Changelog

Changes made in this version not seen in first lecture:

- 6 November: Correct center to edge in several places and be more cagey about whether the edge is faster or not
- 6 November: disk scheduling: put SSTF abbervation on slide
- 6 November: SSDs: remove remarks about set to 1s as confusing

last time

I/O: DMA

FAT filesystem
- divided into clusters (one or more sectors)
- table of integers per cluster
- in file: table entry = number of next cluster
- special value indicates end of file
- out of file: table entry = 0 for free

how disks work (start)
- cylinders, tracks, sectors
- seek time, rotational latency, etc.

missing detail on FAT

multiple copies of file allocation table

typically (but not always) contain same information

idea: part of disk can fail

want to be able to still read the FAT if so

$\rightarrow$ backup copy

note on due dates

FAT due dates moved to Mondays

- caveat: I may not provide much help on weekends

final assignment due last day of class, but...

will not accept submissions after final exam (10 December)

no DMA?

anonymous feedback question: “Can you elaborate on what devices do when they don’t support DMA?”

still connected to CPU via some sort of bus

- typically same bus CPU uses to access memory

CPU writes to/reads from this bus to access device controller

without DMA: this is how *data and status and commands* are transferred

with DMA: this how *status and commands* are transferred

- device retrieves data from memory

why hard drives?

what filesystems were designed for

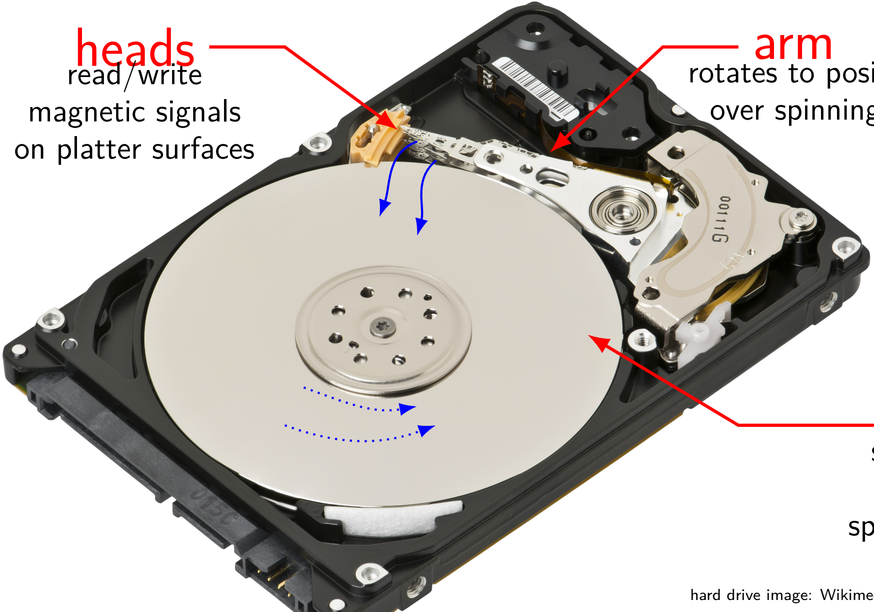
currently most cost-effective way to have a lot of online storage

solid state drives (SSDs) imitate hard drive interfaces

hard drives

heads
read/write
magnetic signals
on platter surfaces

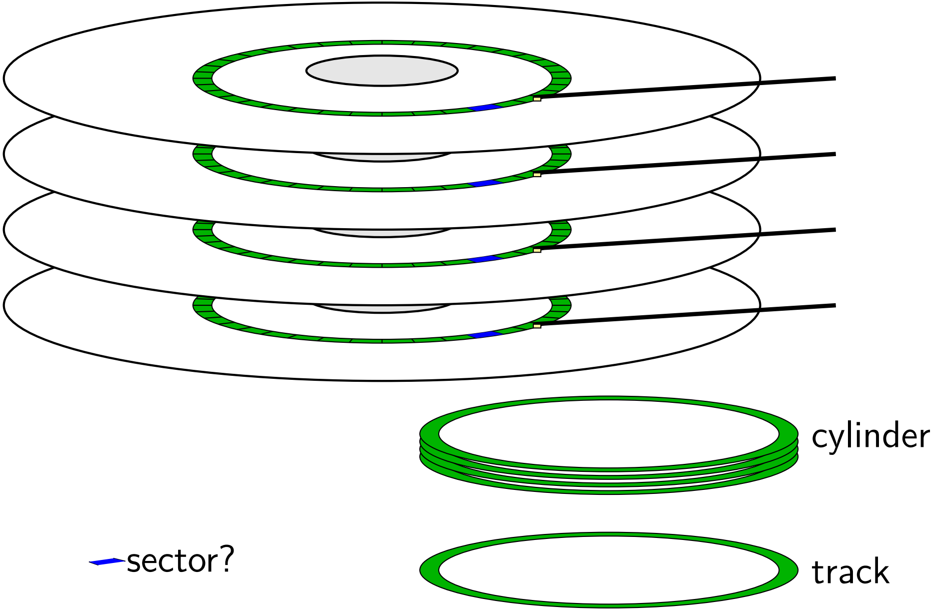
arm
rotates to position heads
over spinning platters



platters
stack of flat discs
(only top visible)
spins when operating

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

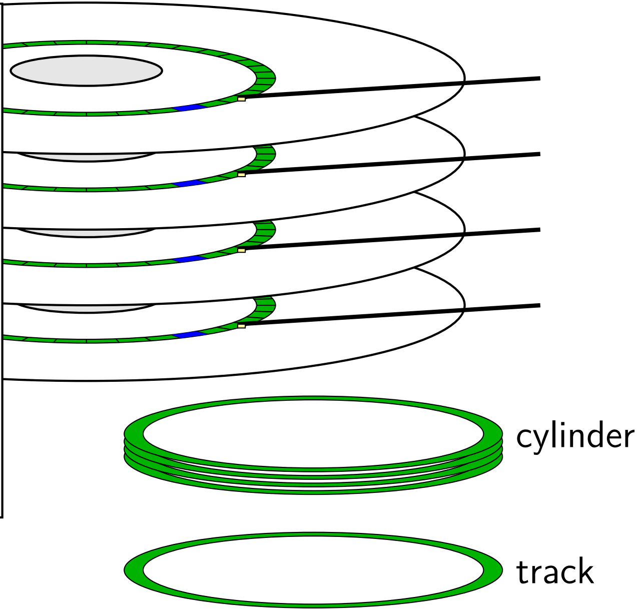
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data



sector?

sectors/cylinders/etc.

seek time — 5–10ms

move heads to cylinder

faster for adjacent accesses

rotational latency — 2–8ms

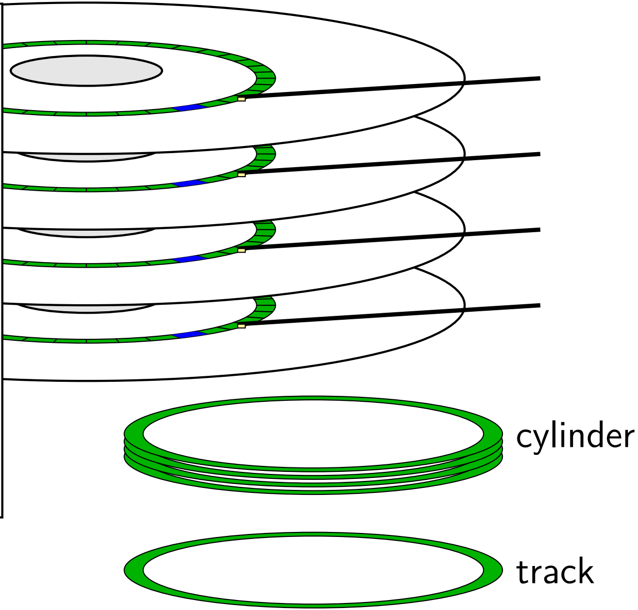
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data



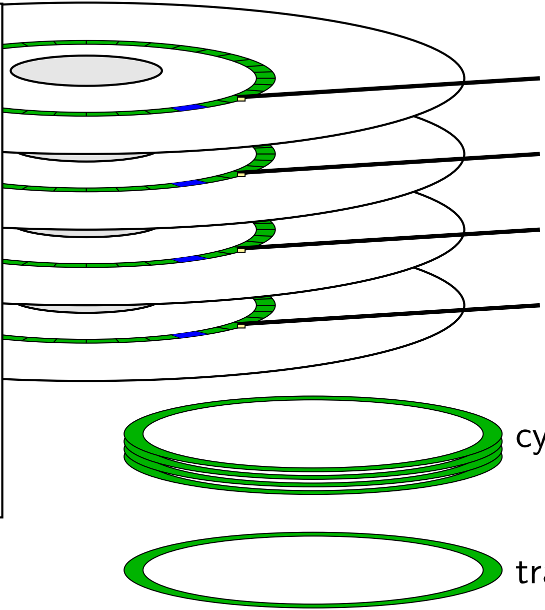
sector?

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

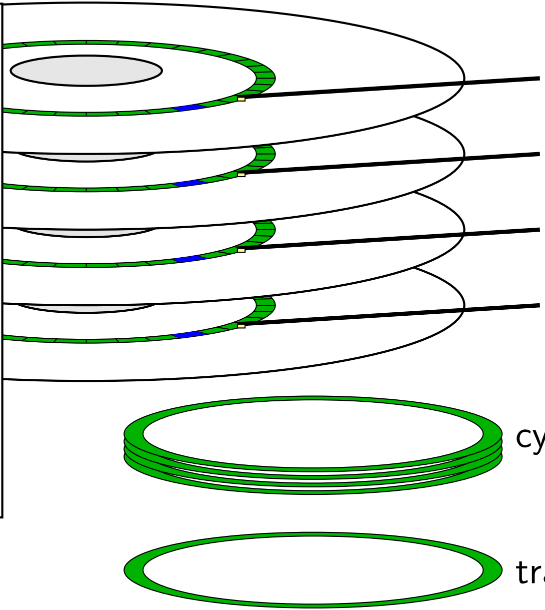


sectors/cylinders/etc.

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 = closer to center

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
- $\approx$ 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

this is expected behavior

maintain mapping (special part of disk)

error correcting codes

disk store 0s/1s magnetically

- very, very, very small and fragile space

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

but use error detecting+correcting codes

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

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

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

typical goals:

minimize seek time

don't starve requiests

some disk scheduling algorithms

SSTF: take request with shortest seek time next

subject to starvation — stuck on one side of disk

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)

take requests when moving from outside of disk to inside
let requests pile up between passes
limits starvation; good overall throughput

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 reads/writes

- bunch of consecutive sectors much faster to read

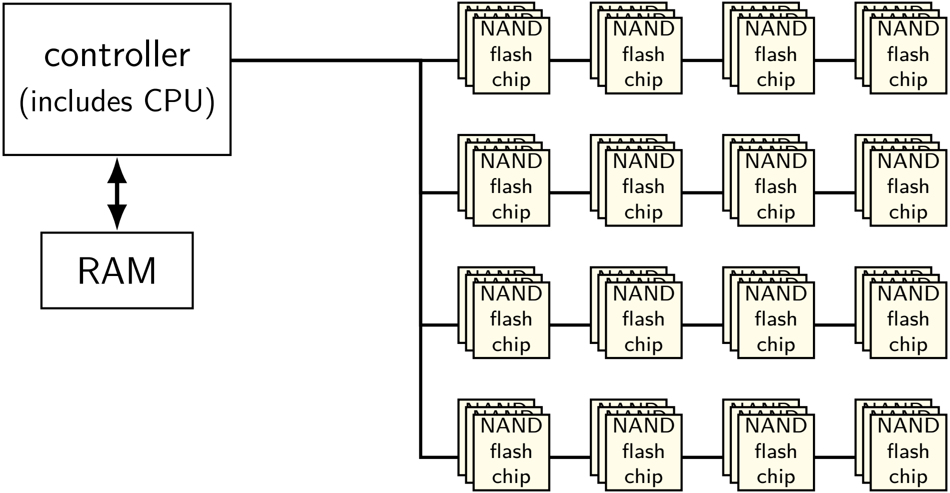
filesystem can start a lot of reads/writes at once

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

filesystem can keep important data close to maybe faster edge of disk

- e.g. disk header/file allocation table
- disk typically has lower sector numbers for faster parts

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

- afte that, flash fails

SSDs: flash as disk

SSDs: implement hard disk interface for NAND flash

- read/write sectors at a time
- 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

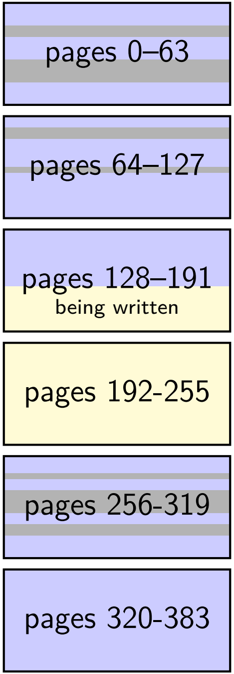
remapping table

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

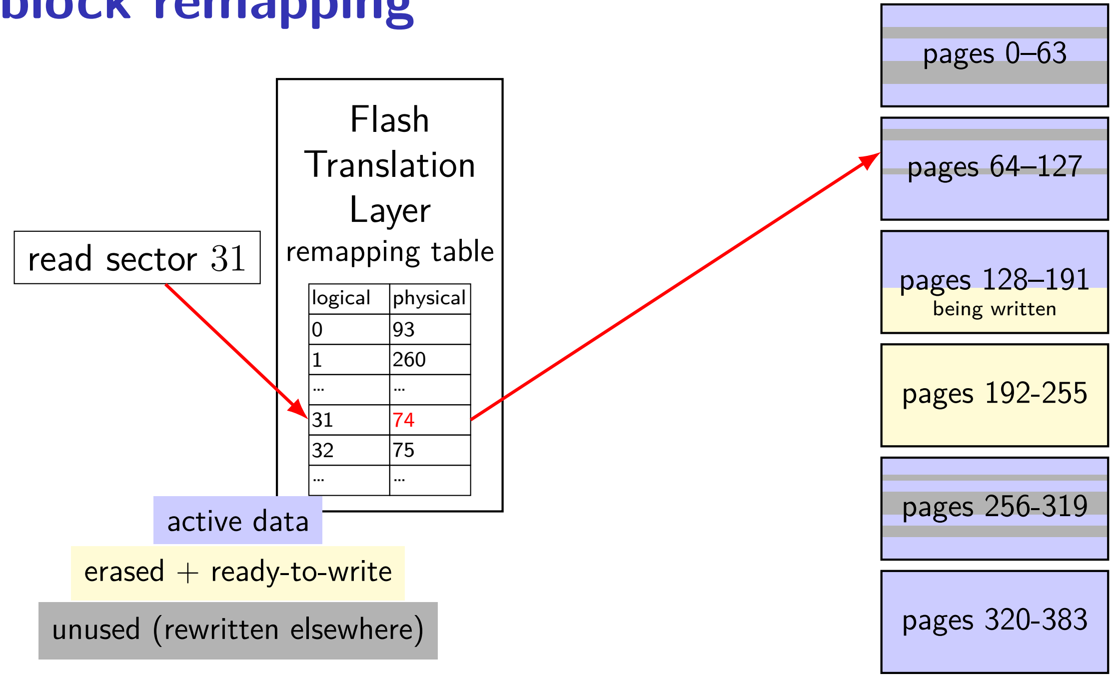
active data

erased + ready-to-write

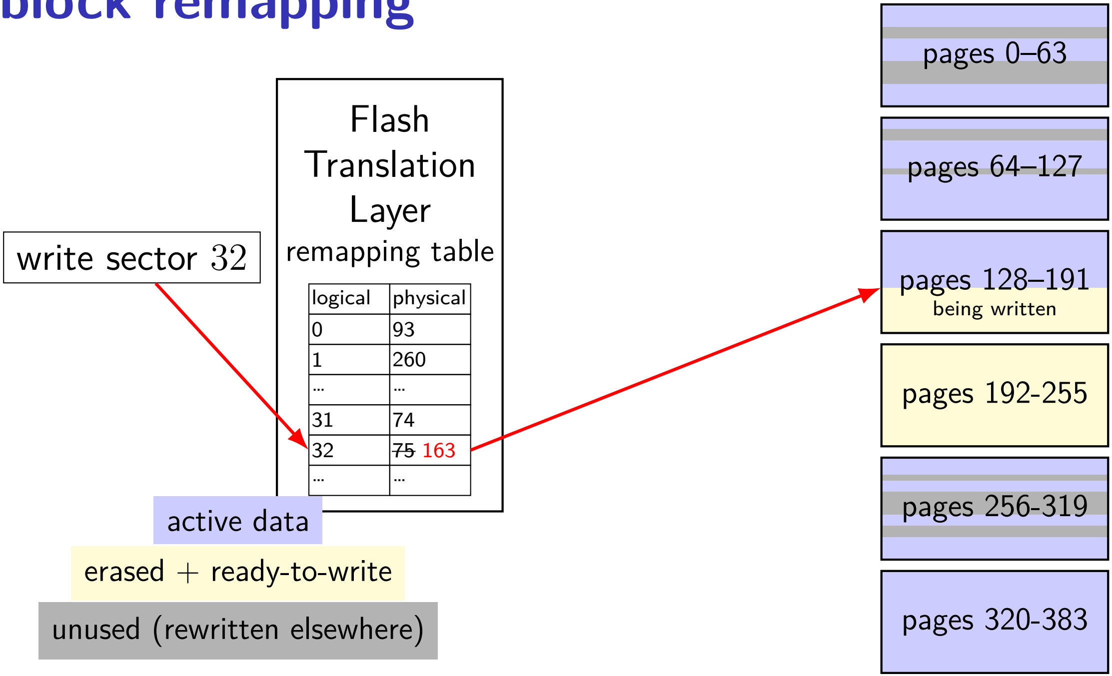
unused (rewritten elsewhere)



block remapping



block remapping



block remapping

Flash Translation Layer

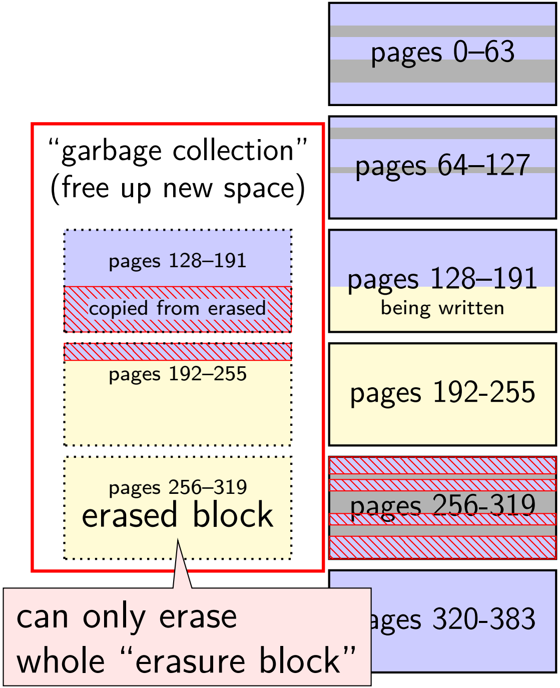
remapping table

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?)

aside: future storage

emerging non-volatile memories…

slower than DRAM (“normal memory”)

faster than SSDs

read/write interface like DRAM but persistent

FAT scattered data

file data and metadata scattered throughout disk

- directory entry
- *many* places in file allocation table

slow to find location of kth cluster of file

- first read FAT entries for clusters 0 to $k-1$

need to scan FAT to allocate new blocks

all not good for contiguous reads/writes

FAT in practice

typically keep entire file allocation table in memory

still pretty slow to find kth cluster of file

xv6 filesystem

xv6's filesystem similar to modern Unix filesytems

better at doing contiguous reads than FAT

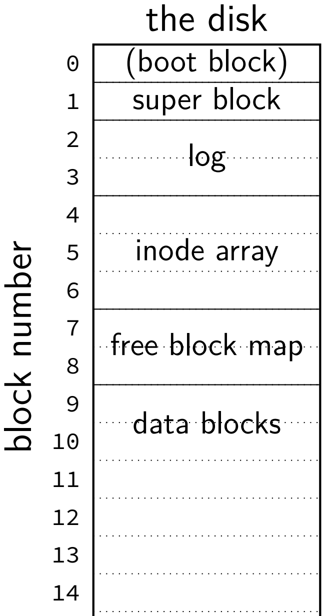
better at handling crashes

supports *hard links* (more on these later)

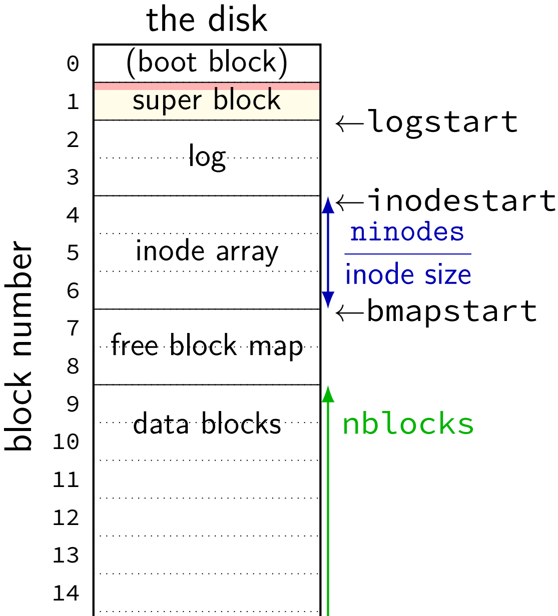
divides disk into *blocks* instead of clusters

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

xv6 disk layout



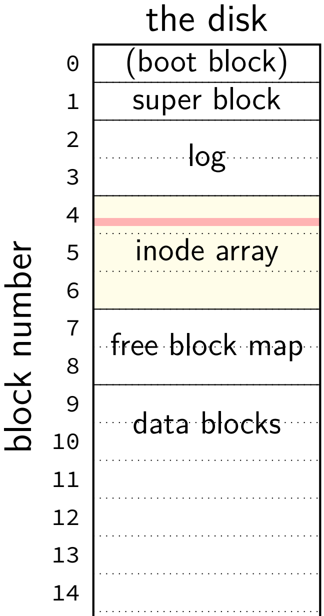
xv6 disk layout



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



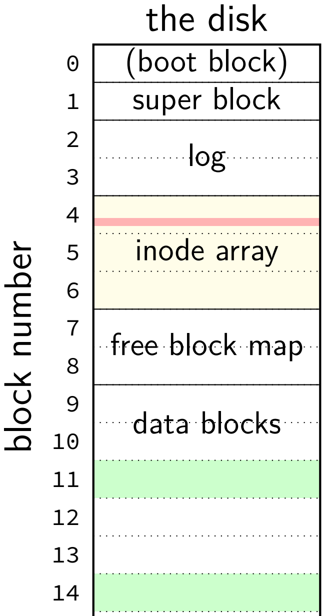
inode — file information

```
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

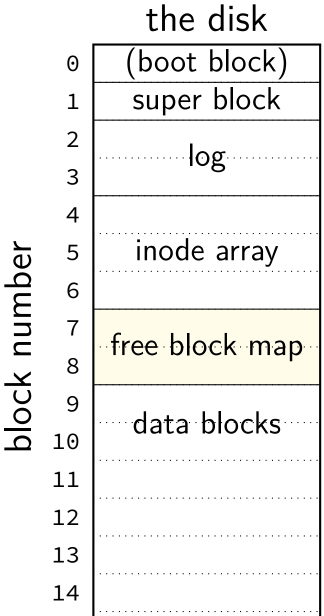
```
struct dinode {
   short type;   // File type
     // T_DIR, T_FILE, T_DEV

   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;

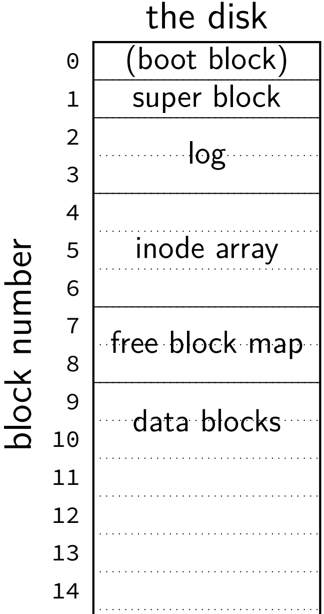
xv6 disk layout



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 FS pros versus FAT

support for reliability — log

- more on this later

possibly easier to scan for free blocks

- more compact free block map

easier to find location of kth block of file

- element of addrs array

file type/size information held with block locations

- inode number = everything about open file

missing pieces

what's the log? (more on that later)

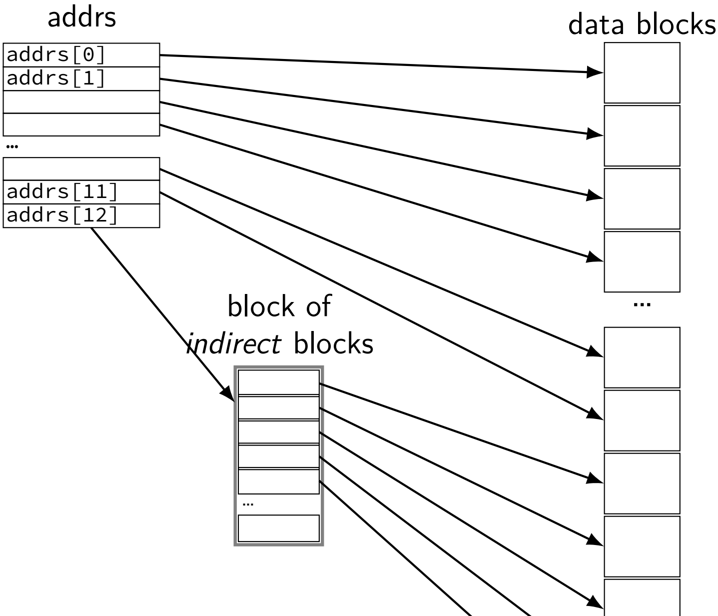
how big is addrs — list of blocks in inode

- what about large files?

other file metadata?

- creation times, etc. — xv6 doesn't have it

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 = 262144 bytes of data referenced

12 direct blocks @ 512 bytes each = 6144 bytes

1 indirect block @ 262144 bytes each = 262144 bytes

maximum file size

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;                  /* File mode */
    __le16 i_uid;                   /* Low 16 bits of Owner Uid */
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Creation time */
    __le32 i_mtime;         /* Modification time */
    __le32 i_dtime;         /* Deletion Time */
    __le16 i_gid;                   /* Low 16 bits of Group Id */
    __le16 i_links_count;           /* Links count */
    __le32 i_blocks;        /* Blocks count */
    __le32 i_flags;         /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;              /* File mode */
    __le16 i_uid;               /* Low 16 bits of Owner Uid */
    __le32 i_size;              /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __l
    __l
    __l
    __l
    __le16 i_gid;               /* Low 16 bits of Group Id */
    __le16 i_links_count;       /* Links count */
    __le32 i_blocks;    /* Blocks count */
    __le32 i_flags;     /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

type (regular, directory, device)
and permissions (read/write/execute for owner/group/others)

Linux ext2 inode

owner and group

```
struct ext2_inode {
    __le16 i_mode;                  /* File mode */
    __le16 i_uid;                   /* Low 16 bits of Owner Uid */
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Creation time */
    __le32 i_mtime;         /* Modification time */
    __le32 i_dtime;         /* Deletion Time */
    __le16 i_gid;                   /* Low 16 bits of Group Id */
    __le16 i_links_count;           /* Links count */
    __le32 i_blocks;        /* Blocks count */
    __le32 i_flags;         /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

Linux ext2 inode

whole bunch of times

```
struct ext2_inode {
    __le16 i_mode;                 /* File mode */
    __le16 i_uid;                  /* Low 16 bits of Owner Uid */
    __le32 i_size;                 /* Size in bytes */
    __le32 i_atime;        /* Access time */
    __le32 i_ctime;        /* Creation time */
    __le32 i_mtime;        /* Modification time */
    __le32 i_dtime;        /* Deletion Time */
    __le16 i_gid;                  /* Low 16 bits of Group Id */
    __le16 i_links_count;          /* Links count */
    __le32 i_blocks;       /* Blocks count */
    __le32 i_flags;        /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

Linux ext2 inode

similar pointers like xv6 FS — but more indirection

```
struct ext2_inod
    __le16 i_mod
    __le16 i_uid;                   /* Low 16 bits of Owner Uid */
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16 i_gid;                   /* Low 16 bits of Group Id */
    __le16 i_links_count;           /* Links count */
    __le32 i_blocks;    /* Blocks count */
    __le32 i_flags;     /* File flags */
    ...
    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};
```

ext2 indirect blocks

12 direct block pointers

1 indirect block pointer

- pointer to block containing more direct block pointers

1 double indirect block pointer

- pointer to block containing more indirect block pointers

1 triple indirect block pointer

- pointer to block containing more double indirect block pointers

ext2 indirect blocks

12 direct block pointers

1 indirect block pointer

- pointer to block containing more direct block pointers

1 double indirect block pointer

- pointer to block containing more indirect block pointers

1 triple indirect block pointer

- pointer to block containing more double indirect block pointers

exercise: if 1K blocks, how big can a file be?

indirect block advantages

small files: all direct blocks + no extra space beyond inode

larger files — more indirection

file should be large enough to hide extra indirection cost

sparse files

the xv6 filesystem and ext2 allow *sparse files*

“holes” with no data blocks

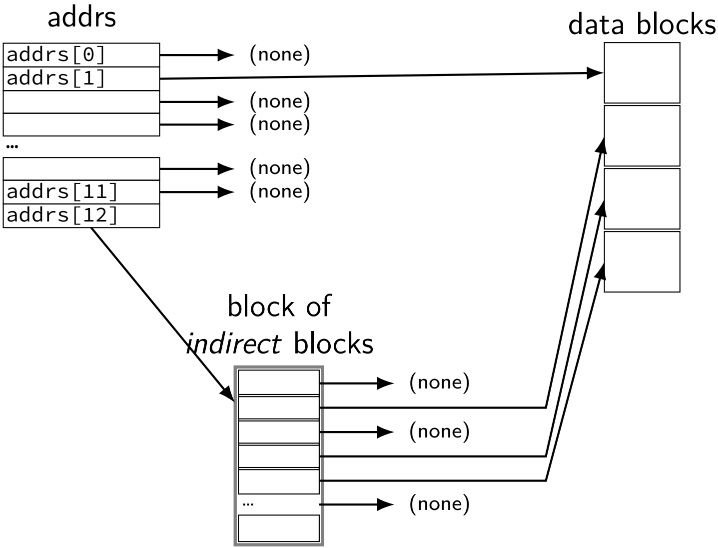
```
#include <stdio.h>
int main(void) {
    FILE *fh = fopen("sparse.dat", "w");
    fseek(fh, 1024 * 1024, SEEK_SET);
    fprintf(fh, "Some␣data␣here\n");
    fclose(fh);
}
```

sparse.dat is 1MB file which uses a handful of blocks

most of its block pointers are some NULL (‘no such block’) value

- including some direct and indirect ones

xv6 inode: sparse file



hard links

xv6/ext2 directory entries: name, inode number

all non-name information: in the inode itself

each directory entry is a *hard link*

a file can have multiple hard links

ln

```
$ echo "This is a test." >test.txt
$ ln test.txt new.txt
$ cat new.txt
This is a test.
$ echo "This is different." >new.txt
$ cat new.txt
This is different.
$ cat test.txt
This is different.
```

`ln OLD NEW` — NEW is the *same file* as OLD

link counts

xv6 and ext2 track number of links

- zero — actually delete file

link counts

xv6 and ext2 track number of links

- zero — actually delete file

also count open files as a link

trick: create file, open it, delete it

- file not really deleted until you close it
- ...but doesn't have a name (no hard link in directory)

link, unlink

`ln OLD NEW` calls the POSIX `link()` function

`rm FOO` calls the POSIX `unlink()` function

soft or symbolic links

POSIX also supports soft/symbolic links

reference a file by name

special type of file whose data is the name

```
$ echo "This is a test." >test.txt
$ ln -s test.txt new.txt
$ ls -l new.txt
lrwxrwxrwx 1 charles charles 8 Oct 29 20:49 new.txt -> test.txt
$ cat new.txt
This is a test.
$ rm test.txt
$ cat new.txt
cat: new.txt: No such file or directory
$ echo "New contents." >test.txt
$ cat new.txt
New contents.
```

xv6 filesystem performance issues

inode, block map stored far away from file data

- long seek times for reading files

unintelligent choice of file/directory data blocks

- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

blocks are pretty small — needs lots of space for metadata

- could change size? but waste space for small files
- large files have giant lists of blocks

linear searches of directory entries to resolve paths

Fast File System

the Berkeley Fast File System (FFS) 'solved' some of these problems

McKusick et al, "A Fast File System for UNIX" https://people.eecs.berkeley.edu/~brewer/cs262/FFS.pdf

Linux's ext2 filesystem based on FFS

xv6 filesystem performance issues

inode, block map stored far away from file data

- long seek times for reading files

unintelligent choice of file/directory data blocks

- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

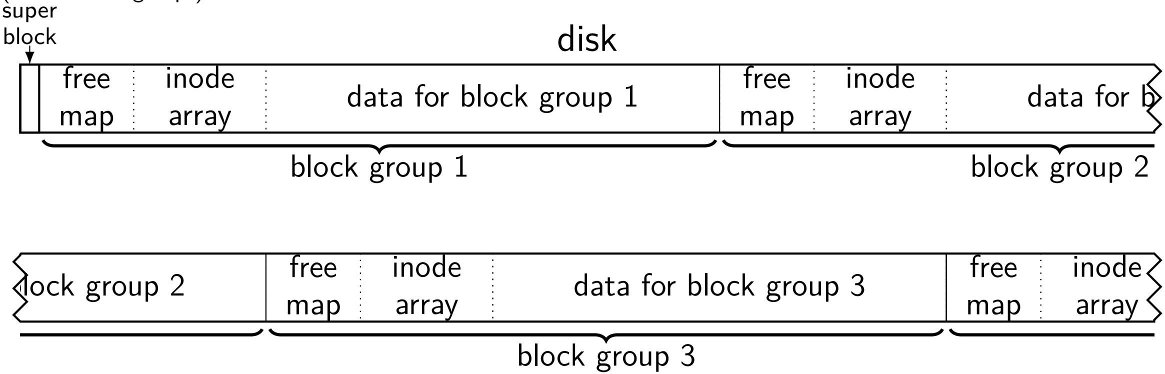
blocks are pretty small — needs lots of space for metadata

- could change size? but waste space for small files
- large files have giant lists of blocks

linear searches of directory entries to resolve paths

block groups

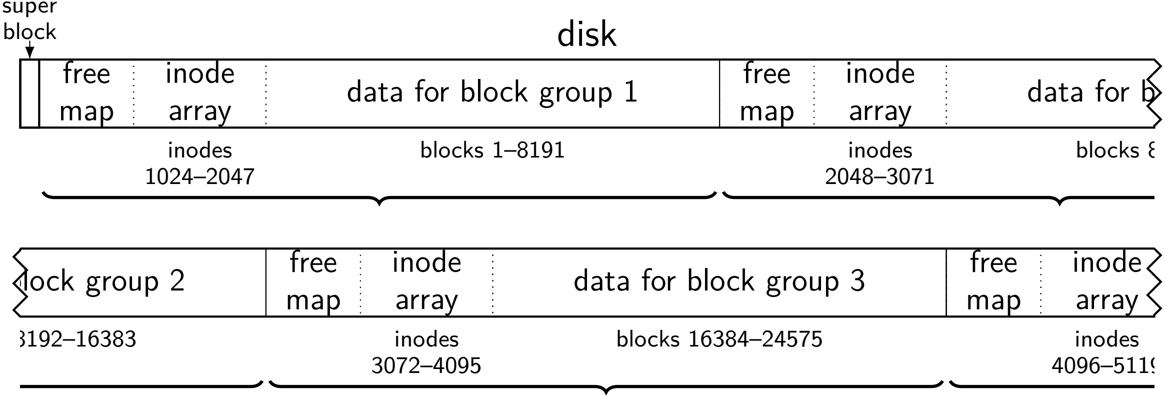
(AKA cluster groups)



split disk into block groups
each block group like a mini-filesystem

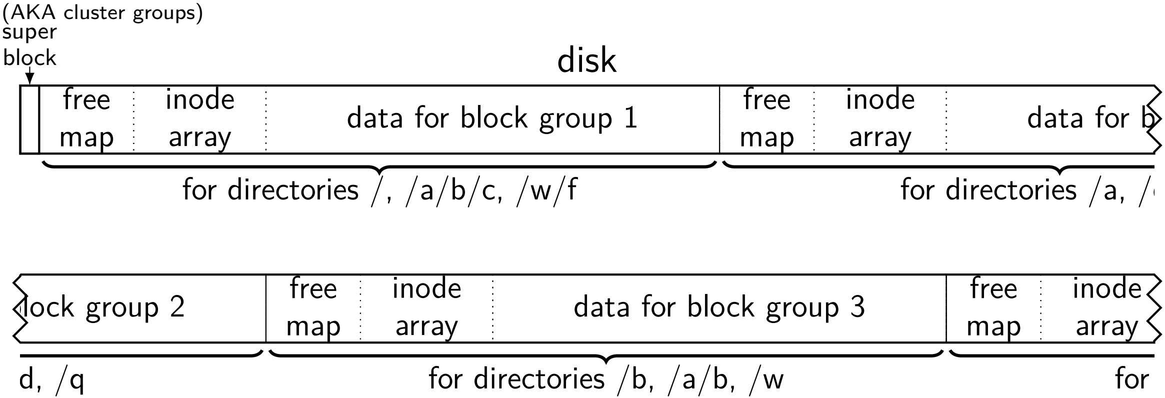
block groups

(AKA cluster groups)



split block + inode numbers across the groups
inode in one block group can reference blocks in another
(but would rather not)

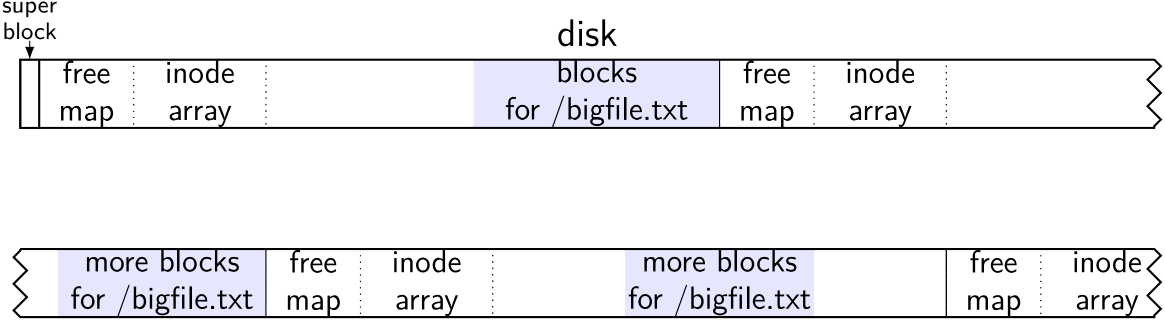
block groups



goal: *most data* for each directory within a block group
directory entries + inodes + file data close on disk
lower seek times!

block groups

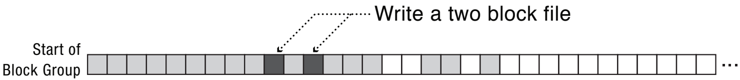
(AKA cluster groups)



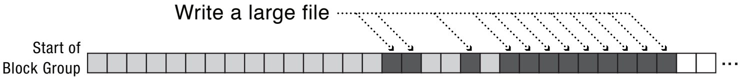
large files might need to be split across block groups

allocation within block groups

Expected typical arrangement.



Small files fill holes near start of block group.



Large files fill holes near start of block group and then write most data to sequential range blocks.

Anderson and Dahlin, *Operating Systems: Principles and Practice* 2nd edition, Figure 13.14

FFS block groups

making a subdirectory: new block group
- for inode + data (entries) in different

writing a file: same block group as directory, first free block
- intuition: non-small files get contiguous groups at end of block
- FFS keeps disk deliberately underutilized (e.g. 10% free) to ensure this

can wait until dirty file data flushed from cache to allocate blocks
- makes it easier to allocate contiguous ranges of blocks

xv6 filesystem performance issues

inode, block map stored far away from file data
- long seek times for reading files

unintelligent choice of file/directory data blocks
- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

blocks are pretty small — needs lots of space for metadata
- could change size? but waste space for small files
- large files have giant lists of blocks

linear searches of directory entries to resolve paths