last time

device files:

- open(/dev/snd/...) + write() to play audio
- open(/dev/input/mouse...) + read() to get keypresses
- etc.

device driver: implements talking to device

top half: handle read()/write() for device file

- typically: read/write kernel buffer
- if needed, setup device for bottom half

bottom half: handle exceptions from device

- typically: if no DMA, copy from kernel buffer to device
- respond to device being having input/ready for output/done

OS to disk interface

disk takes read/write requests
- sector number(s)
- location of data for sector
- modern disk controllers: typically direct memory access

for spinning disks: close sector numbers → close physically
- faster to read/write together

can have queue of pending requests

disk processes them in some order
- OS can say "write X before Y"

filesystems

the FAT filesystem

FAT: File Allocation Table

probably simplest widely used filesystem (family)

named for important data structure: *file allocation table*

FAT and sectors

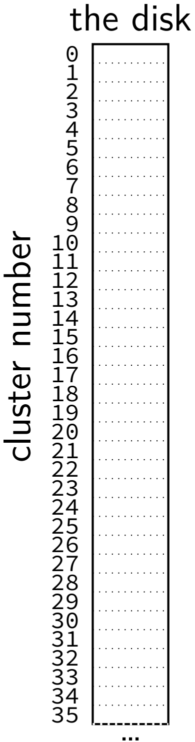
FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read/write

- determined by disk hardware
- historically 512 bytes, but often bigger now

cluster: typically 512 to 4096 bytes



FAT and sectors

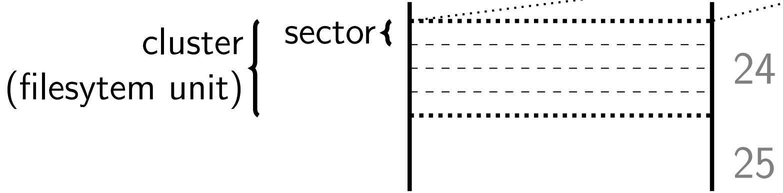
FAT divides disk into *clusters*

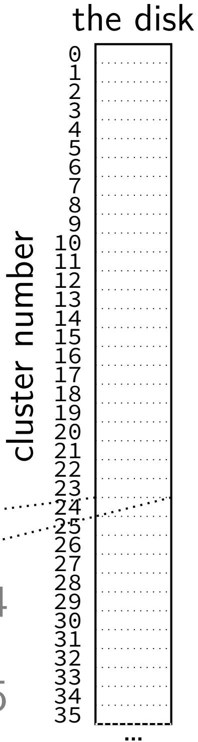
composed of one or more sectors

sector = minimum amount hardware can read/write

- determined by disk hardware
- historically 512 bytes, but often bigger now

cluster: typically 512 to 4096 bytes



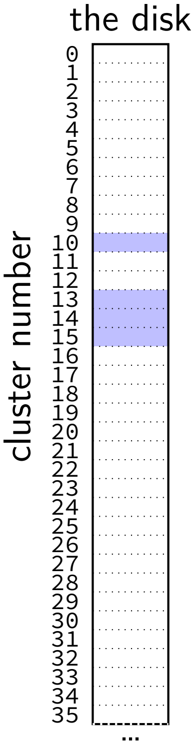


FAT: clusters and files

a file's data stored in a list of clusters

file size isn't multiple of cluster size? waste space

reading a file? need to find the list of clusters

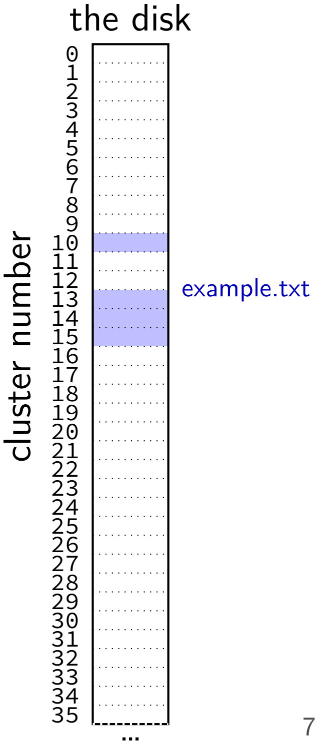


FAT: clusters and files

a file's data stored in a list of clusters

file size isn't multiple of cluster size? waste space

reading a file? need to find the list of clusters



FAT: the file allocation table

big array on disk, one entry per cluster

each entry contains a number — usually "next cluster"

cluster num.	**entry value**
0	4
1	7
2	5
3	1434
...	...
1000	4503
1001	1523
...	...

FAT: reading a file (1)

get (from elsewhere) first cluster of data

linked list of cluster numbers

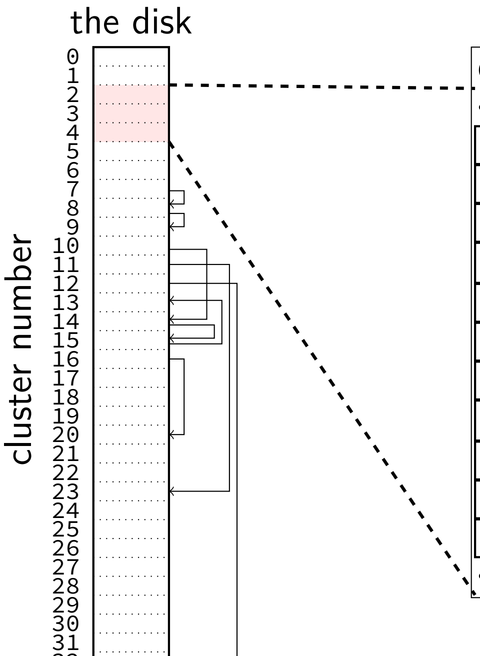
next pointers? file allocation table entry for cluster

special value for NULL (-1 in this example; maybe different in real FAT)

cluster num.	**entry value**
	…
10	14
11	23
12	54
13	(end mark)
14	15
15	13
…	…

file starting at cluster 10 contains data in:
cluster 10, then 14, then 15, then 13

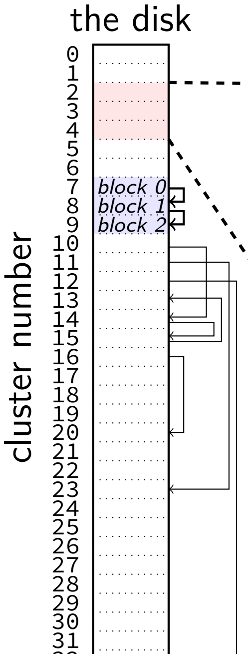
FAT: reading a file (2)



file allocation table

entry value	**index**
...	...
21	6
8	7
9	8
(end mark)	9
14	10
23	11
54	12
(end mark)	15
15	14
13	15
20	16
...	...

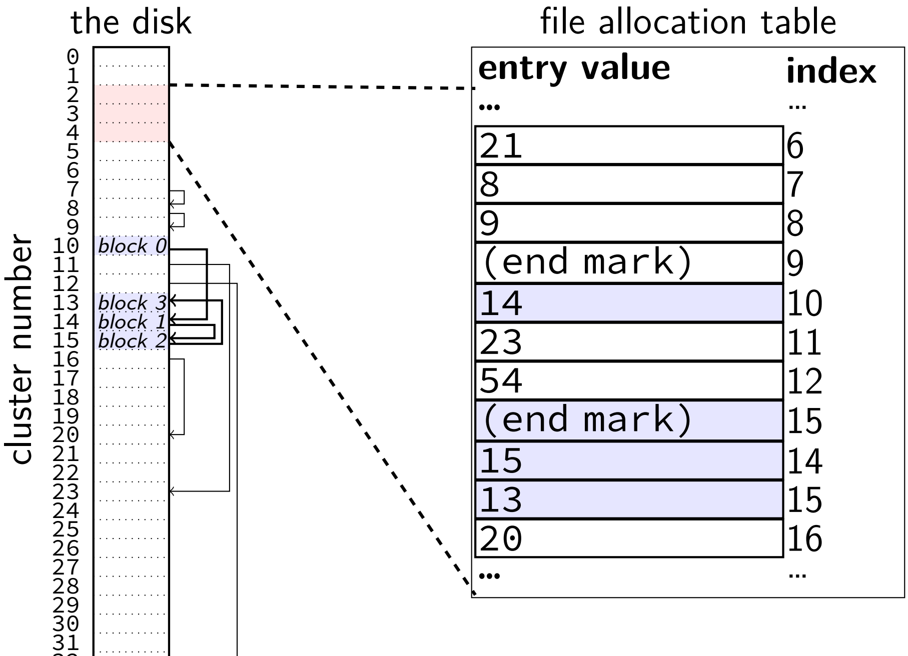
FAT: reading a file (2)



file allocation table

entry value	index
...	...
21	6
8	7
9	8
(end mark)	9
14	10
23	11
54	12
(end mark)	15
15	14
13	15
20	16
...	...

FAT: reading a file (2)



FAT: reading files

to read a file given it's start location

read the starting cluster X

get the next cluster Y from FAT entry X

read the next cluster

get the next cluster from FAT entry Y

...

until you see an end marker

start locations?

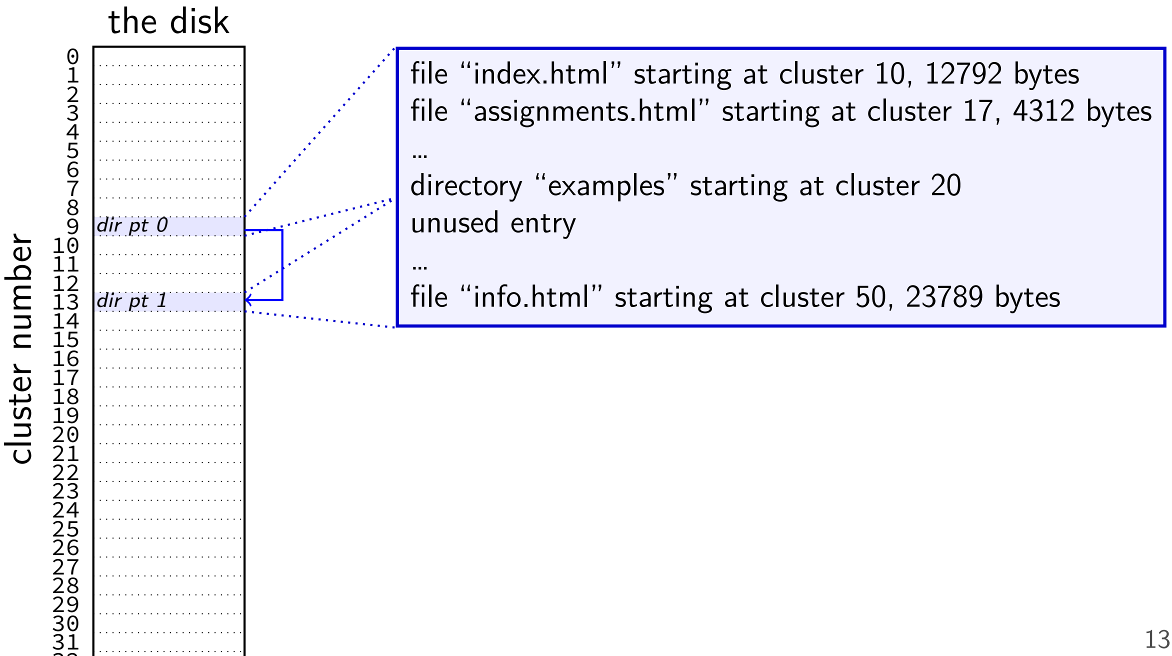
really want filenames

stored in directories!

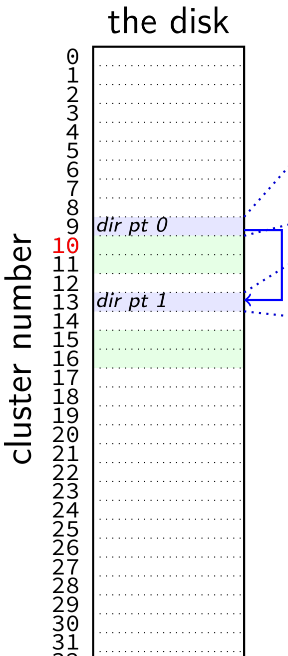
in FAT: directory is a file, but its data is list of:

(name, starting location, other data about file)

finding files with directory

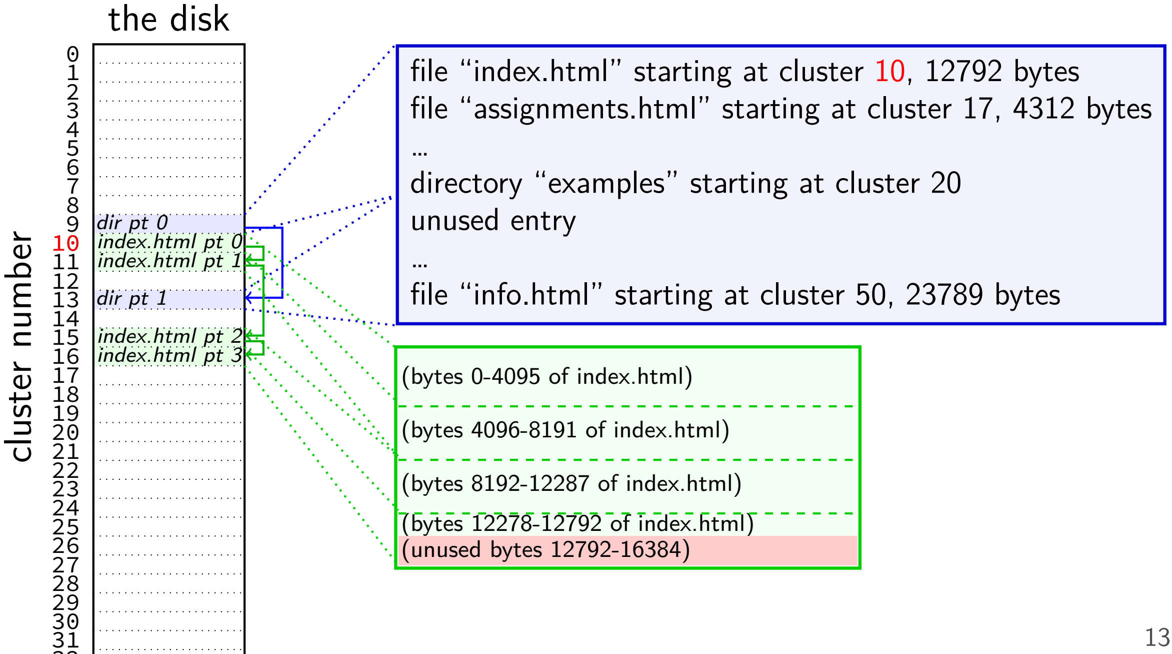


finding files with directory

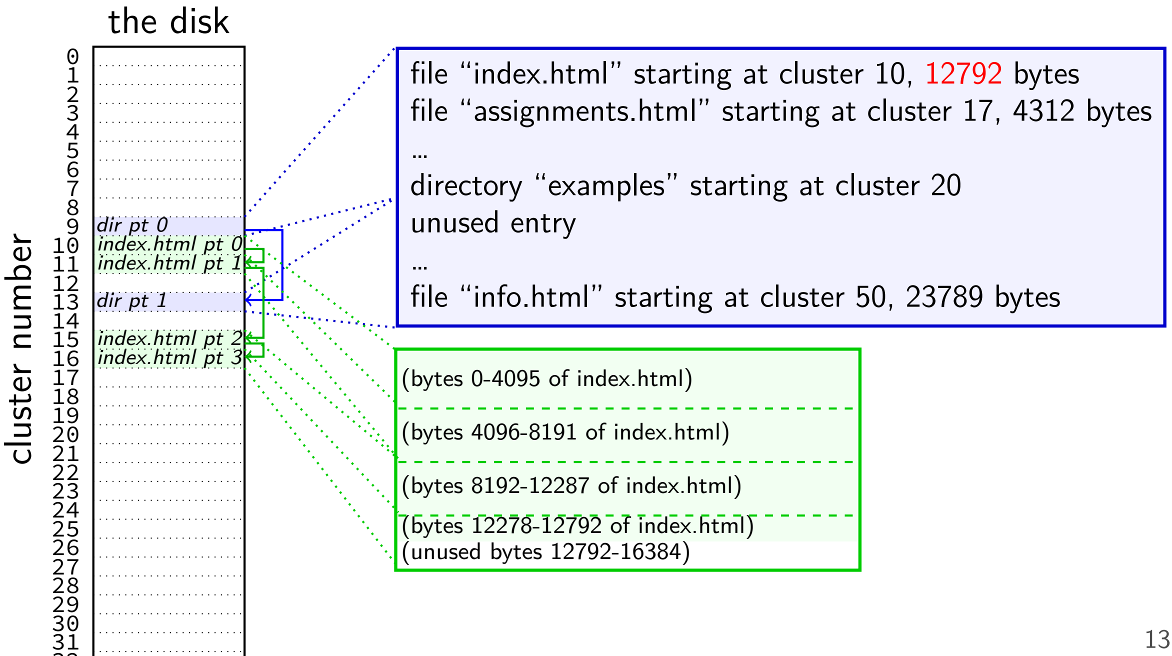


file “index.html” starting at cluster 10, 12792 bytes
file “assignments.html” starting at cluster 17, 4312 bytes
…
directory “examples” starting at cluster 20
unused entry
…
file “info.html” starting at cluster 50, 23789 bytes

finding files with directory



finding files with directory



FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?

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

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

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?

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

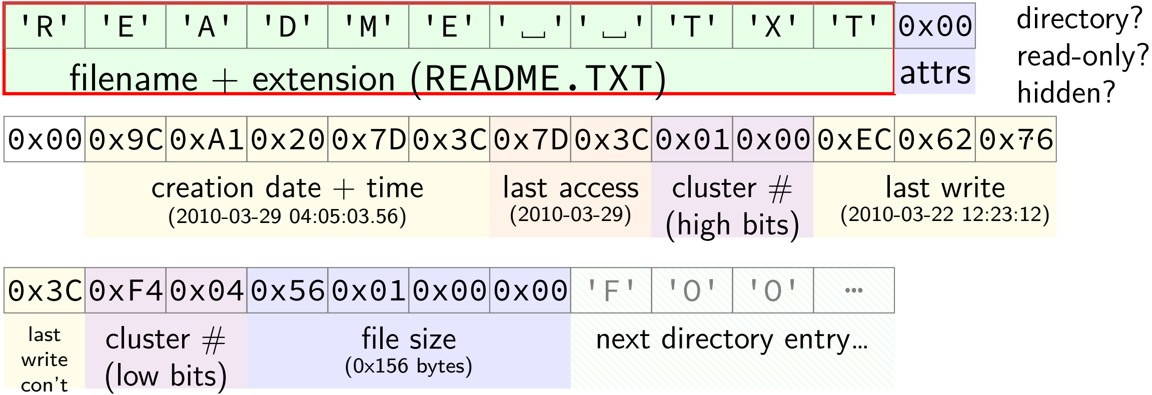
0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	…
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry…			

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

FAT directory entry

box = 1 byte

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



8 character filename + 3 character extension

longer filenames? encoded using extra directory entries

(special `attrs` values to distinguish from normal entries)

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?

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

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

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

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?

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

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

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

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?

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

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

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

FAT directory entries (from C)

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

FAT directory entries (from C)

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

GCC/Clang extension to disable padding
normally compilers add padding to structs
(to avoid splitting values across cache blocks or pages)

FAT directory entries (from C)

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

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

FAT directory entries (from C)

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

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

nested directories

foo/bar/baz/file.txt

read root directory entries to find foo

read foo's directory entries to find bar

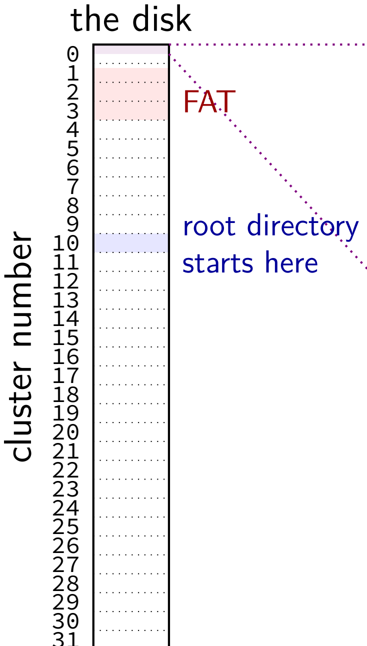
read bar's directory entries to find baz

read baz's directory entries to find file.txt

the root directory?

but where is the first directory?

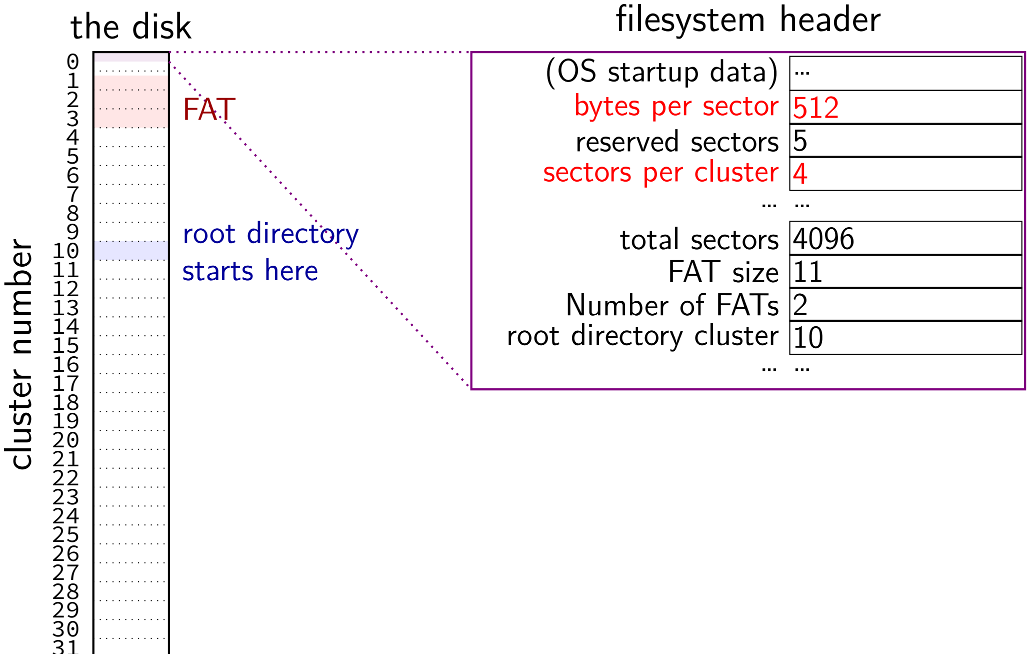
FAT disk header



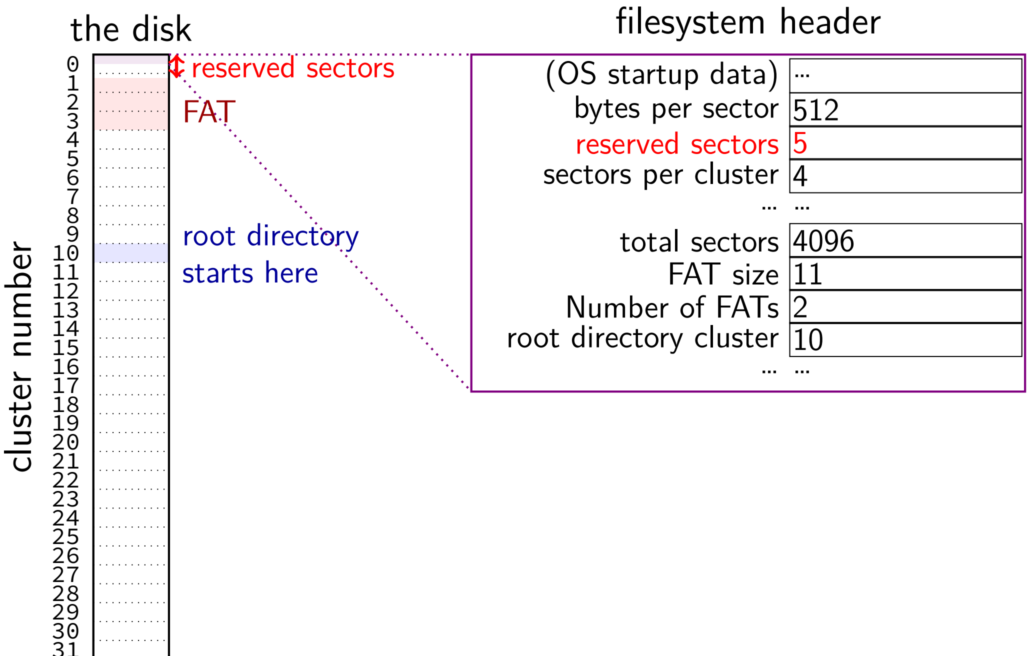
filesystem header

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

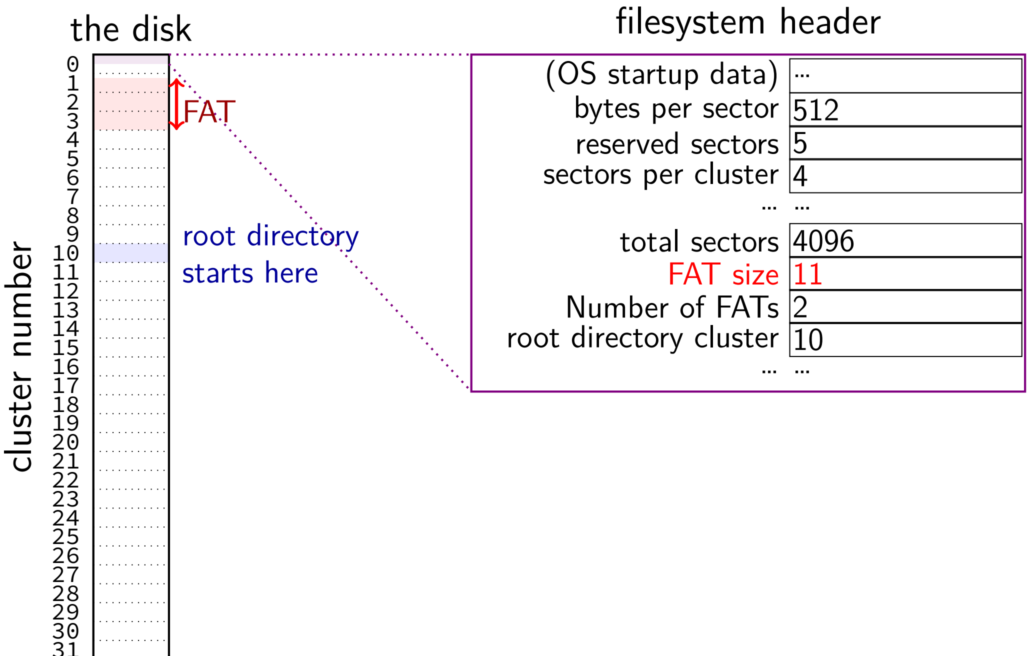
FAT disk header



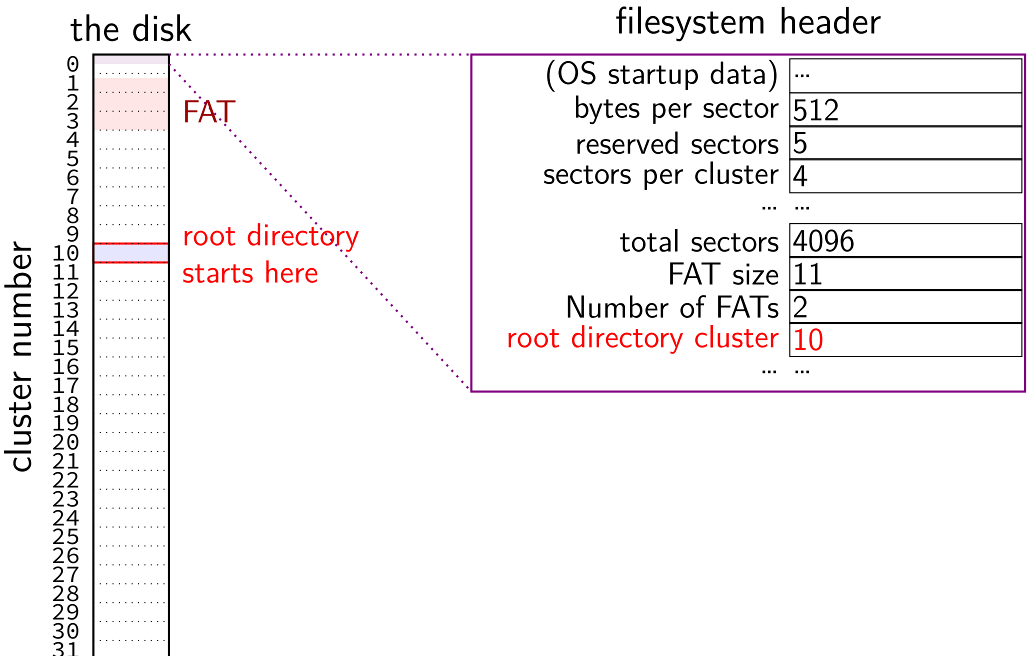
FAT disk header



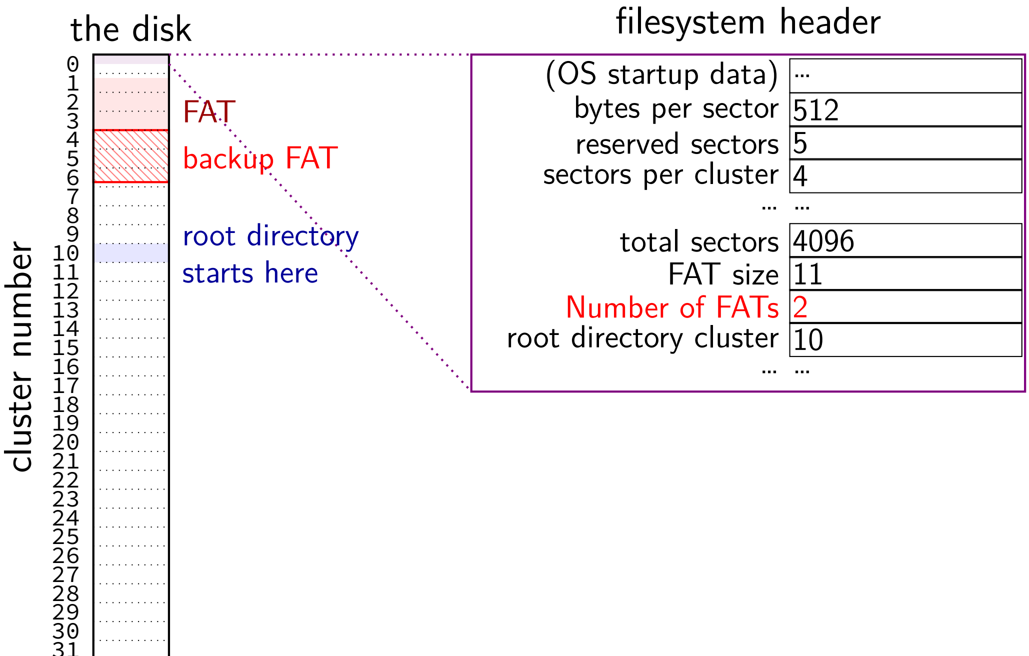
FAT disk header



FAT disk header



FAT disk header



filesystem header

fixed location near beginning of disk

determines size of clusters, etc.

tells where to find FAT, root directory, etc.

FAT header (C)

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

FAT header (C)

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

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

FAT header (C)

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

space before file allocation table

FAT header (C)

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

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

FAT: creating a file

add a directory entry

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

update FAT to link clusters together

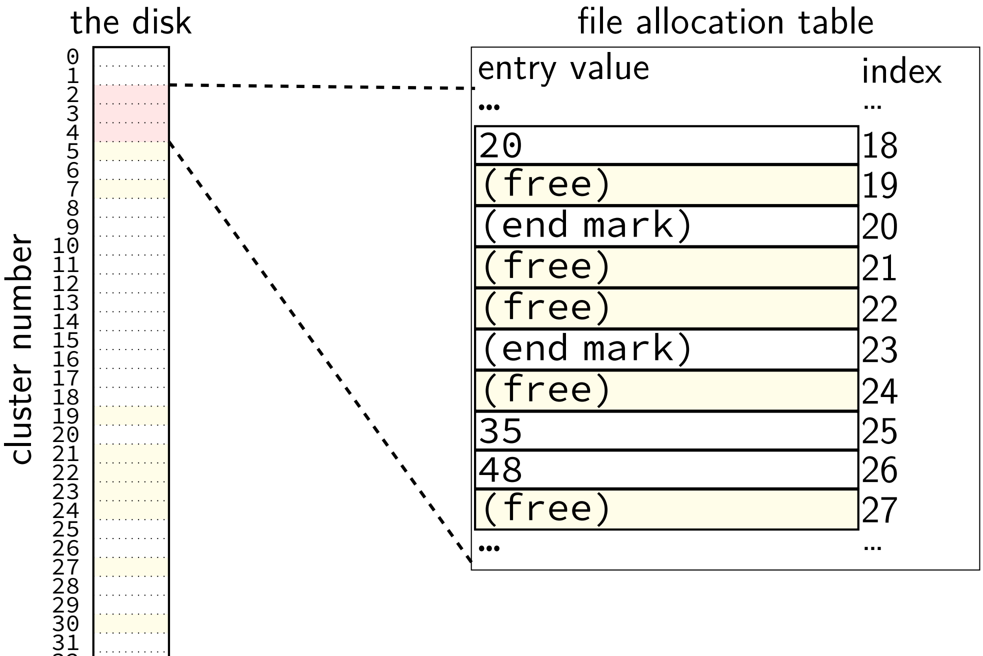
FAT: creating a file

add a directory entry

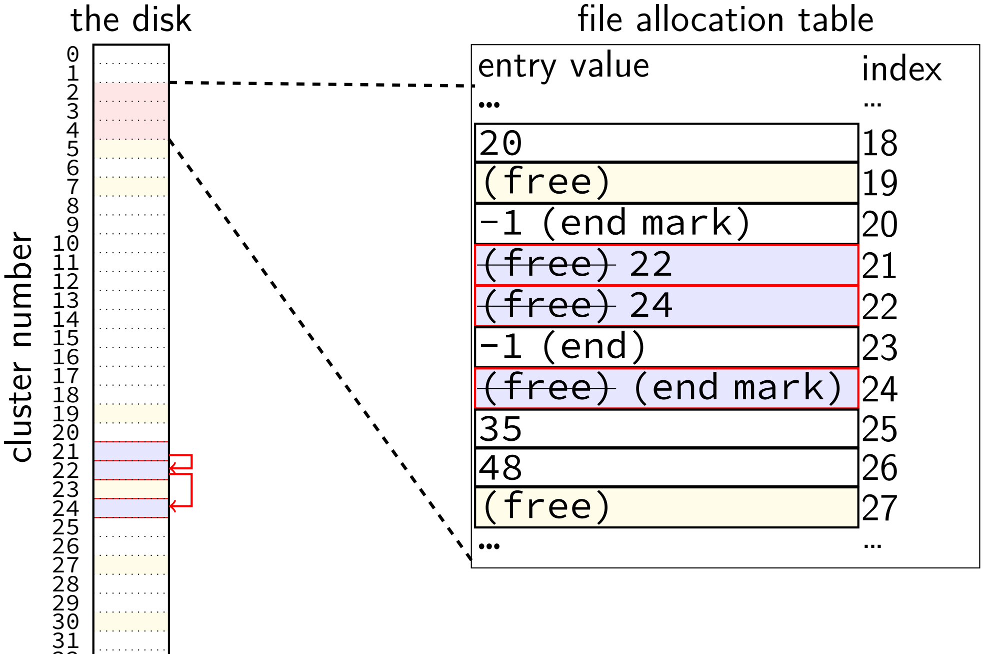
choose clusters to store file data (how???)

update FAT to link clusters together

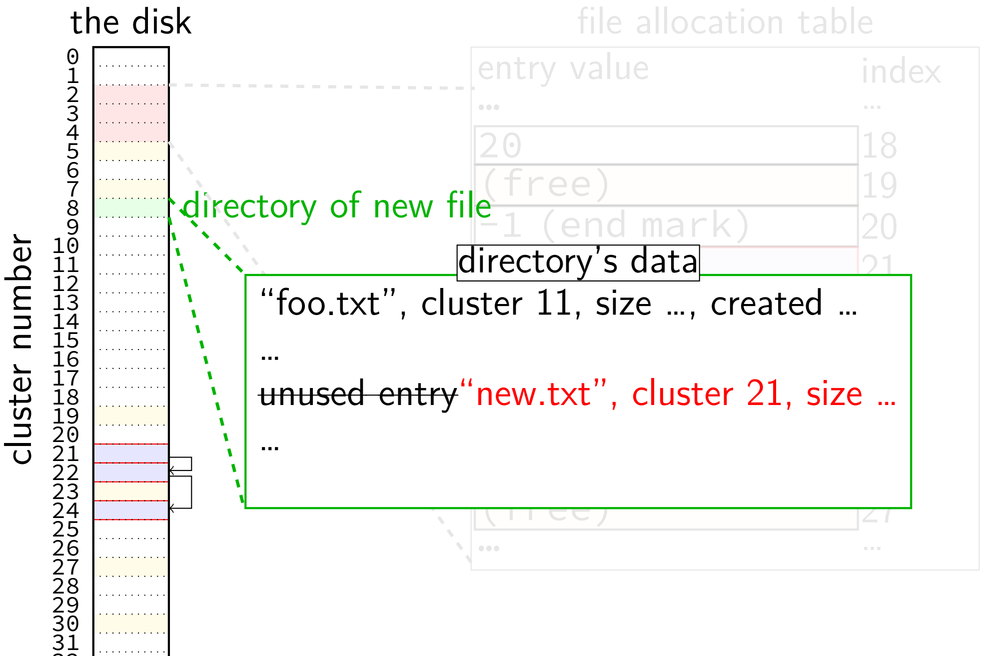
FAT: free clusters



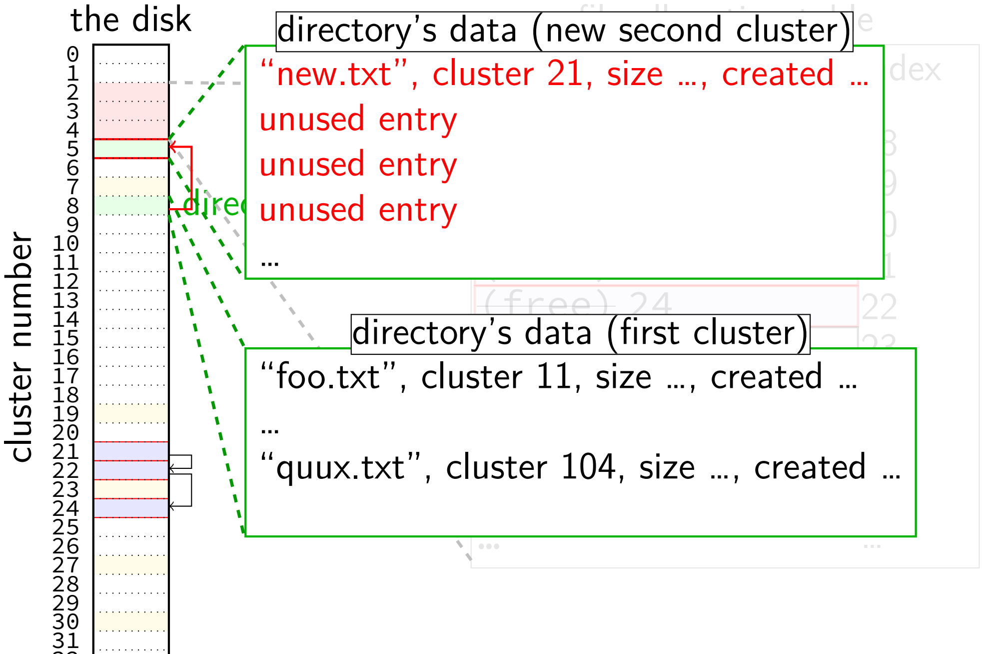
FAT: writing file data



FAT: replacing unused directory entry



FAT: extending directory



FAT: exercise

C.txt is file in directory B which is in directory A

consider the following items on disk:

[a] FAT entries for A's cluster(s)
[b] FAT entries for B's clsuter(s)
[c] FAT entries for C.txt's cluster(s)
[d] data clusters for A
[e] data clusters for B
[f] data clusters for C.txt

Ignoring modification timestamp updates,
which of the above **may** be modified to:

1) assuming directores existed previously, create C.txt
2) truncate C.txt, making it have size 0 bytes (assume prev. not empty)
3) move C.txt from directory B into directory A

FAT: exercise

C.txt is file in directory B which is in directory A

consider the following items on disk:

[a] FAT entries for A's cluster(s)
[b] FAT entries for B's clsuter(s)
[c] FAT entries for C.txt's cluster(s)
[d] data clusters for A
[e] data clusters for B
[f] data clusters for C.txt

Ignoring modification timestamp updates,
which of the above **may** be modified to:

1) assuming directores existed previously, create C.txt

FAT: exercise

C.txt is file in directory B which is in directory A

consider the following items on disk:

[a] FAT entries for A's cluster(s)
[b] FAT entries for B's clsuter(s)
[c] FAT entries for C.txt's cluster(s)
[d] data clusters for A
[e] data clusters for B
[f] data clusters for C.txt

Ignoring modification timestamp updates,
which of the above **may** be modified to:

2) truncate C.txt, making it have size 0 bytes (assume prev. not empty)

FAT: exercise

C.txt is file in directory B which is in directory A

consider the following items on disk:

[a] FAT entries for A's cluster(s)
[b] FAT entries for B's clsuter(s)
[c] FAT entries for C.txt's cluster(s)
[d] data clusters for A
[e] data clusters for B
[f] data clusters for C.txt

Ignoring modification timestamp updates,
which of the above **may** be modified to:

3) move C.txt from directory B into directory A

FAT: deleting files

reset FAT entries for file clusters to free (0)

write "unused" character in filename for directory entry

- maybe rewrite directory if that'll save space?

exercise

say FAT filesystem with:

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

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

- count clusters for both directory entries and the file data

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

xv6 filesystem

xv6's filesystem similar to modern Unix filesytems

better at doing contiguous reads than FAT

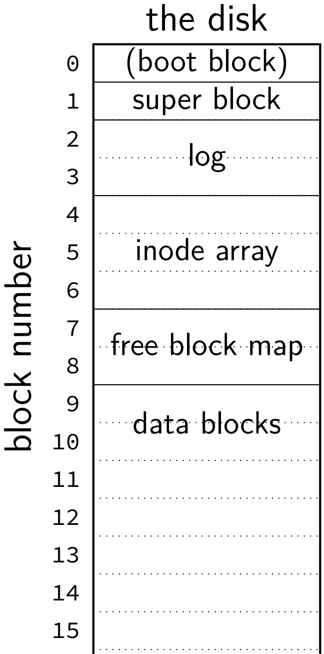
better at handling crashes

supports *hard links*

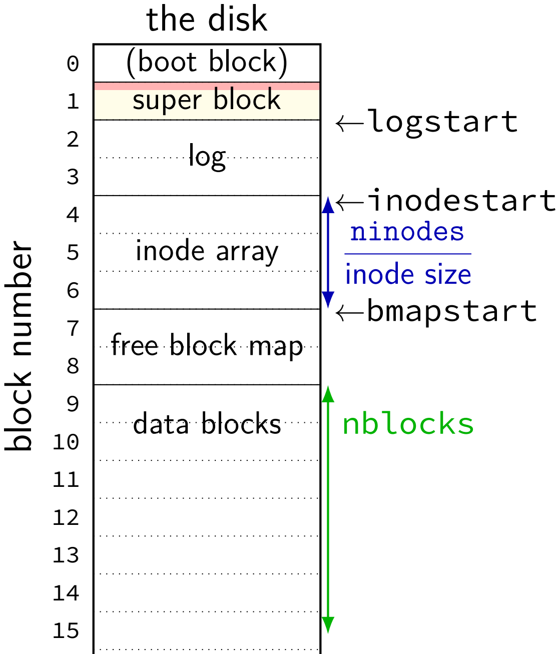
divides disk into *blocks* instead of clusters

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

xv6 disk layout



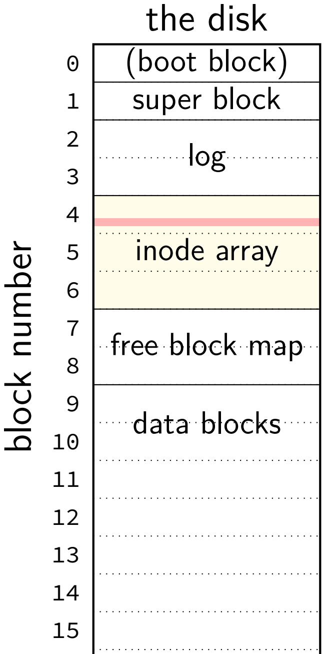
xv6 disk layout



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



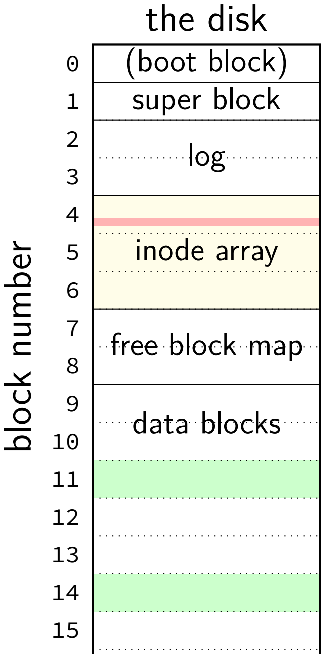
inode — file information

```
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

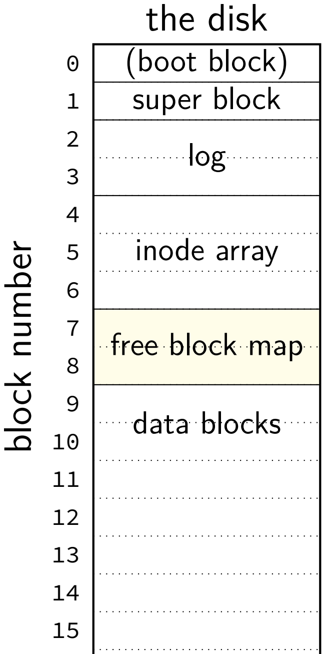
```
struct dinode {
   short type;  // File type
     // T_DIR, T_FILE, T_DEV

   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

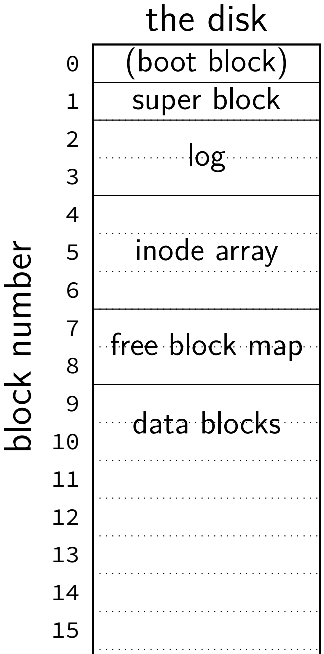
xv6 disk layout



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*

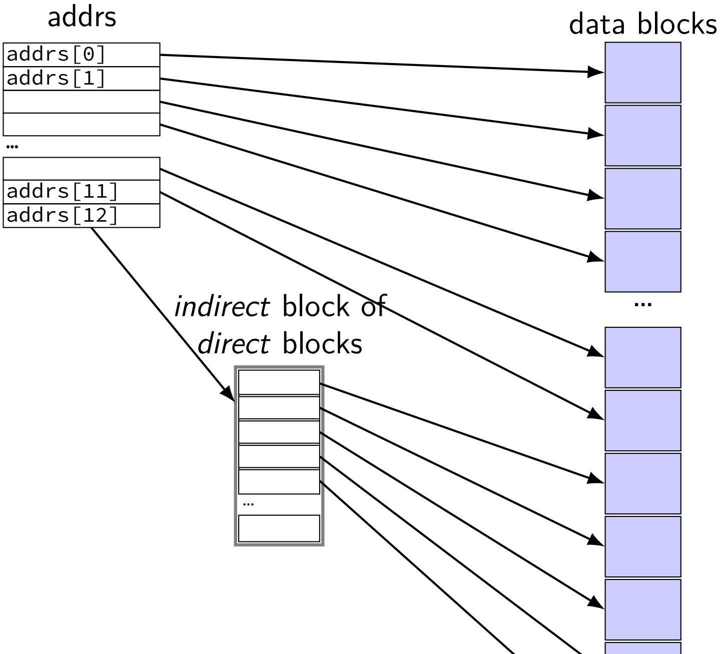
- multiple hard links to file allowed!

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

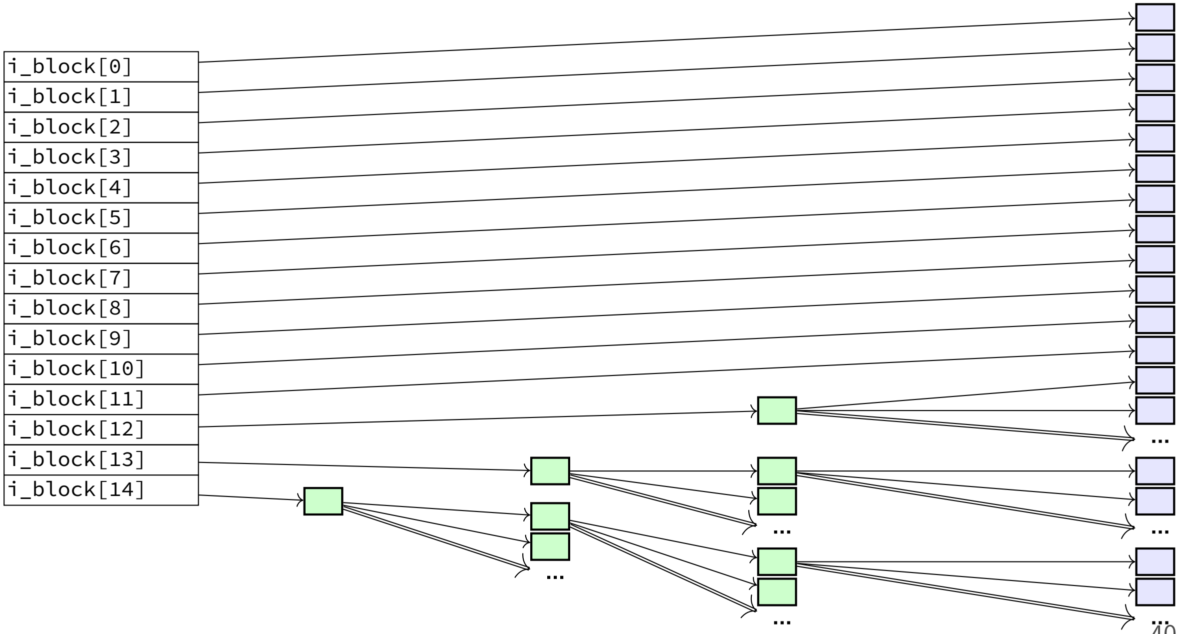
whole bunch of times

Linux ext2 inode

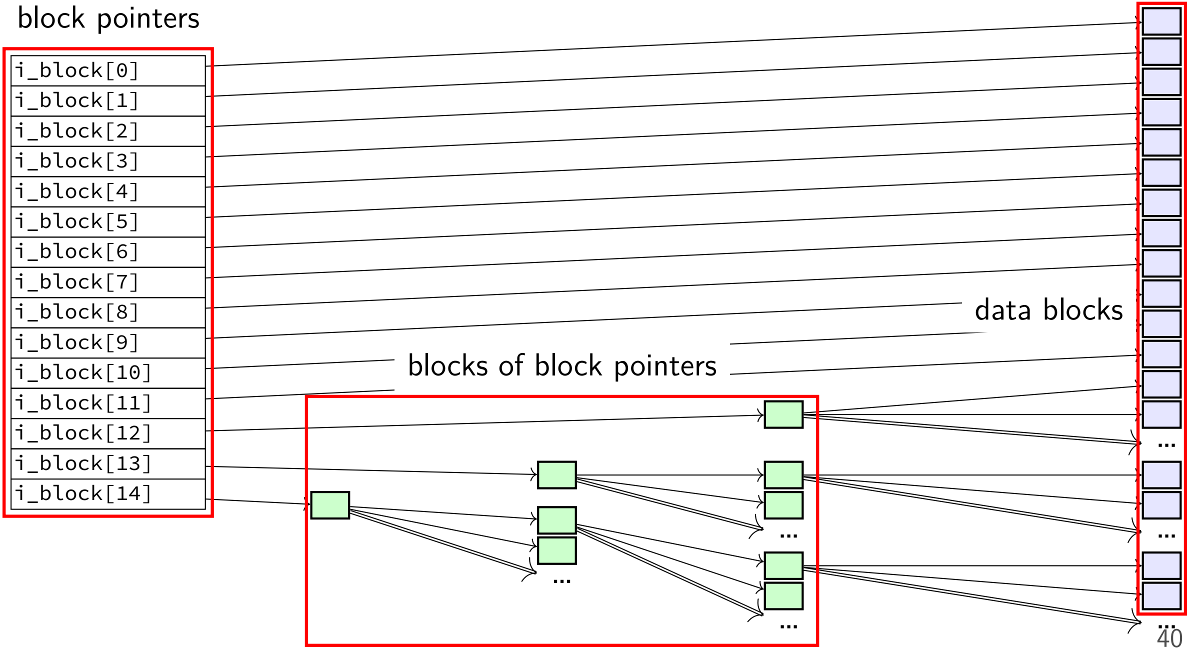
```
struct ext2_inode {
    __le16 i_mod
    __le16 i_uid
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16 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 */
    ...
};
```

similar pointers like xv6 FS — but more indirection

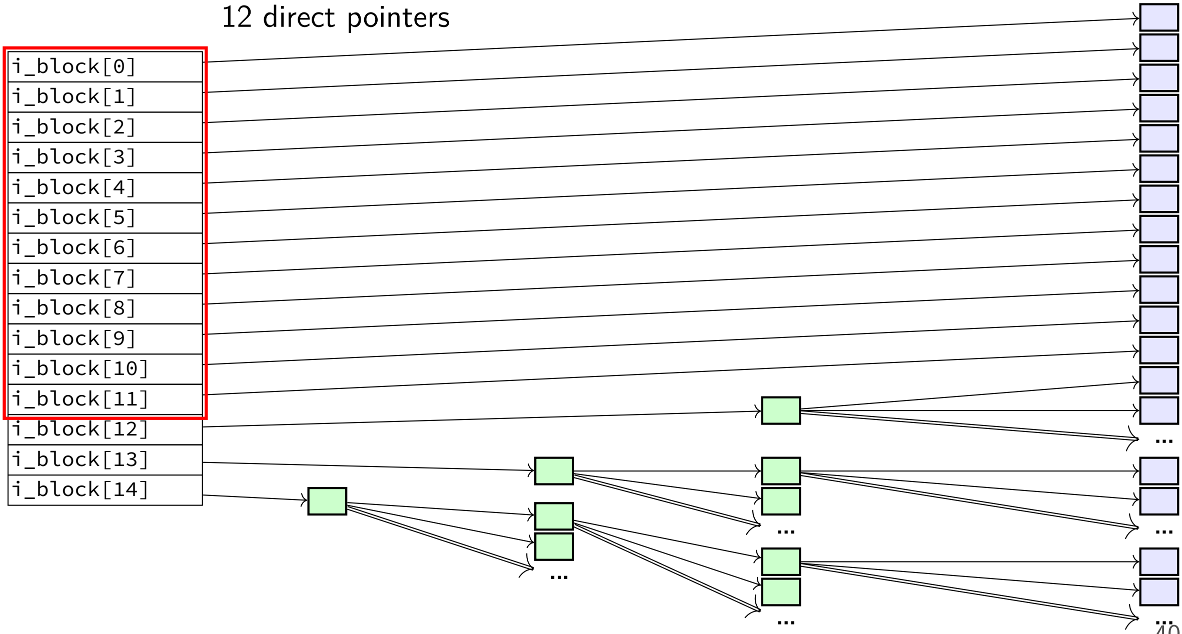
double/triple indirect



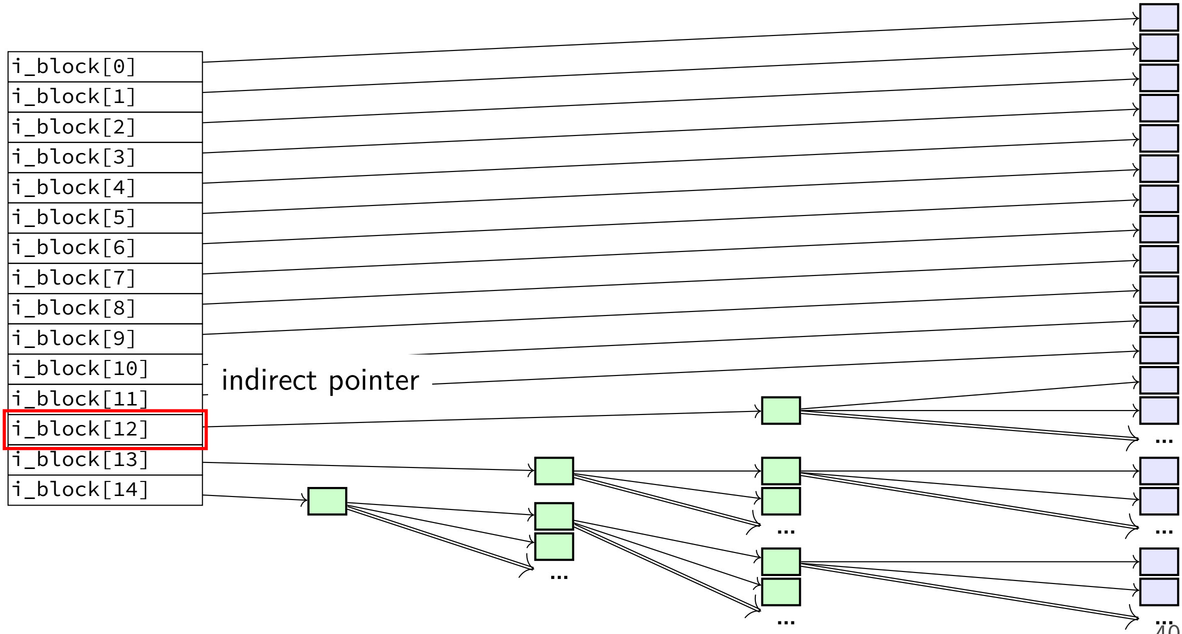
double/triple indirect



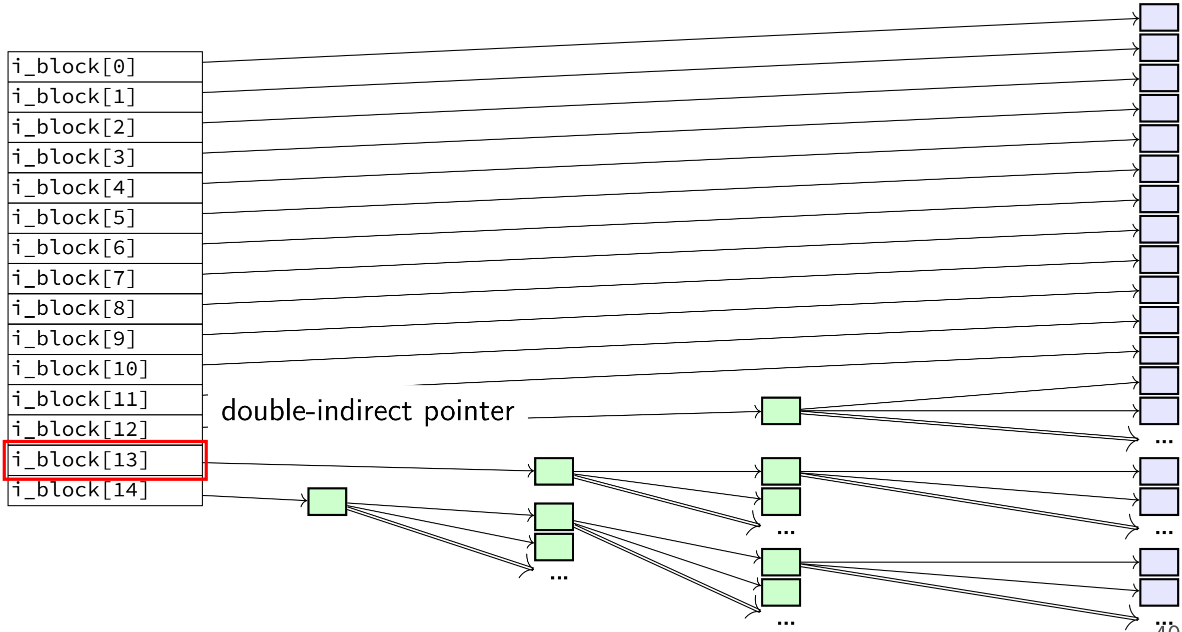
double/triple indirect



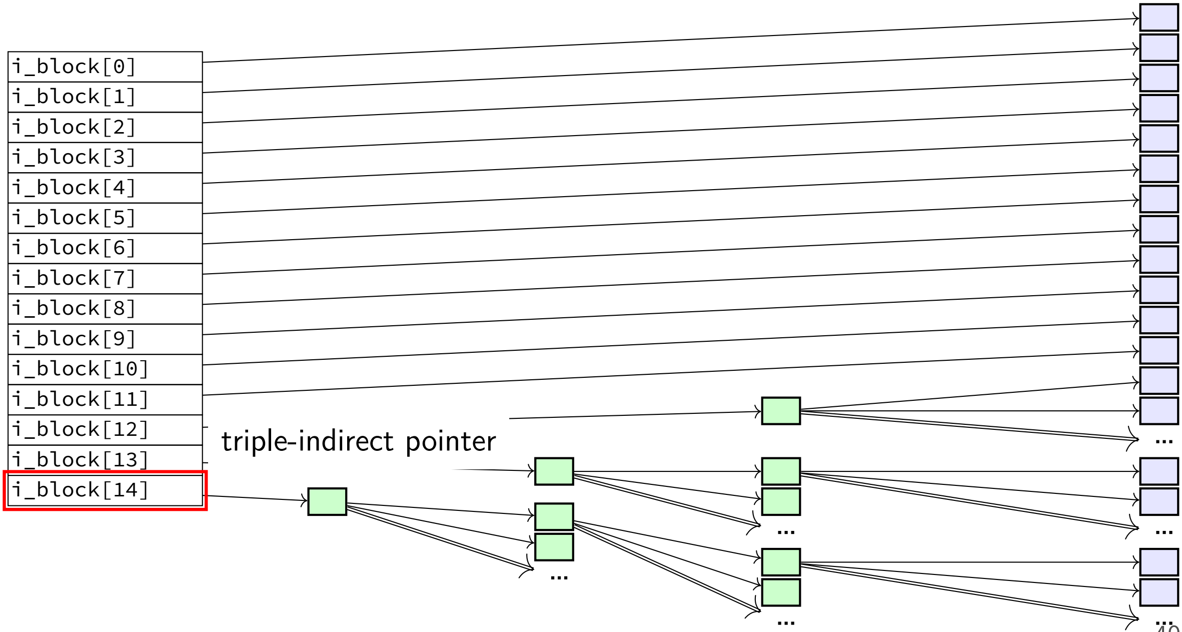
double/triple indirect



double/triple indirect



double/triple indirect



ext2 indirect blocks (1)

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 (1)

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 (solution)

12 direct pointers: first 1K (block size) $\times$ 12 bytes of data

1 indirect pointer:

- points to block with 1K (block size)/4 byte (pointer size) $=$ 256 pointers
- 256 pointers point to 1K blocks
- next 256KB of data

1 double indirect pointer

- points to block with 1K (block size)/4 byte (pointer size) $=$ 256 pointers
- 256 pointers point to pointers that each are like an indirect pointer
- 256KB per indirect pointer $\rightarrow$ next $256 \cdot 256$ KB of data

1 triple indiret

- next $256 \cdot 256 \cdot 256$ KB of data

total size: $12 + 256 + 256^2 + 256^3$ KB $=$ 16843020 KB $\approx$ 16GB

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?

ext2 indirect blocks (2) (solution)

byte 2^{15} = 32KB into file

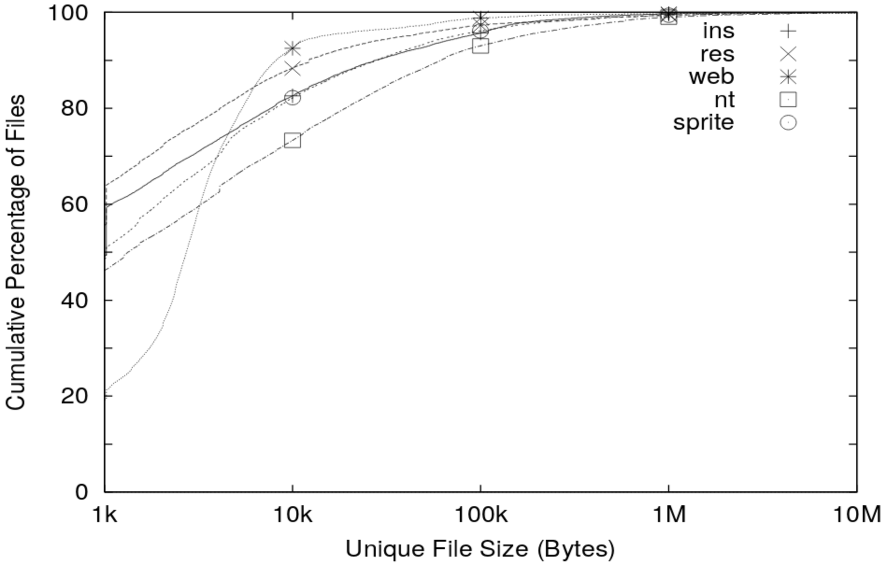
12 direct pointers: first 1K (block size) $\times$ 12 bytes of data

1 indirect pointer:

- points to block with 1K (block size)/4 byte (pointer size) = 256 pointers
- 256 pointers point to 1K blocks
- next 256KB of data

going to be (32 - 12)th element

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
- reason to want small block sizes
- sometimes other optimizations for small files

doens't mean large files are unimportant

- still take up most of the space
- biggest performance problems
- reason to want large block sizes?

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

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

seeking with extents

challenge: finding byte X of the file

with block pointers: can compute index

with extents: need to scan list?

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/SSD stores checksum for each sector

write interrupted? — checksum mismatch

hard drive/SSD returns read error

reliability issues

is the filesystem in a consistent state?

- do we know what blocks are free?
- do we know what files exist?
- is the data for files actually what was written?

also important topics, but won't spend much time on these:

what data will I lose if storage fails?

- mirroring, erasure coding (e.g. RAID) — using multiple storage devices
- idea: if one storage device fails, other(s) still have data

what data will I lose if I make a mistake?

- filesystem can store *multiple versions*
- "snapshots" of what was previously there

several bad options (1)

suppose we're moving a file from one directory to another on xv6

steps:

A: write new directory entry

B: overwrite (remove) old directory entry

several bad options (1)

suppose we're moving a file from one directory to another on xv6

steps:

A: write new directory entry

B: overwrite (remove) old directory entry

if we do A before B and crash happens after A:

- can have extra pointer of file
- problem: if old directory entry removed later, will get confused and free the file!

several bad options (1)

suppose we're moving a file from one directory to another on xv6

steps:

A: write new directory entry

B: overwrite (remove) old directory entry

if we do A before B and crash happens after A:

- can have extra pointer of file
- problem: if old directory entry removed later, will get confused and free the file!

if we do B before A and crash happens after B:

- the file disappeared entirely!

beyond ordering

recall: updating a sector is atomic

- happens entirely or doesn't

can we make filesystem updates work this way?

beyond ordering

recall: updating a sector is atomic

happens entirely or doesn't

can we make filesystem updates work this way?

yes — 'just' make updating one sector do the update

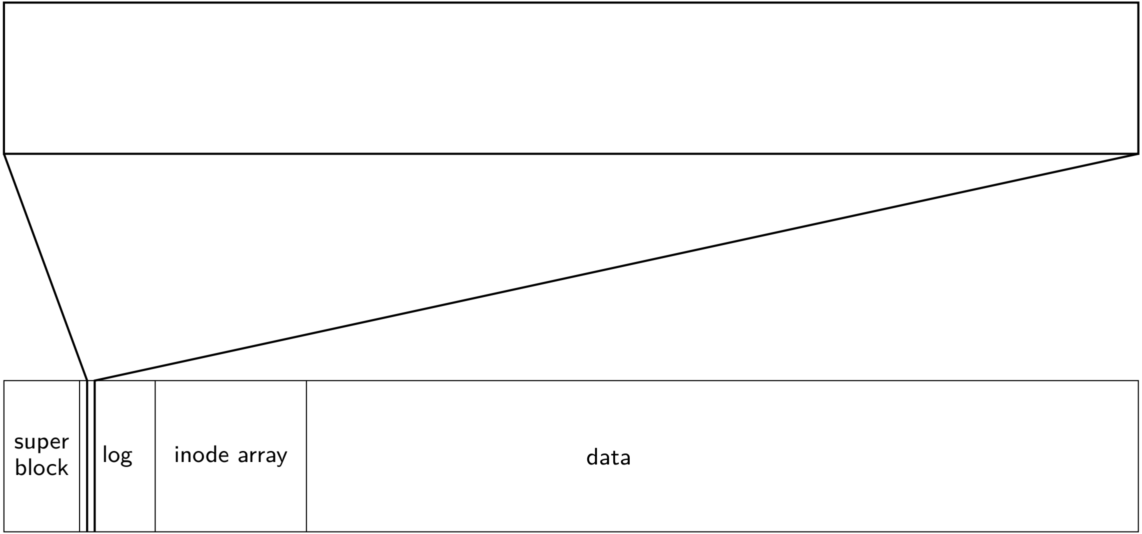
concept: transaction

transaction: bunch of updates that happen all at once

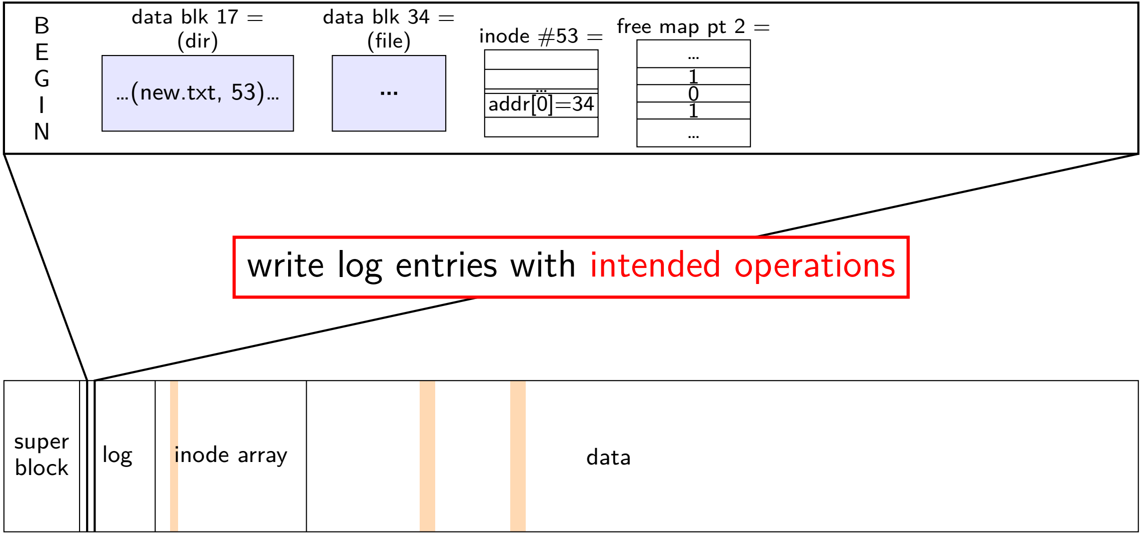
implementation trick: one update means transaction "commits"

- update done — whole transaction happened
- update not done — whole transaction did not happen

redo logging: file creation

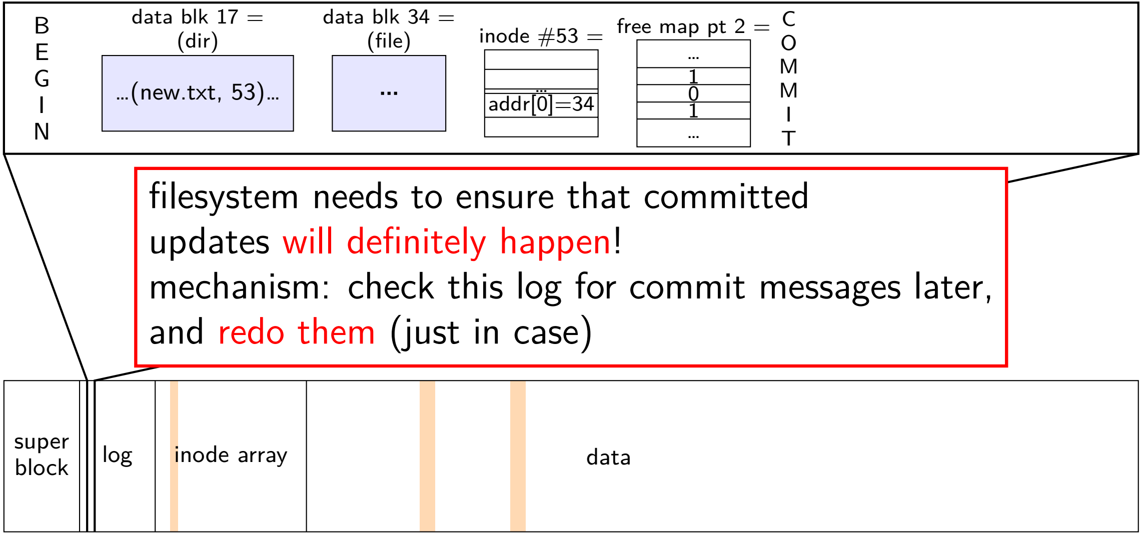


redo logging: file creation



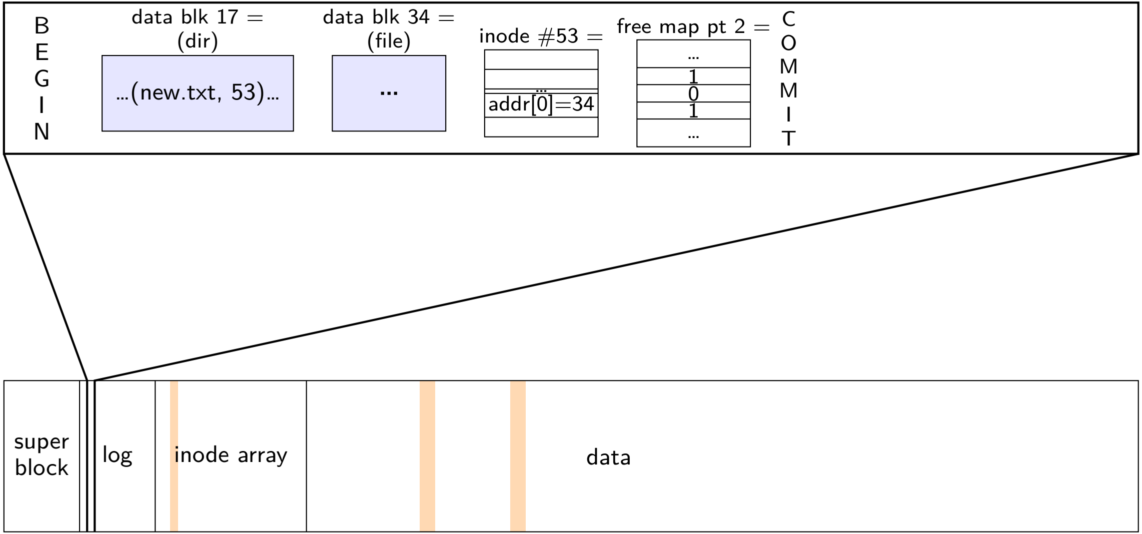
write log entries with intended operations

redo logging: file creation

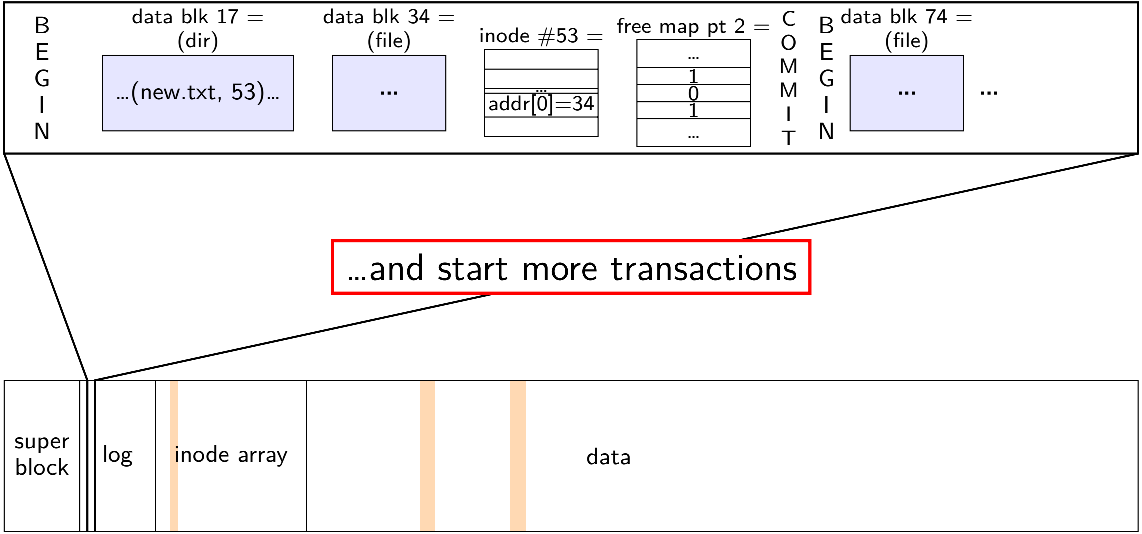


filesystem needs to ensure that committed updates will definitely happen!
mechanism: check this log for commit messages later, and redo them (just in case)

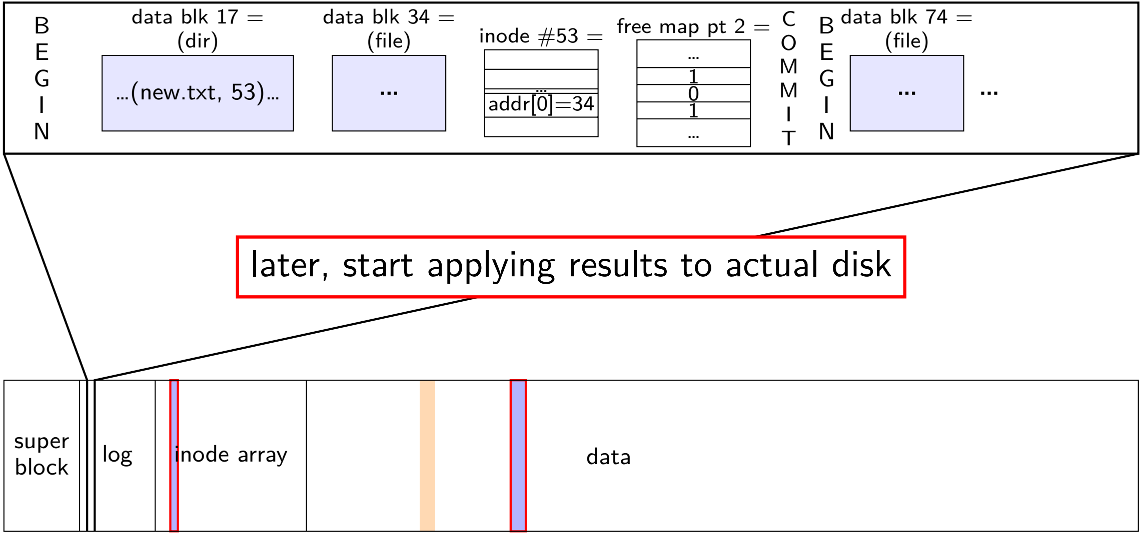
redo logging: file creation



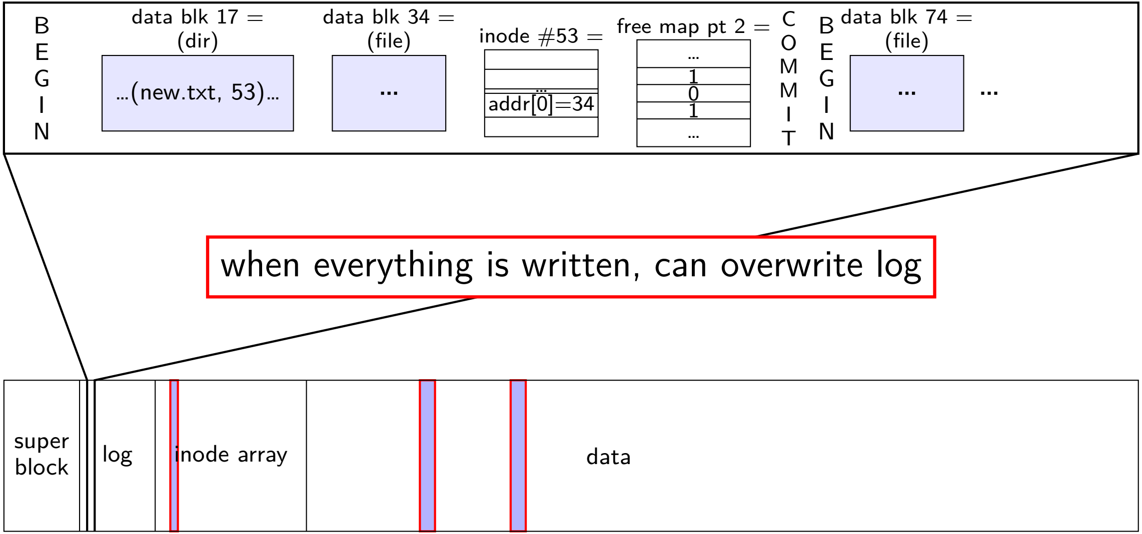
redo logging: file creation



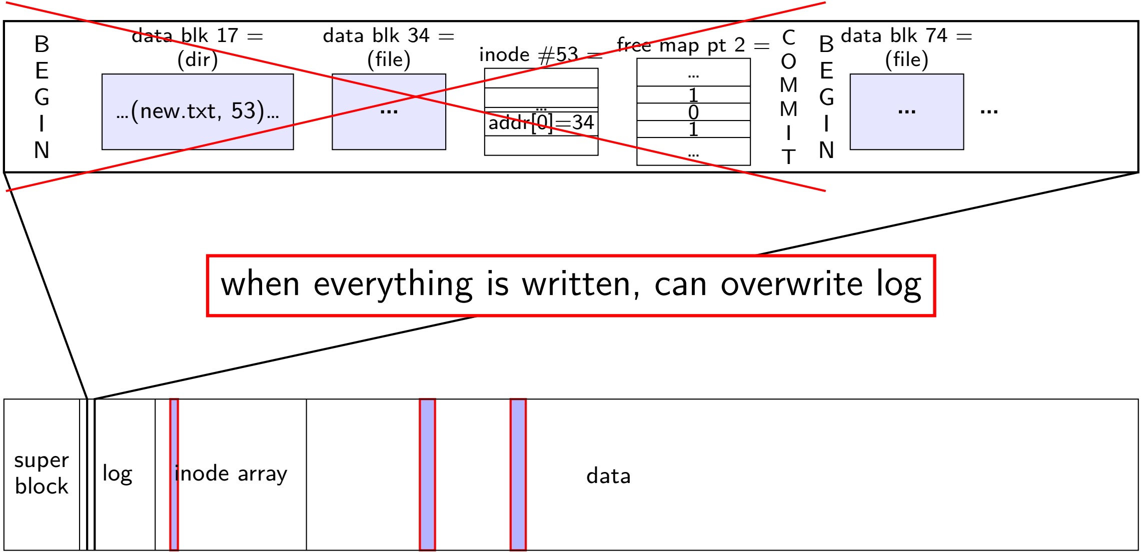
redo logging: file creation



redo logging: file creation



redo logging: file creation



redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

redo logging: file creation

normal operation

write to log transaction steps:

data blocks to create

direcotry entry, inode to write

directory inode (size, time)

update

write to log "commit transaction"

in any order:

update file data blocks

update directory entry

update file inode

update directory inode

reclaim space in log

"garbage collection"

crash before *commit*?

file not created

no partial operation to real data

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

crash after *commit*?

file created

promise: will perform logged updates

(after system reboots/recovers)

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log “commit transaction”

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- “garbage collection”

redo logging: file creation

normal operation

write to log transaction steps:
- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:
- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log
- "garbage collection"

recovery

read log and...

ignore any operation with no "commit"

redo any operation with "commit"
- already done? — okay, setting inode twice

reclaim space in log

idempotency

logged operations should be *okay to do twice* = *idempotent*

good example: set inode link count to 4

bad example: increment inode link count

good example: overwrite inode number X with new value

as long as last committed inode value in log is right...

bad example: allocate new inode with particular contents

good example: overwrite data block with new value

bad example: append data to last used block of file

redo logging summary

write intended operation to the log

- before ever touching 'real' data
- in format that's safe to do twice

write marker to commit to the log

- if exists, the operation *will be done eventually*

actually update the real data

redo logging and filesystems

filesystems that do redo logging are called *journalling filesystems*

exercise (1)

suppose OS performing operation of appending 100KB to a 100KB file X in directory Y and uses redo logging, ext2-like filesystem with 1KB blocks, 4B block pointers

part 1: what's modified?

[A] free block map
[B] data blocks for file
[C] indirect blocks for file
[D] data blocks for directory
[E] inode for file
[F] inode for directory
[G] the log

exercise (2)

suppose OS performing operation of appending 100KB to a 100KB file X in directory Y and uses redo logging

part 2: crash happens after writing:

- log entries for entire operation
- free block map changes
- indirect blocks for file

…what is written after restart as part of this operation?

- [A] free block map
- [B] data blocks for file
- [C] indirect blocks for file
- [D] data blocks for directory
- [E] inode for file
- [F] inode for directory
- [G] the log

degrees of consistency

not all journalling filesystem use redo logging for everything

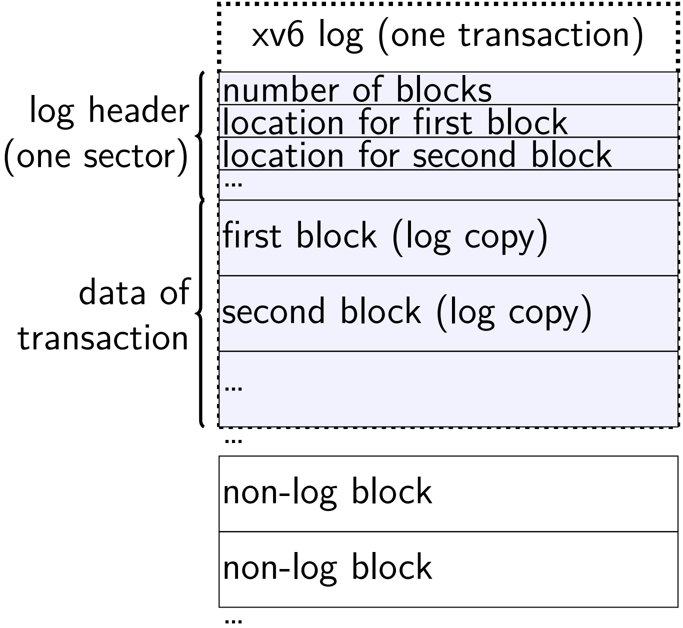
some use it *only for metadata operations*

some use it *for both metadata and user data*

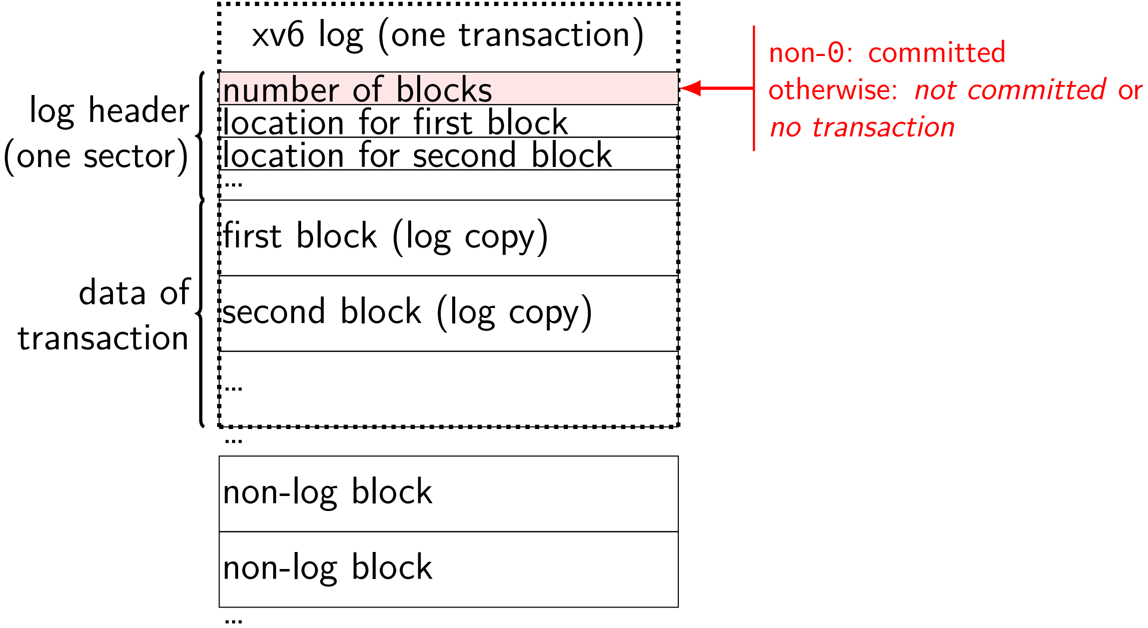
only metadata: avoids lots of duplicate writing

metadata+user data: integrity of user data guaranteed

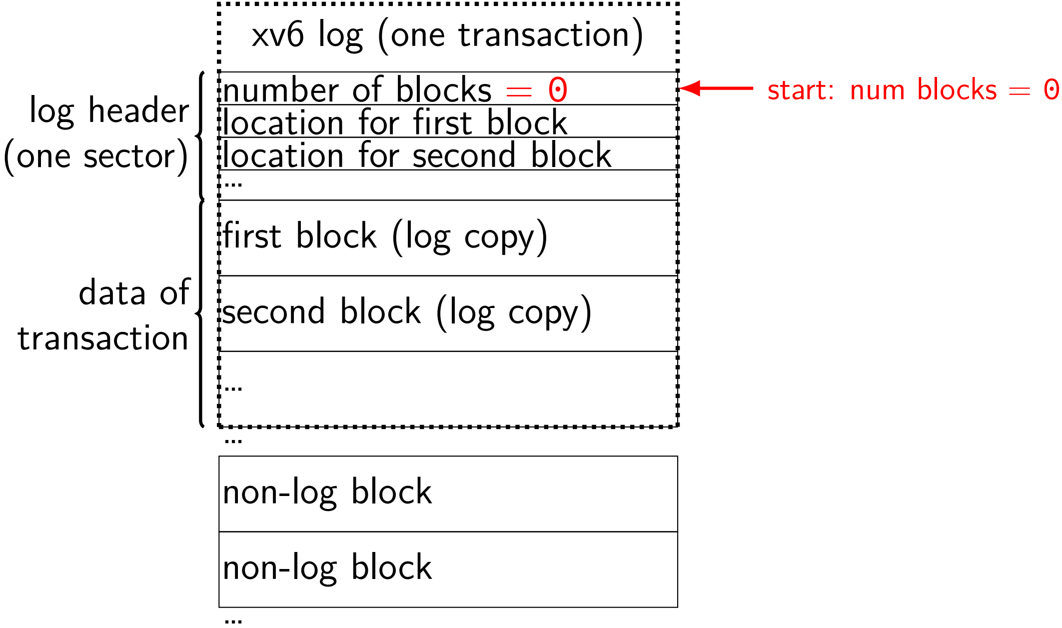
the xv6 journal



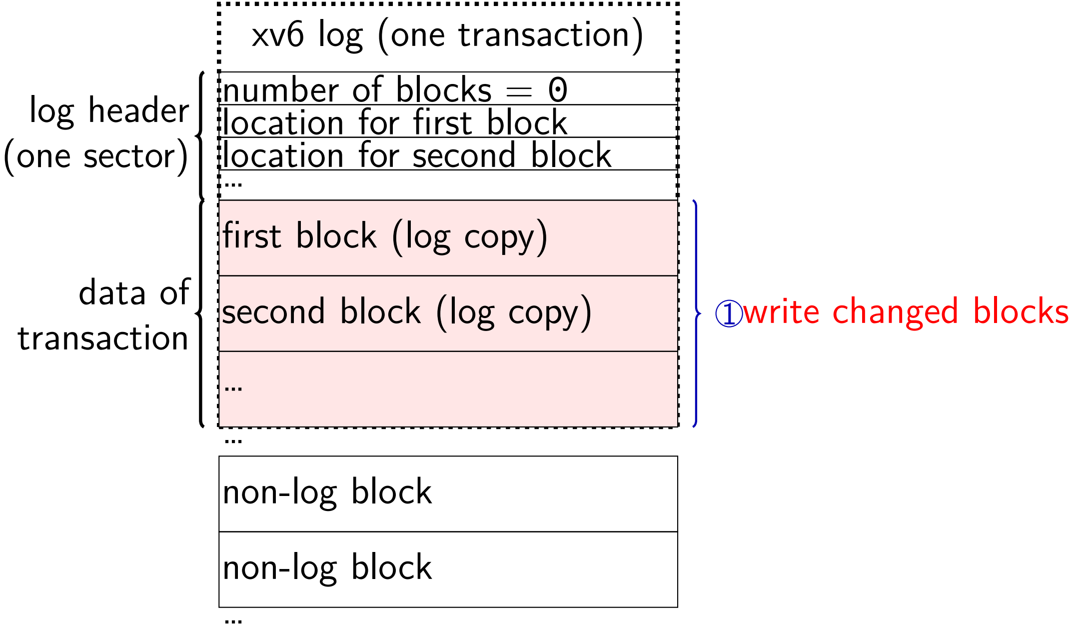
the xv6 journal



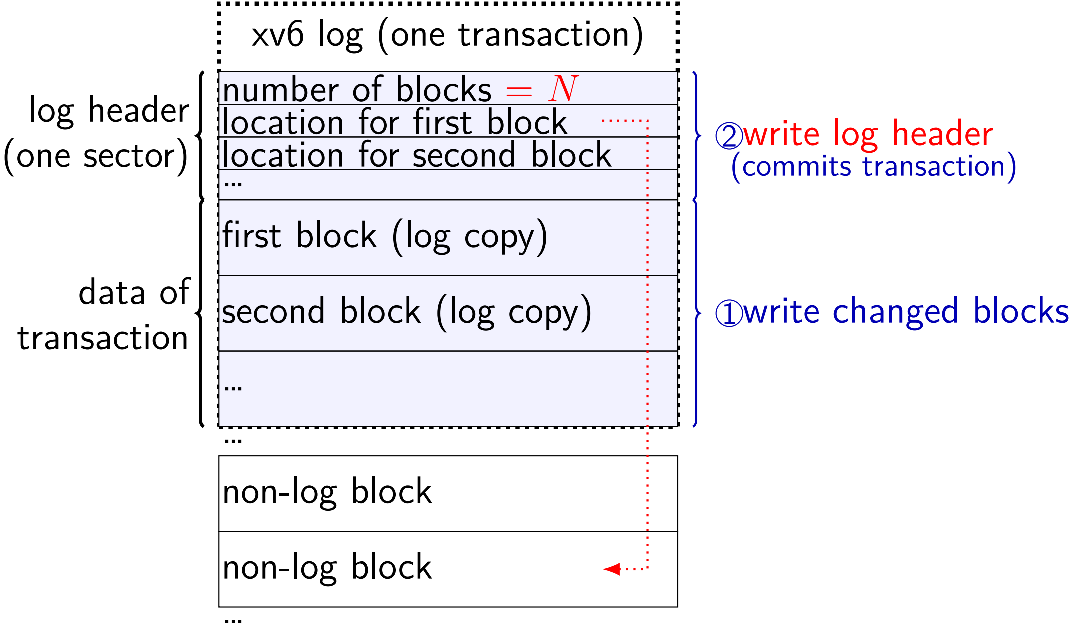
the xv6 journal



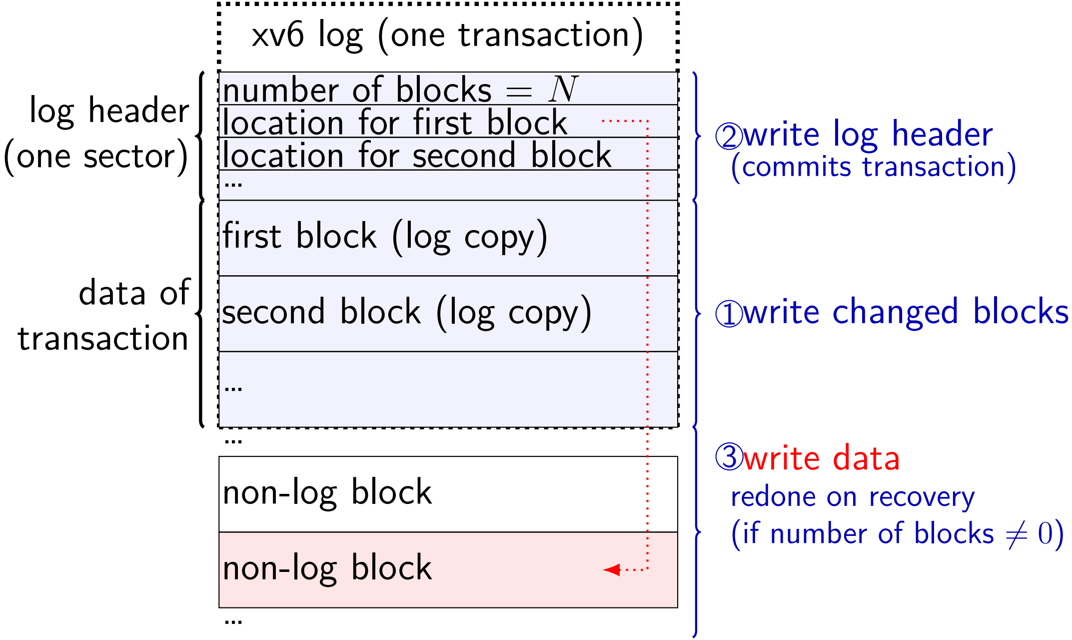
the xv6 journal



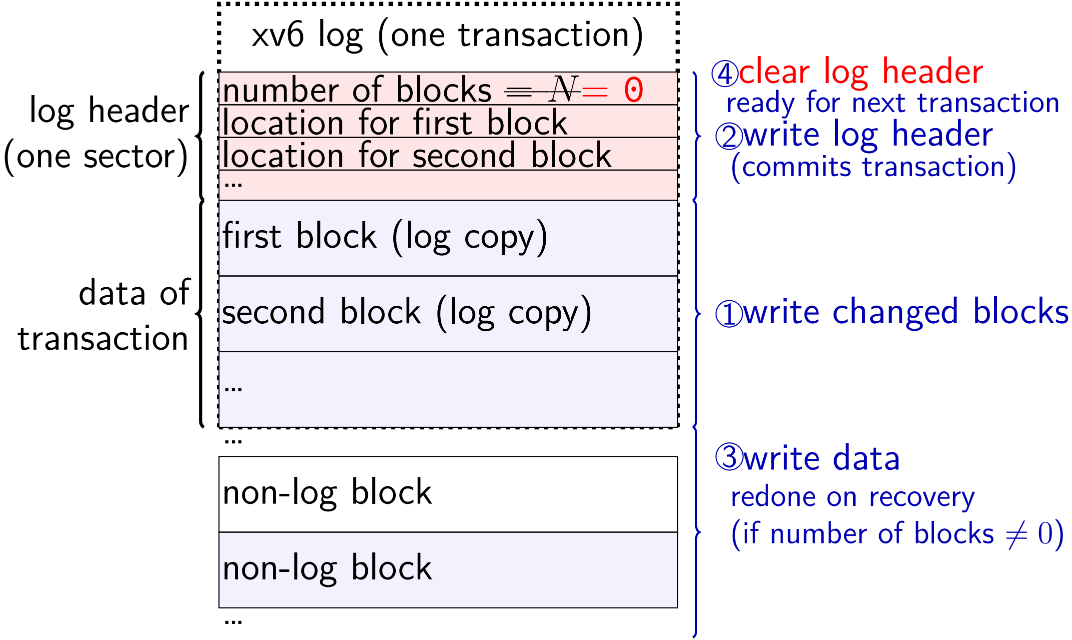
the xv6 journal



the xv6 journal



the xv6 journal



what is a transaction?

so far: each file update?

faster to do batch of updates together

- one log write finishes lots of things
- don't wait to write

xv6 solution: combine lots of updates into one transaction

only commit when...

- no active file operation, *or*
- not enough room left in log for more operations

what is a transaction?

so far: each file update?

faster to do batch of updates together

- one log write finishes lots of things
- don't wait to write

xv6 solution: combine lots of updates into one transaction

only commit when…

- no active file operation, *or*
- not enough room left in log for more operations

mounting filesystems

Unix-like system

root filesystem appears as /

other filesystems *appear as directory*

e.g. lab machines: my home dir is in filesystem at /net/zf15

directories that are filesystems look like normal directories

`/net/zf15/..` is `/net` (even though in different filesystems)

mounts on a dept. machine

```
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
...
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
...
/dev/sda3 on /localtmp type ext4 (rw)
...
zfs1:/zf2 on /net/zf2 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                noacl,sloppy,addr=128.143.136.9)
zfs3:/zf19 on /net/zf19 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                  noacl,sloppy,addr=128.143.67.236)
zfs4:/sw on /net/sw type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                              noacl,sloppy,addr=128.143.136.9)
zfs3:/zf14 on /net/zf14 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                  noacl,sloppy,addr=128.143.67.236)
...
```

kernel FS abstractions

Linux: *virtual file system* API

object-oriented, based on FFS-style filesystem

to implement a filesystem, create object types for:

- superblock (represents “header”)
- inode (represents file)
- dentry (represents cached directory entry)
- file (represents *open file*)

common code handles directory traversal

- and caches directory traversals

common code handles file descriptors, etc.

backup slides

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 blocks (not storing inodes) is used to store a directory of 200 30464B ($29 \cdot 1024 + 256$ byte) files?

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

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

fragments

Linux FS: 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

beyond mirroring

mirroring seems to waste a lot of space

10 disks of data? mirroring $\rightarrow$ 20 disks

10 disks of data? how good can we do with 15 disks?

best possible: lose 5 disks, still okay

- can't do better or it wasn't really 10 disks of data

schemes that do this based on *erasure codes*

- erasure code: encode data in way that handles parts missing (being erased)

erasure code example

store 2 disks of data on 3 disks

recompute original 2 disks of data from any 2 of the 3 disks

extra disk of data: some formula based on the original disks

- common choice: bitwise XOR

common set of schemes like this: RAID

- Redundant Array of Independent Disks

snapshots

filesystem snapshots

idea: filesystem keeps old versions of files around

- accidental deletion? old version stil there
- eventually discard some old versions

can access *snapshot* of files at prior time

snapshots

filesystem snapshots

idea: filesystem keeps old versions of files around

- accidental deletion? old version stil there
- eventually discard some old versions

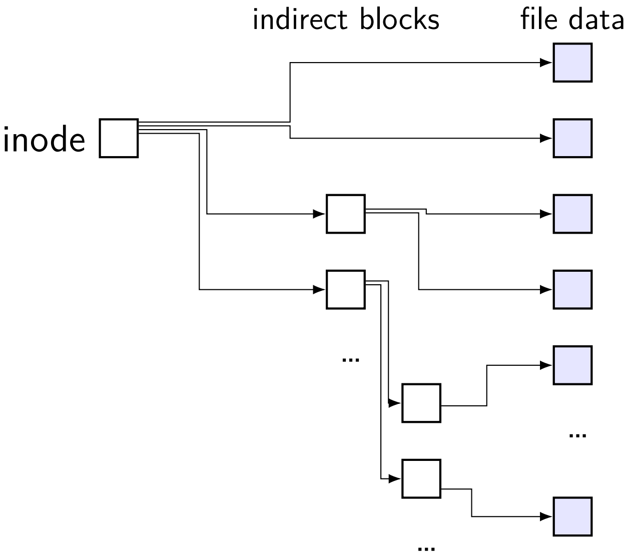
can access *snapshot* of files at prior time

mechanism: copy-on-write

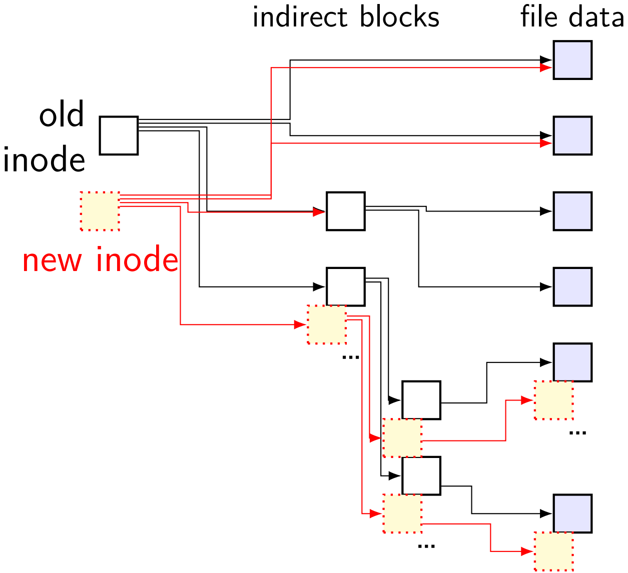
changing file makes new copy of filesystem

common parts shared between versions

inode and copy-on-write



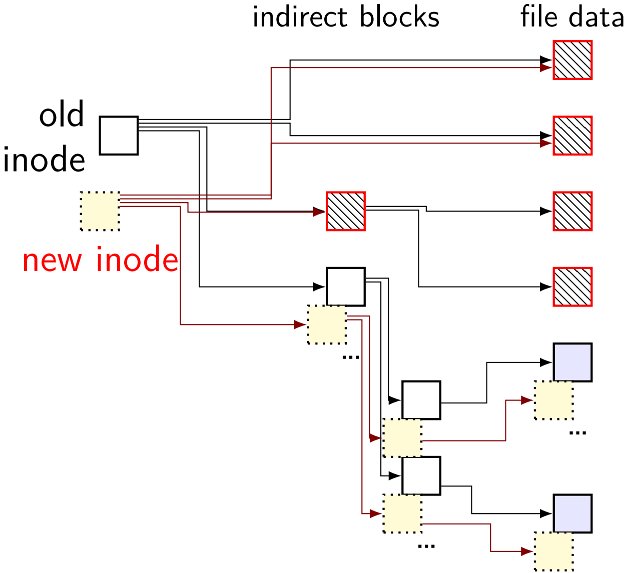
inode and copy-on-write



update: new data blocks
+ new indirect blocks
+ new inode

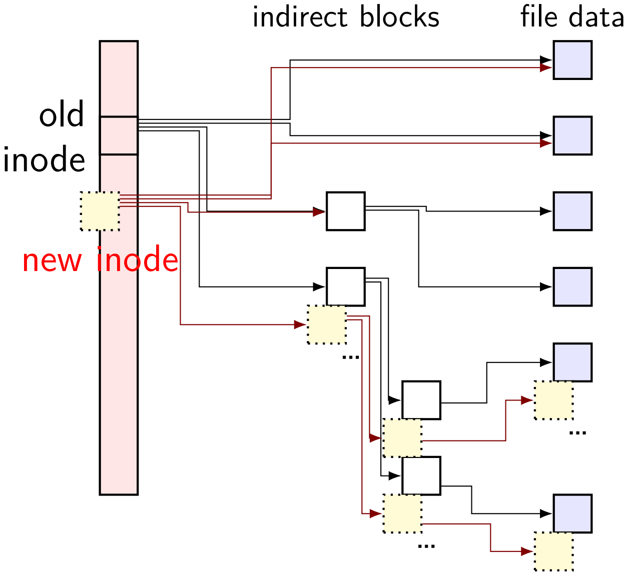
both old+new inode valid

inode and copy-on-write



unchanged parts of file shared

inode and copy-on-write

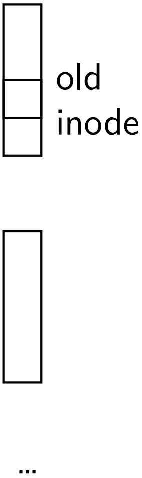


challenge: FFS/xv6/ext2 design has big array of inodes

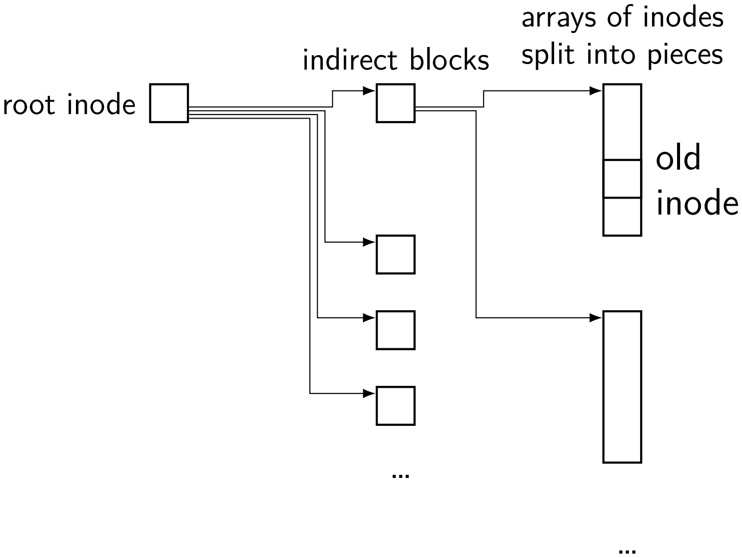
don't want to write new copy of *entire inode array*

extra indirection for inode array

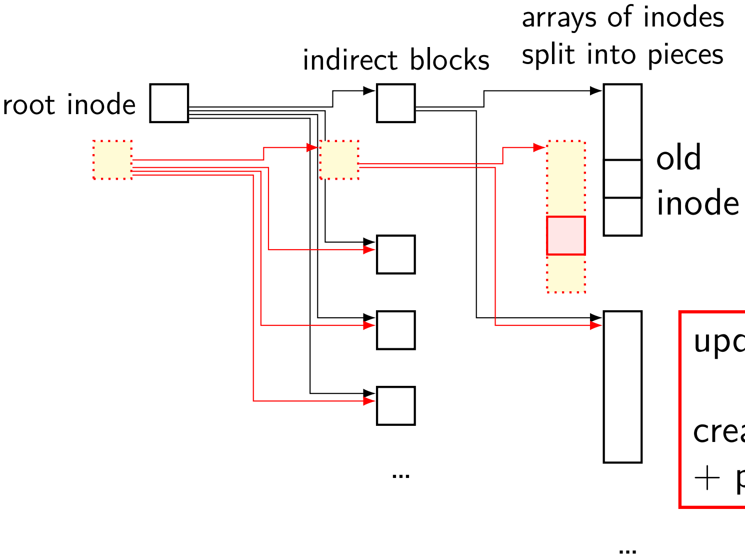
arrays of inodes
split into pieces



extra indirection for inode array



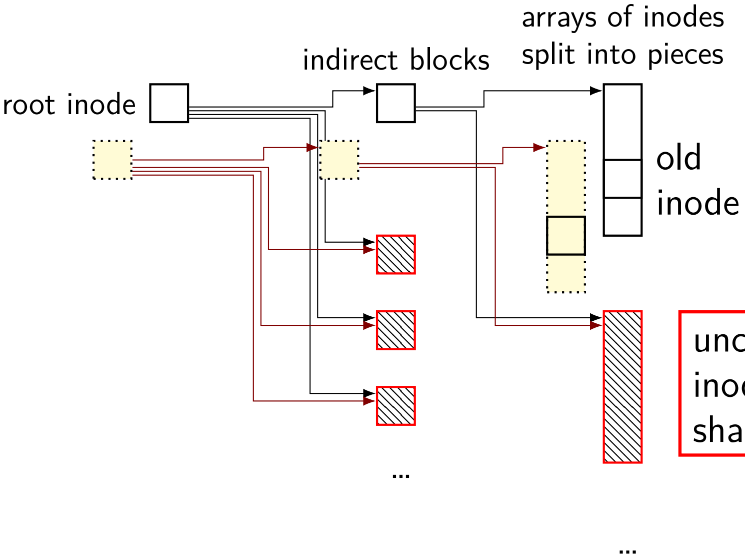
extra indirection for inode array



update one inode?

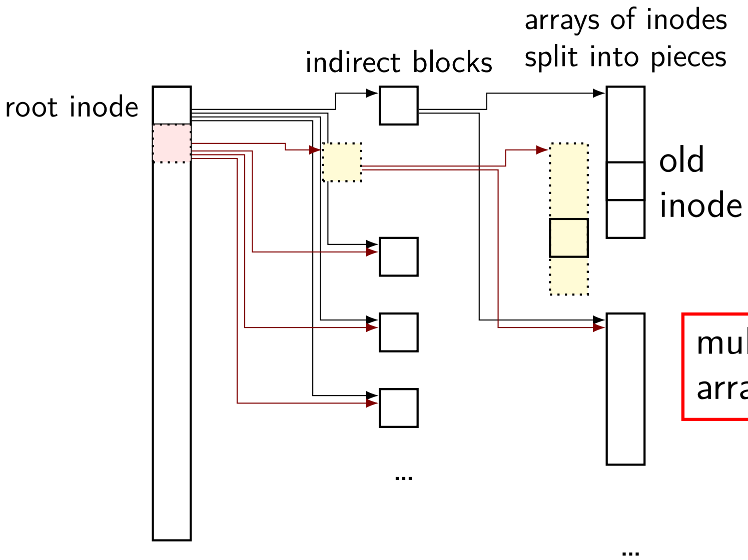
create new root inode
+ pointers

extra indirection for inode array



unchanged parts of
inode array
shared between versions

extra indirection for inode array



multiple snapshots?
array of root inodes

copy-on-write indirection

file update = replace with new version

array of versions of entire filesystem

only copy modified parts

- keep reference counts, like for paging assignment

lots of pointers — only change pointers where modifications happen

snapshots in practice

ZFS supports this (if turned on)

example: `.zfs/snapshots/11.11.18-06` pseudo-directory

contains contents of files at 11 November 2018 6AM

multiple copies

FAT: multiple copies of file allocation table and header

in inode-based filesystems: often multiple copies of 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

- extra copy of metadata that is important for all files
- but won't recover specific files/directories whose data was lost

aside: FAT date encoding

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

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

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

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

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

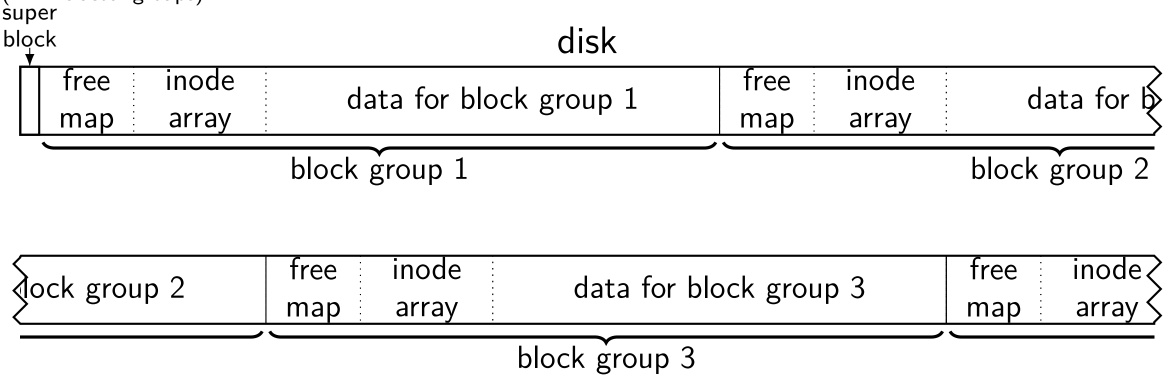
Linux's ext2 filesystem based on FFS

some other notable newer solutions (beyond what FFS/ext2 do)

- better handling of very large files
- avoiding linear directory searches

block groups

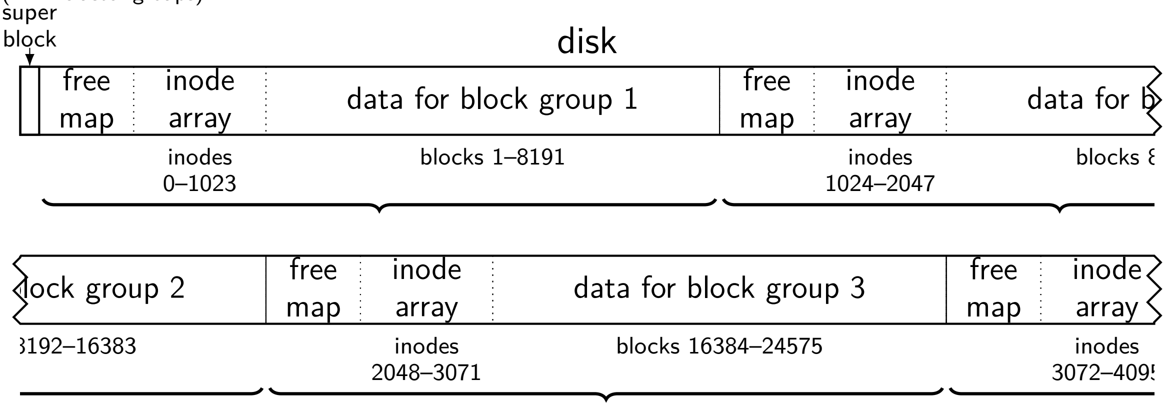
(AKA cluster groups)



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

block groups

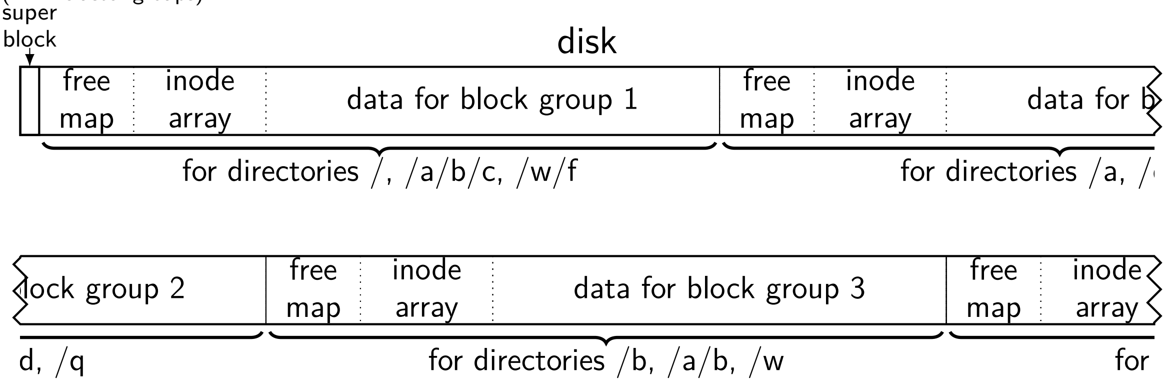
(AKA cluster groups)



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

block groups

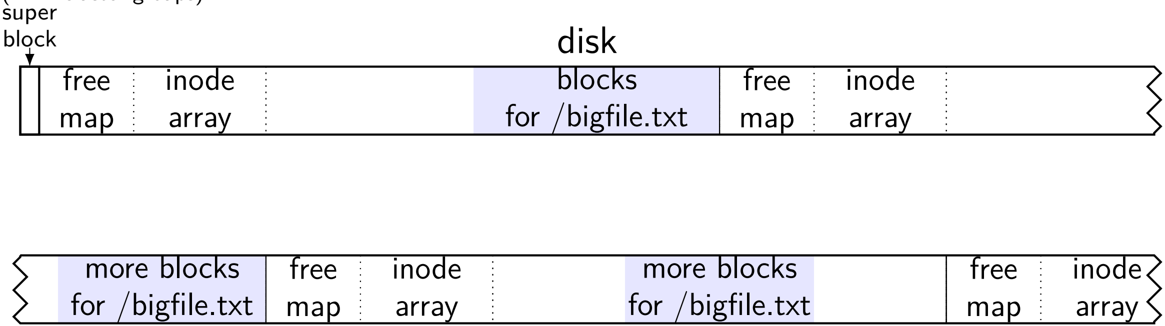
(AKA cluster groups)



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

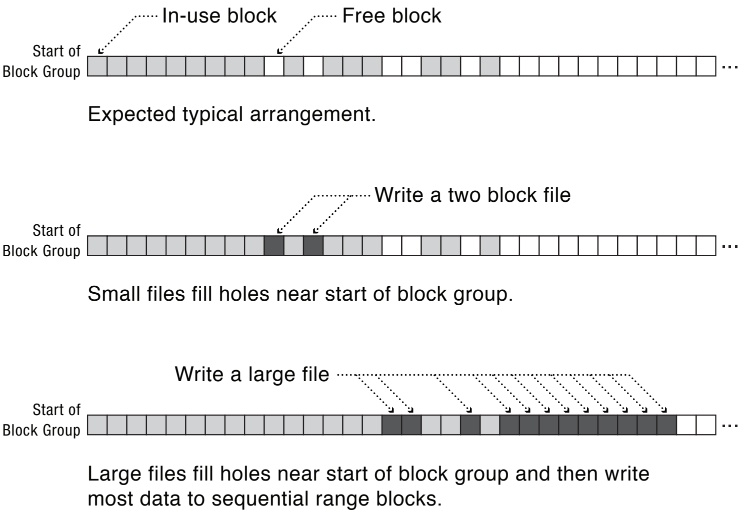
block groups

(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

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

if we do A before B+C and crash happens after A:

have blocks we can't use (not free), but which are unused

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

if we do A before B+C and crash happens after A:
- have blocks we can't use (not free), but which are unused

if we do B before A+C and crash happens after B:
- have inode we can't use (not free), but which is not really used

several bad options (2)

suppose we're creating a new file

A: mark blocks as used in free block map

B: write inode for file

C: write directory entry for file

if we do A before B+C and crash happens after A:
- have blocks we can't use (not free), but which are unused

if we do B before A+C and crash happens after B:
- have inode we can't use (not free), but which is not really used

if we do C before A+B and crash happens after C:
- have directory entry that points to junk — will behave weirdly

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

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

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

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

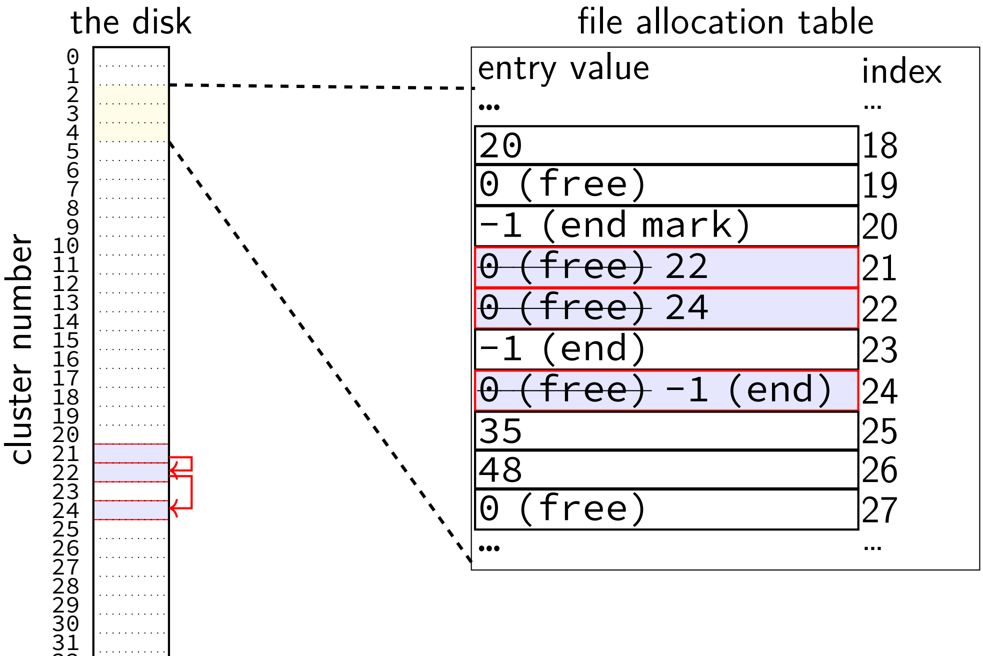
- xv6 finds *first free block/inode*
- result: files/directory entries scattered about

blocks are pretty small — needs lots of space for metadata

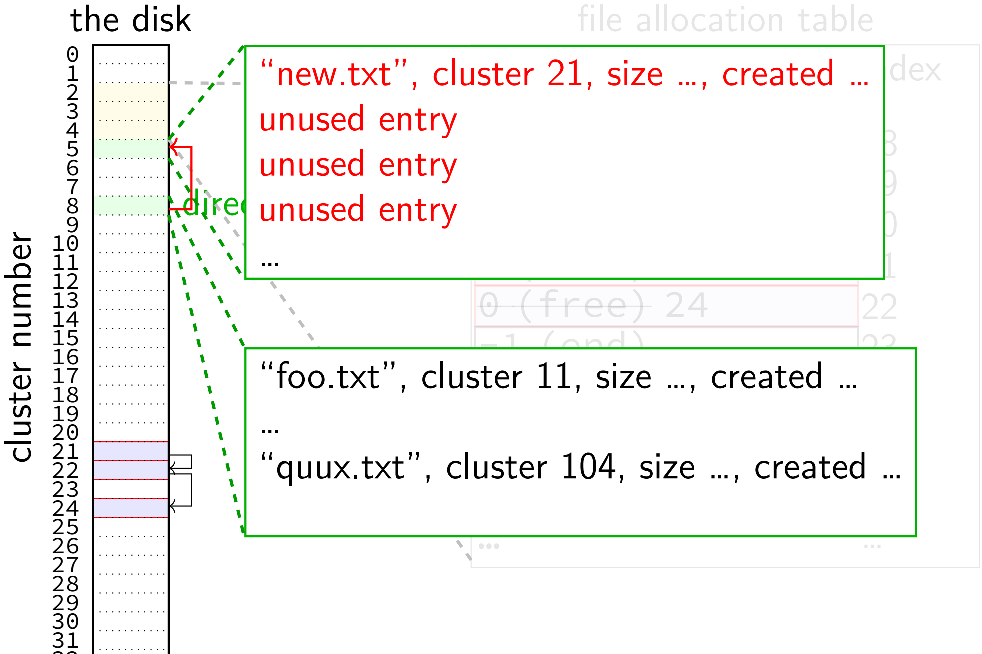
- could change size? but waste space for small files
- large files have giant lists of blocks

linear searches of directory entries to resolve paths

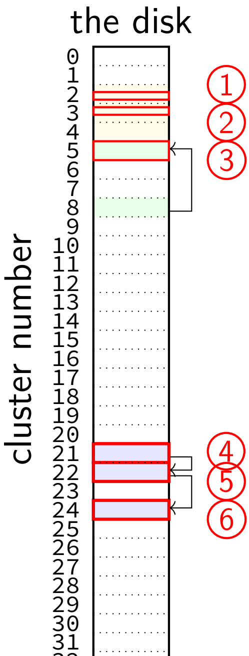
recall: FAT: file creation (1)



recall: FAT: file creation (2)



exercise: FAT file creation



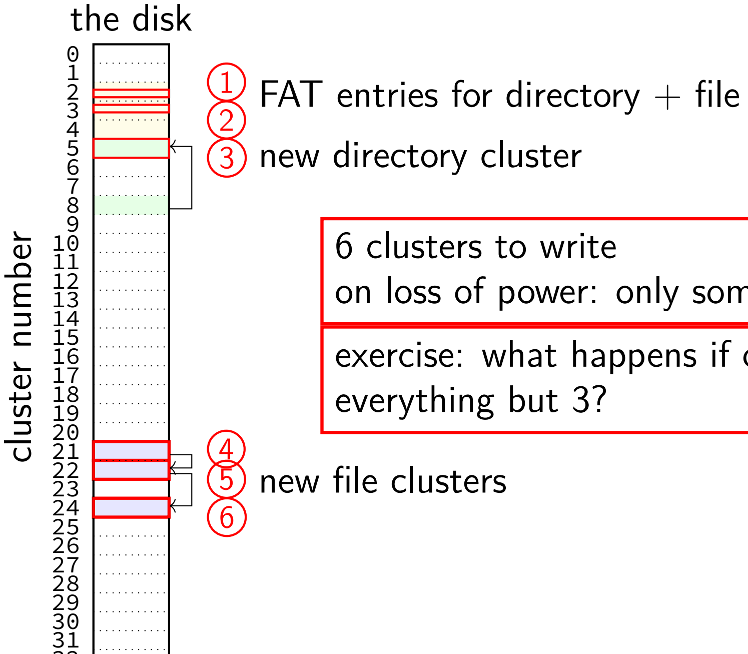
FAT entries for directory + file

new directory cluster

6 clusters to write
on loss of power: only some completed

new file clusters

exercise: FAT file creation



6 clusters to write

on loss of power: only some completed

exercise: what happens if only 1, 2 complete?

everything but 3?

exercise: FAT ordering

(creating a file that needs new cluster of direntries)

1. FAT entry for extra directory cluster
2. FAT entry for new file clusters
3. file clusters
4. file's directory entry (in new directory cluster)

what ordering is best if a crash happens in the middle?

A. 1, 2, 3, 4
B. 4, 3, 1, 2
C. 1, 3, 4, 2
D. 3, 4, 2, 1
E. 3, 1, 4, 2

exercise: xv6 FS ordering

(creating a file that neeeds new block of direntries)

1. free block map for new directory block
2. free block map for new file block
3. directory inode
4. new file inode
5. new directory entry for file (in new directory block)
6. file data blocks

what ordering is best if a crash happens in the middle?

A. 1, 2, 3, 4, 5, 6
B. 6, 5, 4, 3, 2, 1
C. 1, 2, 6, 5, 4, 3
D. 2, 6, 4, 1, 5, 3
E. 3, 4, 1, 2, 5, 6

inode-based FS: careful ordering

mark blocks as allocated before referring to them from directories

write data blocks before writing pointers to them from inodes

write inodes before directory entries pointing to it

remove inode from directory before marking inode as free

  or decreasing link count, if there's another hard link

idea: better to waste space than point to bad data

recovery with careful ordering

avoiding data loss $\rightarrow$ can 'fix' inconsistencies

programs like `fsck` (filesystem check), `chkdsk` (check disk)

run manually or periodically or after abnormal shutdown

inode-based FS: creating a file

normal operation

- allocate data block
- write data block
- update free block map
- update file inode
- update directory entry
  - filename+inode number
- update direcotry inode
  - modification time

inode-based FS: creating a file

normal operation

allocate data block

write data block

update free block map

update file inode

update directory entry
filename+inode number

update direcotry inode
modification time

general rule:
better to waste space
than point to bad data

mark blocks/inodes used before writing

inode-based FS: creating a file

normal operation

- allocate data block
- write data block
- update free block map
- update file inode
- update directory entry
  - filename+inode number
- update direcotry inode
  - modification time

recovery (`fsck`)

- read all directory entries
- scan all inodes
- free unused inodes
  - unused = not in directory
- free unused data blocks
  - unused = not in inode lists
- scan directories for missing update/access times

inode-based FS: exercise: unlink

what order to remove a hard link ($=$ directory entry) for file?

1. overwrite directroy entry for file
2. decrement link count in inode (but link count still > 1 so don't remove)

assume not the last hard link

inode-based FS: exercise: unlink

what order to remove a hard link ($=$ directory entry) for file?

1. overwrite directroy entry for file
2. decrement link count in inode (but link count still > 1 so don't remove)

assume not the last hard link

what does recovery operation do?

inode-based FS: exercise: unlink last

what order to remove a hard link (= directory entry) for file?

1. overwrite last directroy entry for file
2. mark inode as free (link count = 0 now)
3. mark inode's data blocks as free

assume is the last hard link

inode-based FS: exercise: unlink last

what order to remove a hard link (= directory entry) for file?

1. overwrite last directroy entry for file
2. mark inode as free (link count = 0 now)
3. mark inode's data blocks as free

assume is the last hard link

what does recovery operation do?

fsck

Unix typically has an `fsck` utility

  Windows equivalent: `chkdsk`

checks for *filesystem consistency*

  is a data block marked as used that no inodes uses?

  is a data block referred to by two different inodes?

  is a inode marked as used that no directory references?

  is the link count for each inode = number of directories referencing it?

  ...

assuming careful ordering, can fix errors after a crash without loss

maybe can fix other errors, too

fsck costs

my desktop's filesystem:
2.4M used inodes; 379.9M of 472.4M used blocks

recall: check for data block marked as used that no inode uses:

- read blocks containing all of the 2.4M used inodes
- add each block pointer to a list of used blocks
- if they have indirect block pointers, read those blocks, too
- get list of all used blocks (via direct or indirect pointers)
- compare list of used blocks to actual free block bitmap

pretty expensive and slow

running fsck automatically

common to have "clean" bit in superblock

last thing written (to set) on shutdown

first thing written (to clear) on startup

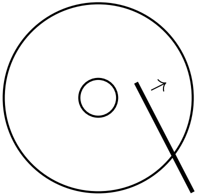
on boot: if clean bit clear, run fsck first

ordering and disk performance

recall: seek times

would like to order writes based on locations on disk

- write many things in one pass of disk head
- write many things in cylinder in one rotation

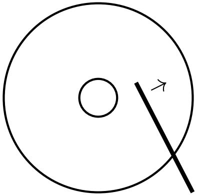


ordering and disk performance

recall: seek times

would like to order writes based on locations on disk

- write many things in one pass of disk head
- write many things in cylinder in one rotation



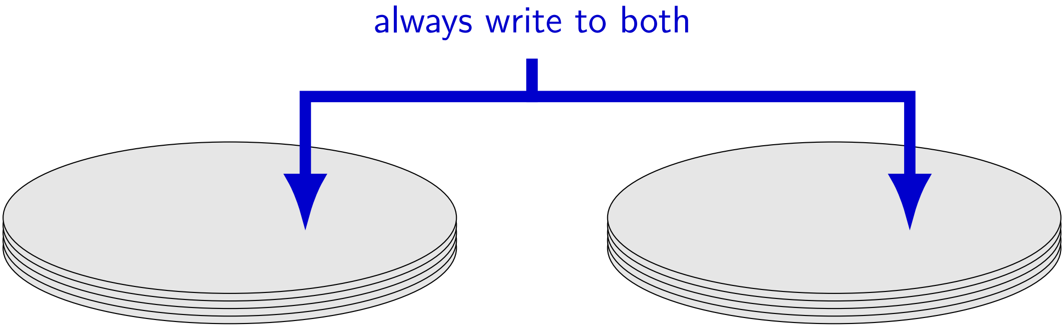
ordering constraints make this hard:

free block map for file (start), then file blocks (middle), then…

file inode (start), then directory (middle), …

mirroring whole disks

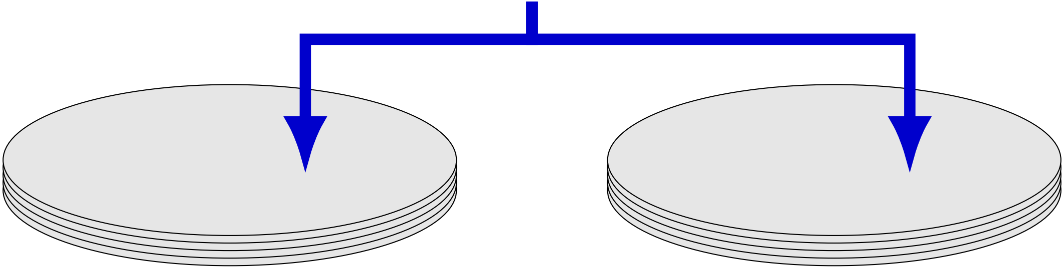
alternate strategy: write everything to two disks



mirroring whole disks

alternate strategy: write everything to two disks

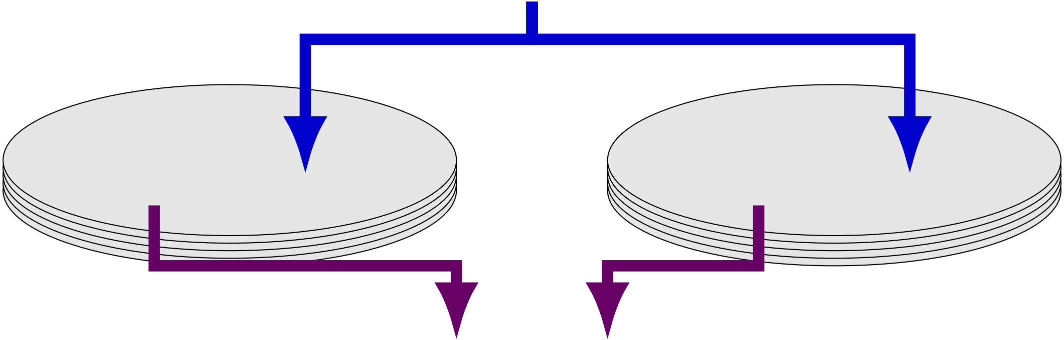
always write to both



mirroring whole disks

alternate strategy: write everything to two disks

always write to both



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

beyond mirroring

mirroring seems to waste a lot of space

10 disks of data? mirroring $\rightarrow$ 20 disks

10 disks of data? how good can we do with 15 disks?

best possible: lose 5 disks, still okay

- can't do better or it wasn't really 10 disks of data

schemes that do this based on *erasure codes*

- erasure code: encode data in way that handles parts missing (being erased)

erasure code example

store 2 disks of data on 3 disks

recompute original 2 disks of data from any 2 of the 3 disks

extra disk of data: some formula based on the original disks

- common choice: bitwise XOR

common set of schemes like this: RAID

- Redundant Array of Independent Disks

exericse

filesystem has:

- root directory with 2 subdirectories
- each subdirectory contains 3 512B files, 2 4MB files
- (1MB = 1024KB; 1KB = 1024B)
- 32B directory entries
- 4B block pointers
- 4KB blocks
- inode: 12 direct pointers, 1 indirect pointer, 1 double-indirect, 1 triple-indirect

(a) how many inodes used?

(b) how many blocks (outside of inodes) with 1KB fragments? [minimum w/partial blocks]

(c) how many blocks (outside of inodes) with block pointers replaced by 8B extents (no fragments)? [compute minimum]

inodes used

per each of 2 subdirectories: 5 files $+$ 1 inode for subdirectory $= 6$

plus 1 for root directory itself

$= 12 + 1 = 13$

blocks with fragments

each of 6 512B files uses a single 1KB fragment

wastes 512Bs of it

each of 2 subdirectory needs $32\text{B} \cdot 5 \ll 1\text{KB}$ (1 fragment)

(5 directory entries; probably also additional entries for `..`)

root directory needs $32\text{B} \cdot 2 \ll 1\text{KB}$ (1 fragment)

9 1KB fragments $\rightarrow$ minimum 3 (4KB) blocks

each of 4 4MB file uses 1024 data blocks

1 indirect block for blocks 13-(1024+13) [last 12 pointers unused]

$=$ 4096 blocks (4MB files data) $+$ 4 (4MB file indirects) $+$ 3 (for fragments)
$=$ 4103 blocks

blocks with extents

each of 6 512B files uses a single 4KB block

extent specifying block

each of 2 subdirectory needs 32B $\cdot$ 5 $\ll$ 4KB (1 block)

root directory needs 32B $\cdot$ 2 $\ll$ 4KB (1 block)

each of 2 4MB file uses 2048 data blocks

no indirect blocks assuming 2048 data blocks are contiguous (one extent in inode)

$=$ 4096 blocks (4MB files data) $+$ 6 (small files) $+$ 3 (directory entries) $=$ 4105 blocks

redo logging problems

doesn't the log get infinitely big?

writing everything twice?

redo logging problems

doesn't the log get infinitely big?

writing everything twice?

limiting log size

once transaction is written to real data, can discard

sometimes called "garbage collecting" the log

may sometimes need to block to free up log space

- perform logged updates before adding more to log

hope: usually log cleanup happens "in the background"

redo logging problems

doesn't the log get infinitely big?

writing everything twice?

lots of writing?

entire log can be written sequentially

- ideal for hard disk performance
- also pretty good for SSDs

no waiting for 'real' updates

- application can proceed while updates are happening
- files will be updated even if system crashes

often better for performance!

readahead implementation ideas?

which of these is probably best?

(a) when there's a page fault requring reading page X of a file from disk, read pages X and $X+1$

(b) when there's a page fault requring reading page $X > 200$ of a file from disk, read the rest of the file

(c) when page fault occurs for page X of a file, read pages X through $X+200$ and proactively add all to the current program's page table

(d) when page fault occurs for page X of a file, read pages X through $X+200$ but don't place pages $X+1$ through $X+200$ in the page table yet

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

when to start reads?

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

- takes some time to read in data — well before needed

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

- takes some time to read in data — well before needed

how much to readahead?

- if too much: evict other stuff programs need
- if too little: won't keep up with program
- if too little: won't make efficient use of HDD/SSD/etc.

problems with LRU

question: when does LRU perform poorly?

exercise: which of these is LRU bad for?

code in a text editor for handling out-of-disk-space errors

initial values of the shell's global variales

on a desktop, long movies that are too big to fit in memory and played from beginning to end

on web server, long movies that are too big to fit in memory and frequently downloaded by clients

files that are parsed when loaded and overwritten when saved

on web server, frequently requested HTML files

problems with LRU

question: when does LRU perform poorly?

only reading things once

repeated scans of large amounts of data

problems with LRU

question: when does LRU perform poorly?

only reading things once

repeated scans of large amounts of data

both common access patterns for files

solution for LRU being bad?

one idea that Linux uses:

for *file data*, use different replacement policy

tries to avoid keeping around file data accessed only once

CLOCK-Pro: special casing for one-use pages

by default, Linux tries to handle scanning of files

- one read of file data — e.g. play a video, load file into memory

basic idea: delay considering pages active until second access

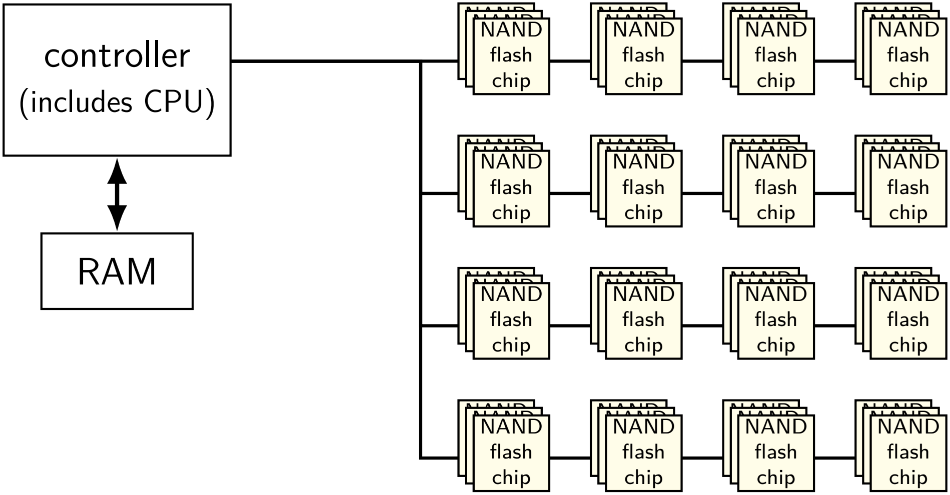
- second access = second scan of accessed bits/etc.

single scans of file won't "pollute" cache

without this change: reading large files slows down other programs

- recently read part of large file steals space from active programs

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

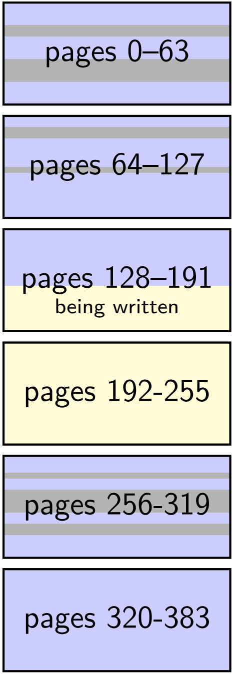
remapping table

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

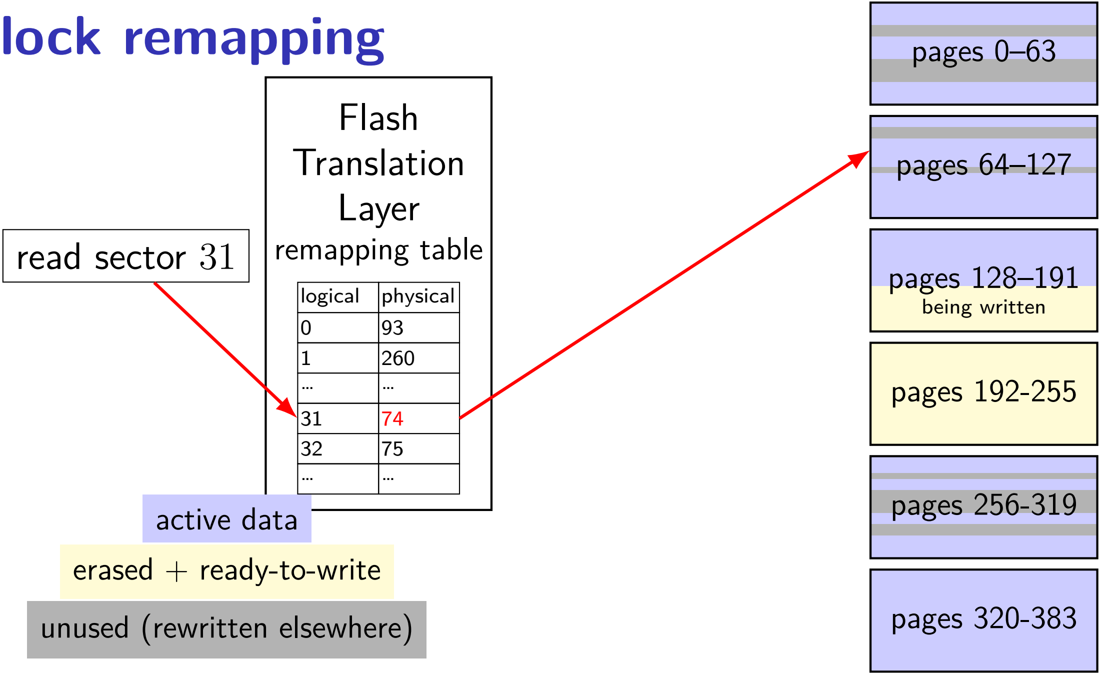
active data

erased + ready-to-write

unused (rewritten elsewhere)

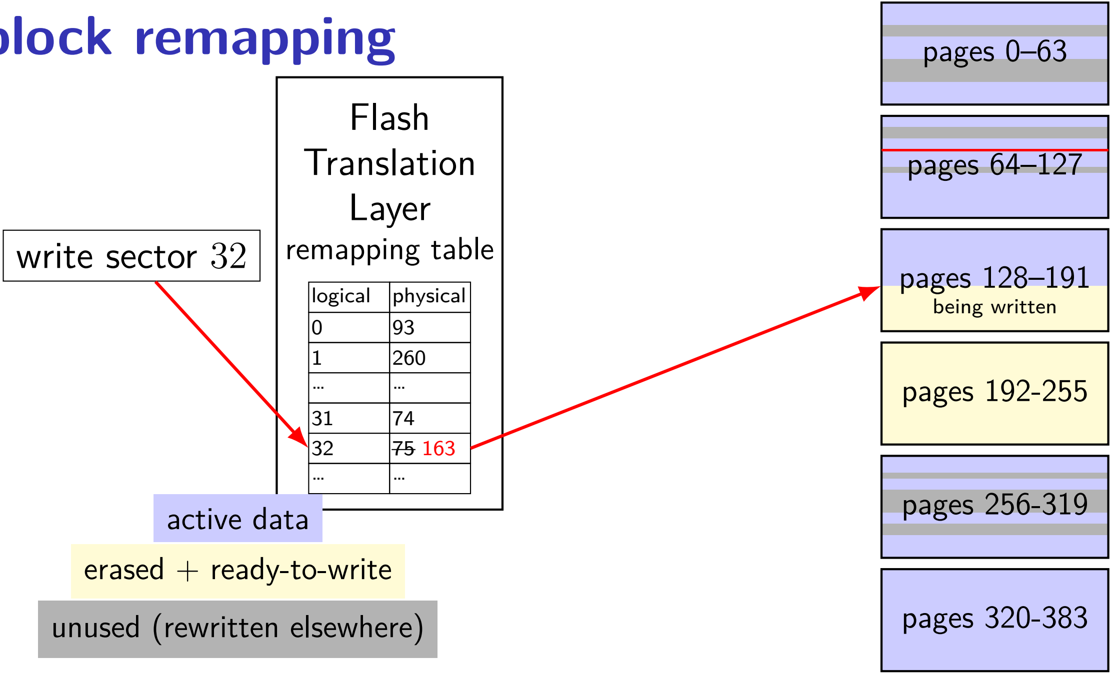


block remapping



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

block remapping



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

block remapping

Flash Translation Layer

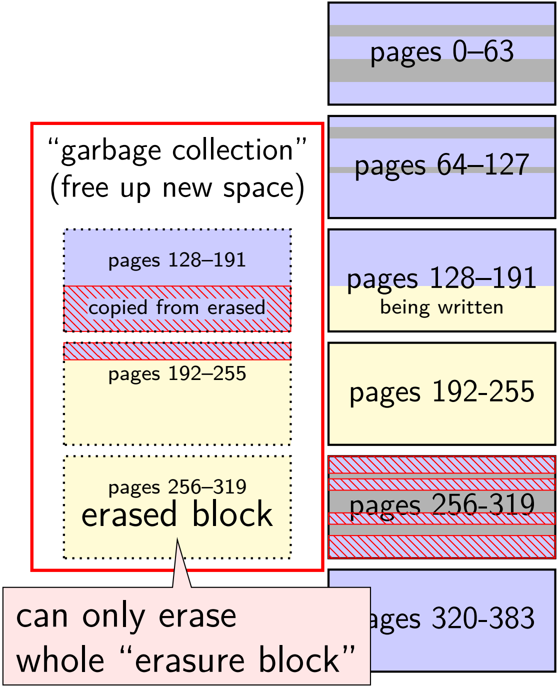
remapping table

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

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