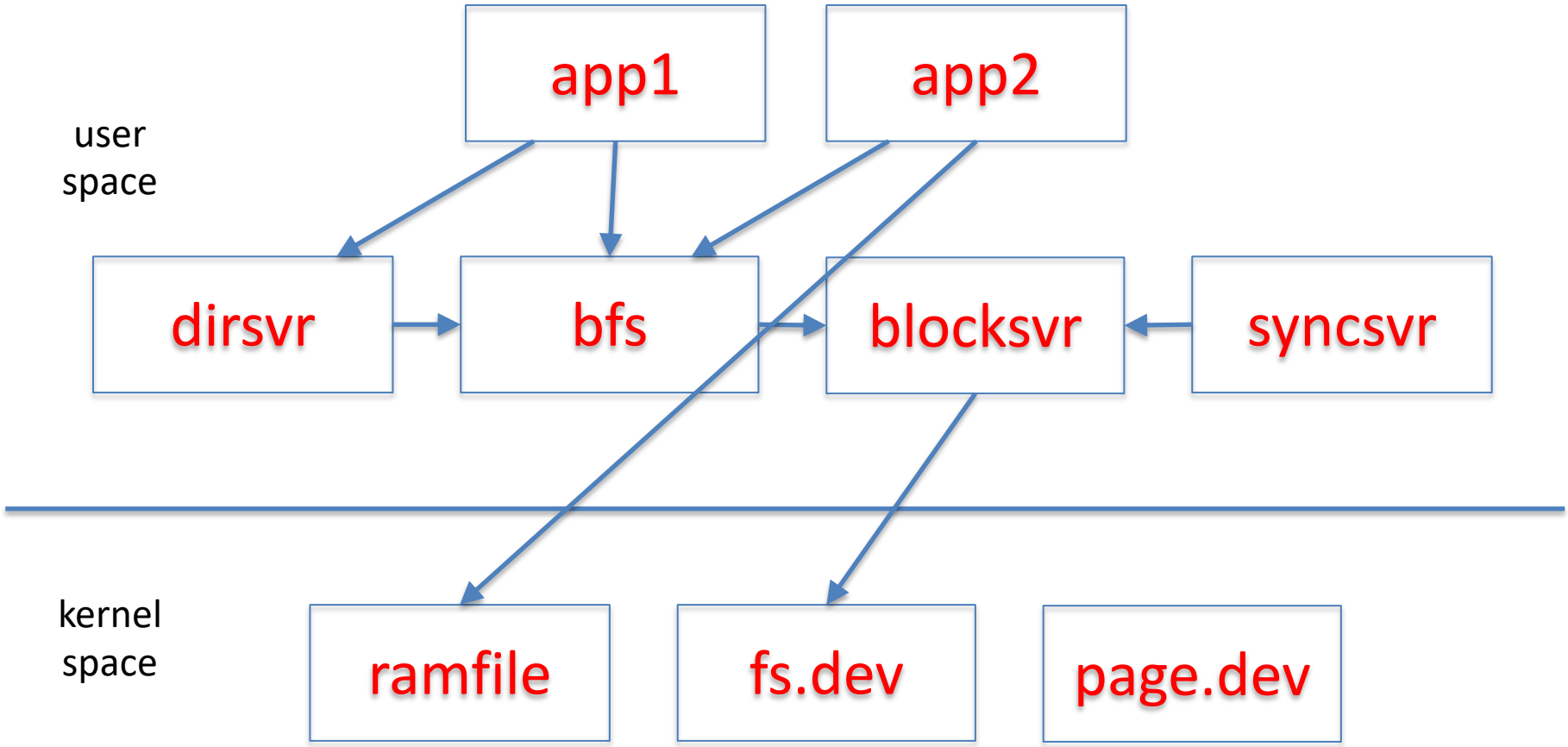
F.A.T. File System

Robbert van Renesse
Yunhao Zhang

File System

- Mapping a file name to its content and metadata
  - A file may be stored in HDD, SSD, or RAMDISK
  - Metadata like permission, owner, etc

EGOS Storage Architecture



Block Store Abstraction

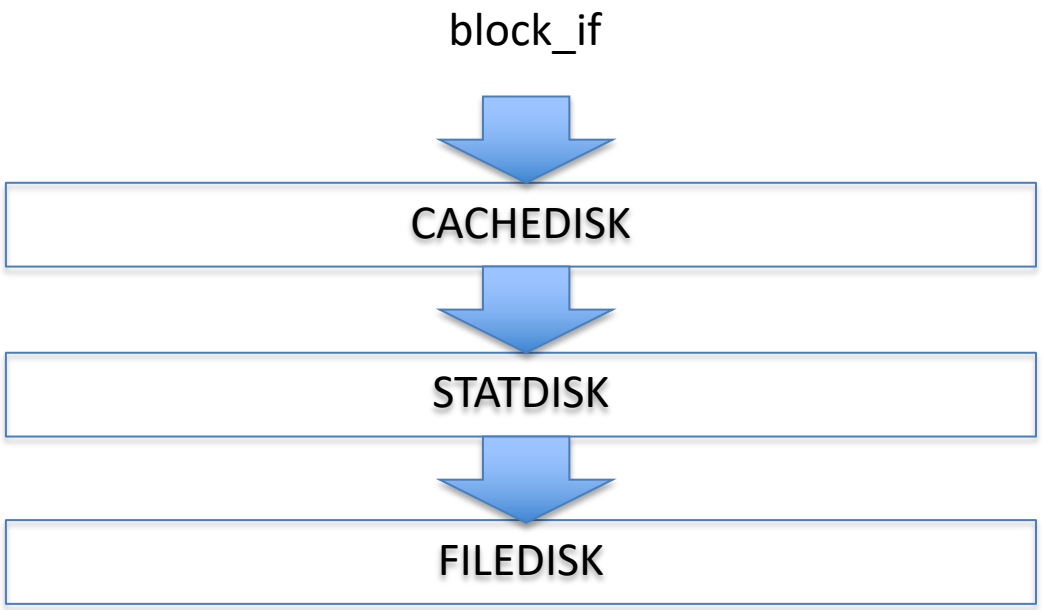
- A block store consists of a collection of *i-nodes*
- Each i-node is a finite sequence of *blocks*
- Simple interface:
  - block_t block
    - block of size BLOCK_SIZE
  - getninodes() → integer
    - returns the number of i-nodes on this block store
  - getsize(inode number) → integer
    - returns 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

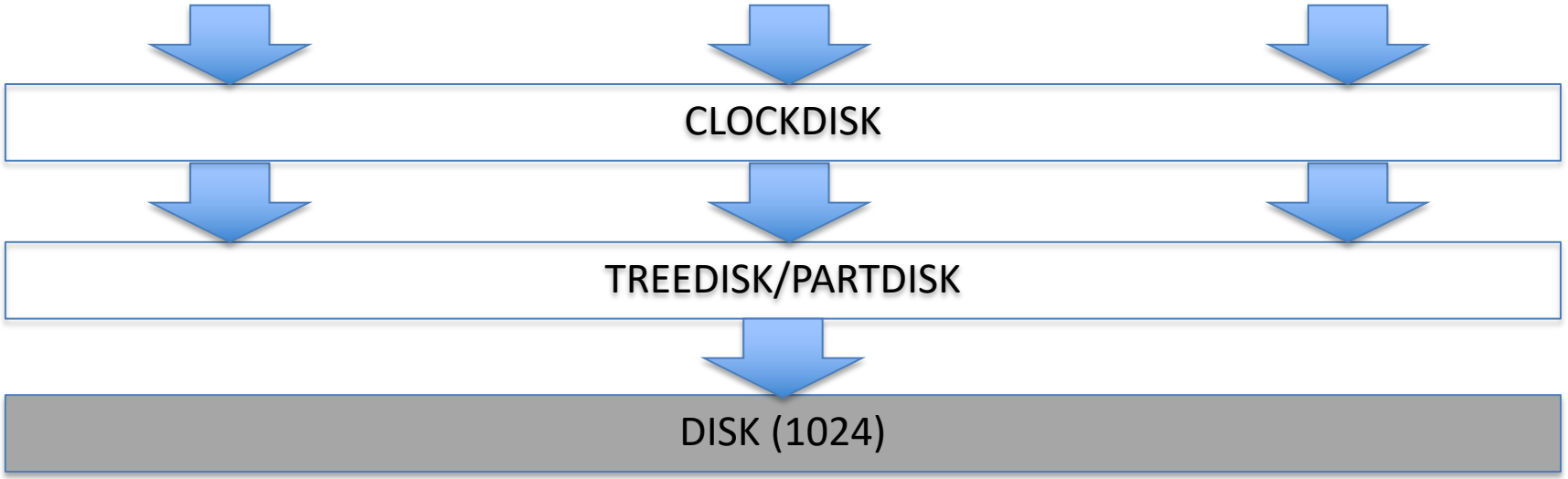
keeps track of #reads and #writes for statistics

keeps blocks in a Posix file

Multiplexing

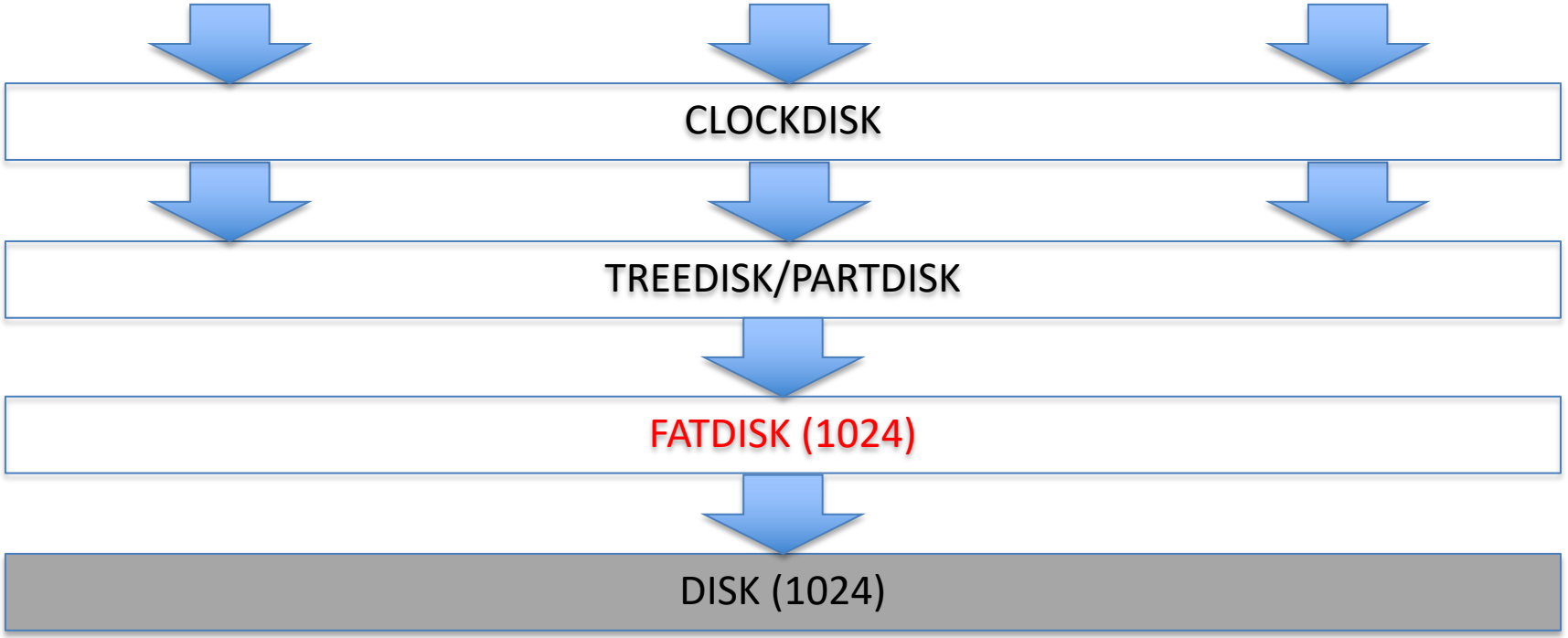
- A single block store can be “multiplexed”, offering multiple virtual block stores
- One way is simply partitioning the underlying block store into multiple disjoint sections
  - Treedisk
  - Partdisk

Partitioning



FAT file system

- Manage a disk using File Allocation Table



FAT file system

- Given an inode, our goal is to read all content (blocks) of that inode.
- All the information needed to locate the blocks belonging to an inode is **stored on disk**.

Linked List Allocation

Each file is stored as linked list of **blocks**

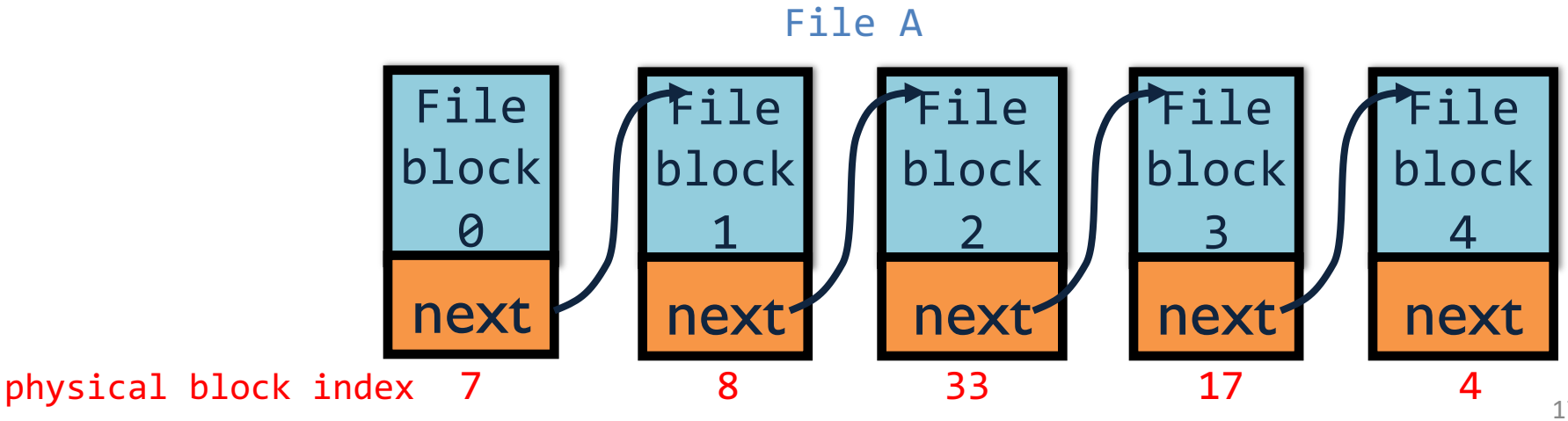
- First word of each block points to next block
- Rest of disk block is file data

+ 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)

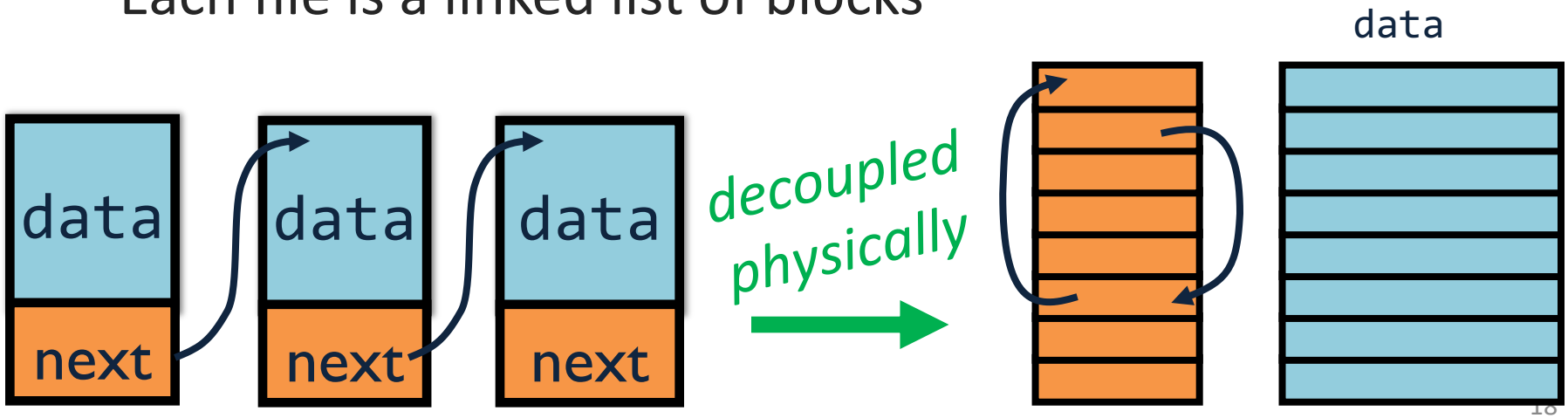
[late 70's]

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:

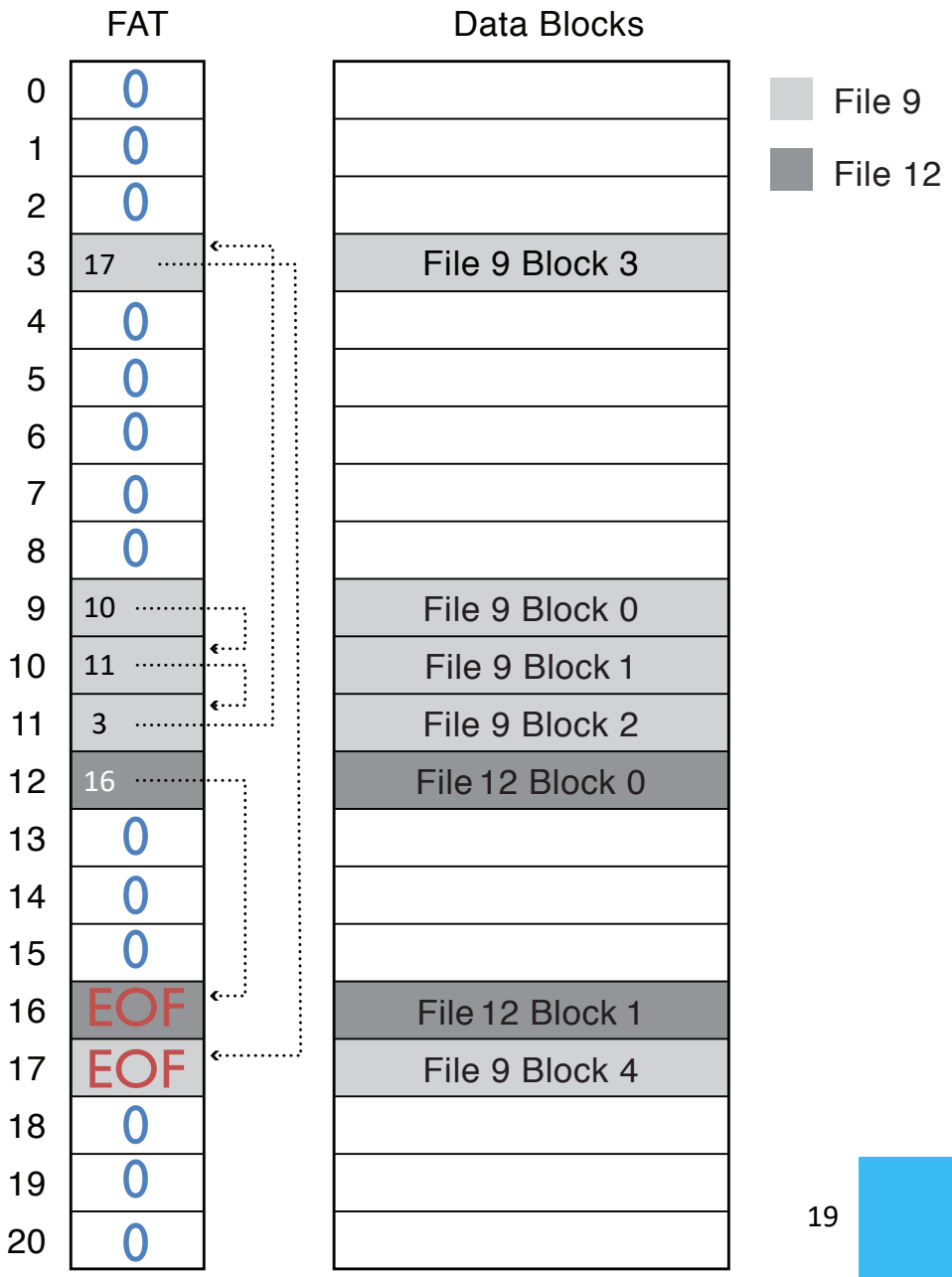
- Linear map of all blocks on disk
- Each file is a linked list of blocks



FAT File System

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

What is missing?



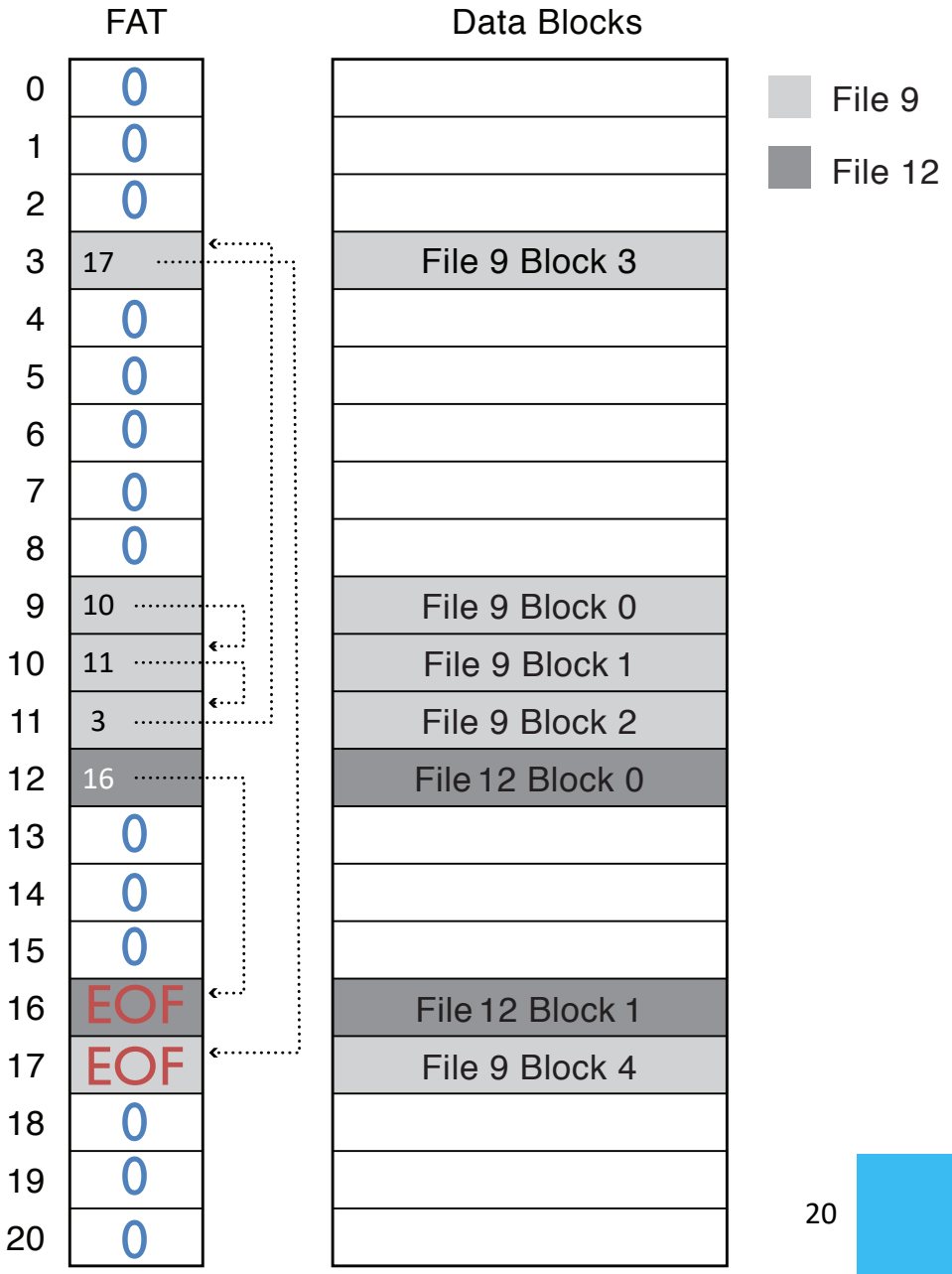
FAT File System

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

What is missing?

At file level
- start index of the file
- size

At disk level
- freelist
- size



P6:FAT disk layout

- 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 of blocks is given in the superblock
  4. the remaining blocks
     - *data blocks, free blocks*

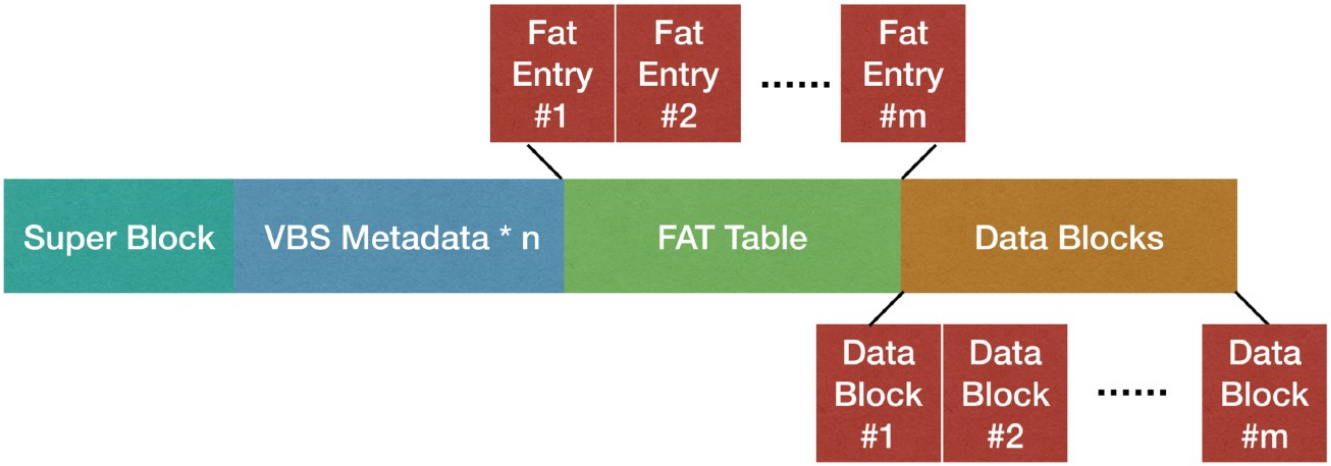
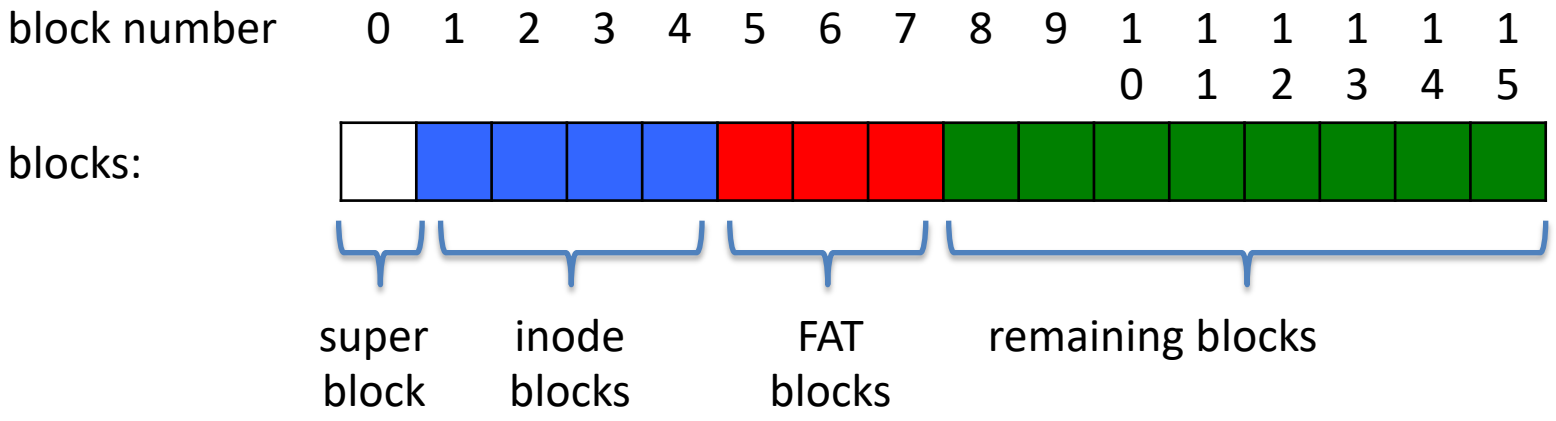


Figure 2: Physical Disk Layout

- **[Super Block]** This is the first block on the physical disk. It stores some meta-data such as the number of virtual block stores (i.e. n), the number of data blocks (i.e. m), and information keeping track of unused data blocks (i.e., the *free list*).
- **[VBS Metadata * n]** This region is usually called the *inode table*, where each *inode* stores the metadata of a virtual block store. Each inode contains a pair of integers: the size of the VBS (in number of blocks) and the data block index of the *first* data block in this VBS. For example, (5, 9) means that the VBS contains 5 blocks and the first block is stored in data block #9 of the "Data Blocks" region.
- **[FAT Table]** There are m FAT entries, corresponding to the m data blocks. Each VBS is organized as a linked list of FAT entries, and each FAT entry contains the *next* pointer of the linked list. In Figure 3, the size of VBS #1 is 5, stored in data block #9, #5, #23, #2, #8. In this case, FAT entry #9 would contain integer 5, FAT entry #5 would contain integer 23, etc. FAT entry #8 would contain 0 marking the end of the linked list. One may also use -1 instead of 0.
- **[Data Blocks]** Data blocks store the actual data of each VBS. Some data blocks hold valid data while others are *unused* and contain arbitrary content.


block number
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
blocks:
super block
inode blocks
FAT blocks
remaining blocks

Data structures

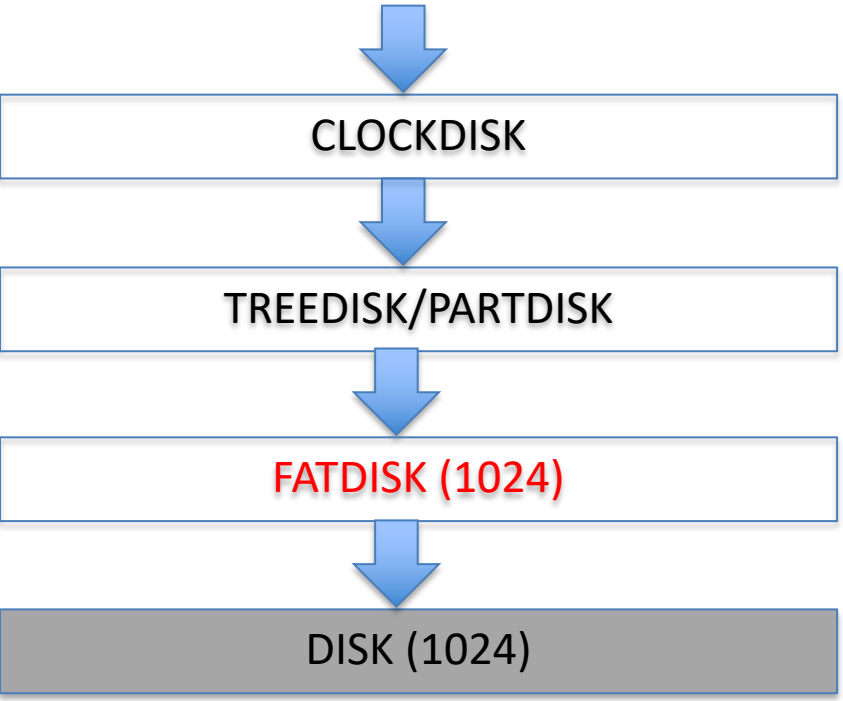
`src/block/fatdisk.h`

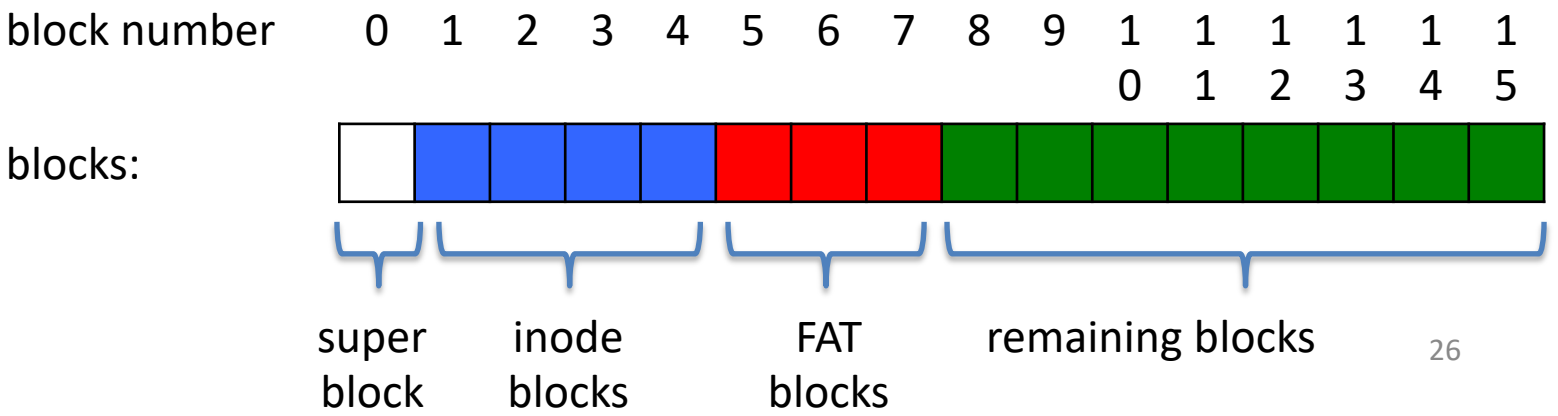
- struct fatdisk_superblock
- struct fatdisk_inode
- struct fatdisk_fatentry
- struct fatdisk_inodeblock
- struct fatdisk_fatblock

Implement 4 functions

- fatdisk_create(block_if below, unsigned int below_ino, unsigned int ninodes);
  - Use below->getsize(below_ino) to know the capacity of underlying disk.
  - Initialize the FAT layout described before.
- fatdisk_read(block_if this_bs, unsigned int ino, block_no offset, block_t *block);
  - Read content from (ino, offset)
- fatdisk_write(block_if this_bs, unsigned int ino, block_no offset, block_t *block);
  - Write content to (ino, offset)
- fatdisk_free_file(struct fatdisk_snapshot *snapshot, struct fatdisk_state *fs);
  - Free a file and return all its block back to freelist.

below_ino and **ino** are not the same!





Today

- P3, P5 due
- P6 release, due on May 12th

- Thanks for filling out the course evaluation!