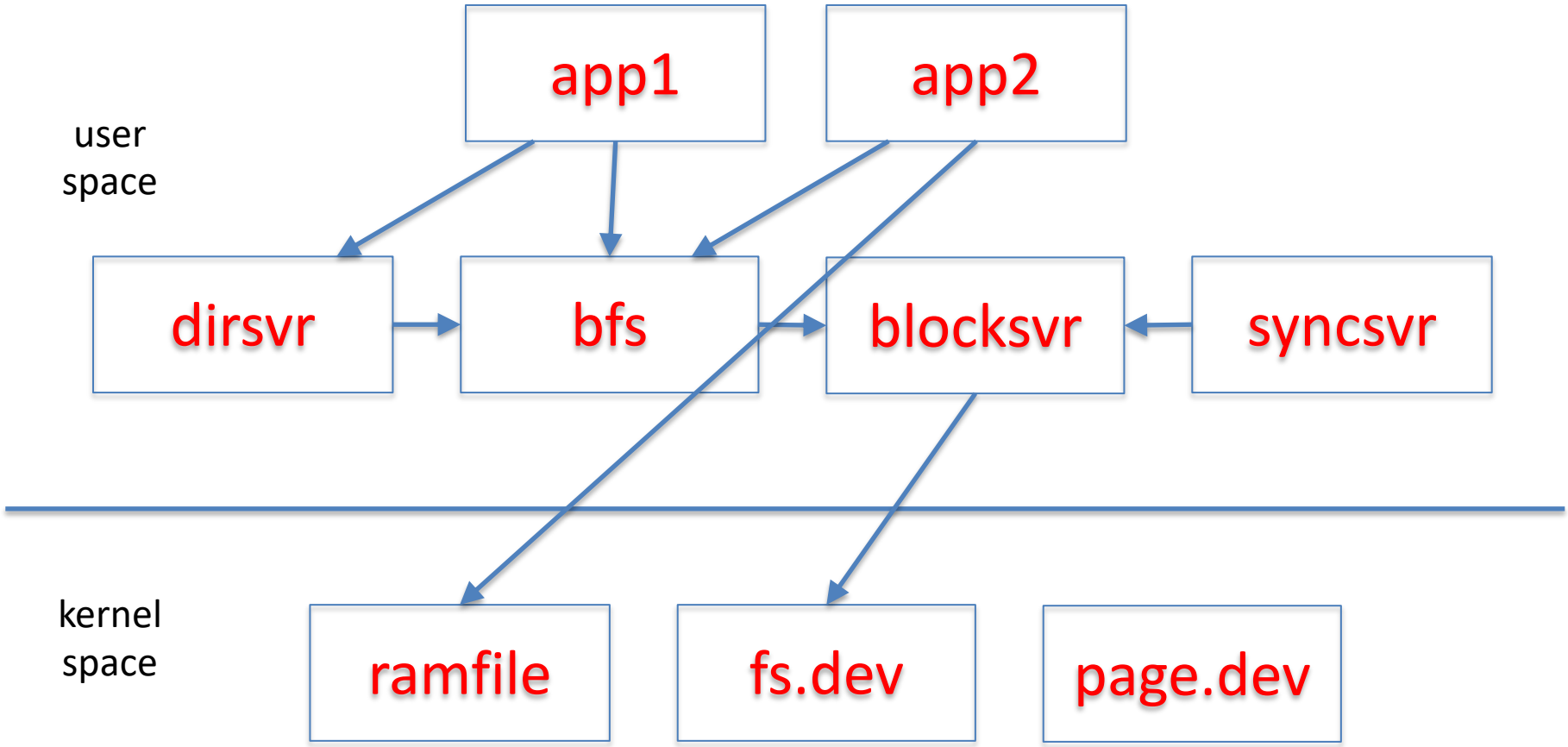
FAT File System

Robbert van Renesse

Intro

- Underneath any file system, database system, etc. there are one or more *block stores*
- A block store provides a disk-like interface:
  - a storage object is a sequence of blocks
    - typically, a few kilobytes
  - you can read or write a block at a time
- The block store abstraction doesn't deal with file naming, security, etc., just storage

EGOS Storage Architecture



bfs: block file server

- Stores all its user and meta data in blocksvr
- Maintains for each file a “stat structure”:
  - size in bytes
  - owner
  - modification time
  - access control information
  - etc.
- files are indexed by i-node numbers
  - 0, 1, 2, …
  - #i-nodes determined by blocksvr

Block Store Abstraction

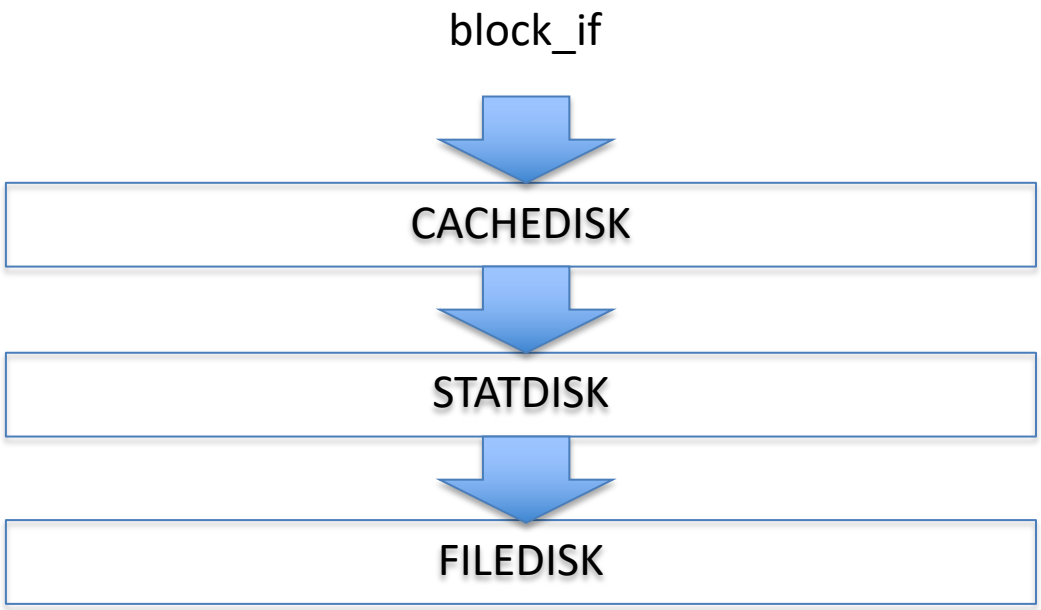
- A block store consists of a collection of *i-nodes*
- Each i-node is a finite sequence of *blocks*
- Simple interface:
  - block_t block
    - block of size BLOCK_SIZE
  - getninodes() → integer
    - returns the number of i-nodes on this block store
  - getsize(inode number) → integer
    - returns the number of of block on the given inode
  - setsize(inode number, nblocks)
    - set the number of blocks on the given inode
  - release()
    - give up reference to the block store

Block Store Abstraction, cont'd

- read(inode, block number) → block
  - returns the contents of the given block number
- write(inode, block number, block)
  - writes the block contents at the given block number
- sync(inode)
  - make sure all blocks are persistent
    - if inode == -1, then all blocks on all inodes

Block Stores can be Layered!

Each layer presents a block_if abstraction



keeps a cache of recently used blocks

keeps track of #reads and #writes for statistics

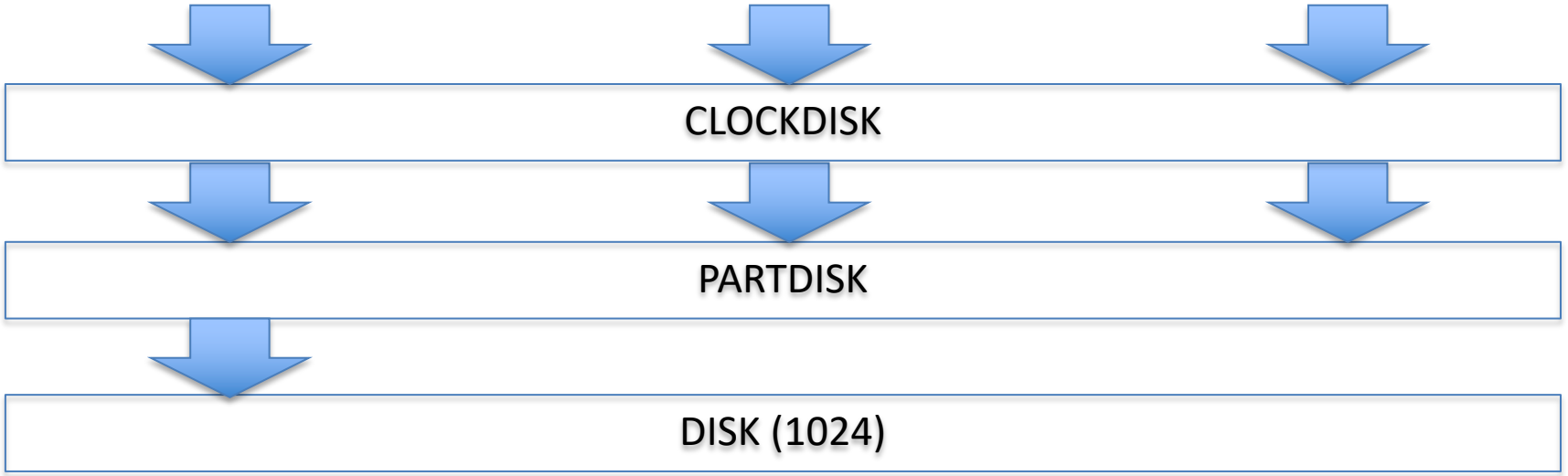
keeps blocks in a Posix file

Multiplexing

- A single block store can be “multiplexed”, offering multiple virtual block stores
- One way is simply partitioning the underlying block store into multiple disjoint sections

```
block_if partdisk_init(block_if below,
        unsigned int ninodes, block_no partsizes[])
```

Partitioning



Sharing a Block Store

- partdisk creates multiple fixed partitions, one for each file, but this has very similar problems to partitioning physical memory among processes
- You want something similar to paging
  - more efficient and flexible sharing
  - techniques are very similar!

Linked List Allocation

Each file is stored as linked list of blocks

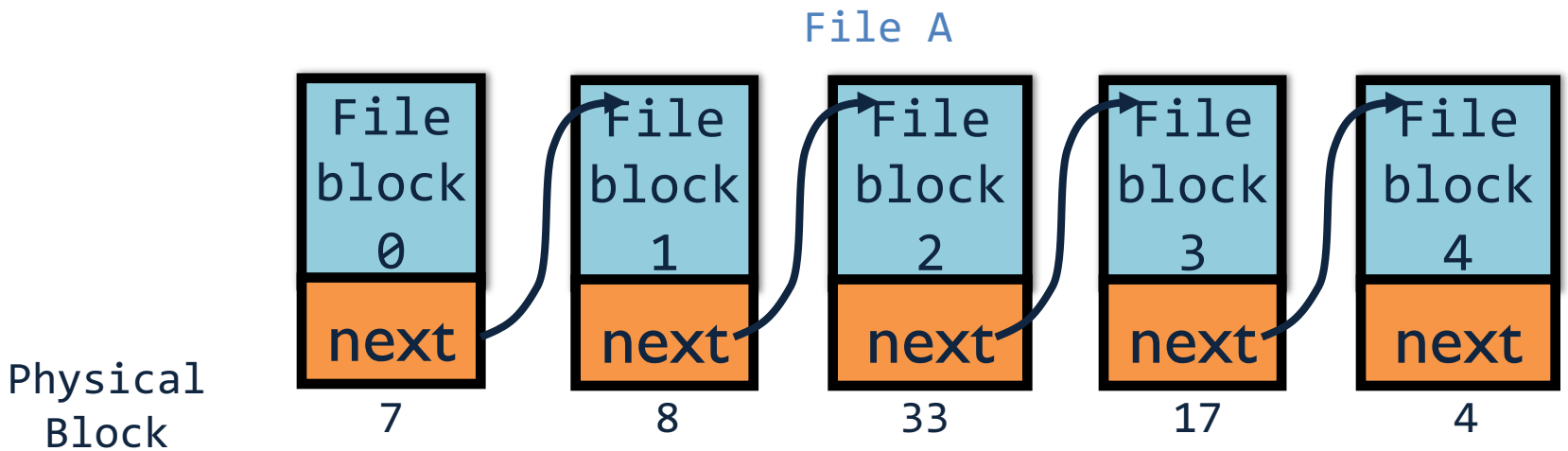
- First word of each block points to next block
- Rest of disk block is file data

+ Space Utilization: no space lost to external fragmentation

+ Simple: only need to find 1st block of each file

– Performance: random access is slow

– Implementation: blocks mix meta-data and data



File Allocation Table (FAT) FS

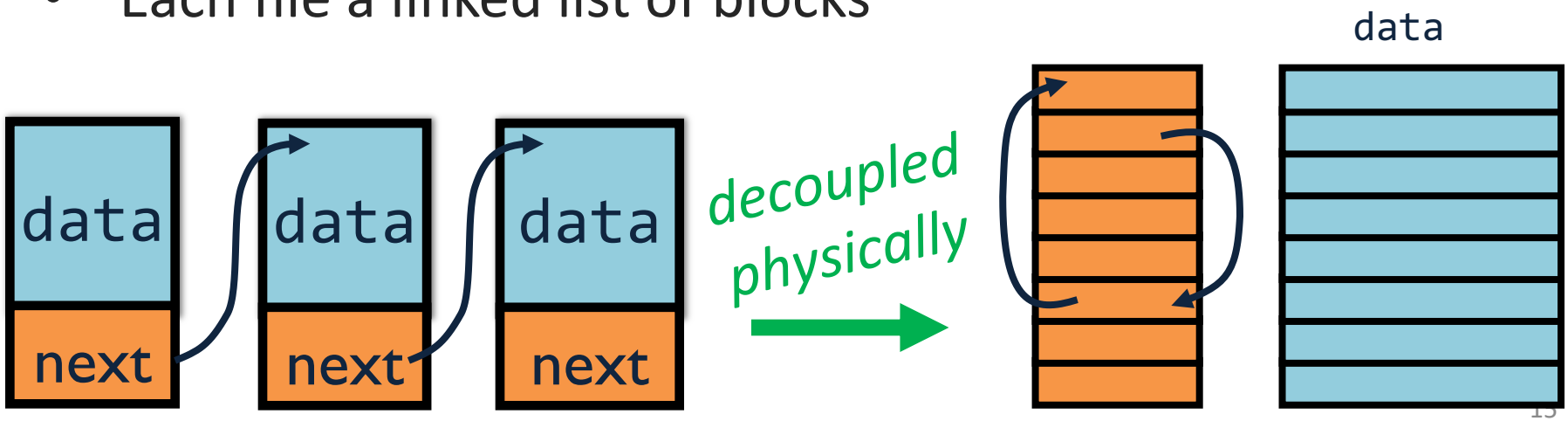
[late 70's]

Microsoft File Allocation Table

- originally: MS-DOS, early version of Windows
- today: still widely used (e.g., CD-ROMs, thumb drives, camera cards)

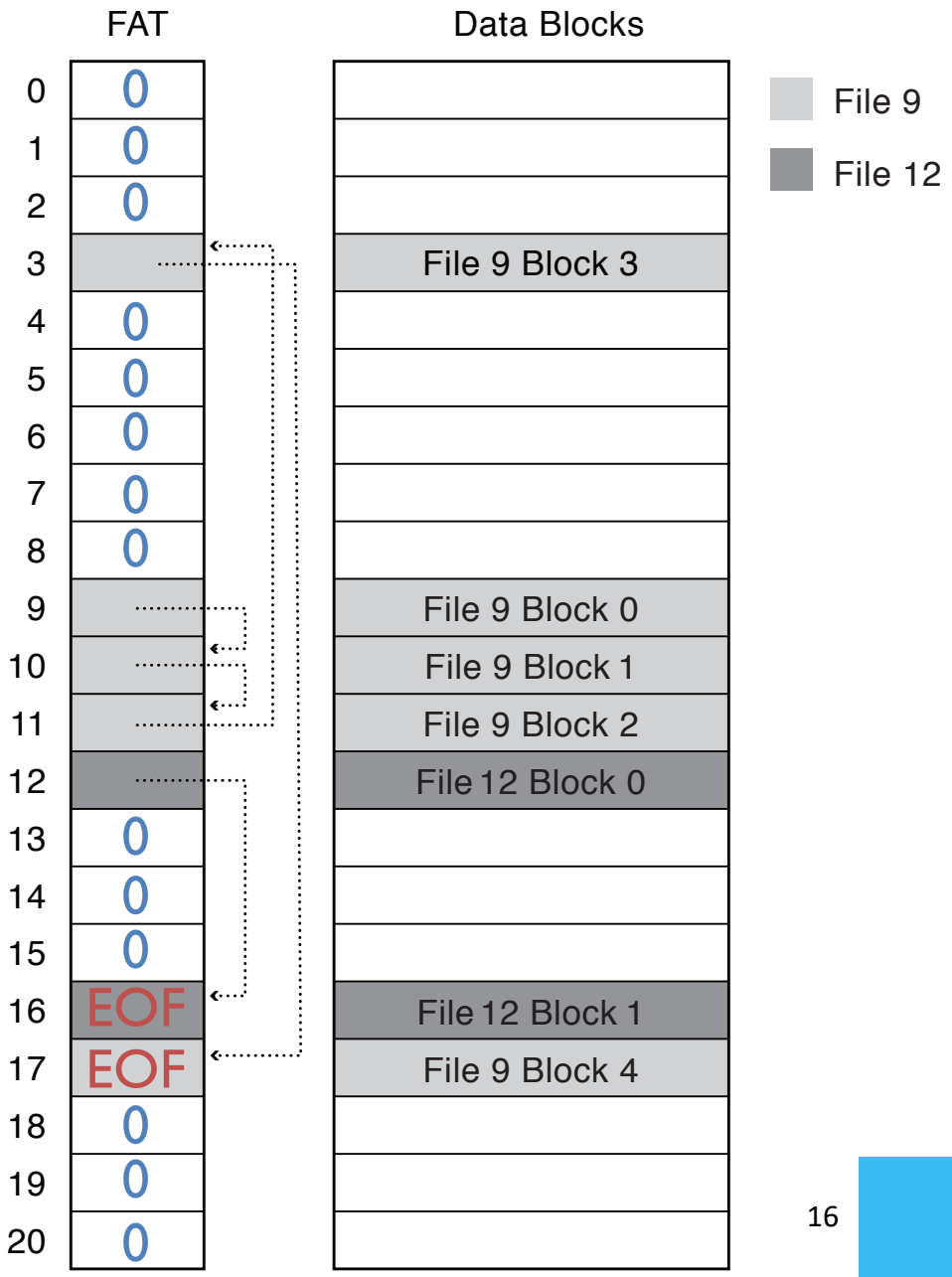
File table:

- Linear map of all blocks on disk
- Each file a linked list of blocks



FAT File System

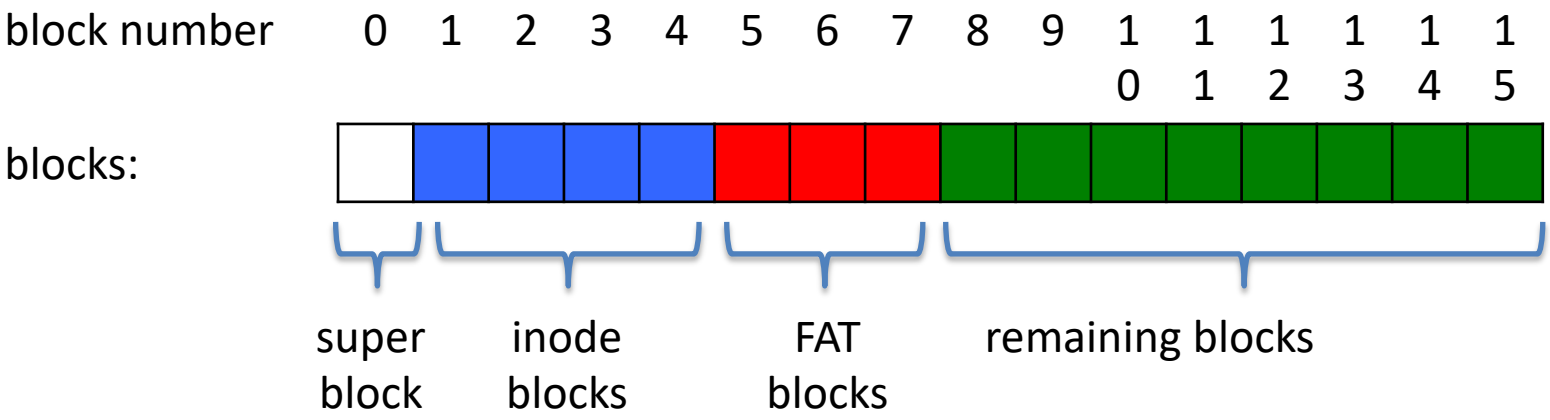
- 1 entry per block
- EOF for last block
- 0 indicates free block



P4: Partitioning with *fatdisk*

- fatdisk offers multiple virtual block stores
- The underlying block store is partitioned into four sections:
  1. *superblock*
     - at block #0
  2. a fixed number of *i-node blocks*
     - start at block #1
     - the number is given in the superblock
  3. the FAT table
     - the number is given in the superblock
  4. the remaining blocks
     - *data blocks, free blocks*

fatdisk: layout



fatdisk superblock

```
struct fatdisk_superblock {
    block_no n_inodeblocks;
         // # blocks containing inodes

    block_no n_fatblocks;
         // # blocks containing fat entries

    block_no fat_free_list;
         // fat index of the first free fat entry};

   …
}
```

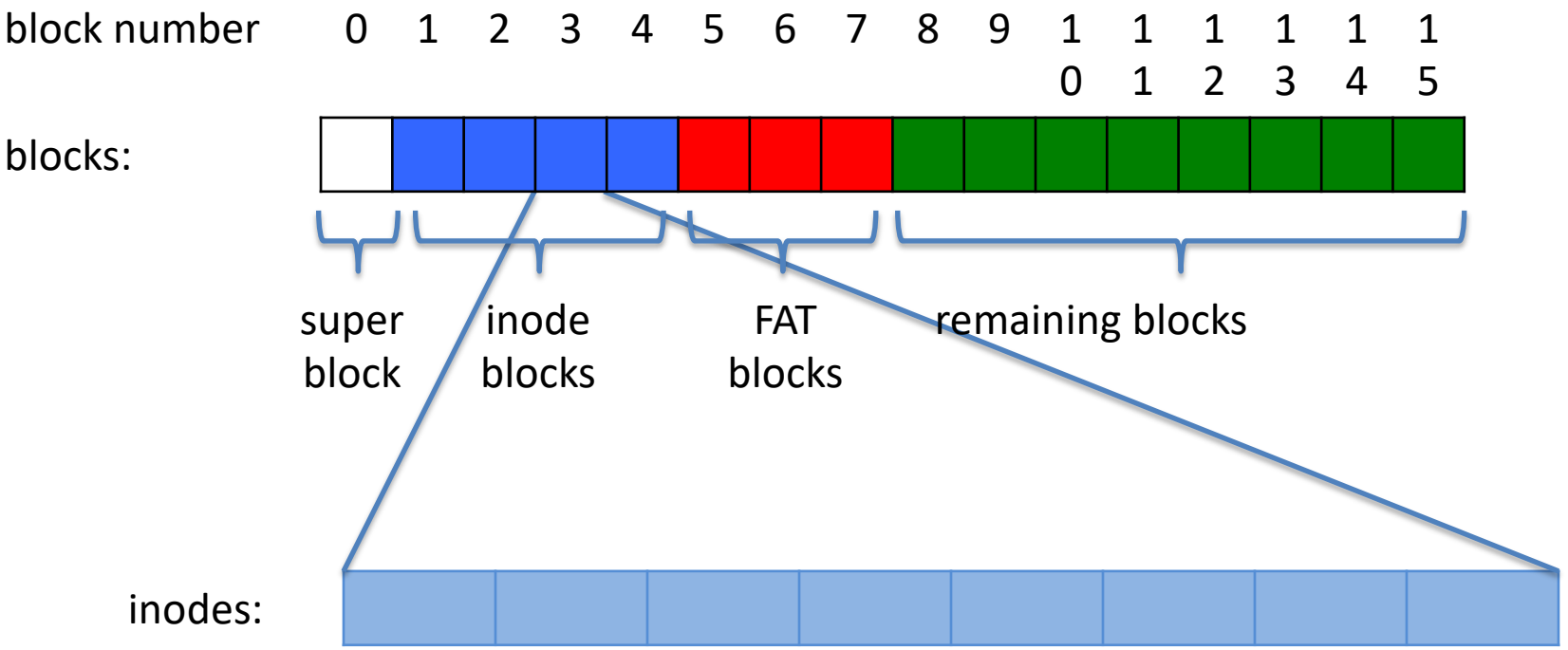
fatdisk i-node
(one per virtual block store)

```
struct treedisk_inode {
    block_no head;
            // block number of first block
            // should be 0 if nblocks == 0

    block_no nblocks;
            // #blocks in the virtual block store
};
```

fatdisk i-node block

```
#define INODES_PER_BLOCK    (BLOCK_SIZE /   \
         sizeof(struct fatdisk_inode))

struct fatdisk_inodeblock {
    struct fatdisk_inode inodes[INODES_PER_BLOCK];
};
```

fatdisk: i-node blocks



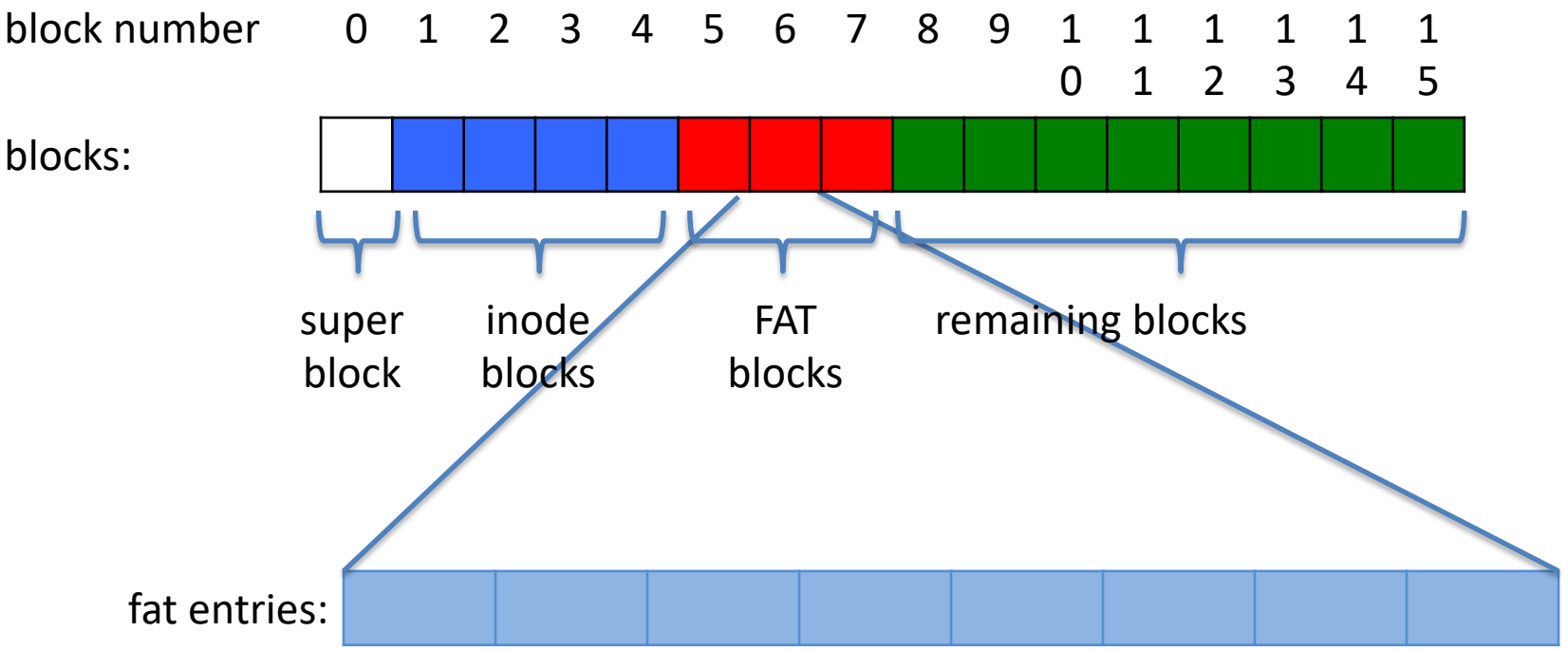
fatdisk fat-entry
(one per virtual block)

```
struct fatdisk_fatentry {
    block_no next;
      // next entry in the file or in the free list
      // 0 (or -1) for EOF or end of free list
};
```

fatdisk FAT block

```
#define FAT_PER_BLOCK        (BLOCK_SIZE /
            sizeof(struct fatdisk_fatentry))

struct fatdisk_fatblock {
    struct fatdisk_fatentry entries[FAT_PER_BLOCK];
};
```

fatdisk: FAT blocks



General purpose block

```
union fatdisk_block {
    struct fatdisk_superblock superblock;
    struct fatdisk_inodeblock inodeblock;
    struct fatdisk_fatblock fatblock;
    block_t datablock;
};
```

free list

- Essentially a file containing the unused blocks

```
struct fatdisk_superblock {
    block_no n_inodeblocks;
         // # blocks containing inodes

    block_no n_fatblocks;
         // # blocks containing fat entries

    block_no fat_free_list;
         // fat index of the first free fat entry};

    …
}
```

fatdisk.c

```
int fatdisk_create(block_store_t *below,
    unsigned int below_ino, unsigned int ninodes);
```

- initializes the fatdisk on-disk data structure
  - superblock, inode table, FAT table, free list

```
block_store_t *fatdisk_init(block_store_t *below,
    unsigned int below_ino);
```

- the fatdisk layer interface

Don’t overwrite existing file systems

```
int fatdisk_create(block_store_t *below,
                   unsigned int below_ino, unsigned int ninodes) {
  union fatdisk_block superblock;
  if ((*below->read)(below, below_ino, 0, (block_t *) &superblock) < 0) {
    return -1;
  }
  if (superblock.superblock.n_inodeblocks != 0) {
    printf("fatdisk: one already exists with %lu inodes\n",
      superblock.superblock.n_inodeblocks * INODES_PER_BLOCK);
    assert(superblock.superblock.n_inodeblocks >= n_inodeblocks);
    return 0;
  }
```

How do you change a byte in a block?