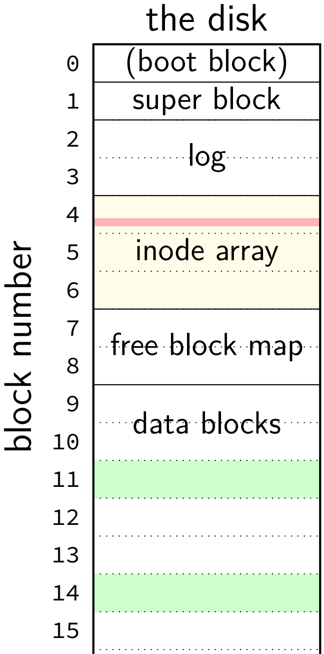
inode — file information

```
struct dinode {
   short type;   // File type
     // T_DIR, T_FILE, T_DEV

   short major; short minor; // T_DEV only

   short nlink;
     // Number of links to inode in file syst
   uint size;    // Size of file (bytes)
   uint addrs[NDIRECT+1];
     // Data block addresses
};
```

xv6 disk layout



inode — file information

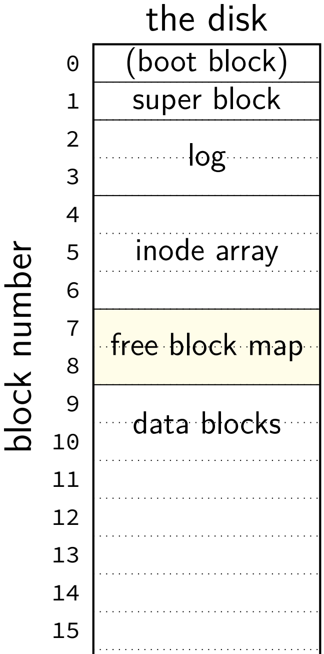
```
struct dinode {
   short type;  // File type
     // T_DIR, T_FILE, T_DEV

   short major; short minor; // T_DEV only

   short nlink;
     // Number of links to inode in file syst
   uint size;   // Size of file (bytes)
   uint addrs[NDIRECT+1];
     // Data block addresses
};
```

location of data as block numbers:
e.g. addrs[0] = 11; addrs[1] = 14;
special case for larger files

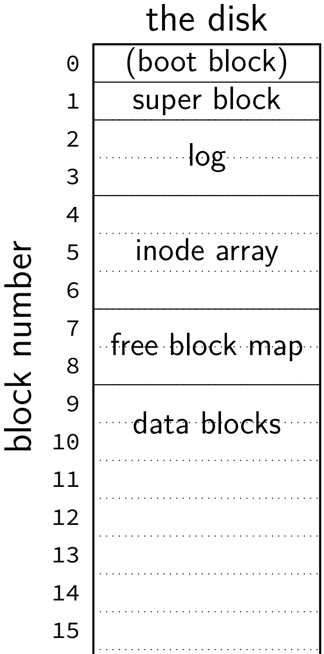
xv6 disk layout



free block map — 1 bit per data block
1 if available, 0 if used

allocating blocks: scan for 1 bits
contiguous 1s — contigous blocks

xv6 disk layout



what about finding free inodes

xv6 solution: scan for type = 0

typical Unix solution: separate free inode map

xv6 directory entries

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

inum — index into inode array on disk

name — name of file or directory

each directory reference to inode called a *hard link*

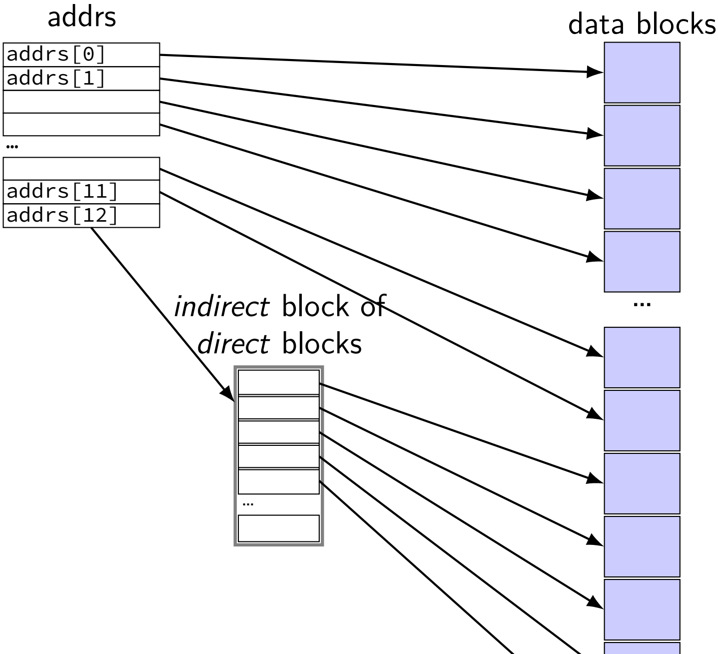
  multiple hard links to file allowed!

xv6 allocating inodes/blocks

need new inode or data block: linear search

simplest solution: xv6 always takes the first one that's free

xv6 inode: direct and indirect blocks



xv6 file sizes

512 byte blocks

2-byte block pointers: 256 block pointers in the indirect block

256 blocks = 131072 bytes of data referenced

12 direct blocks @ 512 bytes each = 6144 bytes

1 indirect block @ 131072 bytes each = 131072 bytes

maximum file size

backup slides

error correcting codes

disk store 0s/1s magnetically

- very, very, very small and fragile

magnetic signals can fade over time/be damaged/interfere/etc.

but use error detecting+correcting codes

- details? CS/ECE 4434 covers this

error detecting — can tell OS “don’t have data”

- result: data corruption is very rare
- data loss much more common

error correcting codes — extra copies to fix problems

- only works if not too many bits damaged