filesystems 3

last time

FAT headers, free space, allocating space

hard disk performance
> seek times from physical movement of disk head
> queues of requests, scheduled to control seek time
> smarts in controller: bad blocks, scheduling

solid state disks
> block remapping to hide erasure blocks in flash

xv6 filesystem
> *inodes* contain info about file blocks, type, size, etc. (instead of directory entries)

xv6 filesystem

xv6's filesystem similar to modern Unix filesytems

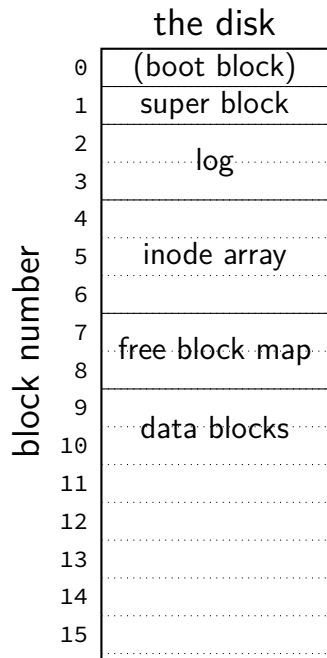better at doing contiguous reads than FAT

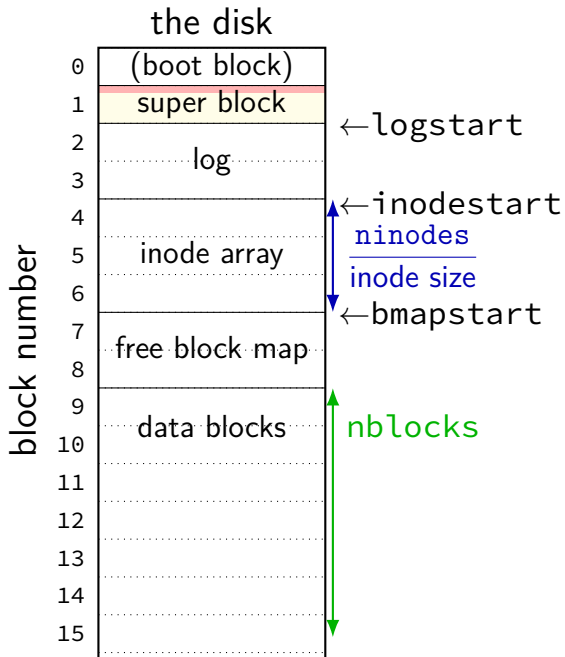better at handling crashes

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
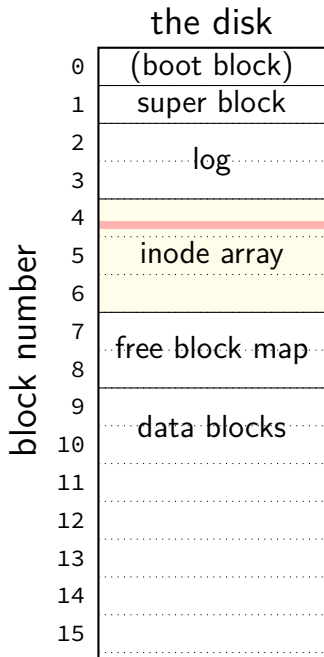
the disk

block number	
0	(boot block)
1	super block
2	log
3	
4	
5	inode array
6	
7	free block map
8	
9	data blocks
10	
11	
12	
13	
14	
15	

xv6 disk layout

the disk

block number

0	(boot block)
1	super block
2	log
3	log
4	
5	inode array
6	
7	free block map
8	
9	data blocks
10	
11	
12	
13	
14	
15	

←logstart

←inodestart
ninodes
inode size
←bmapstart

nblocks

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
};
```
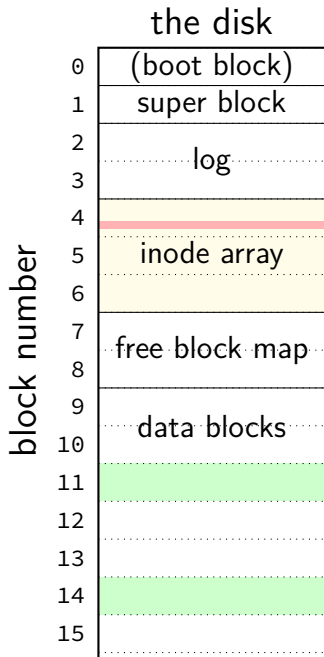
5

xv6 disk layout

the disk

block number	
0	(boot block)
1	super block
2	log
3	
4	
5	inode array
6	
7	free block map
8	
9	data blocks
10	
11	
12	
13	
14	
15	

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

the disk

block number

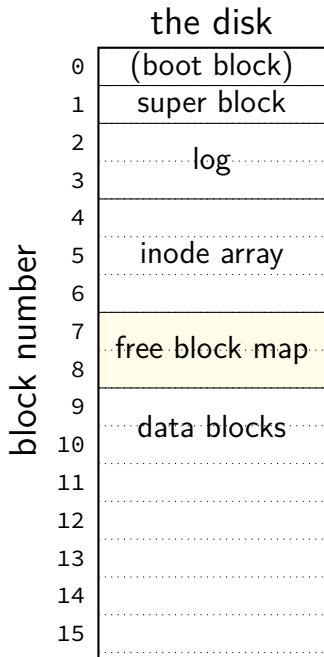|	
0	(boot block)
1	super block
2	log
3	
4	
5	inode array
6	
7	free block map
8	
9	data blocks
10	
11	
12	
13	
14	
15	

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:
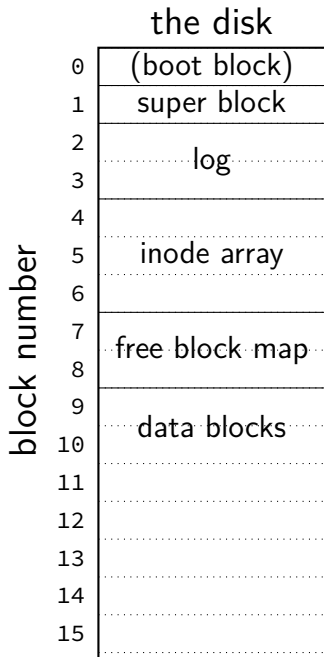e.g. addrs[0] = 11; addrs[1] = 14;
special case for larger files

5

xv6 disk layout

the disk

block number	
0	(boot block)
1	super block
2	log
3	log
4	
5	inode array
6	
7	free block map
8	
9	data blocks
10	
11	
12	
13	
14	
15	

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

the disk

block number	
0	(boot block)
1	super block
2	log
3	
4	
5	inode array
6	
7	free block map
8	
9	data blocks
10	
11	
12	
13	
14	
15	

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

addrs

addrs[0]
addrs[1]

…

addrs[11]
addrs[12]

data blocks

indirect block of
direct blocks

…

…
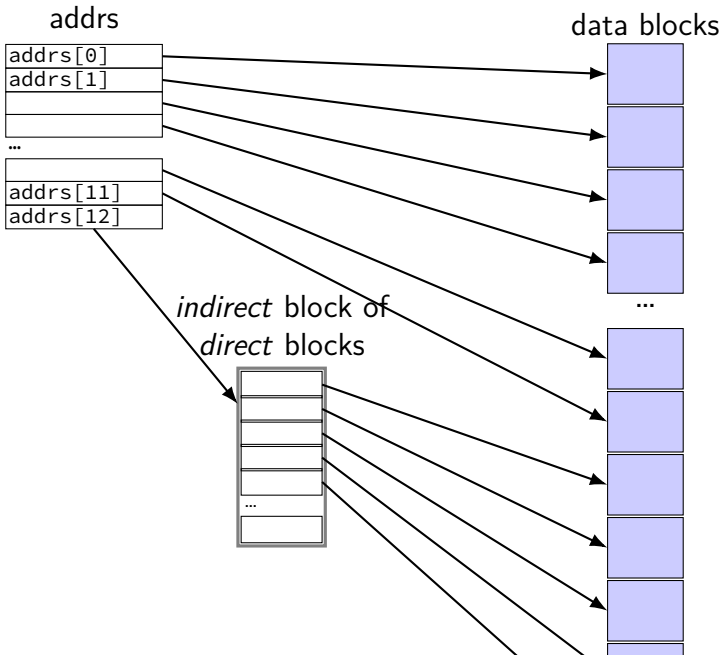
xv6 file sizes

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

```
struct ext2_inode {
    __le16 i_mod
    __le16 i_uid   similar pointers like xv6 FS — but more indirection
    __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 */
    ...
};
```
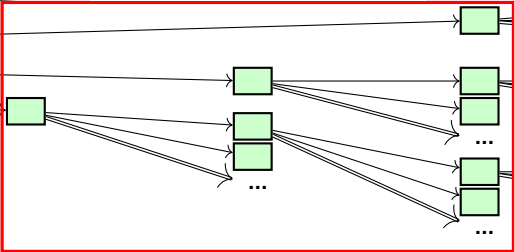
double/triple indirect
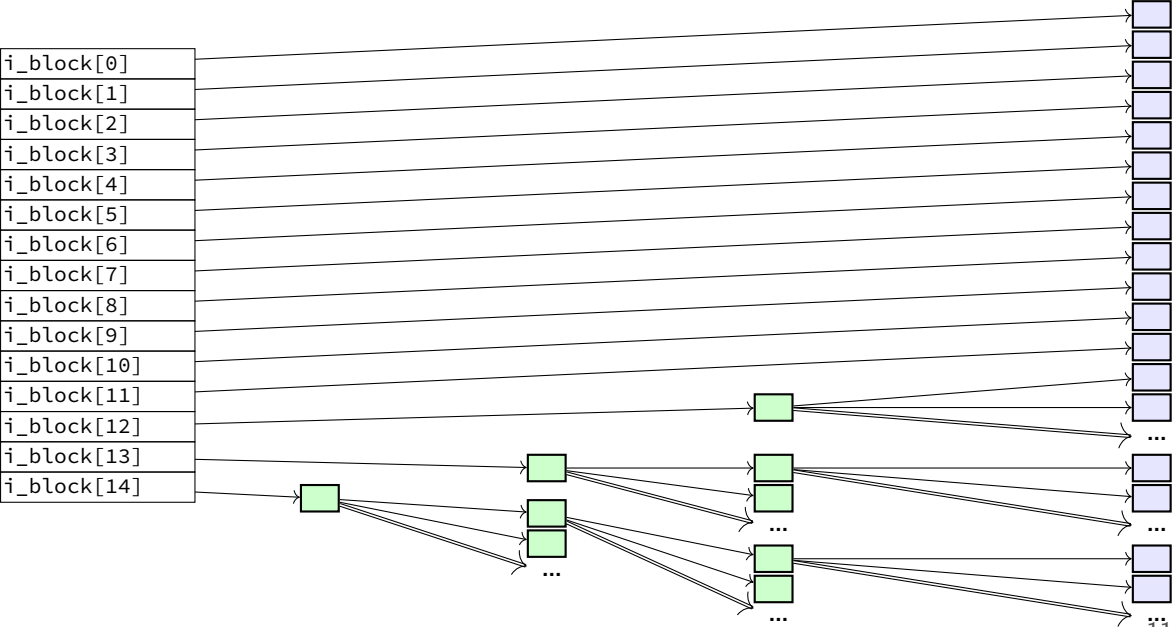
double/triple indirect

block pointers

i_block[0]
i_block[1]
i_block[2]
i_block[3]
i_block[4]
i_block[5]
i_block[6]
i_block[7]
i_block[8]
i_block[9]
i_block[10]
i_block[11]
i_block[12]
i_block[13]
i_block[14]

data blocks

blocks of block pointers

...

...

...

...

...

11
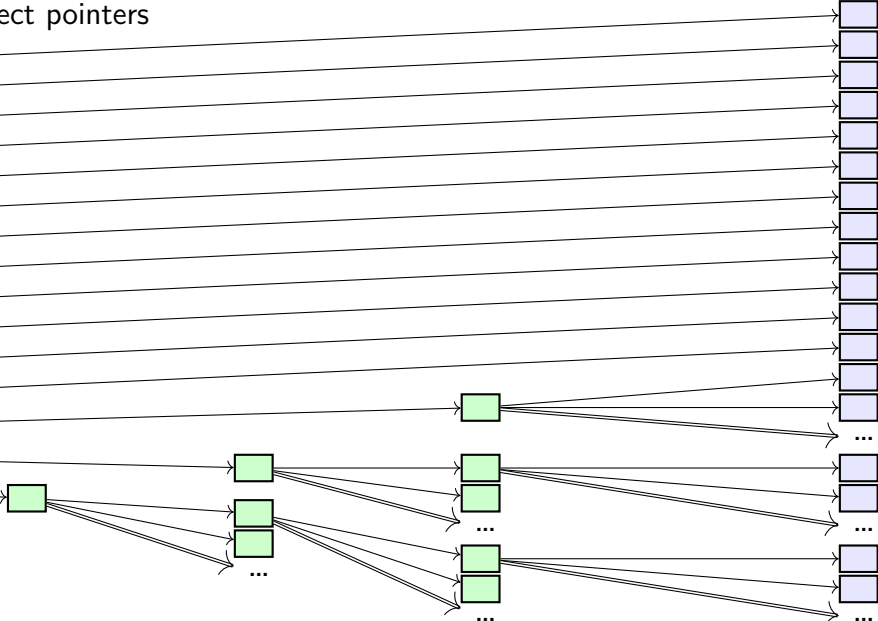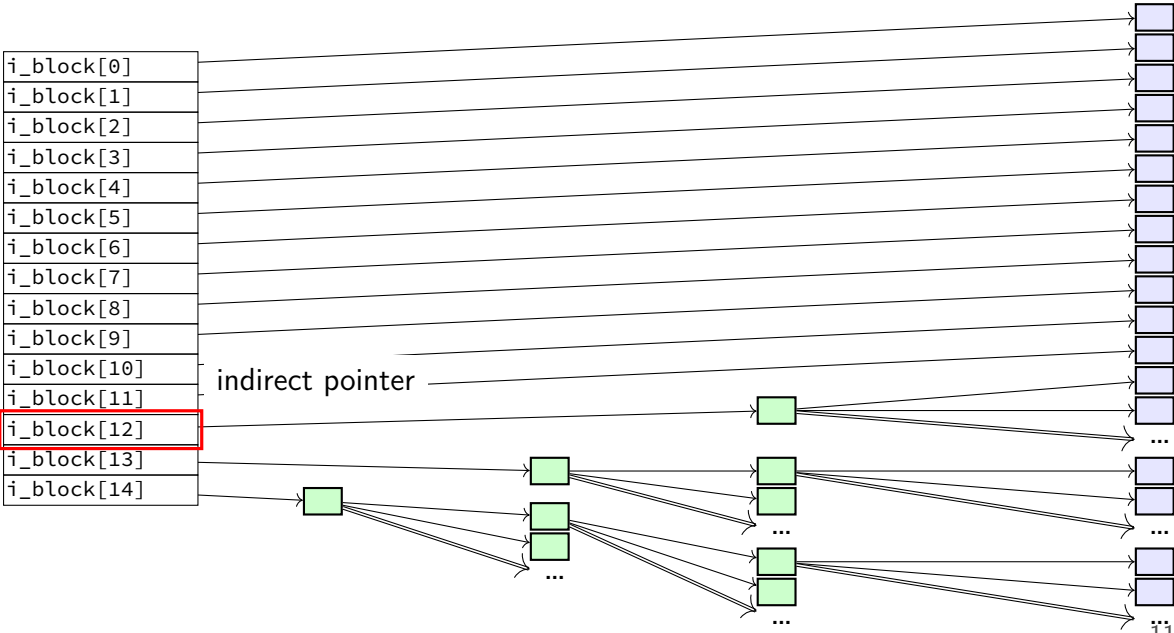
double/triple indirect

12 direct pointers

i_block[0]
i_block[1]
i_block[2]
i_block[3]
i_block[4]
i_block[5]
i_block[6]
i_block[7]
i_block[8]
i_block[9]
i_block[10]
i_block[11]
i_block[12]
i_block[13]
i_block[14]

double/triple indirect

double/triple indirect

double-indirect pointer
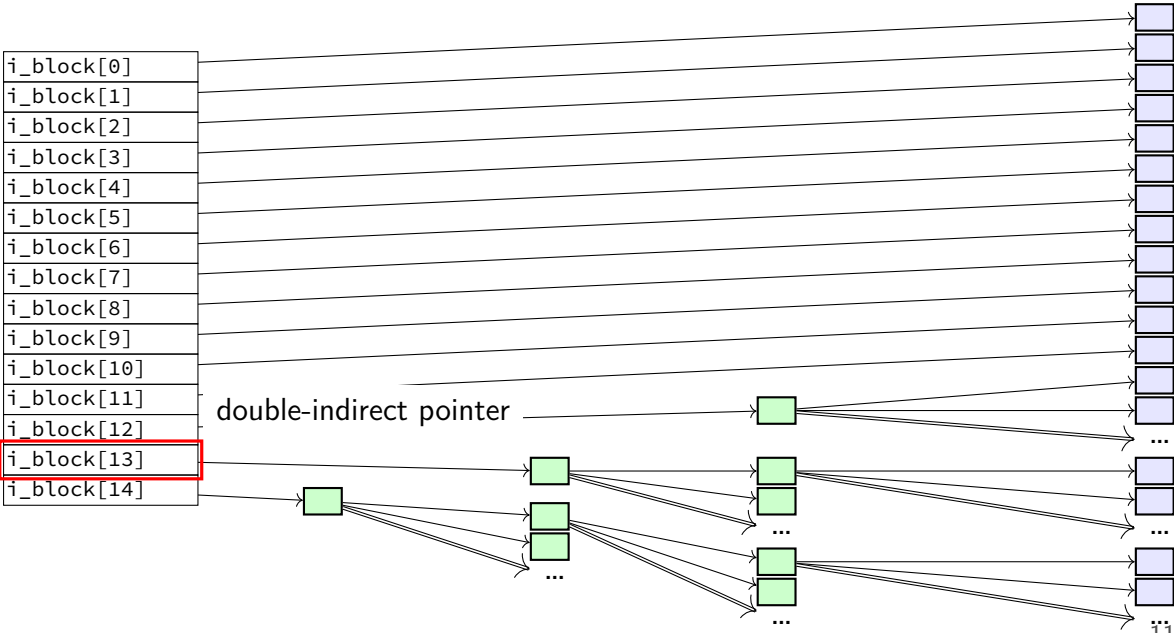
double/triple indirect

i_block[0]
i_block[1]
i_block[2]
i_block[3]
i_block[4]
i_block[5]
i_block[6]
i_block[7]
i_block[8]
i_block[9]
i_block[10]
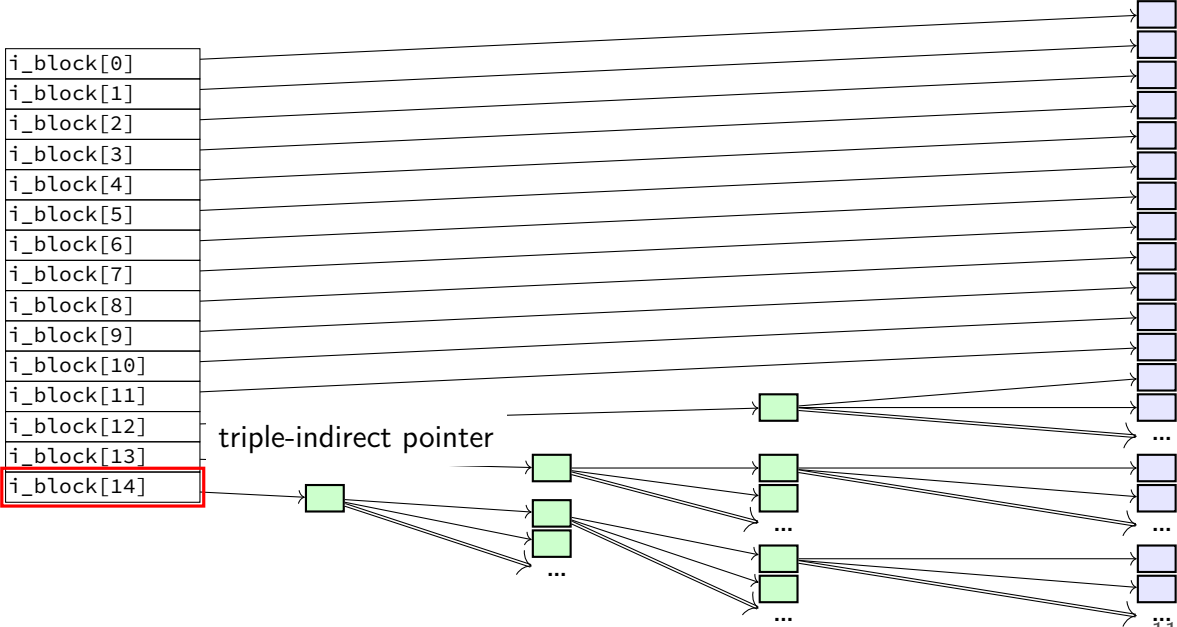i_block[11]
i_block[12]
i_block[13]
i_block[14]

triple-indirect pointer

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?

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

(log N)-like time to find block for particular offset
  no linear search like FAT

sparse files

the xv6 filesystem and ext2 allow *sparse files*
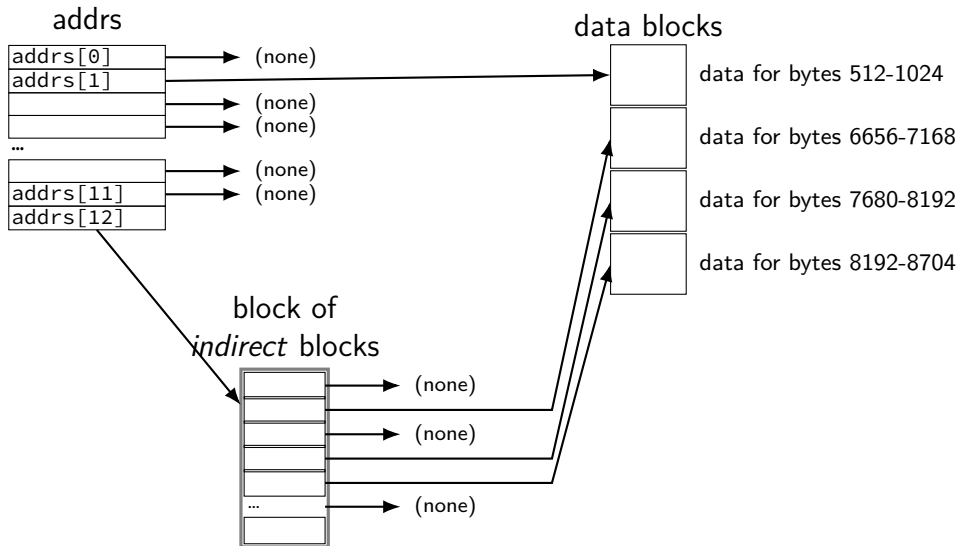
"holes" with no data blocks

```c
#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

addrs

addrs[0]	→ (none)
addrs[1]	
	→ (none)
	→ (none)
…	
	→ (none)
addrs[11]	→ (none)
addrs[12]	

data blocks

data for bytes 512-1024

data for bytes 6656-7168

data for bytes 7680-8192

data for bytes 8192-8704

block of
indirect blocks

	→ (none)
	→ (none)
…	→ (none)

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.
```

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)

other file metadata?
    creation times, etc. — xv6 doesn't have it

not good at taking advantage of HDD architecture

exercise

say xv6 filesystem with:

   64-byte inodes (12 direct + 1 indirect pointer)
   16-byte directory entries
   512 byte blocks
   2-byte block pointers

how many inodes + blocks (not storing inodes) is used to store a directory of 200 30KB files?

   remember: blocks could include blocks storing data or block pointers or directory enties

how many inodes + blocks is used to store a directory of 2000 3KB files?

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`
avoids long seek times, wasting space for tiny files

Linux's ext2 filesystem based on FFS

some other notable newer solutions (beyond what FFS/ext2 do)
better handling of very large files
avoiding linear directory searches

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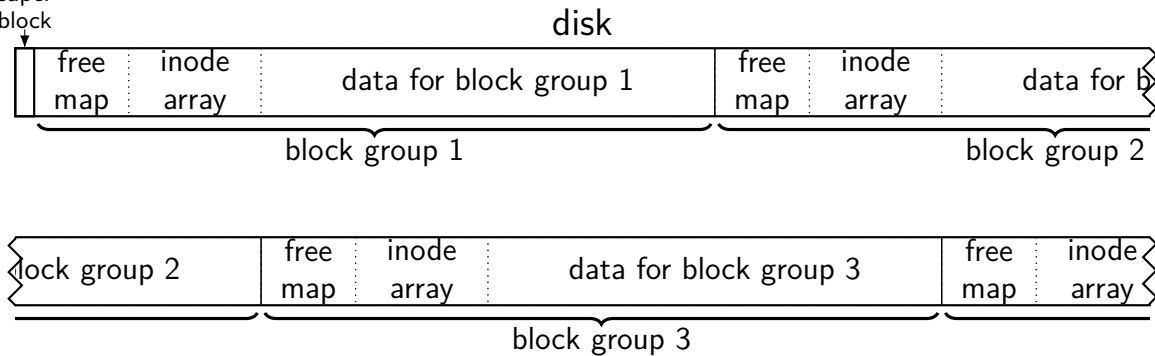   could change size? but waste space for small files
     large files have giant lists of blocks

linear searches of directory entries to resolve paths

block groups

(AKA cluster groups)

super
block

disk

free map	inode array	data for block group 1	free map	inode array	data for b

block group 1

block group 2

lock group 2	free map	inode array	data for block group 3	free map	inode array

block group 3

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

block groups
(AKA cluster groups)

super
block

disk

free map	inode array	data for block group 1	free map	inode array	data for b

inodes 0–1023   blocks 1–8191   inodes 1024–2047   blocks 8

lock group 2	free map	inode array	data for block group 3	free map	inode array

3192–16383   inodes 2048–3071   blocks 16384–24575   inodes 3072–409!
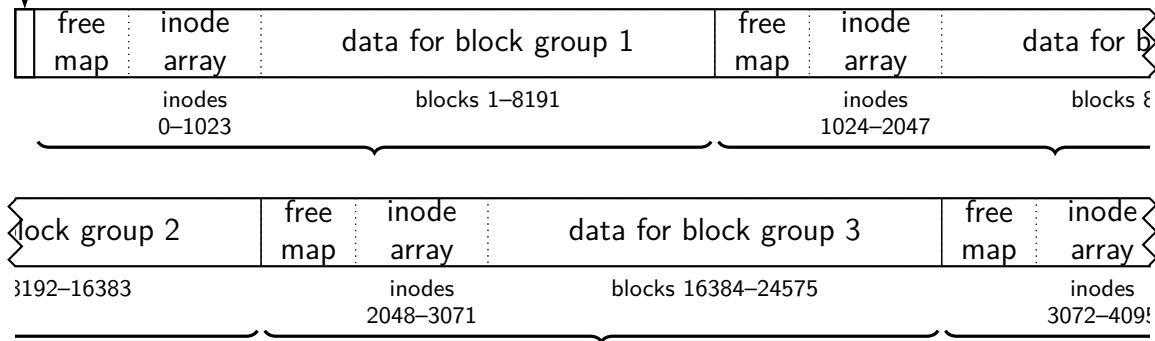
split block + inode numbers across the groups
inode in one block group can reference blocks in another
(but would rather not)

block groups
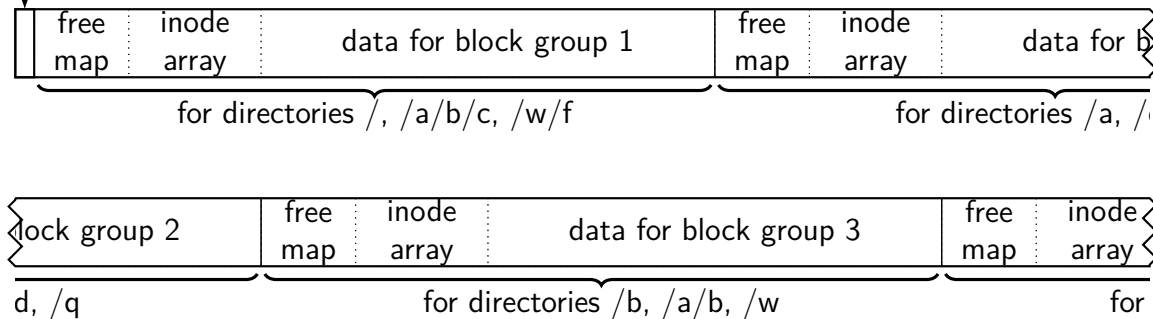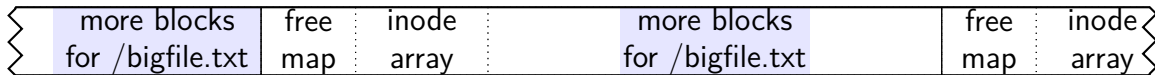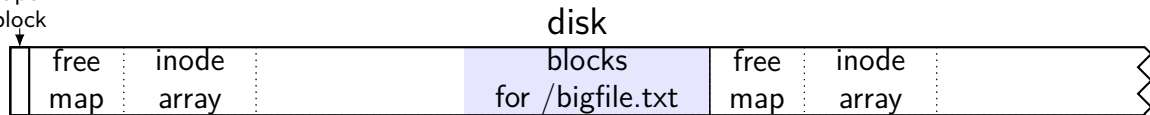(AKA cluster groups)

super
block

disk

	free map	inode array	data for block group 1	free map	inode array	data for b

$\underbrace{\qquad\qquad\qquad\qquad}$ for directories /, /a/b/c, /w/f $\qquad\qquad\qquad\qquad\underbrace{\qquad}$ for directories /a, /

lock group 2	free map	inode array	data for block group 3	free map	inode array

d, /q $\qquad\qquad\underbrace{\qquad\qquad\qquad}$ for directories /b, /a/b, /w $\qquad\qquad\qquad$ for

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)

super
block

disk

| free map | inode array | | blocks for /bigfile.txt | free map | inode array | |

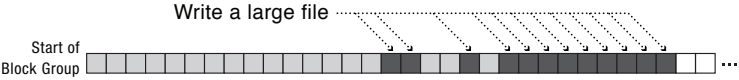| more blocks for /bigfile.txt | free map | inode array | | more blocks for /bigfile.txt | | free map | inode array |

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.

Write a large file

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

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*
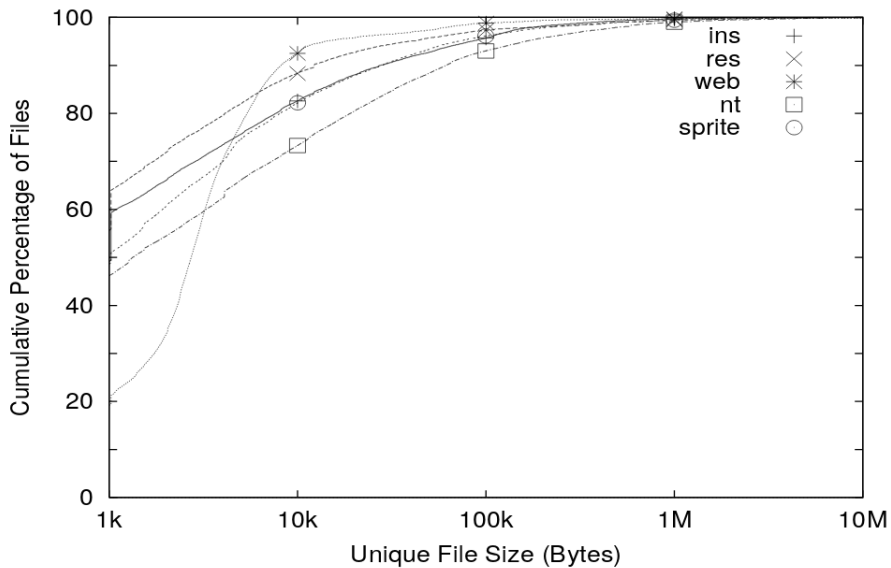   result: files/directory entries scattered about

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

empirical file sizes

typical file sizes

most files are small
  sometimes 50+% less than 1kbyte
  often 80-95% less than 10kbyte

doens't mean large files are unimportant
  still take up most of the space
  biggest performance problems

fragments

FFS: a file's last block can be a *fragment* — only part of a block

each block split into approx. 4 fragments
    each fragment has its own index

extra field in inode indicates that last block is fragment

allows one block to store data for several small files

non-FFS changes

now some techniques beyond FFS

some of these supported by current filesystems, like
Microsoft's NTFS
Linux's ext4 (successor to ext2)

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

extents

large file? lists of many thousands of blocks is awkward
    ...and requires multiple reads from disk to get

solution: store extents: (start disk block, size)
    replaces or supplements block list

Linux's ext4 and Windows's NTFS both use this

allocating extents

challenge: finding contiguous sets of free blocks

FFS's strategy "first in block group" doesn't work well
    first several blocks likely to be 'holes' from deleted files

NTFS: scan block map for "best fit"
    look for big enough chunk of free blocks
    choose smallest among all the candidates

don't find any? okay: use more than one extent

efficient seeking with extents

suppose a file has long list of extents

how to seek to byte X?

efficient seeking with extents

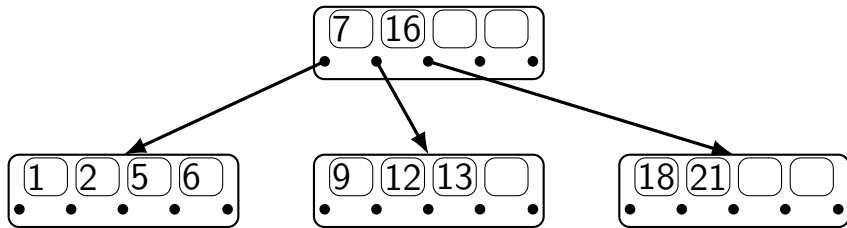suppose a file has long list of extents

how to seek to byte X?

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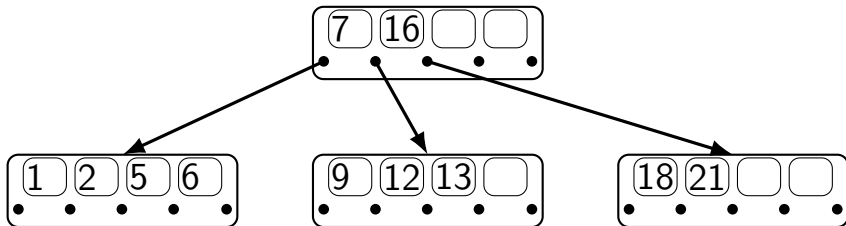   ext4: each node stores extent value=(start data block+size)
    ext4: each node has pointer (disk block) to its children

non-binary search trees

non-binary search trees
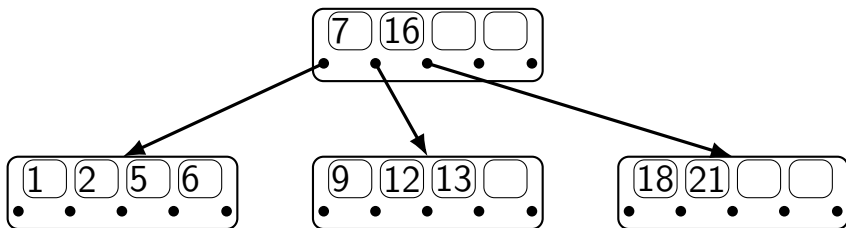

each node can be one block on disk

    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

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, $z = x \oplus y$
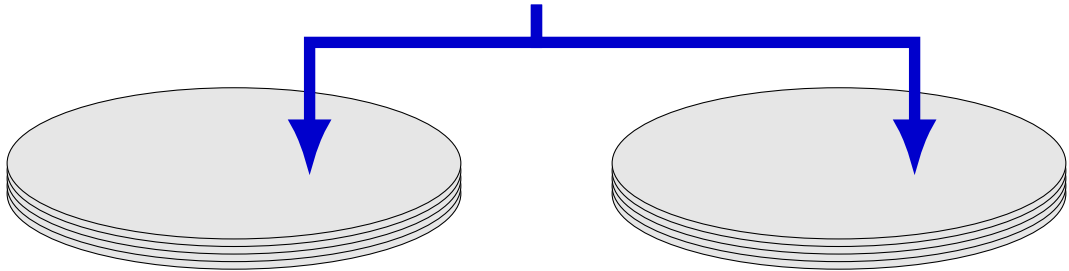
recover x: $x = y \oplus z$

recover y: $y = x \oplus z$

recover z: $y = x \oplus y$

mirroring whole disks
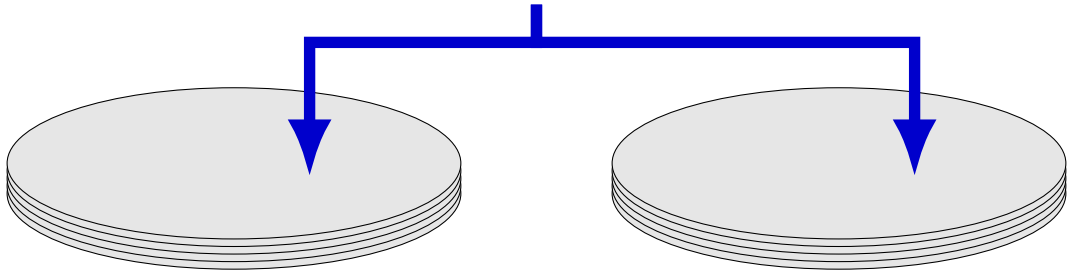
alternate strategy: write everything to two disks

always write to both

mirroring whole disks

alternate strategy: write everything to two disks

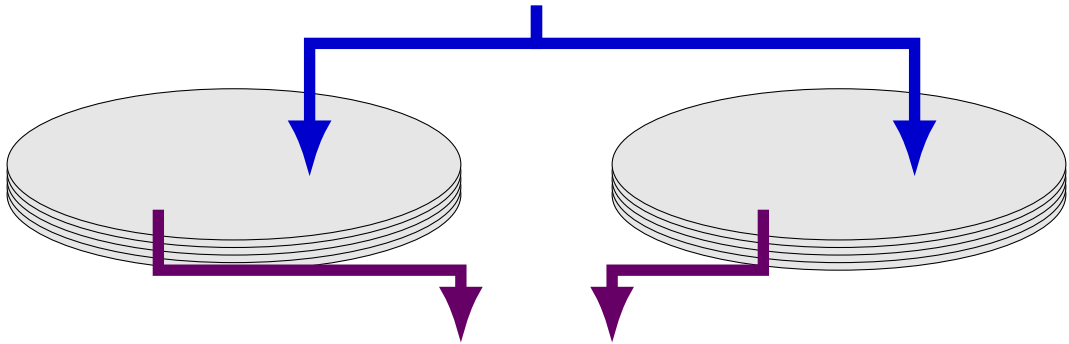always write to both

mirroring whole disks

alternate strategy: write everything to two disks

always write to both

read from either
(or different parts of both – faster!)

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