FAT / inodes

the FAT filesystem

FAT: File Allocation Table

probably simplest widely used filesystem (family)

named for important data structure: *file allocation table*

FAT and sectors

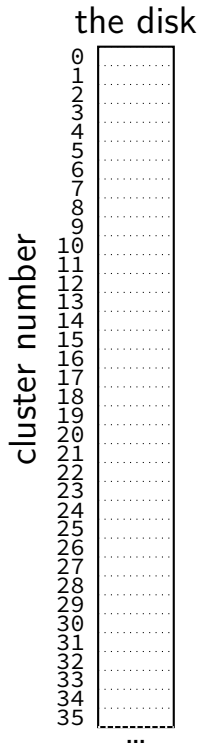
FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read

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

cluster: typically 512 to 4096 bytes



FAT and sectors

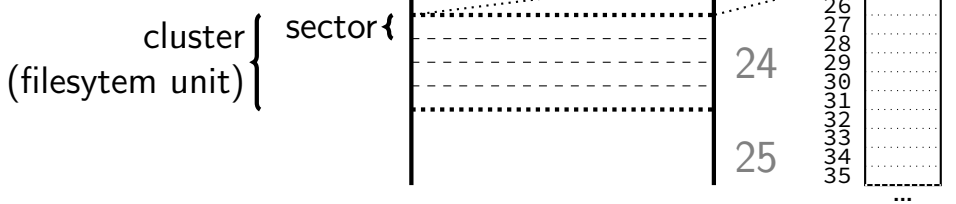
FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read

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

cluster: typically 512 to 4096 bytes

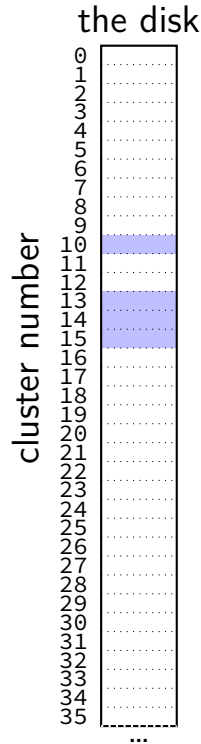


FAT: clusters and files

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

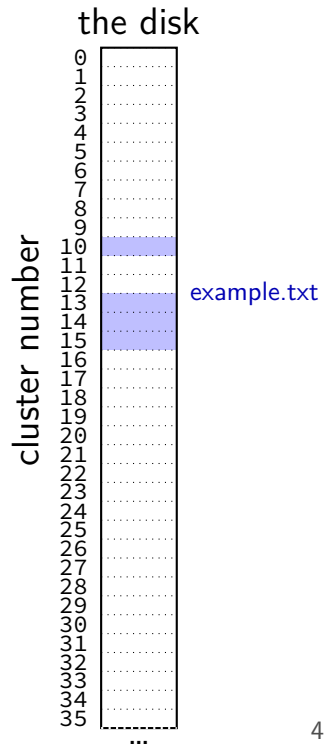


FAT: clusters and files

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

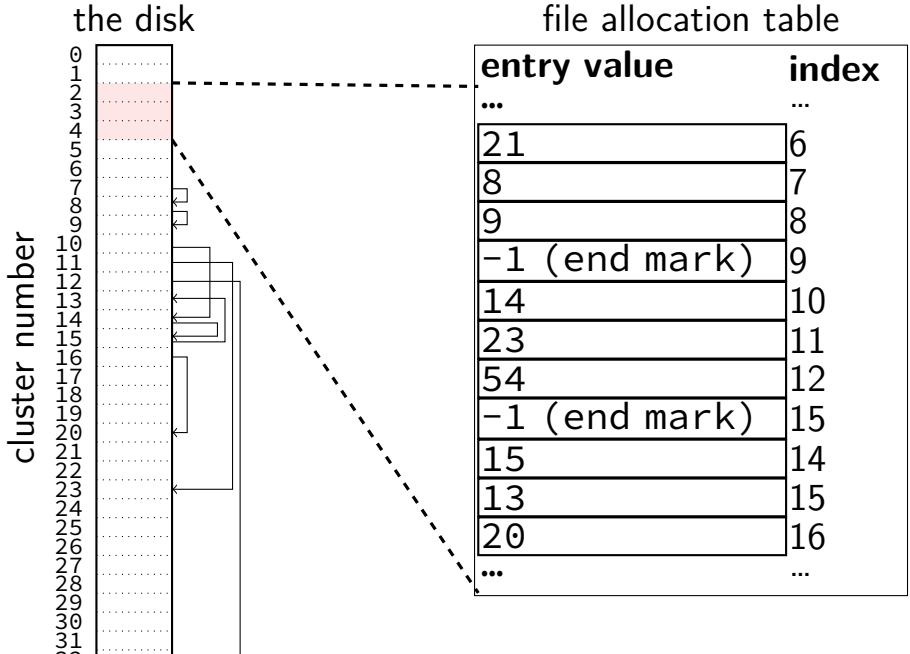
next pointers? file allocation table entry for cluster

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

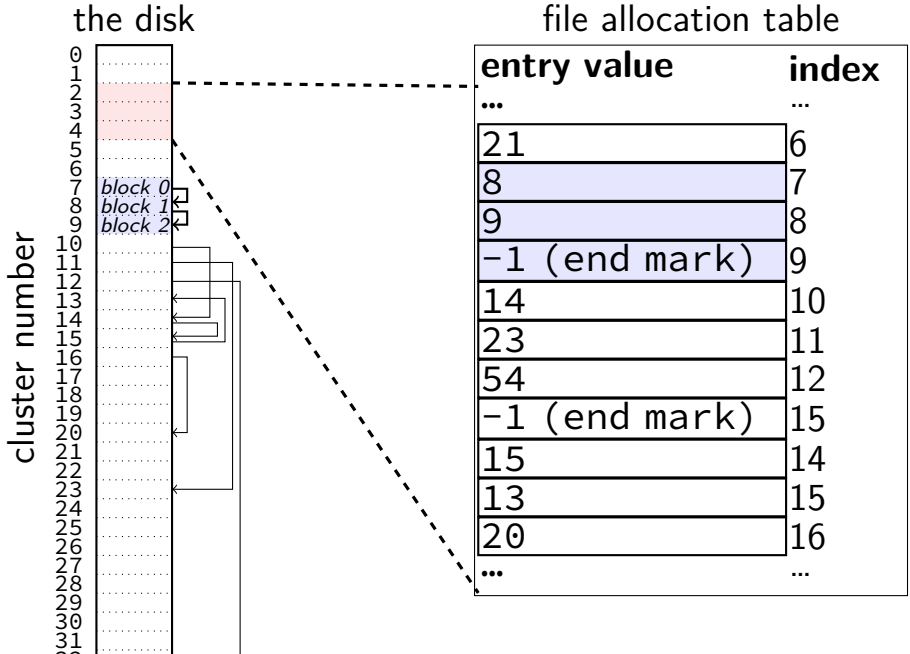
cluster num.	entry value
...	...
10	14
11	23
12	54
13	-1 (end mark)
14	15
15	13
...	...

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

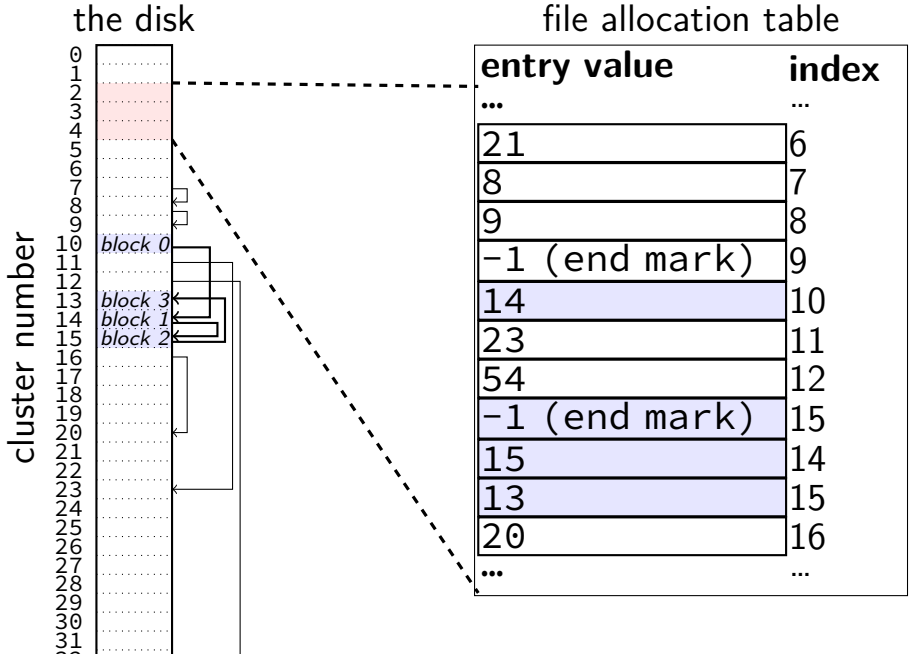
FAT: reading a file (2)



FAT: reading a file (2)



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?

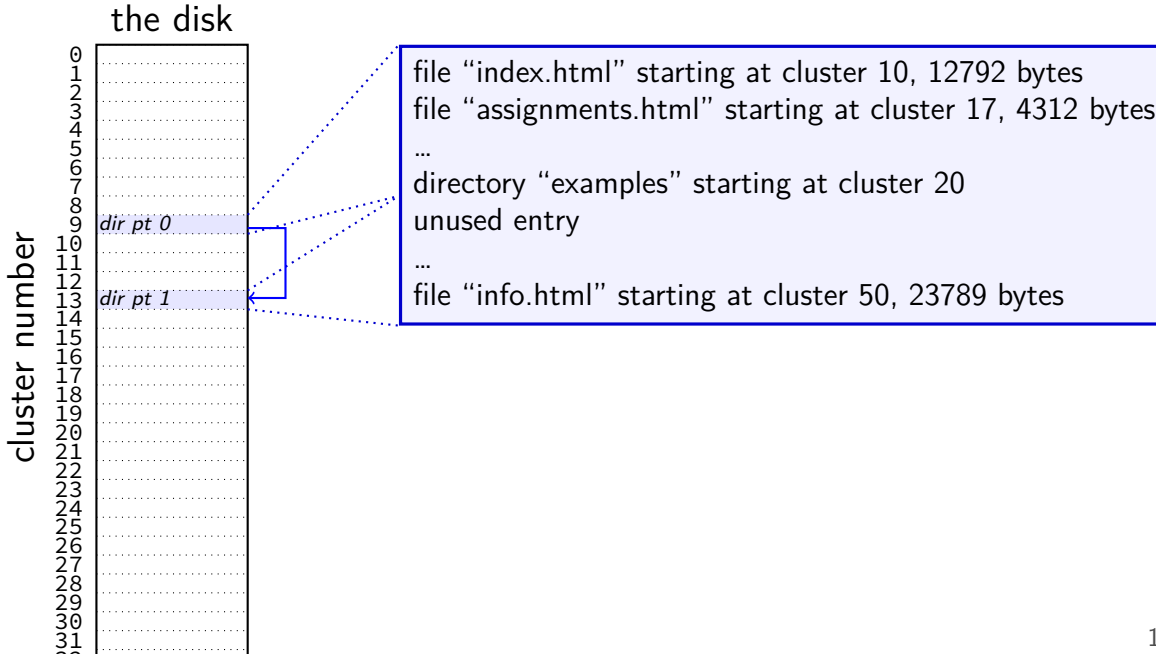
really want filenames

stored in directories!

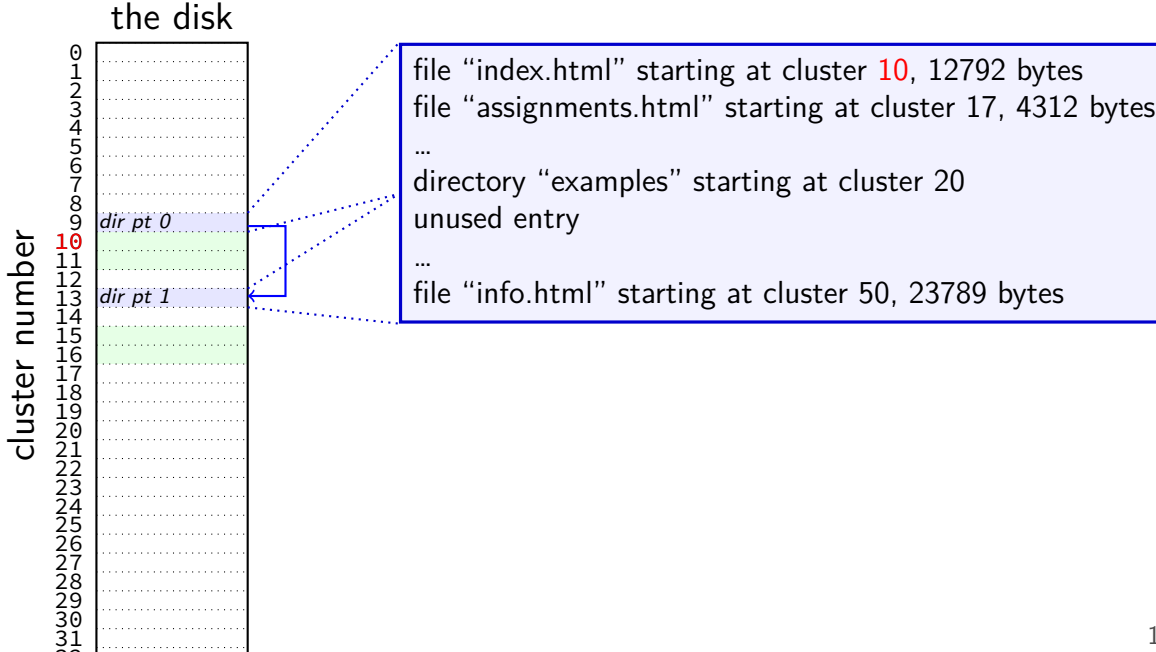
in FAT: directory is a file, but its data is list of:

(name, starting location, other data about file)

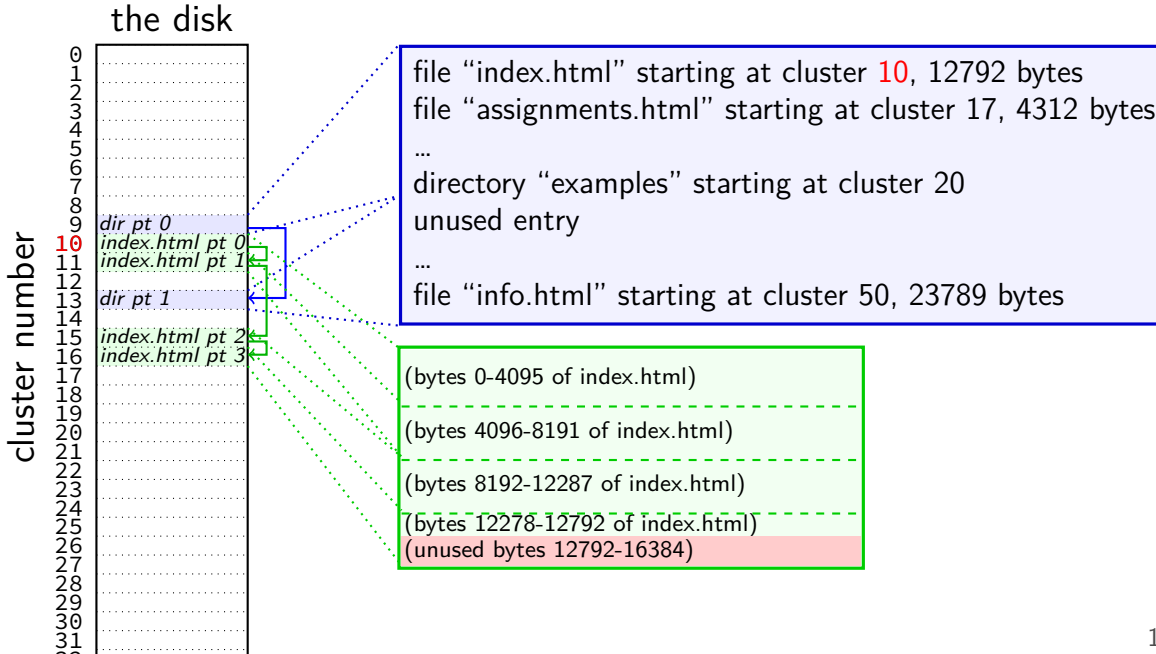
finding files with directory



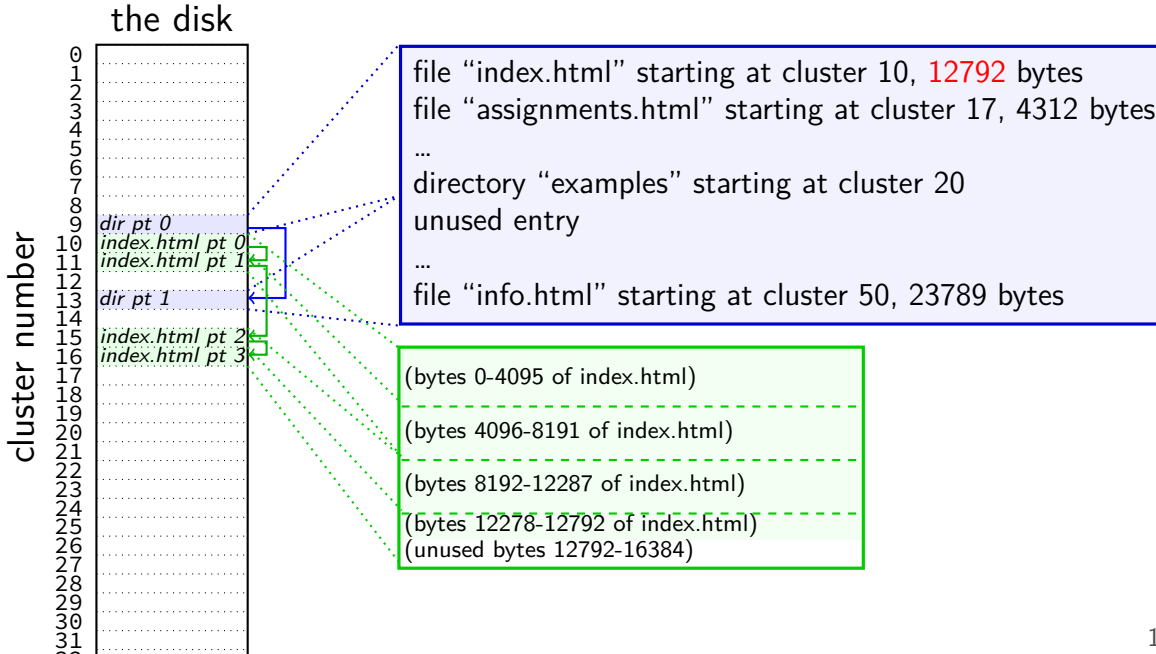
finding files with directory



finding files with directory



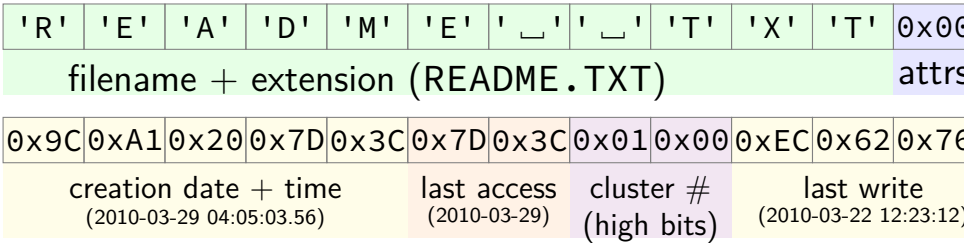
finding files with directory



FAT directory entry

box = 1 byte

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

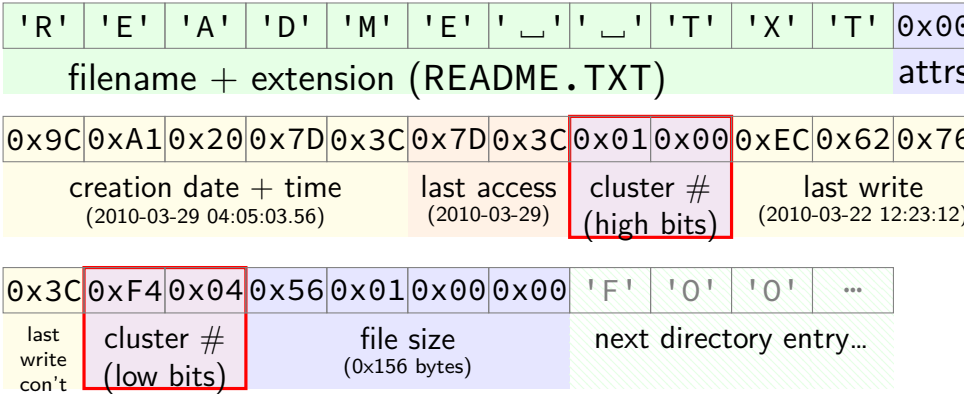


directory?
read-only?
hidden?
...

FAT directory entry

box = 1 byte

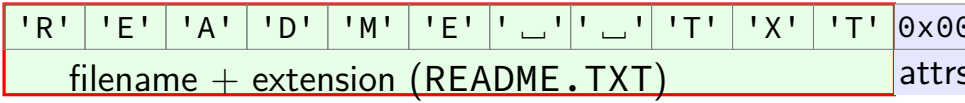
entry for README.TXT, 342 byte file, starting at cluster 0x104F4



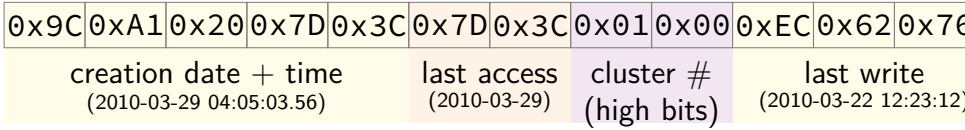
FAT directory entry

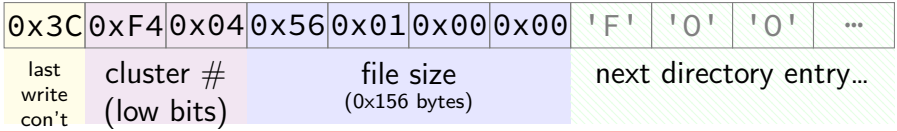
box = 1 byte

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



directory?
read-only?
hidden?
...



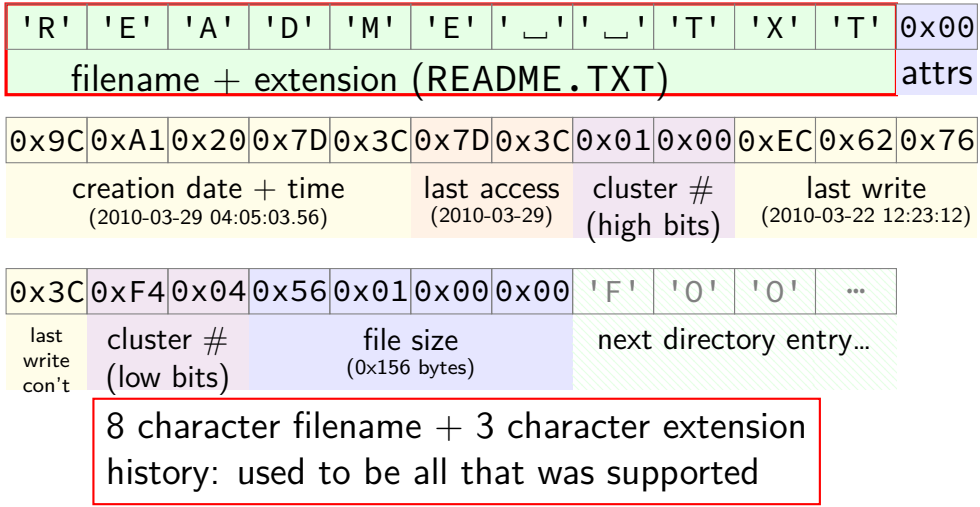


8 character filename + 3 character extension
longer filenames? encoded using extra directory entries
(special `attrs` values to distinguish from normal entries)

FAT directory entry

box = 1 byte

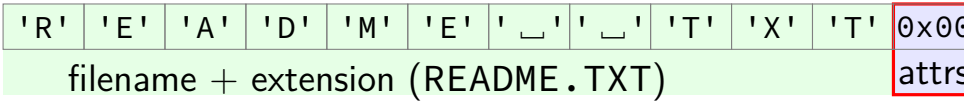
entry for README.TXT, 342 byte file, starting at cluster 0x104F4



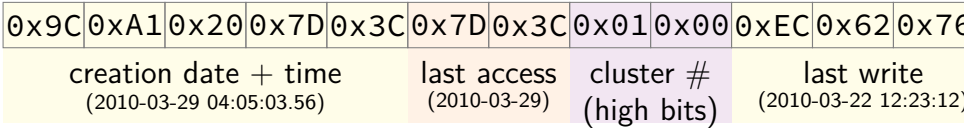
FAT directory entry

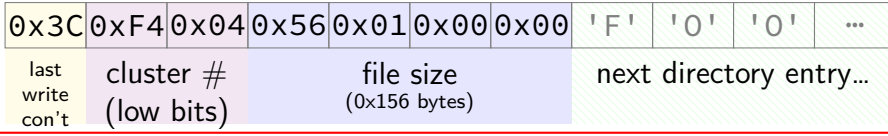
box = 1 byte

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



directory?
read-only?
hidden?
...



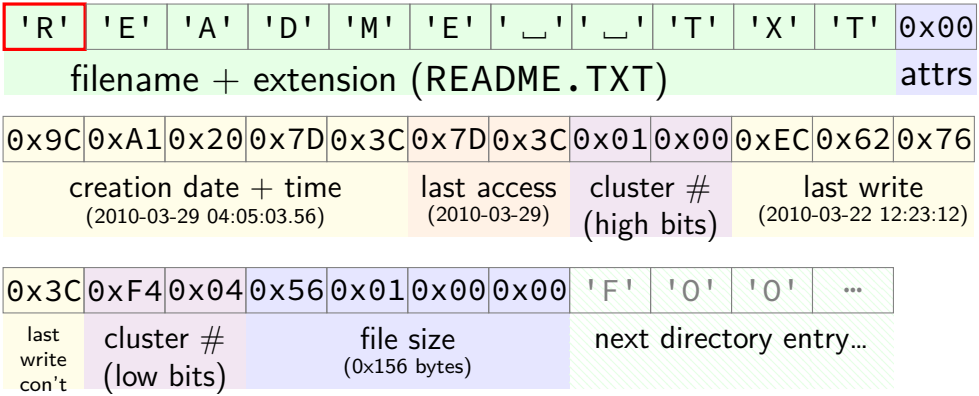


attributes: is a subdirectory, read-only, ...
also marks directory entries used to hold extra filename data

FAT directory entry

box = 1 byte

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



directory?
read-only?
hidden?
...

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

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

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)

```
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;
  uint8_t DIR_CrtTimeTenth;
  uint16_t DIR_CrtTime;
  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_Nam
  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

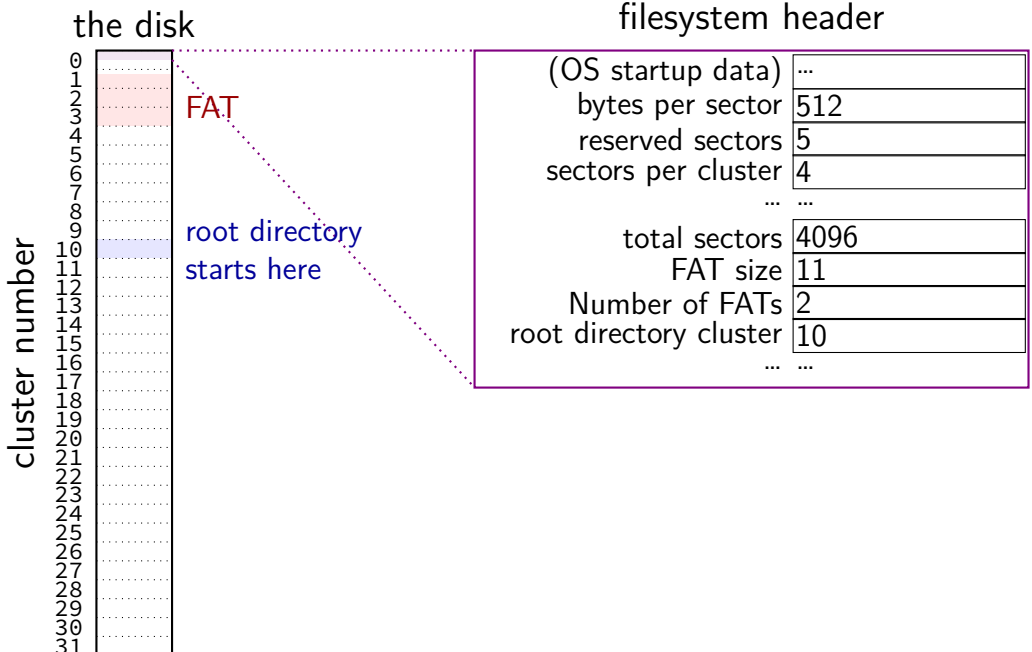
read bar's directory entries to find baz

read baz's directory entries to find file.txt

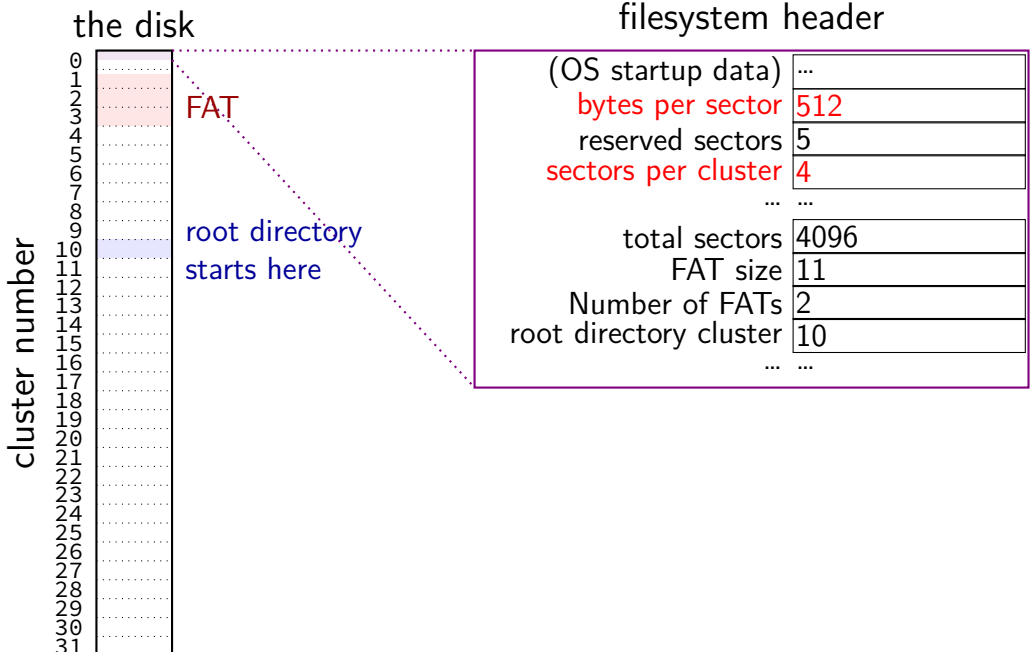
the root directory?

but where is the first directory?

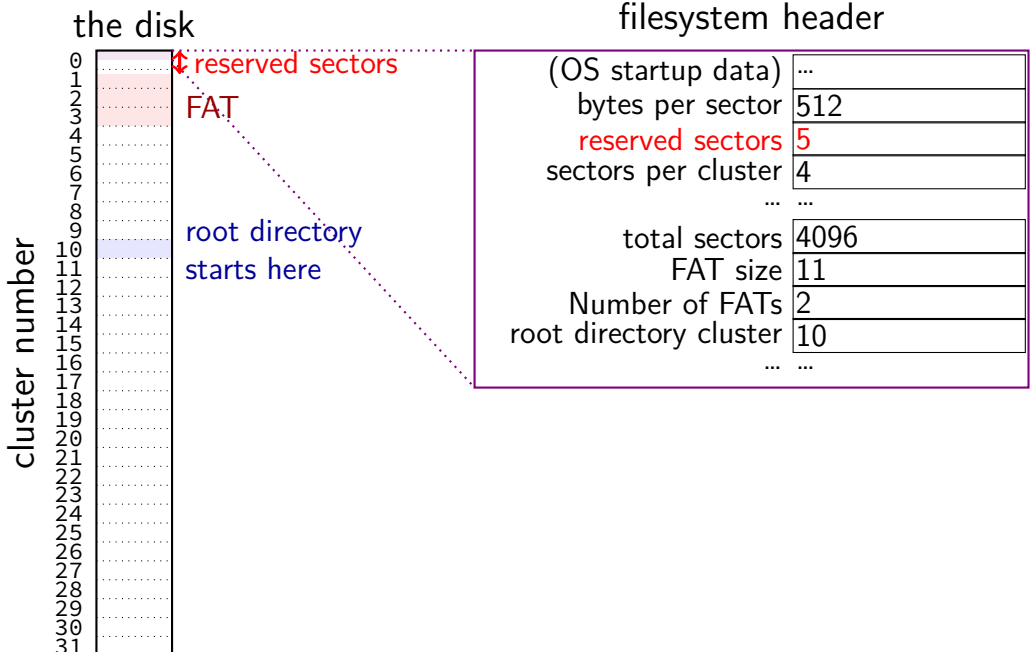
FAT disk header



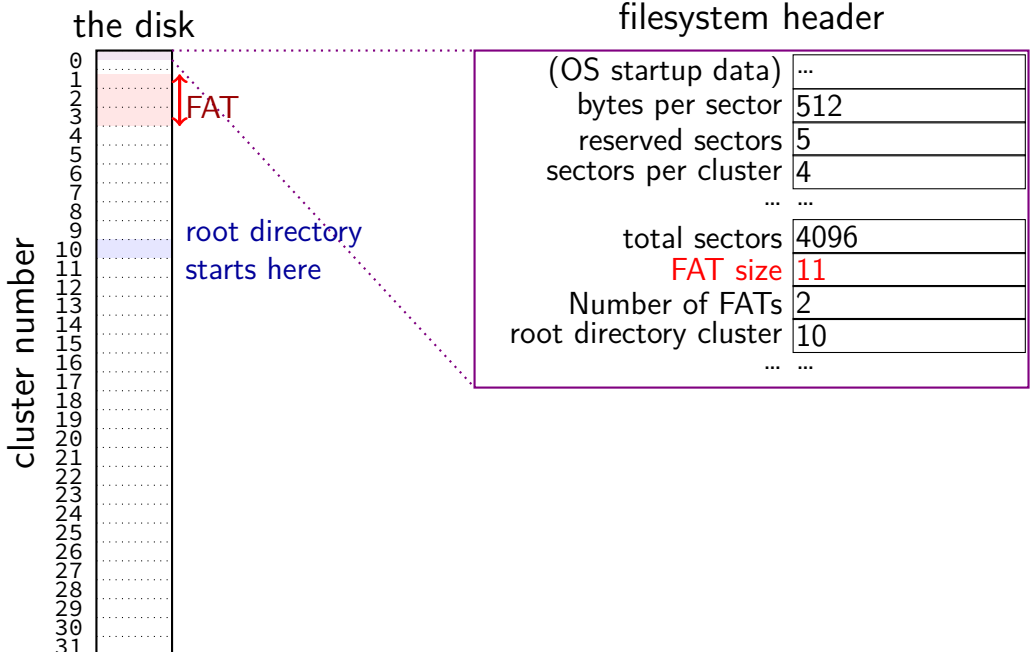
FAT disk header



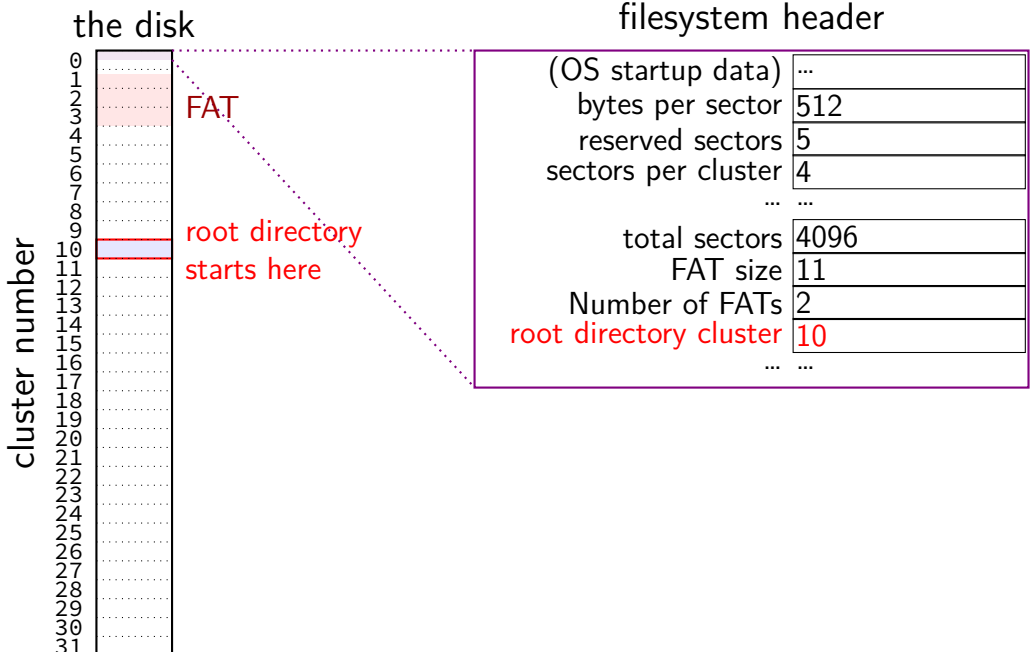
FAT disk header



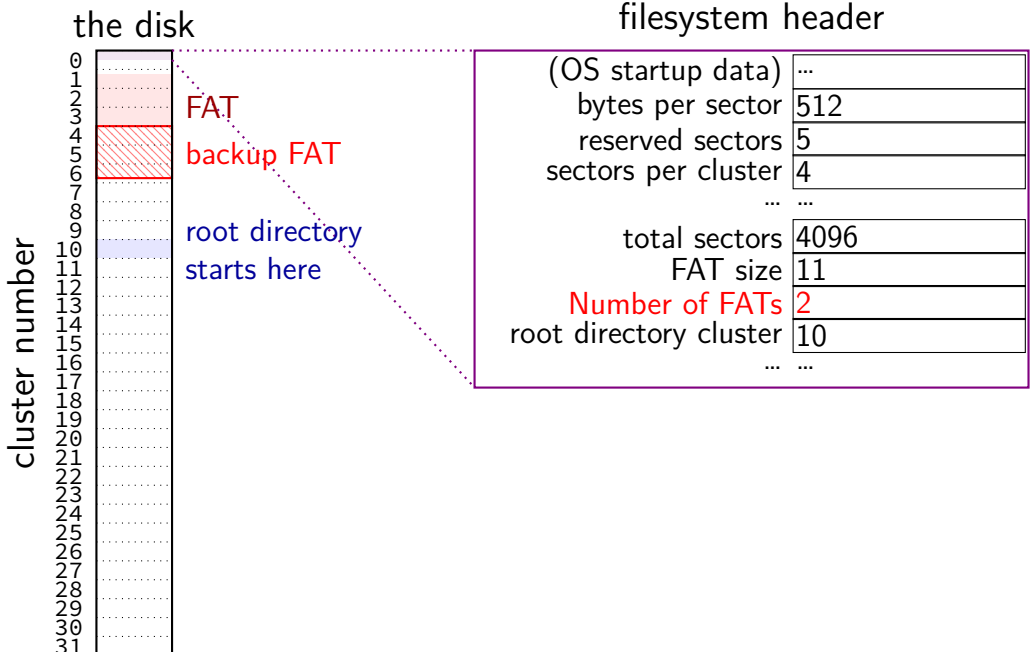
FAT disk header



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 activ
```

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;    // no
  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;
  uint16_t BPB_RsvdSecCnt;
  uint8_t BPB_NumFATs;
  uint16_t BPB_rootEntCnt;
  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

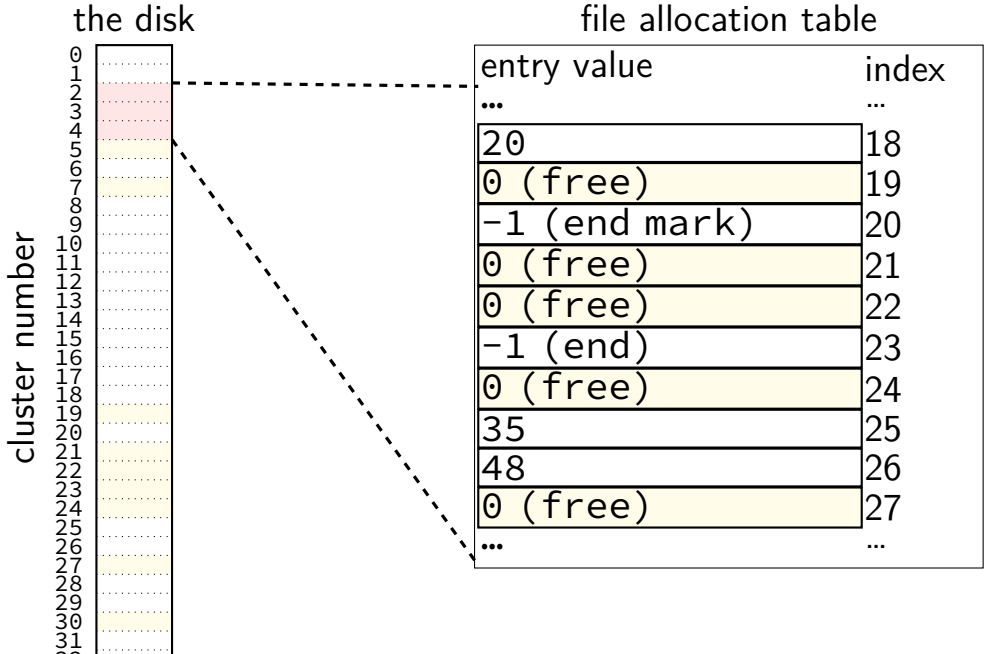
FAT: creating a file

add a directory entry

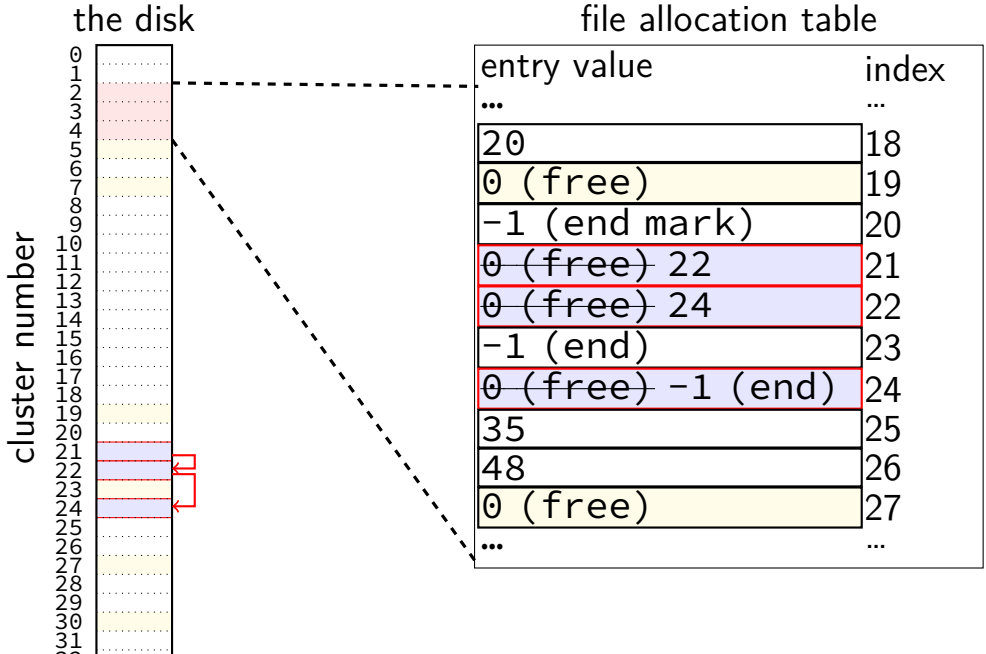
choose clusters to store file data (how???)

update FAT to link clusters together

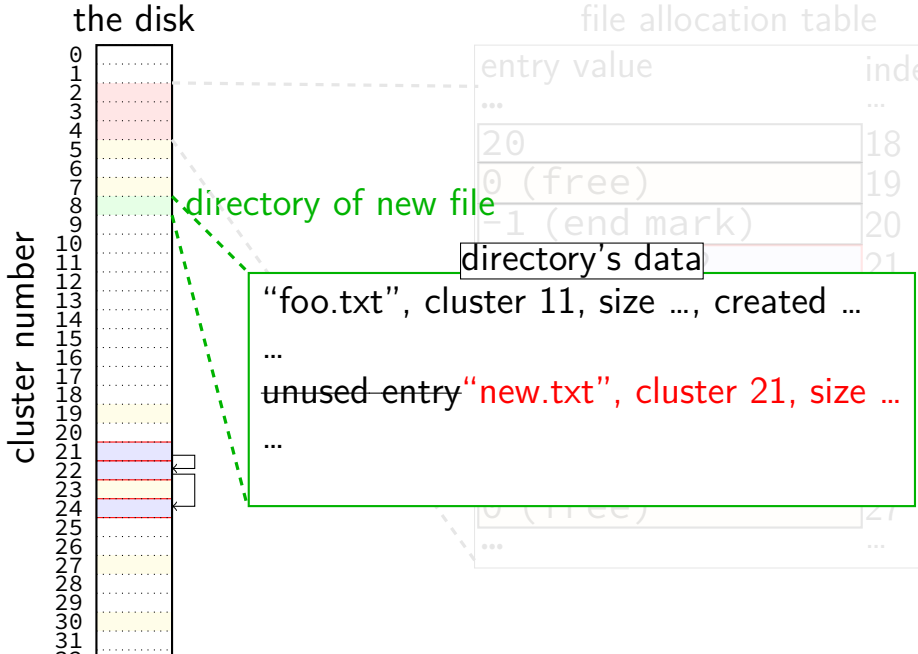
FAT: free clusters



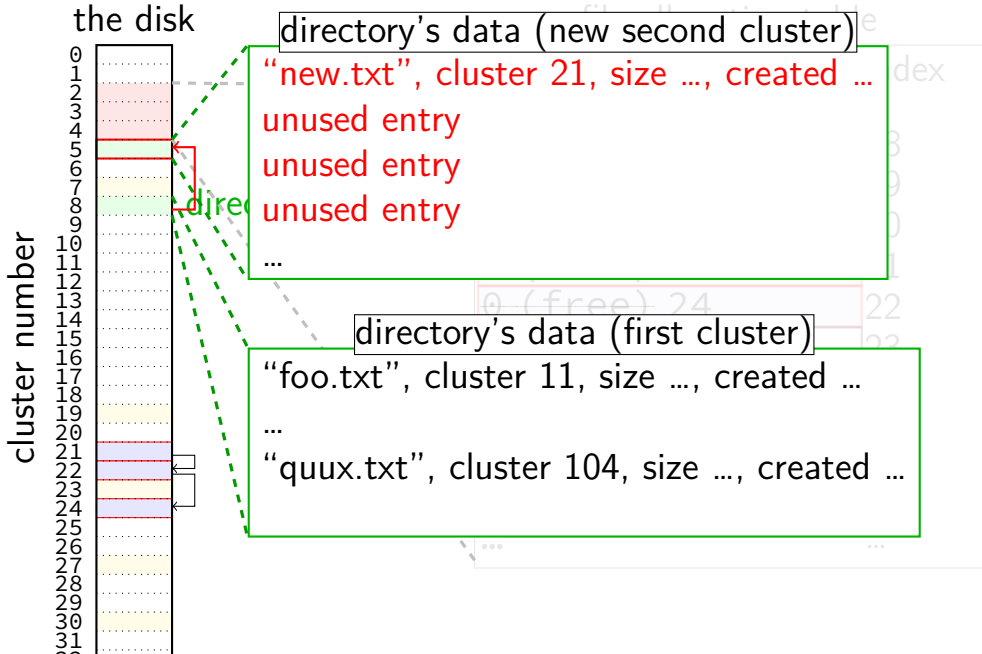
FAT: writing file data



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
[b] FAT entries for B
[c] FAT entries for C.txt
[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: 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?

FAT pros and cons?

xv6 filesystem

xv6's filesystem similar to modern Unix filesytems

better at doing contiguous reads than FAT

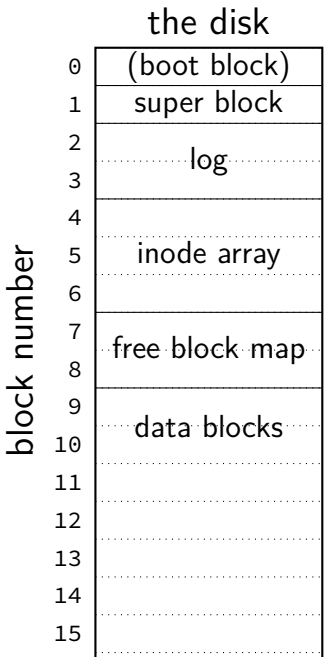
better at handling crashes

supports *hard links*

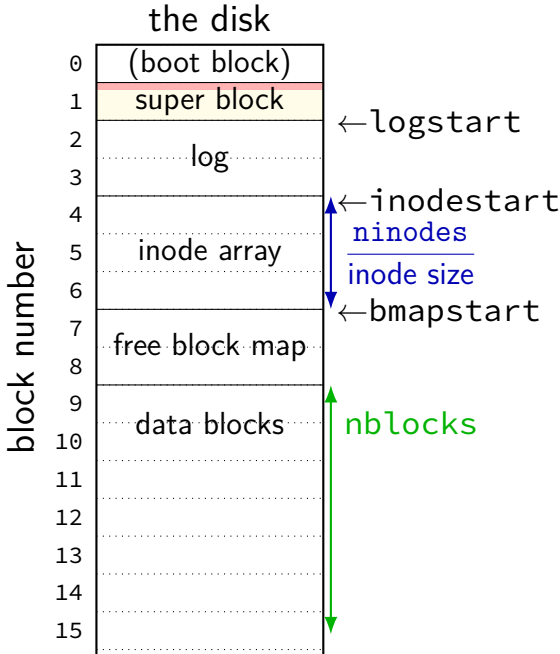
divides disk into *blocks* instead of clusters

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

xv6 disk layout



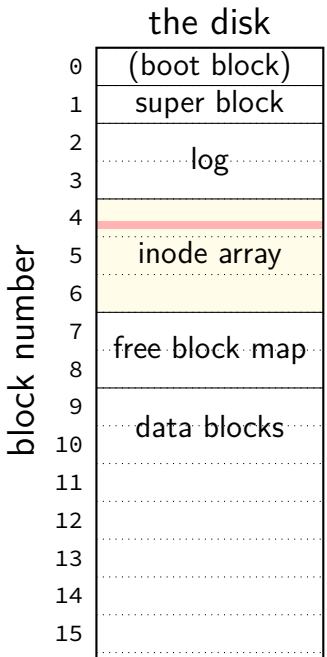
xv6 disk layout



superblock — "header"

```
struct superblock {
  uint size;
    // Size of file system image (b
  uint nblocks;
    // # of data blocks
  uint ninodes;
    // # of inodes
  uint nlog;
    // # of log blocks
  uint logstart;
    // block # of first log block
  uint inodestart;
    // block # of first inode block
  uint bmapstart;
    // block # of first free map bl
};
```

xv6 disk layout



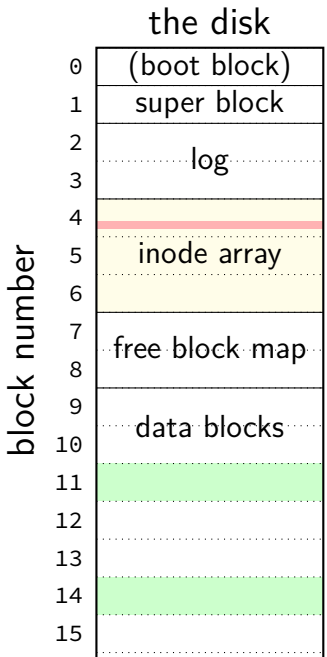
inode — file information

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

  short major; short minor; // T_DEV only

  short nlink;
    // Number of links to inode in file syst
  uint size;   // Size of file (bytes)
  uint addrs[NDIRECT+1];
    // Data block addresses
};
```

xv6 disk layout



inode — file information

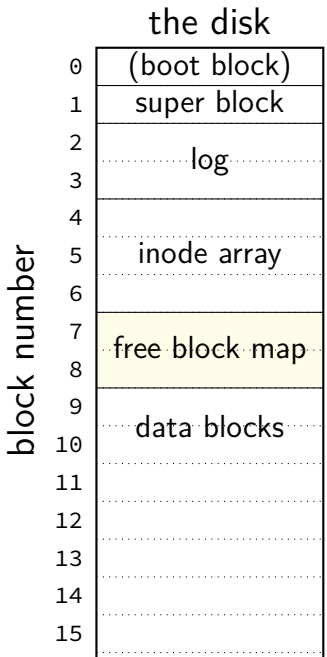
```
struct dinode {
  short type;  // File type
    // T_DIR, T_FILE, T_DEV

  short major; short minor; // T_DEV only

  short nlink;
    // Number of links to inode in file syst
  uint size;   // Size of file (bytes)
  uint addrs[NDIRECT+1];
    // Data block addresses
};
```

location of data as block numbers:
e.g. addrs[0] = 11; addrs[1] = 14;
special case for larger files

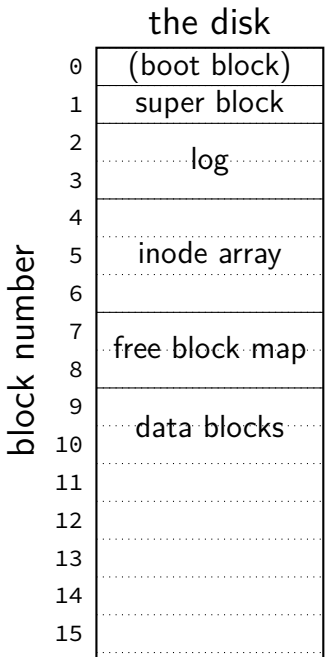
xv6 disk layout



free block map — 1 bit per data block
1 if available, 0 if used

allocating blocks: scan for 1 bits
contiguous 1s — contigous blocks

xv6 disk layout



what about finding free inodes
xv6 solution: scan for type $= 0$

typical Unix solution: separate free inode map

xv6 directory entries

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

inum — index into inode array on disk

name — name of file or directory

each directory reference to inode called a *hard link*

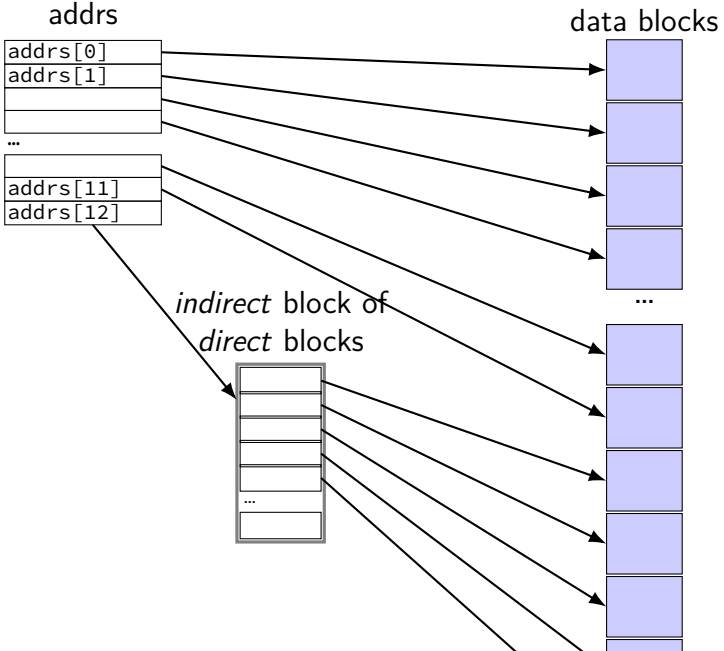
- multiple hard links to file allowed!

xv6 allocating inodes/blocks

need new inode or data block: linear search

simplest solution: xv6 always takes the first one that's free

xv6 inode: direct and indirect blocks



xv6 file sizes

512 byte blocks

2-byte block pointers: 256 block pointers in the indirect block

256 blocks = 131072 bytes of data referenced

12 direct blocks @ 512 bytes each = 6144 bytes

1 indirect block @ 131072 bytes each = 131072 bytes

maximum file size = 6144 + 131072 bytes

Linux ext2 inode

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

Linux ext2 inode

```
struct ext2_inode {
    __le16 i_mode;              /* File mode */
    __le16 i_uid;               /* Low 16 bits of Owner Uid */
    __le32 i_size;              /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __l
    __l
    __l
    __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 */
    ...
};
```

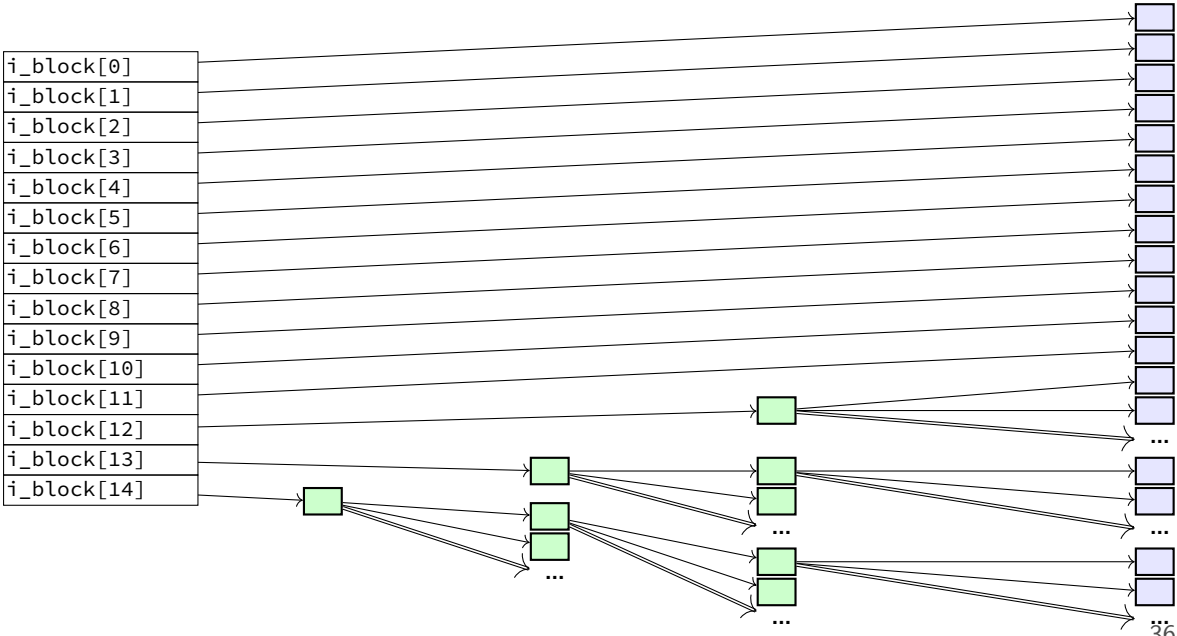
whole bunch of times

Linux ext2 inode

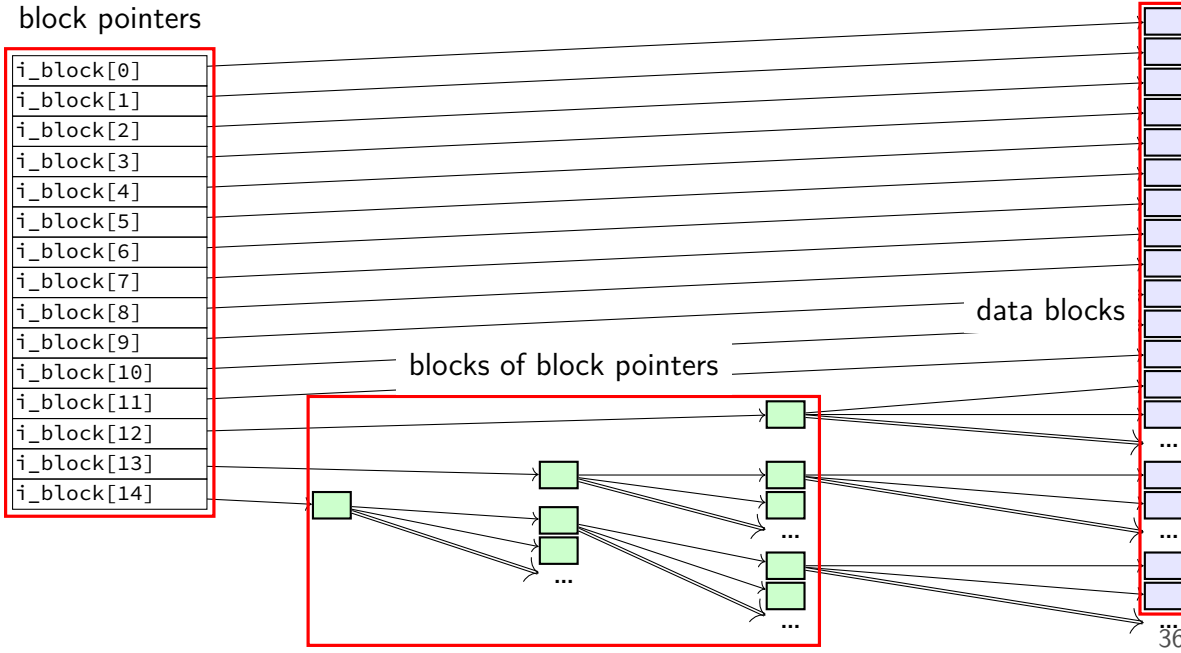
```
struct ext2_inode {
    __le16 i_mod
    __le16 i_uid
    __le32 i_size;                  /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16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

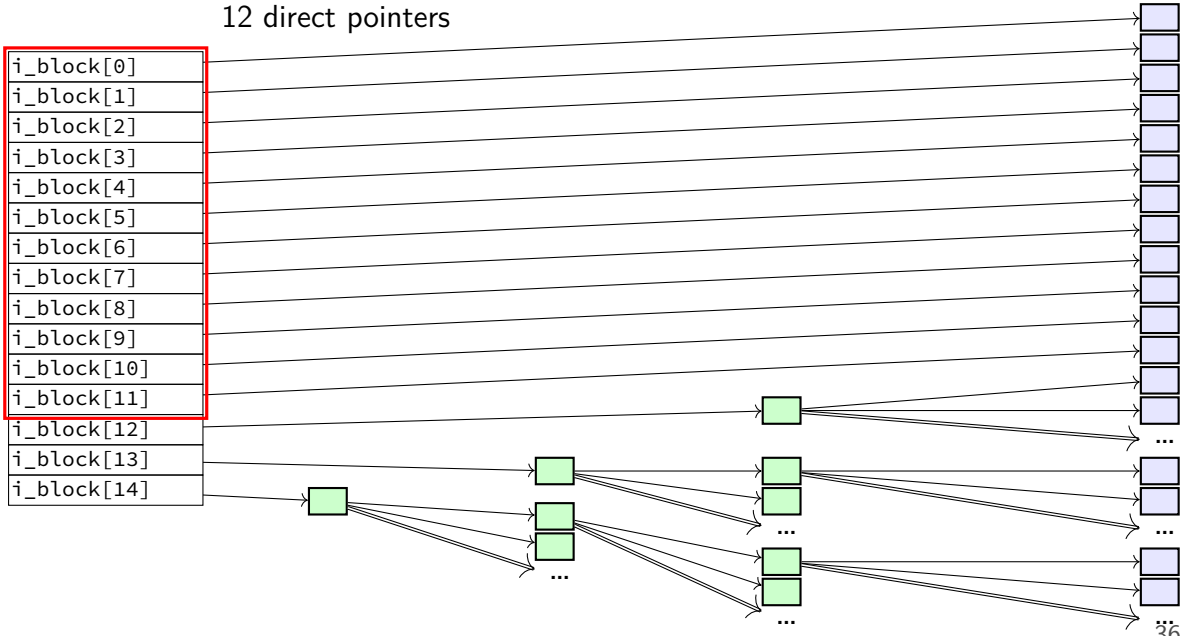
double/triple indirect



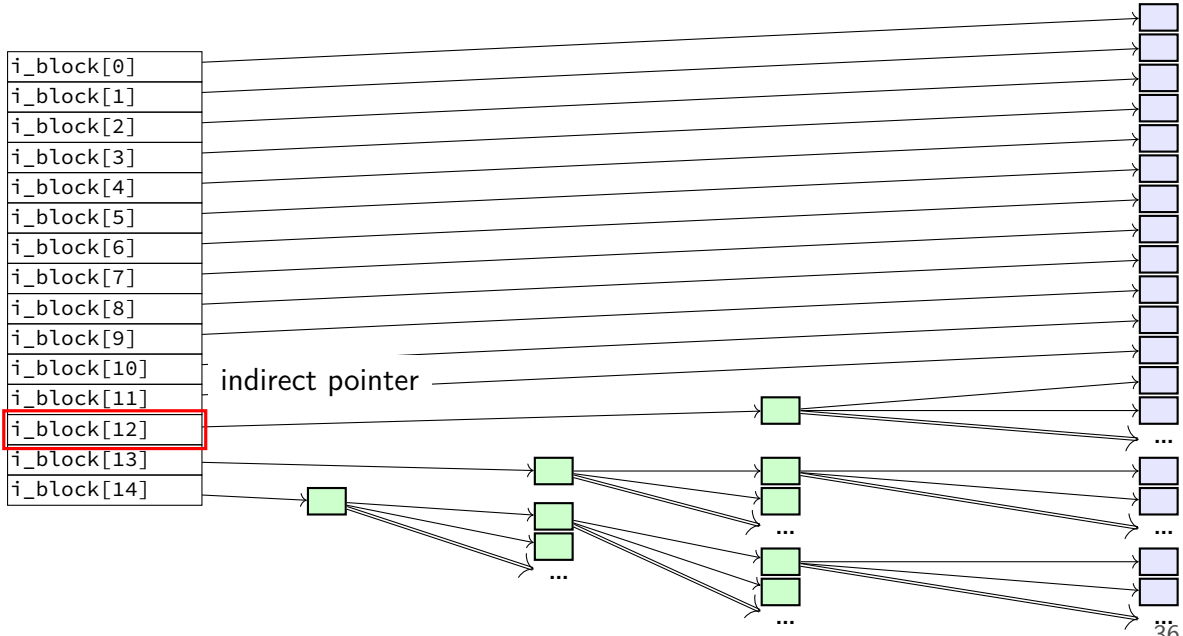
double/triple indirect



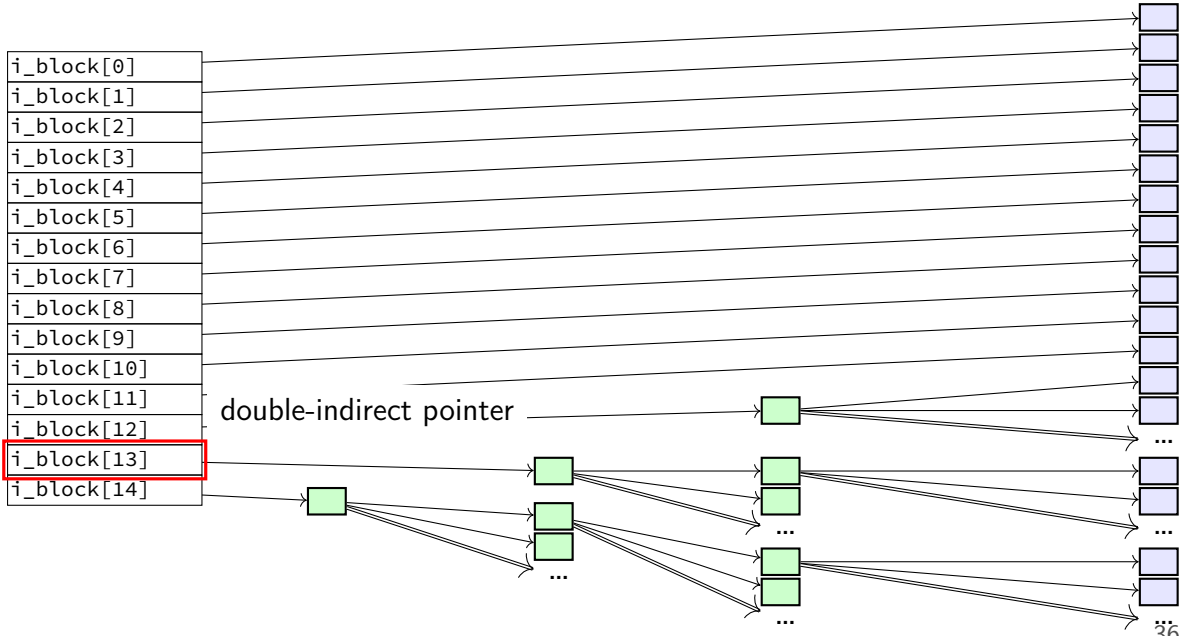
double/triple indirect



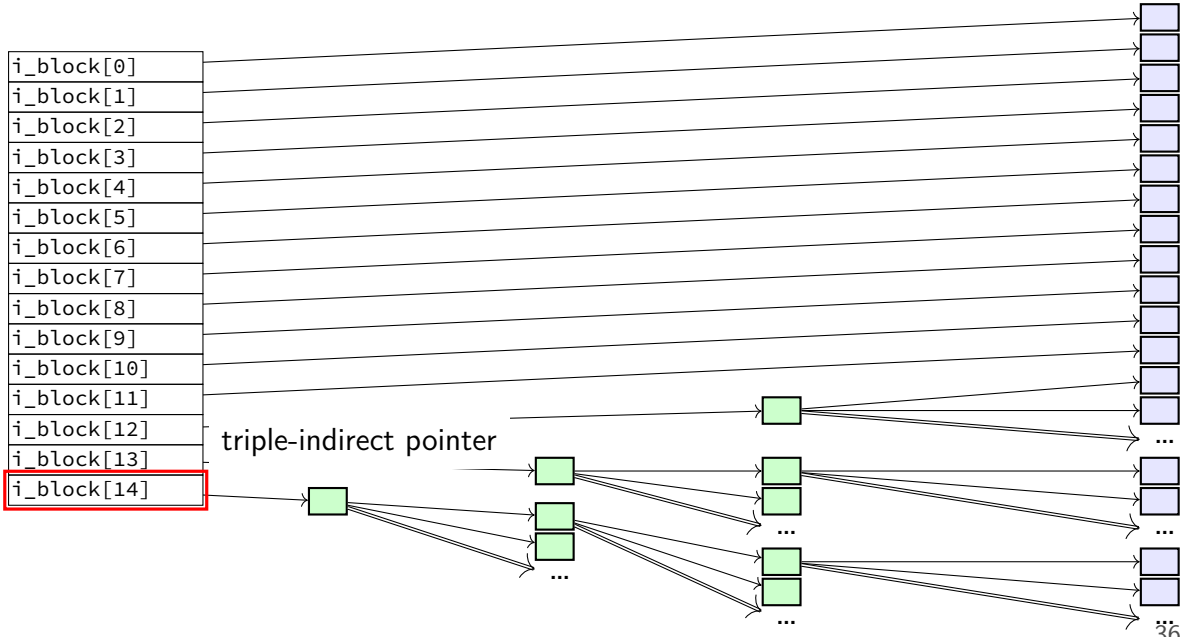
double/triple indirect



double/triple indirect



double/triple indirect



ext2 indirect blocks

12 direct block pointers

1 indirect block pointer

- pointer to block containing more direct block pointers

1 double indirect block pointer

- pointer to block containing more indirect block pointers

1 triple indirect block pointer

- pointer to block containing more double indirect block pointers

ext2 indirect blocks

12 direct block pointers

1 indirect block pointer

- pointer to block containing more direct block pointers

1 double indirect block pointer

- pointer to block containing more indirect block pointers

1 triple indirect block pointer

- pointer to block containing more double indirect block pointers

exercise: if 1K blocks, 4 byte block pointers, how big can a file be?

ext2 indirect blocks (2)

12 direct block pointers

1 indirect block pointer

1 double indirect block pointer

1 triple indirect block pointer

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

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