Changelog

Changes made in this version not seen in first lecture:

- 2 April 2019: block groups: change inode number labels to start at 0, not 1024
- 2 April 2019: typical file size: fix “5-20%” to “80-95%” (portion less, not more)
- 2 April 2019: efficient seeking with extents: state what is contained in nodes as key/value part

last time (1)

FAT continued

- identifying free sectors
- updating directories

hard disks

- cylinders, tracks, sectors
- seek time, rotational latency
- closer is better
- error-detecting/correcting codes

last time (2)

solid state disks

- sector remapping
- relocating data to work around erasure blocks

inodes-based FS — file data in one place

- one array of inodes
- directories contain *hard links* — name + inode number

block pointer arrays + (double/triple)-indirect 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 */
    ...
};
```

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

```
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 */
    ...
};
```

owner and group

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 */
    ...
};
```

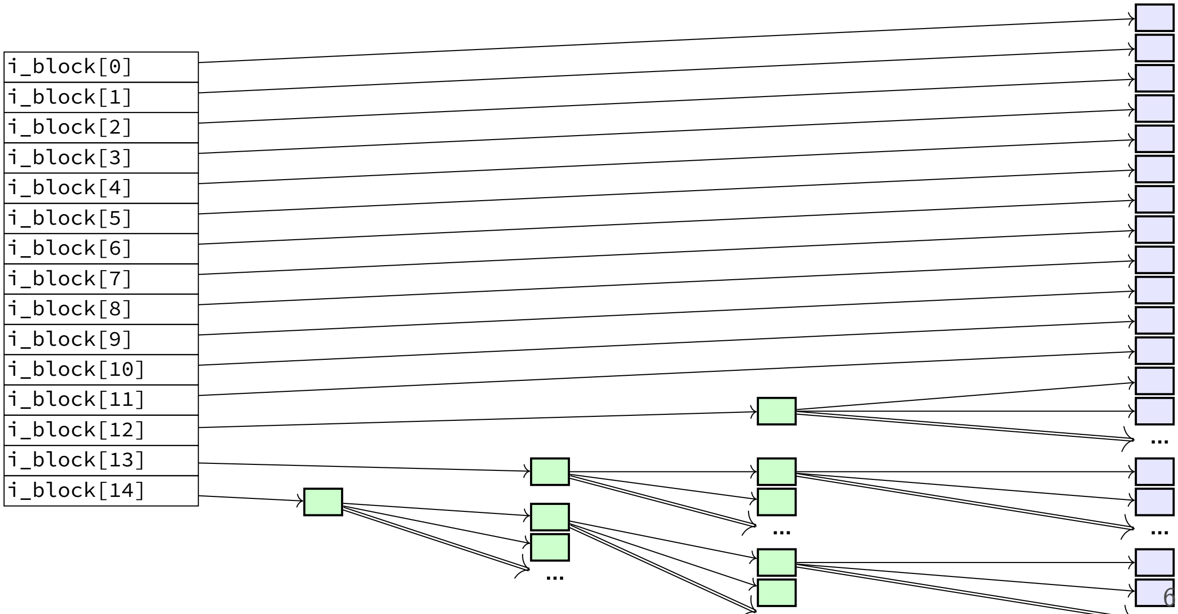
whole bunch of times

Linux ext2 inode

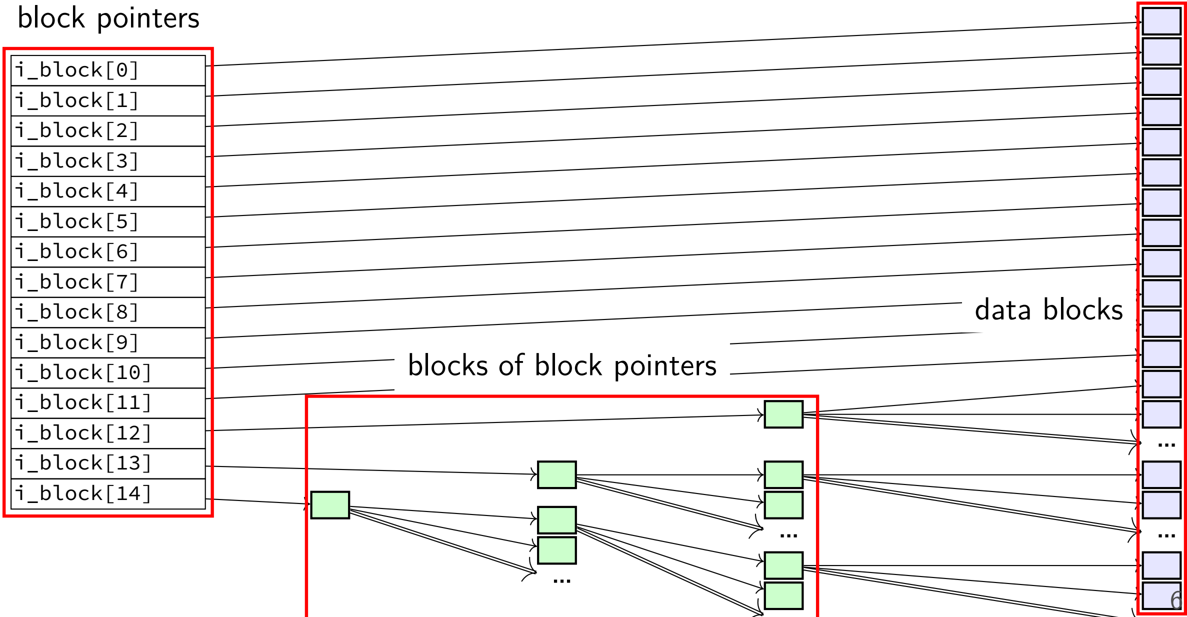
similar pointers like xv6 FS — but more indirection

```
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 */
    ...
};
```

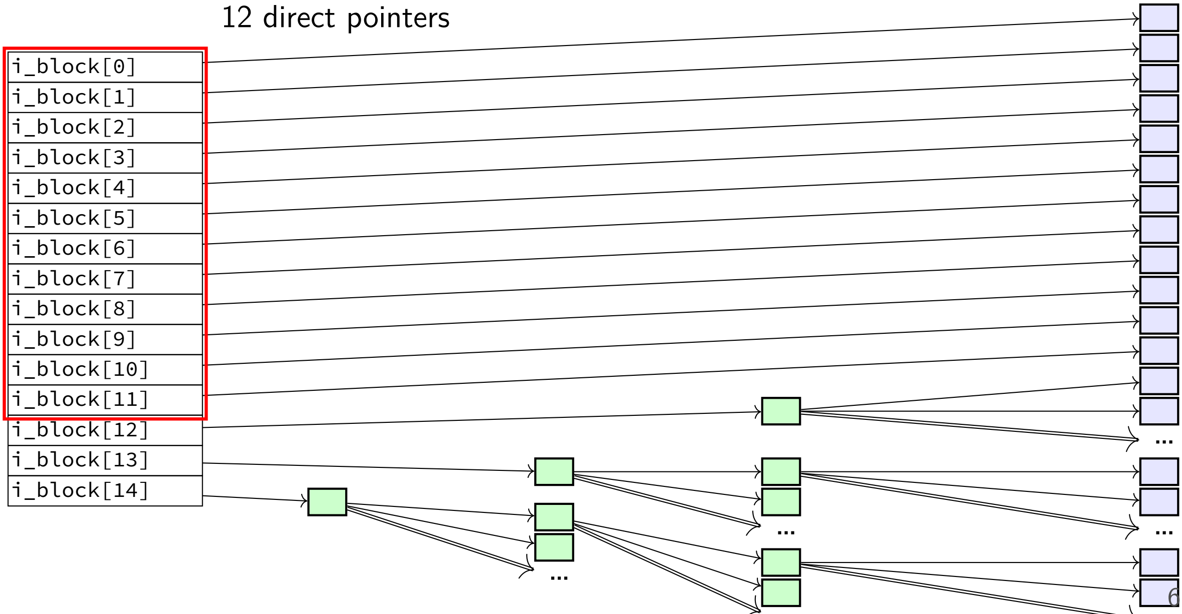
double/triple indirect



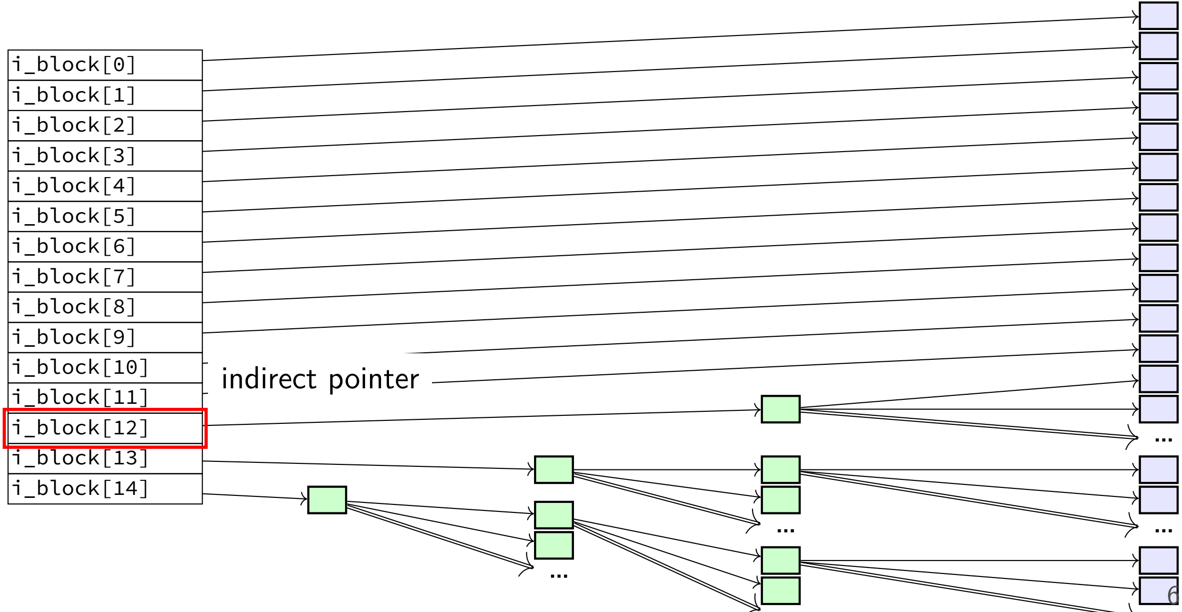
double/triple indirect



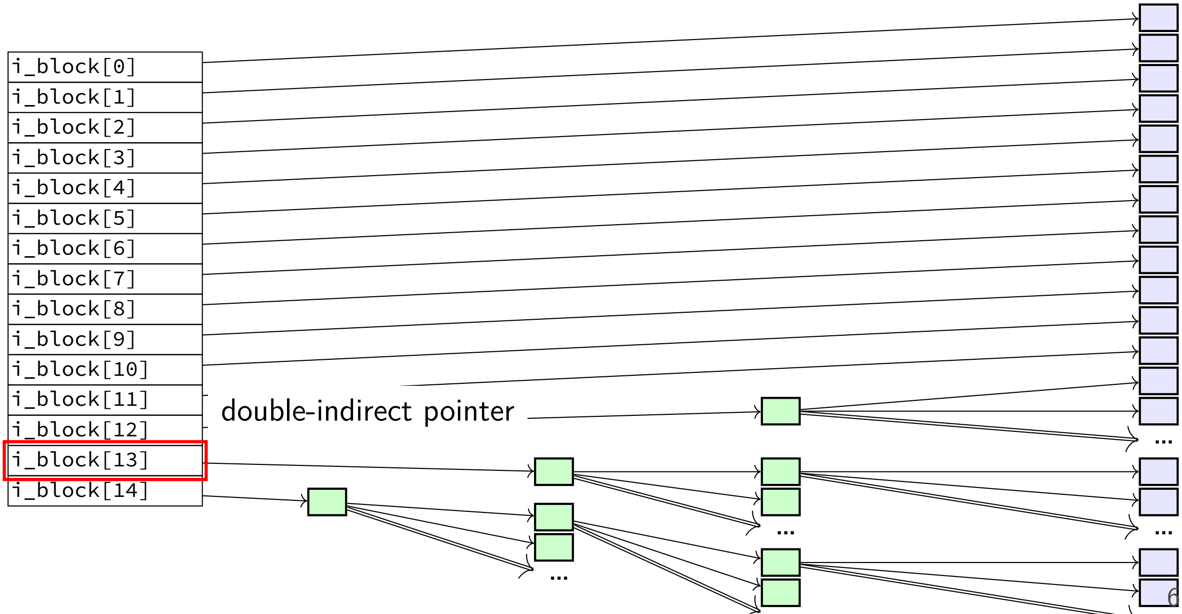
double/triple indirect



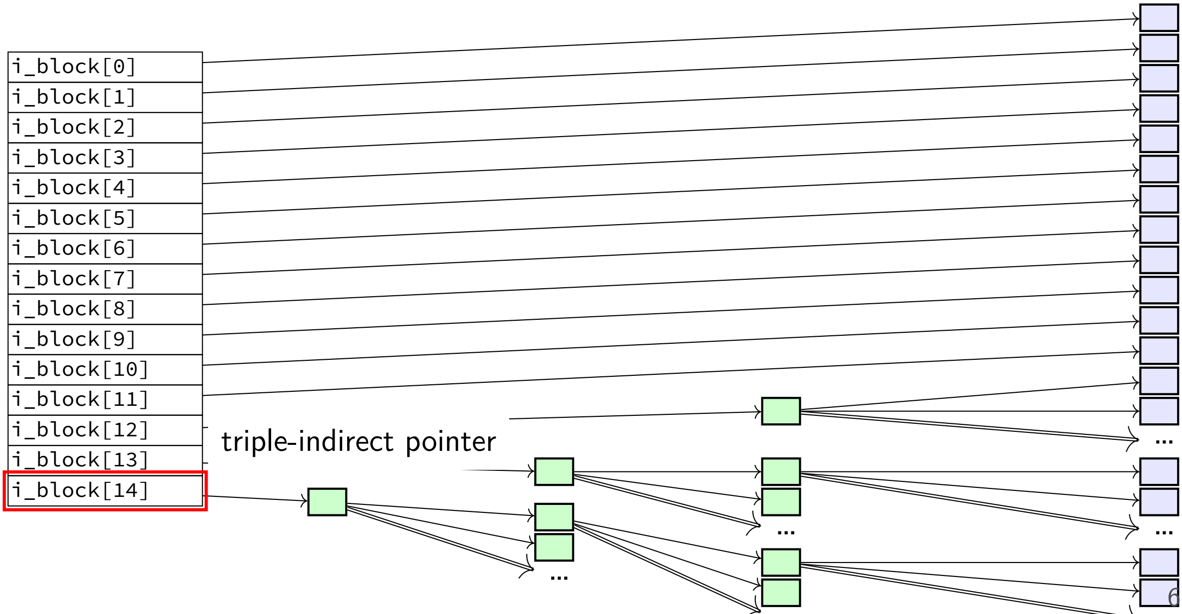
double/triple indirect



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?

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

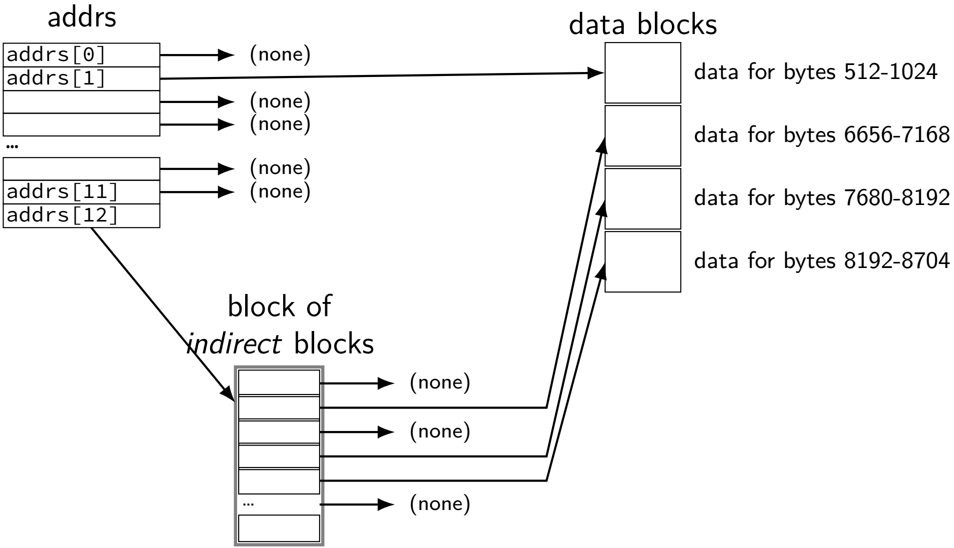
```
#include <stdio.h>
int main(void) {
    FILE *fh = fopen("sparse.dat", "w");
    fseek(fh, 1024 * 1024, SEEK_SET);
    fprintf(fh, "Some data here\n");
    fclose(fh);
}
```

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.
```

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

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

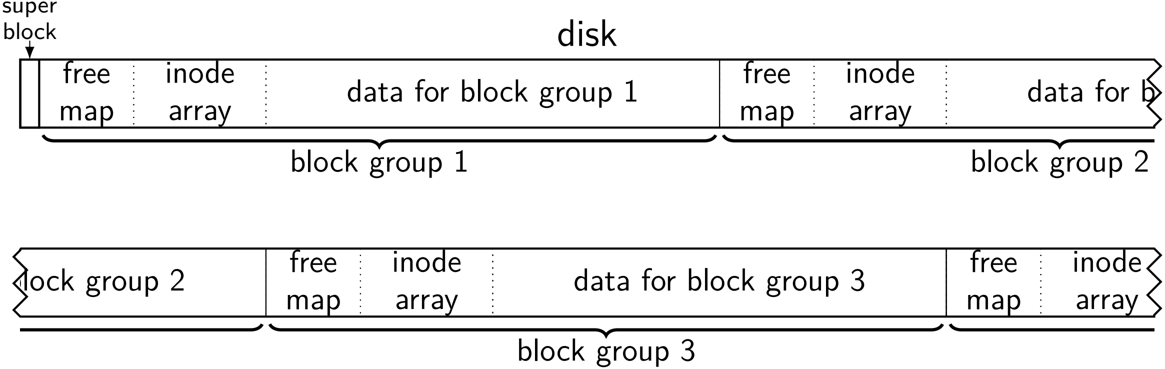
blocks are pretty small — needs lots of space for metadata

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

linear searches of directory entries to resolve paths

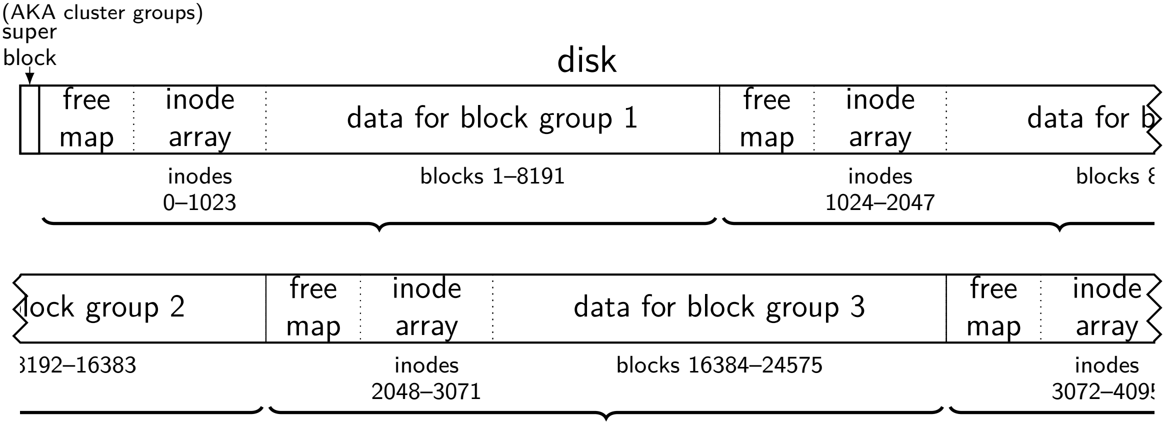
block groups

(AKA cluster groups)



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

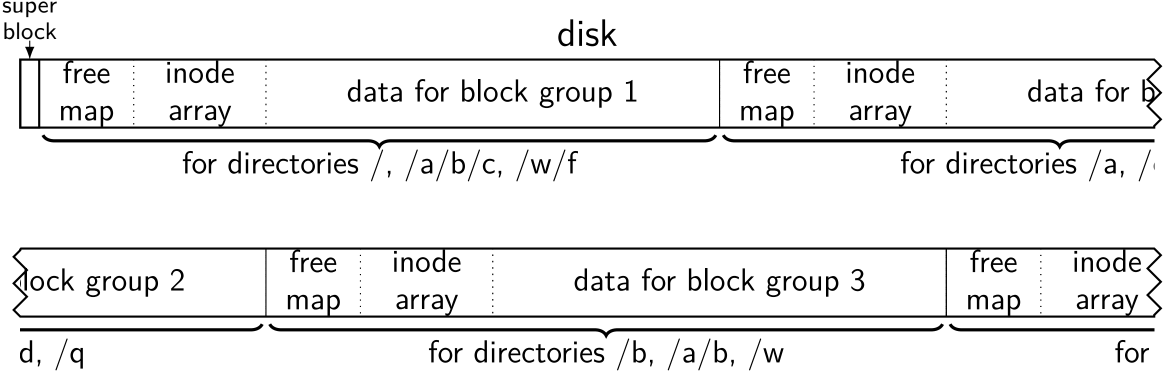
block groups



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

block groups

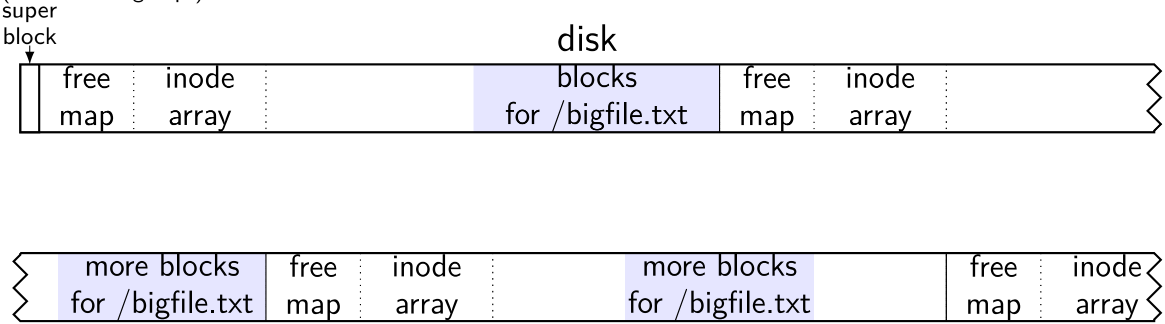
(AKA cluster groups)



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

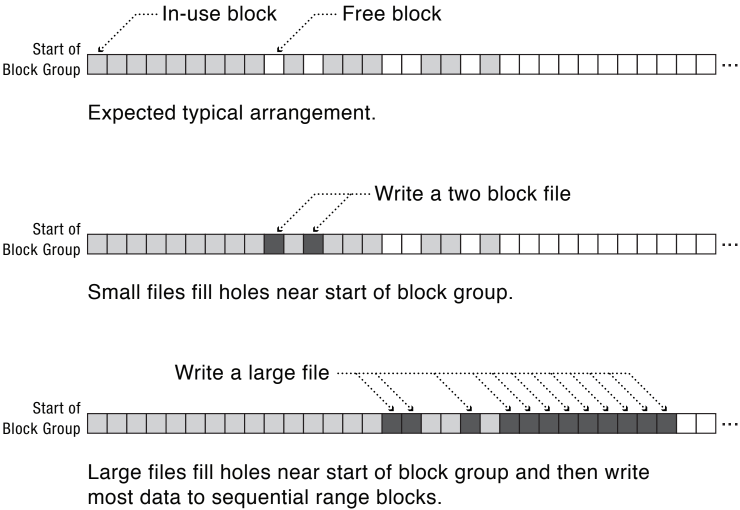
block groups

(AKA cluster groups)



large files might need to be split across block groups

allocation within block groups



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

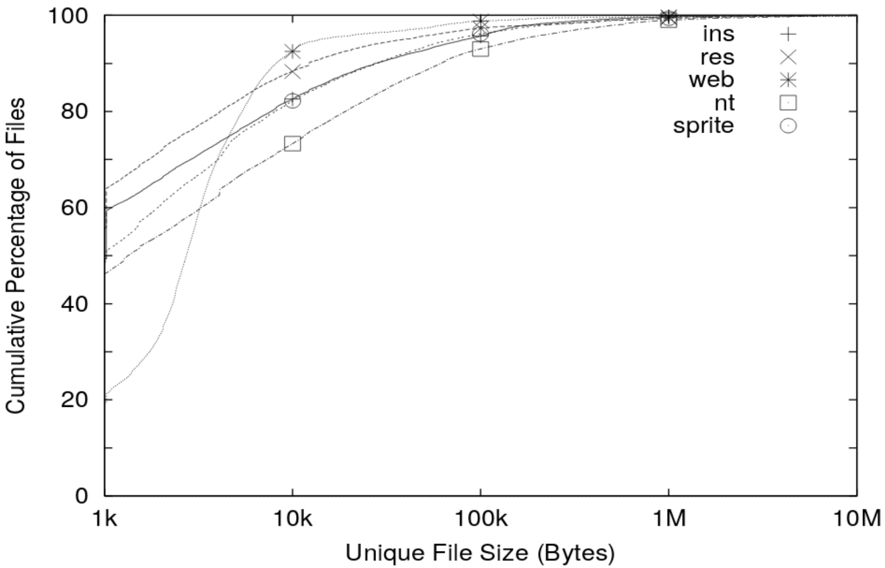
- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

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

empirical file sizes



Roselli et al, “A Comparison of Filesystem Workloads”, in FAST 2000

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

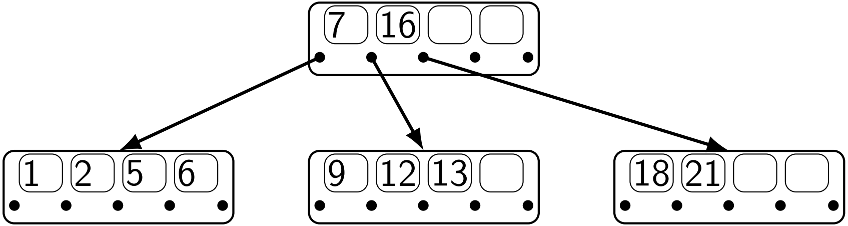
suppose a file has long list of extents

how to seek to byte X?

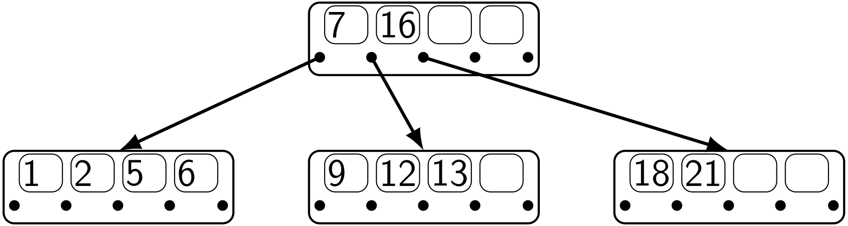
solution: store a (search) tree

- 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



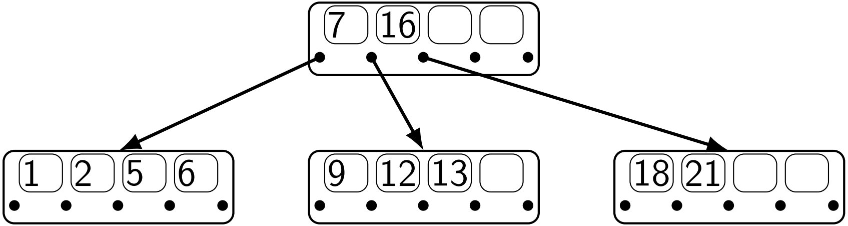
each node can be one block on disk

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

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

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?

multiple copies

FAT: multiple copies of file allocation table and header

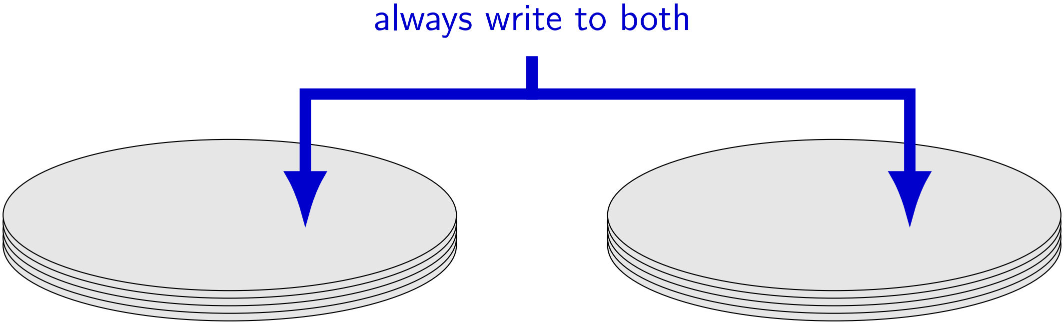
in inode-based filesystems: often multiple superblocks

if part of disk's data is lost, have an extra copy
- always update both copies
- hope: disk failure to small group of sectors

hope: enough to recover most files on disk failure

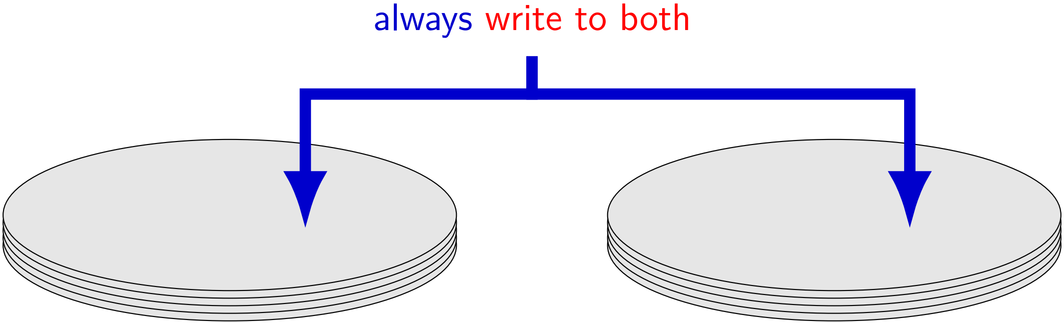
mirroring whole disks

alternate strategy: write everything to two disks



mirroring whole disks

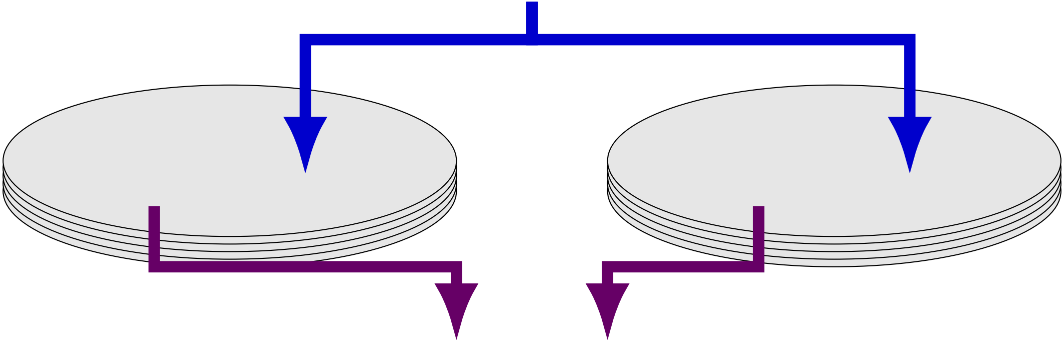
alternate strategy: write everything to two disks



mirroring whole disks

alternate strategy: write everything to two disks

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