hard drives / filesystems 2

last time

direct memory access

- write directy to device driver buffers
- OS supplies physical address
- maybe avoid more copies if really clever?

disk interface: sectors

FAT filesystem

- dividing disk into clusters
- files as linked list of cluster numbers
- file alloc table: linked list next pointers + free cluster info
- directory entries: file info + first clutser number

on extension requests

there was already a paging assignment extension...

- and I know several students started the assignment with enough time...
- don't want students to play "guess what the real due date is" when making plans

I wish we had more effective OH help, but our general assumption is that you should be able complete the assignment without it

- ...and that you won't start working in the last day or so to give time for getting answers to questions...

for particular difficulty to work assignment, case-by-case extensions (email or submit on kytos)

- computer/Internet availability issues, sudden moves, illness, ...

late policy still applies (3, 5 days)

on office hours

hopefully we're learning to be more efficient in virtual OH

e.g. switching between students to avoid spending too much time at once

please help us make them efficient:

good "task" descriptions may let us group students together for help

simplify your question: narrow down/simplify test cases

simplify your question: figure out what of your code is running/doing

(via debug prints, GDB, …)

use OH time other than in the last 24 hours before the due time

note on FAT assignment

read from disk image (file with contents of hard drive/SSD)

use real specs from Microsoft

implement FAT32 version; specs describe several variants

mapping from cluster numbers to location on disk different

end-of-file in FAT could be values other than -1

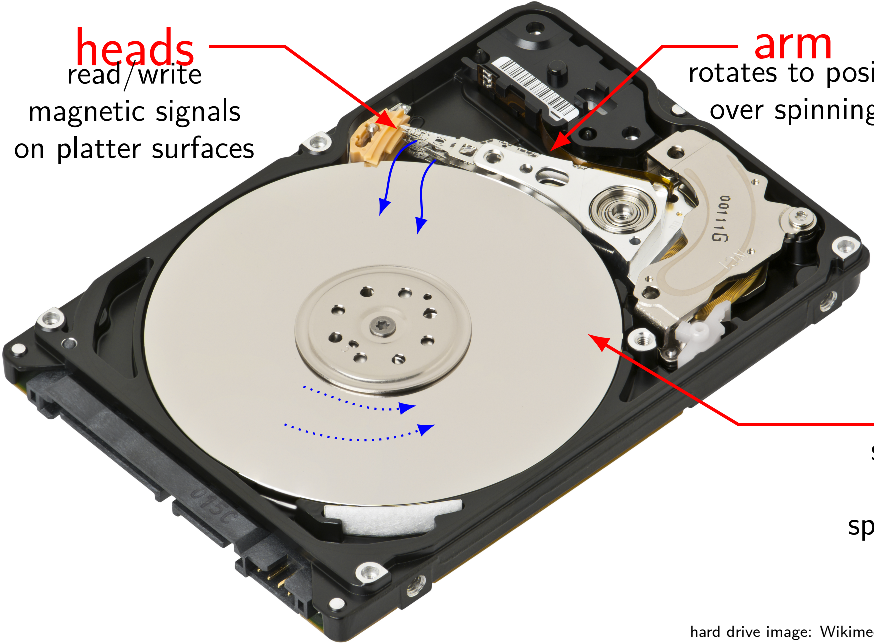
why hard drives?

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



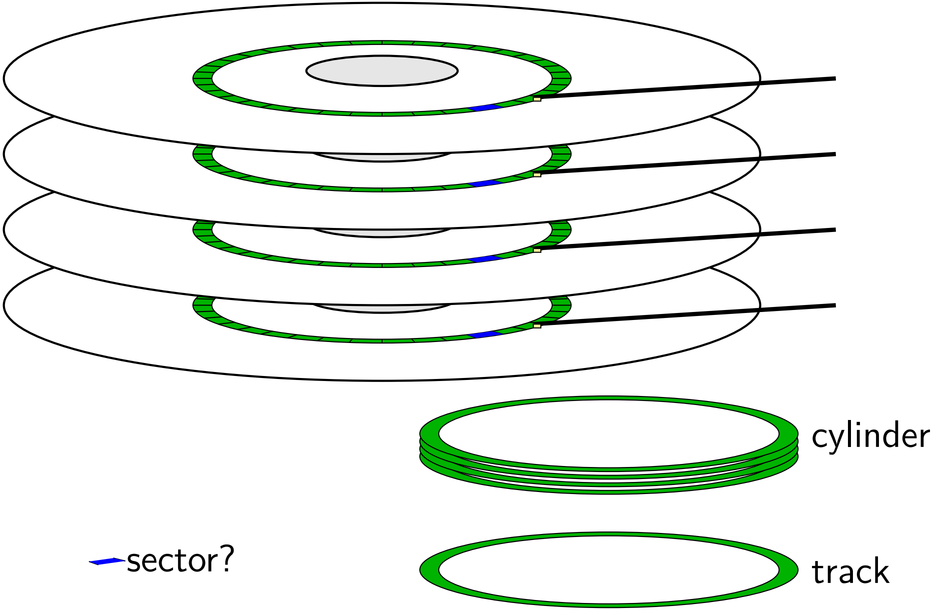
heads
read/write
magnetic signals
on platter surfaces

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

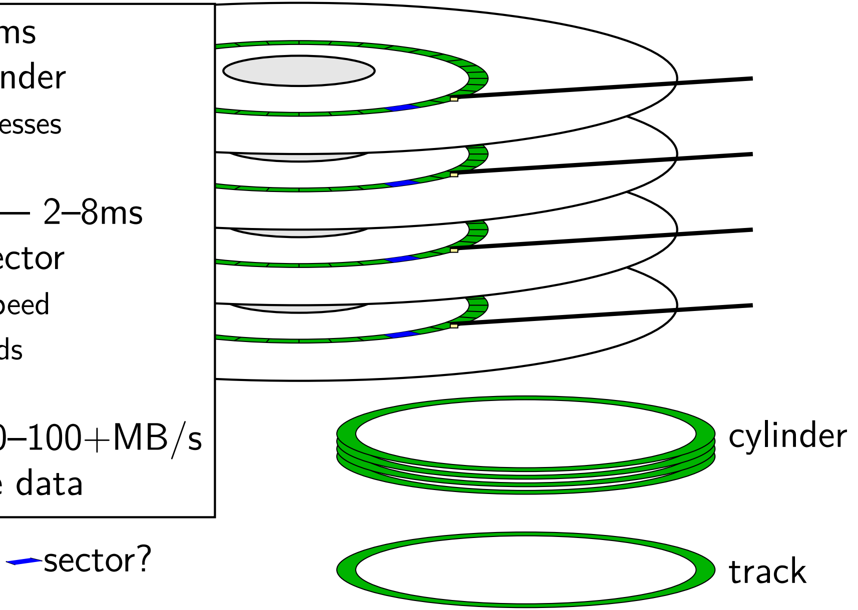
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data



sectors/cylinders/etc.

seek time — 5–10ms

move heads to cylinder

faster for adjacent accesses

rotational latency — 2–8ms

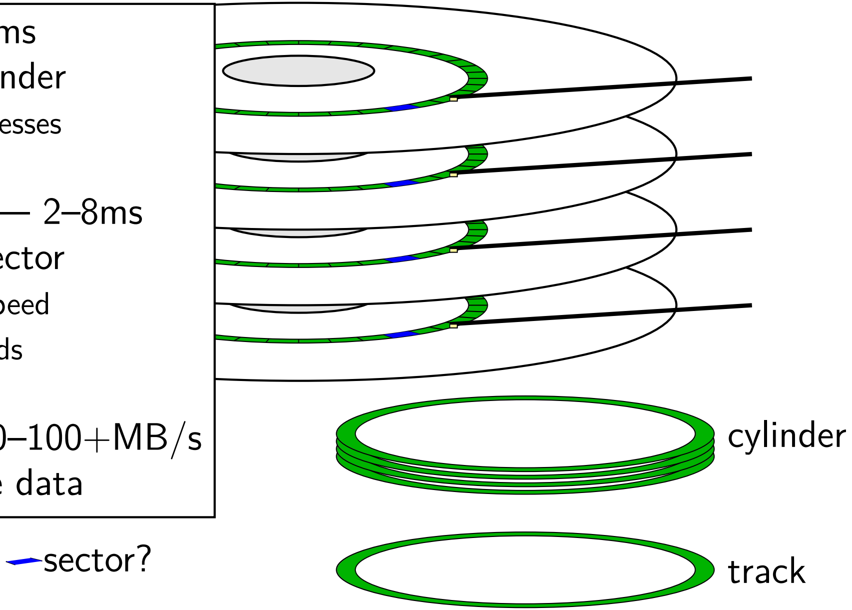
rotate platter to sector

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

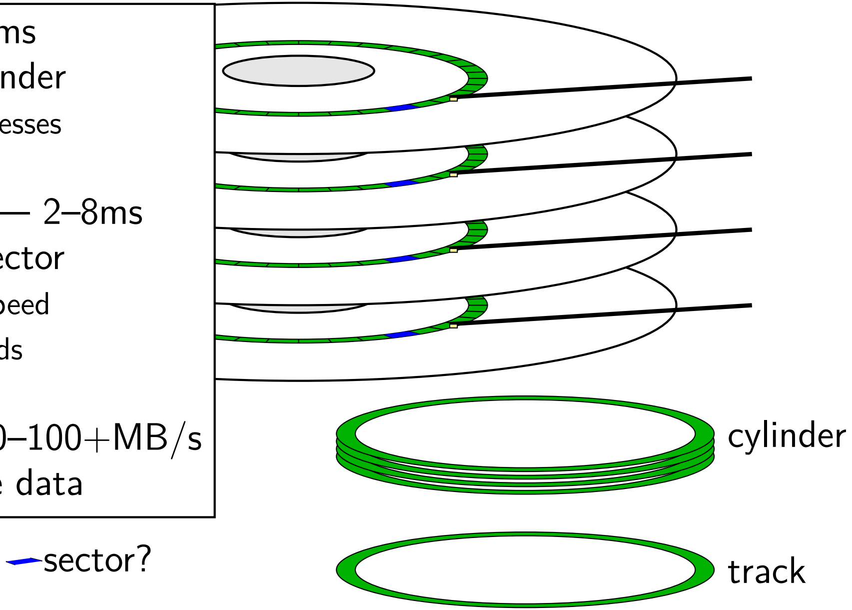
rotate platter to sector

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

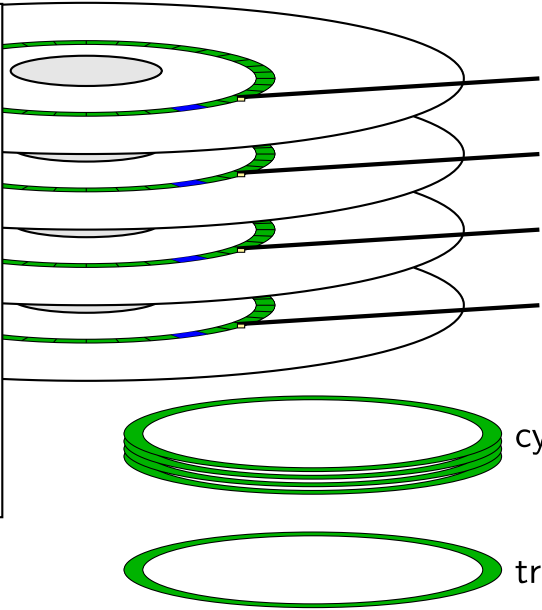
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data



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 controller and/or OS may need *schedule* requests

- group nearby requests together

as user of disk: better to request multiple things at a time

disk performance and filesystems

filesystem can…

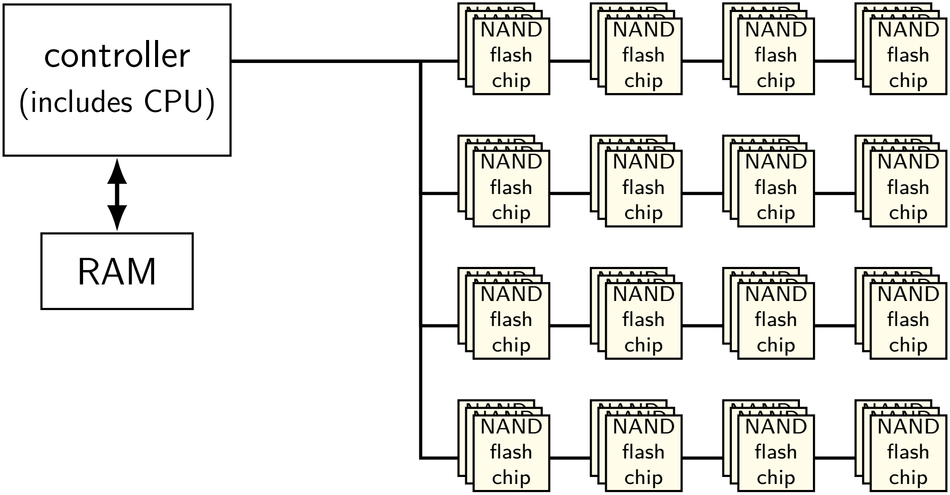
do contiguous or nearby reads/writes

- 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

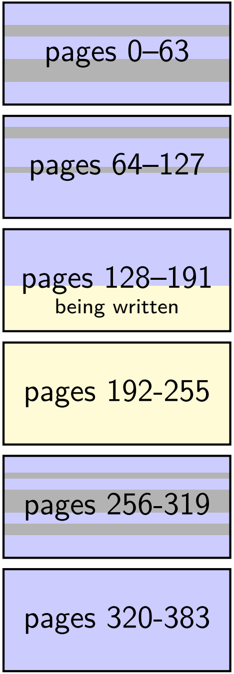
remapping table

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

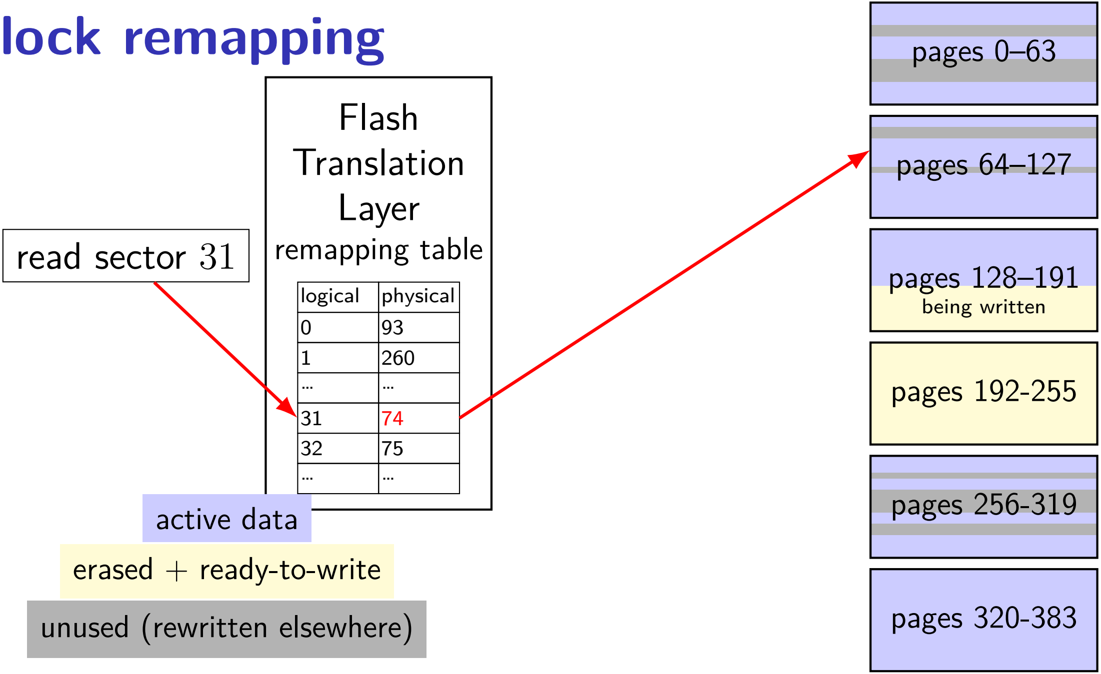
active data

erased + ready-to-write

unused (rewritten elsewhere)



block remapping



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

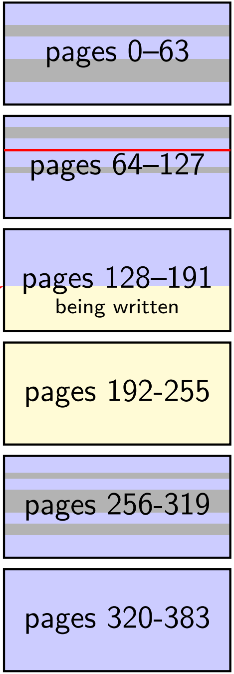
block remapping

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

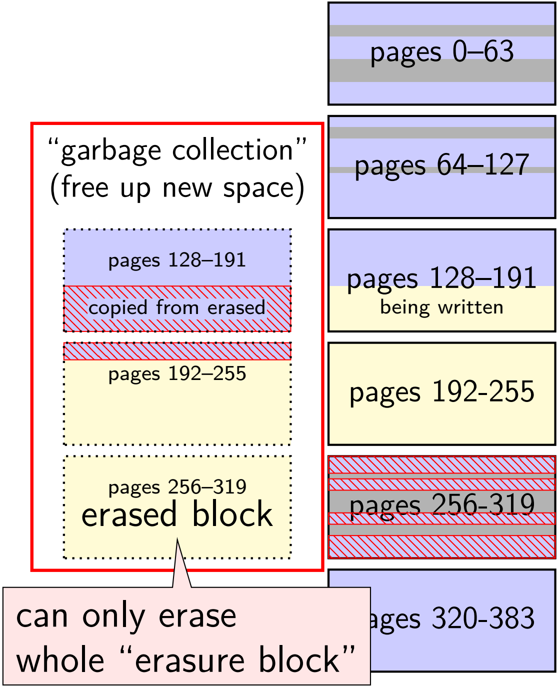
remapping table

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 → 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

exercise

Assuming a FAT-like filesystem on an SSD, which of the following are likely to be stored in the same (or very small number of) erasure block?

[a] the clusters of a set of log file all in one directory written continuously over months by a server and assigned a contiguous range of cluster numbers

[b] the data clusters of a set of images, copied all at once from a camera and assigned a variety of cluster numbers

[c] all the entires of the FAT (assume the OS only rewrites a sector of the FAT if it is changed)

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

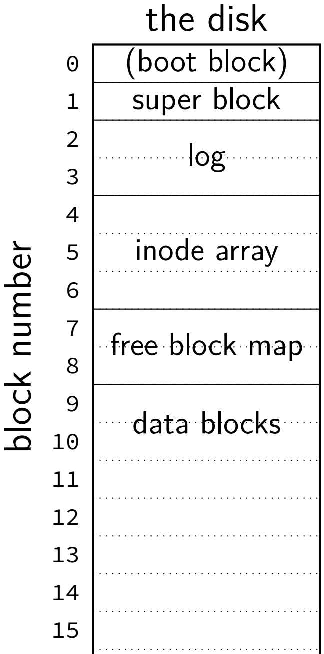
better at handling crashes

supports *hard links*

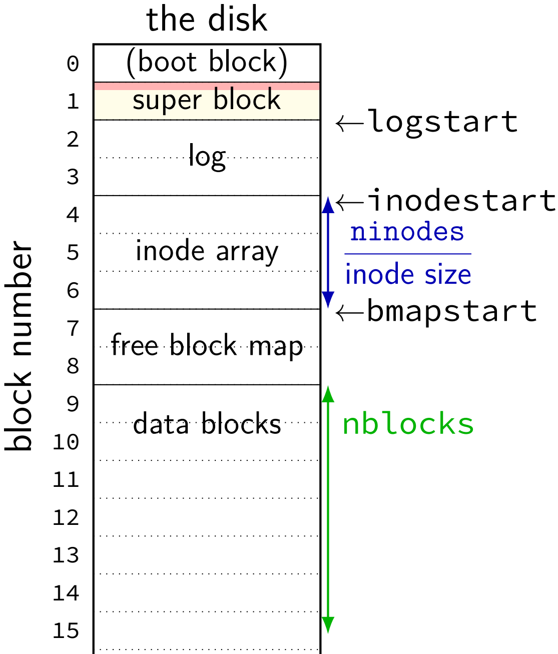
divides disk into *blocks* instead of clusters

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

xv6 disk layout



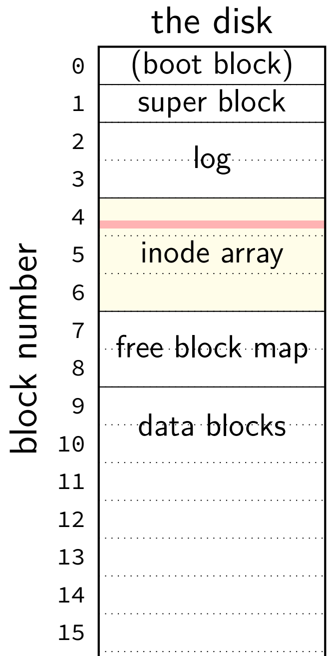
xv6 disk layout



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



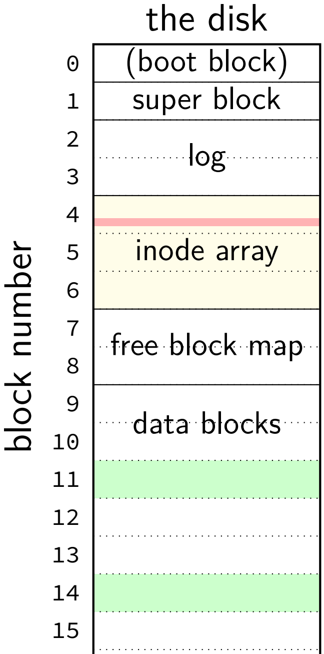
inode — file information

```
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

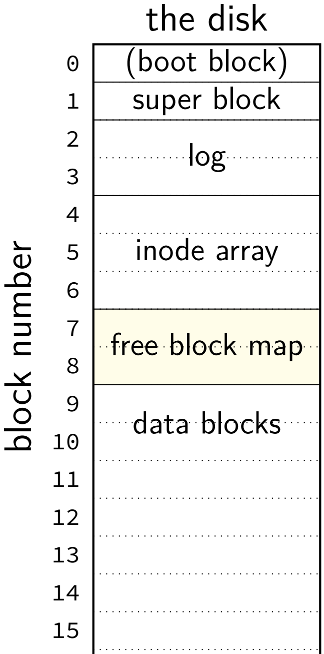
```
struct dinode {
   short type;  // File type
     // T_DIR, T_FILE, T_DEV

   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

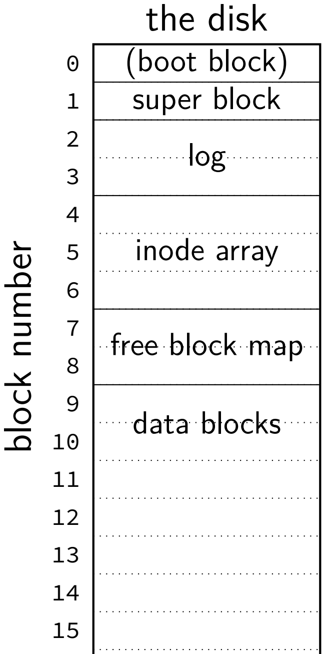
xv6 disk layout



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*

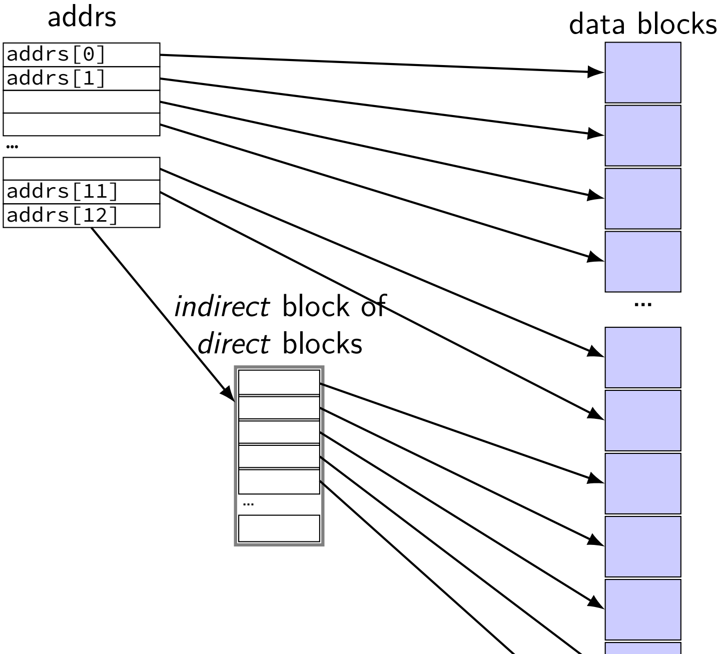
multiple hard links to file allowed!

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 = 6144 + 131072 bytes

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 */
    __le32 i_ctime;   /* Creation time */
    __l
    __l
    __l
    __l
    __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

```
struct ext2_inode {
    __le16 i_mode;                  /* File mode */
    __le16 i_uid;                   /* Low 16 bits
    __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 */
    ...
};
```

owner and group

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;              /* File mod
    __le16 i_uid;               /* Low 16 b
    __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 */
    ...
};
```

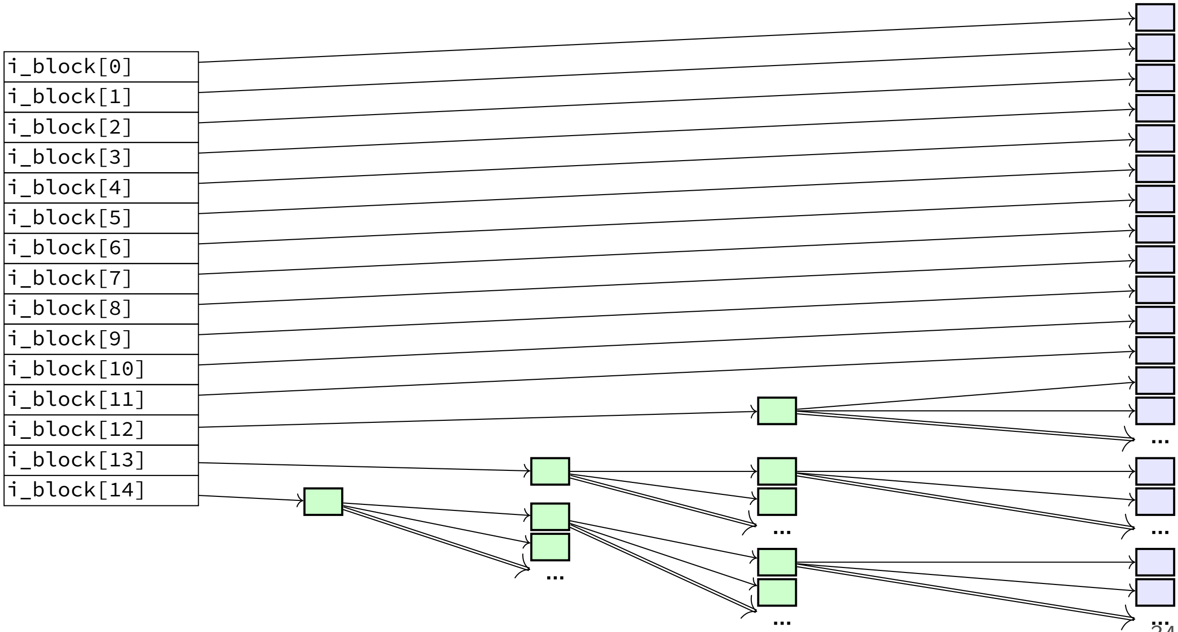
whole bunch of times

Linux ext2 inode

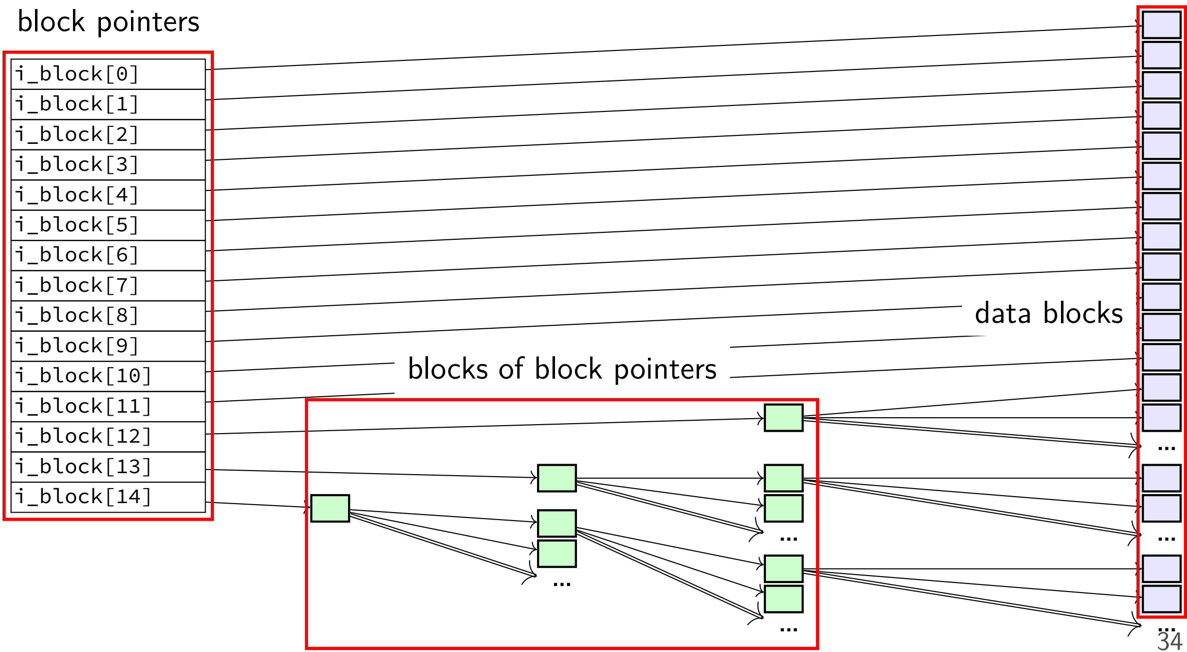
similar pointers like xv6 FS — but more indirection

```
struct ext2_inode {
    __le16 i_mod
    __le16 i_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 */
    ...
};
```

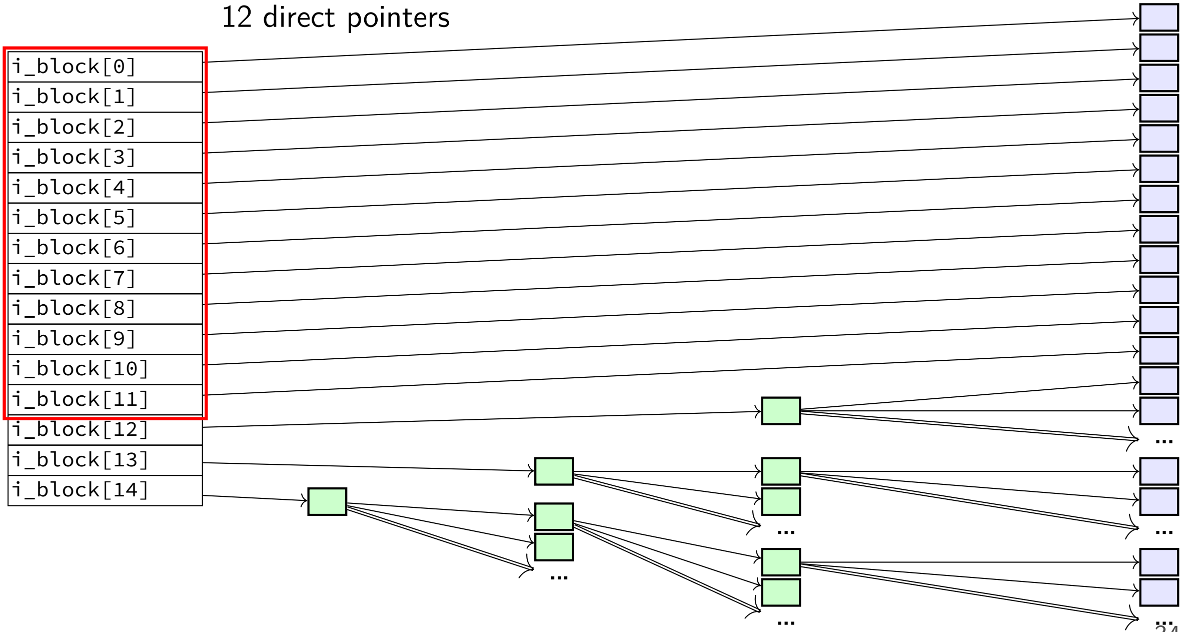
double/triple indirect



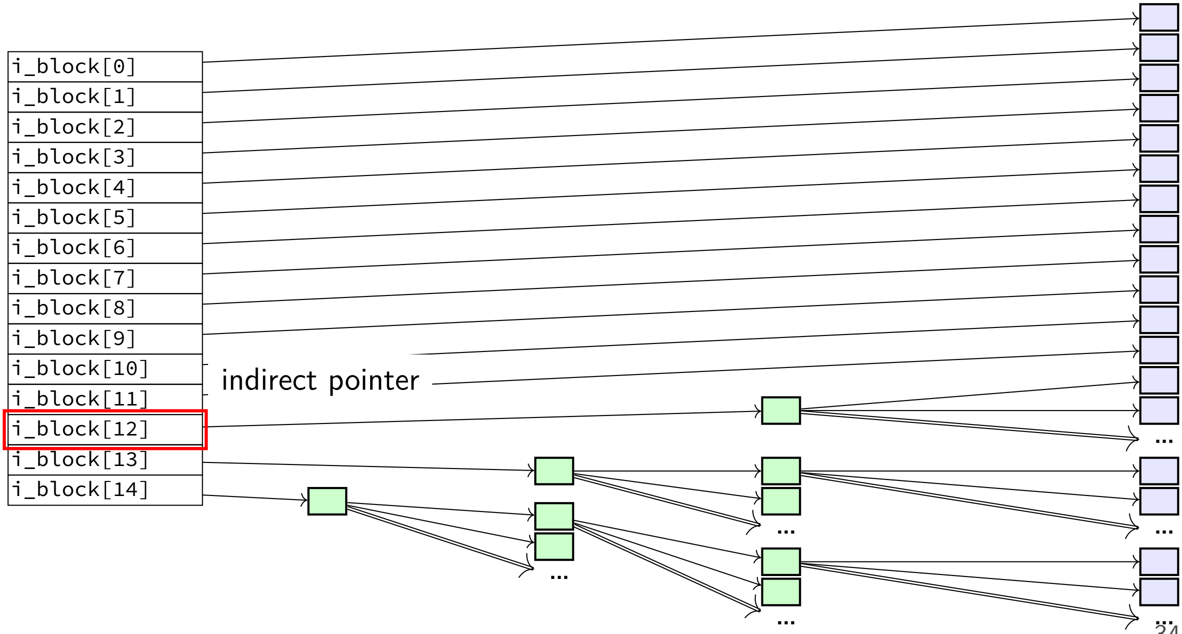
double/triple indirect



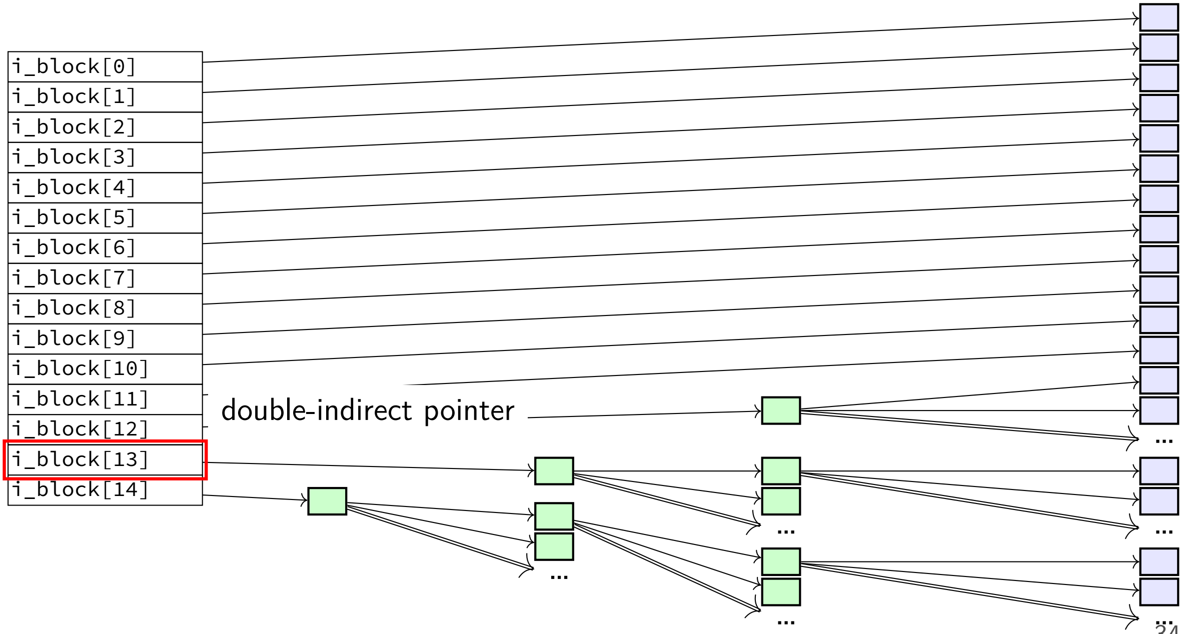
double/triple indirect



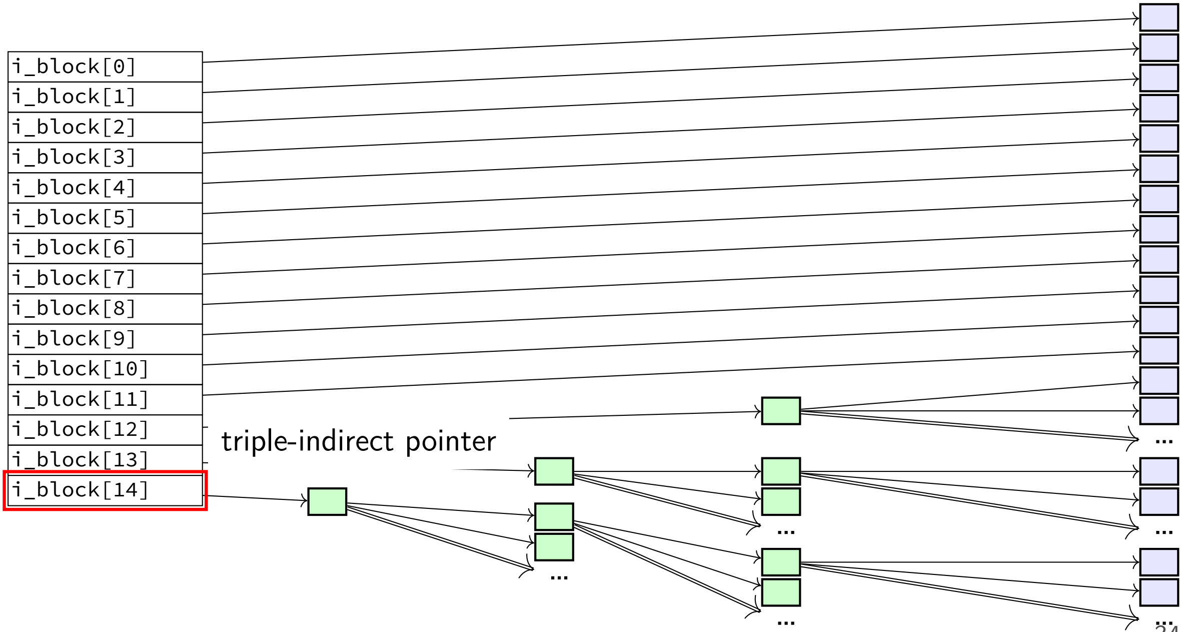
double/triple indirect



double/triple indirect



double/triple indirect



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, 4 byte block pointers, how big can a file be?

ext2 indirect blocks (2)

12 direct block pointers

1 indirect block pointer

1 double indirect block pointer

1 triple indirect block pointer

exercise: if 1K (2^10 byte) blocks, 4 byte block pointers,
how does OS find byte 2^15 of the file?

(1) using indirect pointer or double-indirect pointer in inode?
(2) what index of block pointer array pointed to by pointer in inode?

filesystem reliability

a crash happens — what's the state of my filesystem?

hard disk atomicity

interrupt a hard drive write?

write whole disk sector or corrupt it

hard drive stores checksum for each sector

write interrupted? — checksum mismatch

- hard drive returns read error

reliability issues

is the data there?

- can we find the file, etc.?

is the filesystem in a consistent state?

- do we know what blocks are free?

backup slides

erasure coding with xor

storing 2 bits xy using 3

choose x, y, $z = x \oplus y$

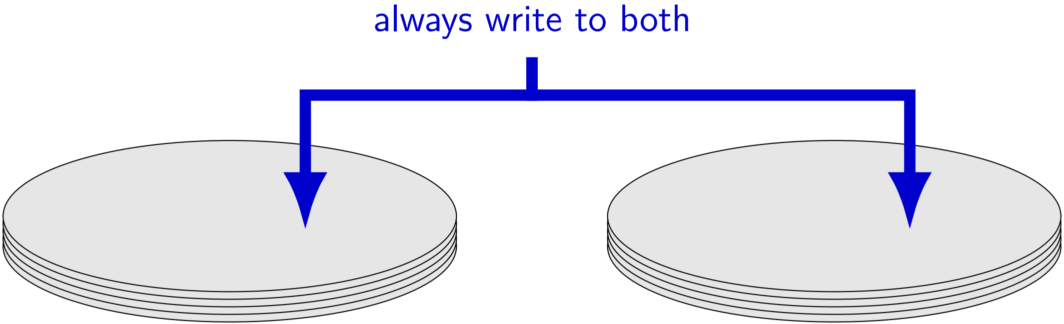
recover x: $x = y \oplus z$

recover y: $y = x \oplus z$

recover z: $y = x \oplus y$

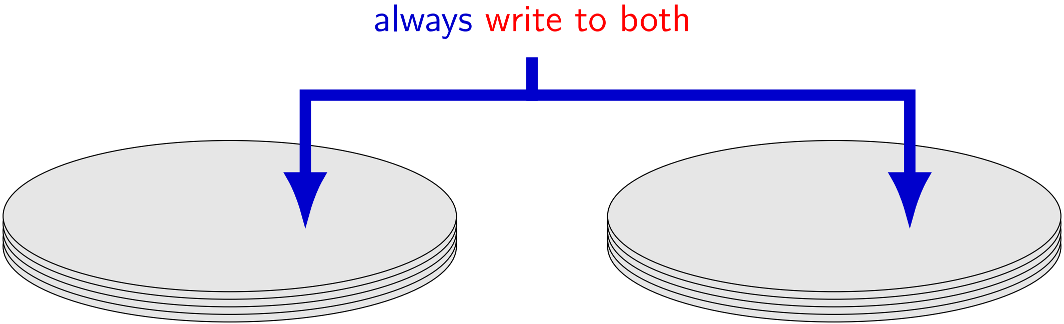
mirroring whole disks

alternate strategy: write everything to two disks



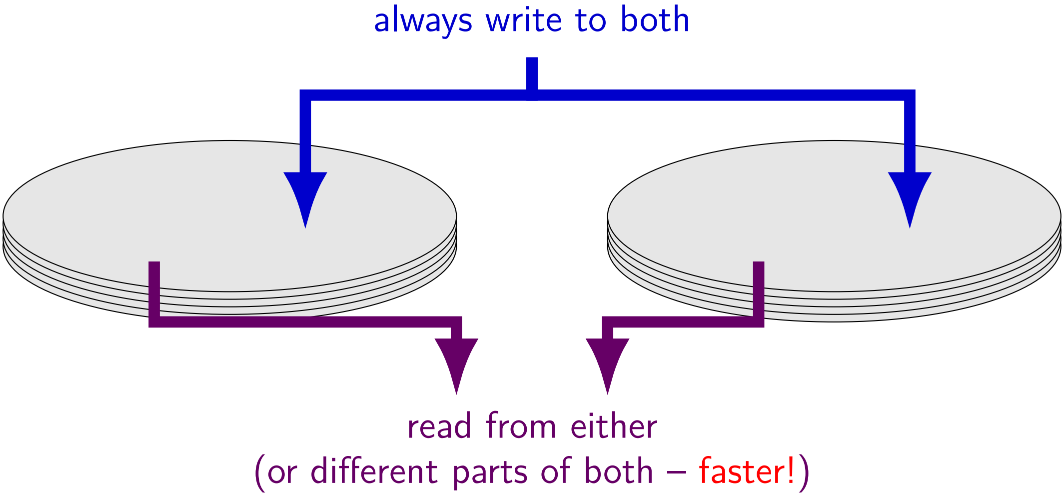
mirroring whole disks

alternate strategy: write everything to two disks



mirroring whole disks

alternate strategy: write everything to two disks



RAID 4 parity

$\oplus$ — bitwise xor

disk 1	**disk 2**	**disk 3**
A_1: sector 0	A_2: sector 1	A_p: $A_1 \oplus A_2$
B_1: sector 2	B_2: sector 3	B_p: $B_1 \oplus B_2$
...	...	...

RAID 4 parity

$\oplus$ — bitwise xor

disk 1	**disk 2**	**disk 3**
A_1: sector 0	A_2: sector 1	A_p: $A_1 \oplus A_2$
B_1: sector 2	B_2: sector 3	B_p: $B_1 \oplus B_2$
...	...	...

$$A_p = A_1 \oplus A_2$$
$$A_1 = A_p \oplus A_2$$
$$A_2 = A_1 \oplus A_p$$

can compute contents of any disk!

RAID 4 parity

$\oplus$ — bitwise xor

disk 1	**disk 2**	**disk 3**
A_1: sector 0	A_2: sector 1	A_p: $A_1 \oplus A_2$
B_1: sector 2	B_2: sector 3	B_p: $B_1 \oplus B_2$
...	...	...

exercise: how to replace sector 3 (B_2)with new value?
how many writes? how many reads?

RAID 4 parity (more disks)

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3 sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_3: sector 5	B_p: $B_1 \oplus B_2 \oplus B_3$
...	...	...	

RAID 4 parity (more disks)

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3 sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_3: sector 5	B_p: $B_1 \oplus B_2 \oplus B_3$
...	...	...	

$$A_p = A_1 \oplus A_2 \oplus A_3$$
$$A_1 = A_p \oplus A_2 \oplus A_3$$
$$A_2 = A_1 \oplus A_p \oplus A_3$$
$$A_3 = A_1 \oplus A_2 \oplus A_p$$

can still compute contents of any disk!

RAID 4 parity (more disks)

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3 sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_3: sector 5	B_p: $B_1 \oplus B_2 \oplus B_3$
...	...	...	

exercise: how to replace sector 3 (B_1) with new value now?
how many writes? how many reads?

RAID 5 parity

disk 1	disk 2	disk 3	disk 4
A_1: sector 0	A_2: sector 1	A_3: sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_p: $B_1 \oplus B_2 \oplus B_3$	B_3:sector 5
C_1: sector 6	C_p: $C_1 \oplus C_2 \oplus C_3$	C_2: sector 7	C_3: sector 8
...	...	...	

RAID 5 parity

disk 1	**disk 2**	**disk 3**	**disk 4**
A_1: sector 0	A_2: sector 1	A_3: sector 2	A_p: $A_1 \oplus A_2 \oplus A_3$
B_1: sector 3	B_2: sector 4	B_p: $B_1 \oplus B_2 \oplus B_3$	B_3:sector 5
C_1: sector 6	C_p: $C_1 \oplus C_2 \oplus C_3$	C_2: sector 7	C_3: sector 8
...	...	...	

spread out parity updates across disks
so each disk has about same amount of work

more general schemes

RAID 6: tolerate loss of any two disks

can generalize to 3 or more failures

- justification: takes days/weeks to replace data on missing disk
- ...giving time for more disks to fail

probably more in CS 4434?

but none of this addresses consistency

RAID-like redundancy

usually appears to filesystem as 'more reliable disk'

- hardware or software layers to implement extra copies/parity

some filesystems (e.g. ZFS) implement this themselves

- more flexibility — e.g. change redundancy file-by-file
- ZFS combines with its own checksums — don't trust disks!

RAID: missing piece

what about losing data while blocks being updated

very tricky/failure-prone part of RAID implementations

efficient seeking with extents

suppose a file has long list of extents

how to seek to byte X?

efficient seeking with extents

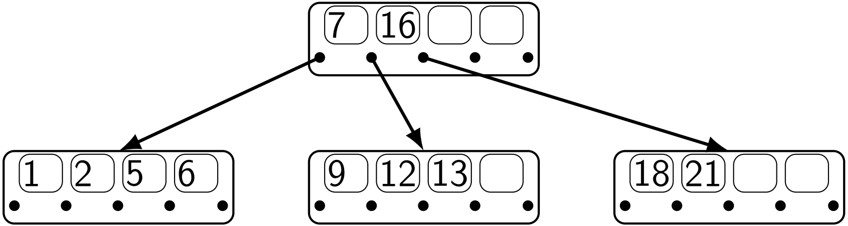
suppose a file has long list of extents

how to seek to byte X?

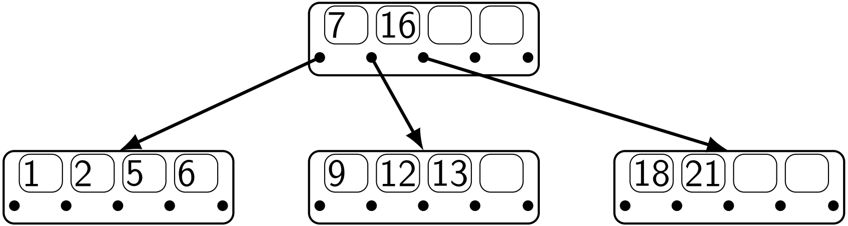
solution: store a (search) tree

- ext4: each node stores key=minimum file index it covers
- ext4: each node stores extent value=(start data block+size)
- ext4: each node has pointer (disk block) to its children

non-binary search trees



non-binary search trees



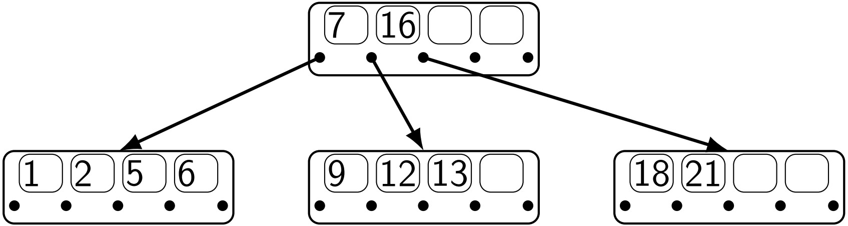
each node can be one block on disk

choose number of entries in node based on block size

avoid large or random accesses to disk and linear searches

can do binary search within a node

non-binary search trees



each node can be one block on disk

choose number of entries in node based on block size

avoid large or random accesses to disk and linear searches

can do binary search within a node

algorithms for adding to tree while keeping it balanced

similar idea to AVL trees

using trees on disk

linear search to find extent at offset X

- store index by offset of extent within file

linear search to find file in directory?

- index by filename

both problems — solved with non-binary tree on disk

sparse files

the xv6 filesystem and ext2 allow *sparse files*

“holes” with no data blocks

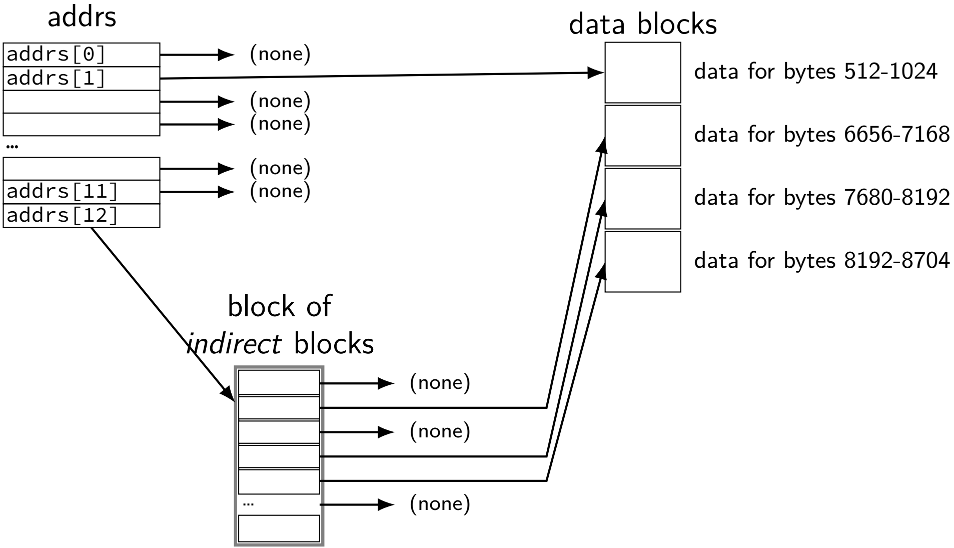
```
#include <stdio.h>
int main(void) {
    FILE *fh = fopen("sparse.dat", "w");
    fseek(fh, 1024 * 1024, SEEK_SET);
    fprintf(fh, "Some data here\n");
    fclose(fh);
}
```

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 called a *hard link*

a file can have multiple hard links

ln

```
$ echo "Text A." >test.txt
$ ln test.txt new.txt
$ cat new.txt
Text A.
$ echo "Text B." >new.txt
$ cat new.txt
Text B.
$ cat test.txt
Text B.
```

`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.
```

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