Bilkent University
Department of Computer Engineering
CS342 Operating Systems

File Systems: Implementation

Last Update: May 16, 2023

Objectives and Outline

Objectives

- To describe the details of implementing local file systems and directory structures
- To describe block allocation methods and free-block algorithms and trade-offs

Outline

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

File System Design

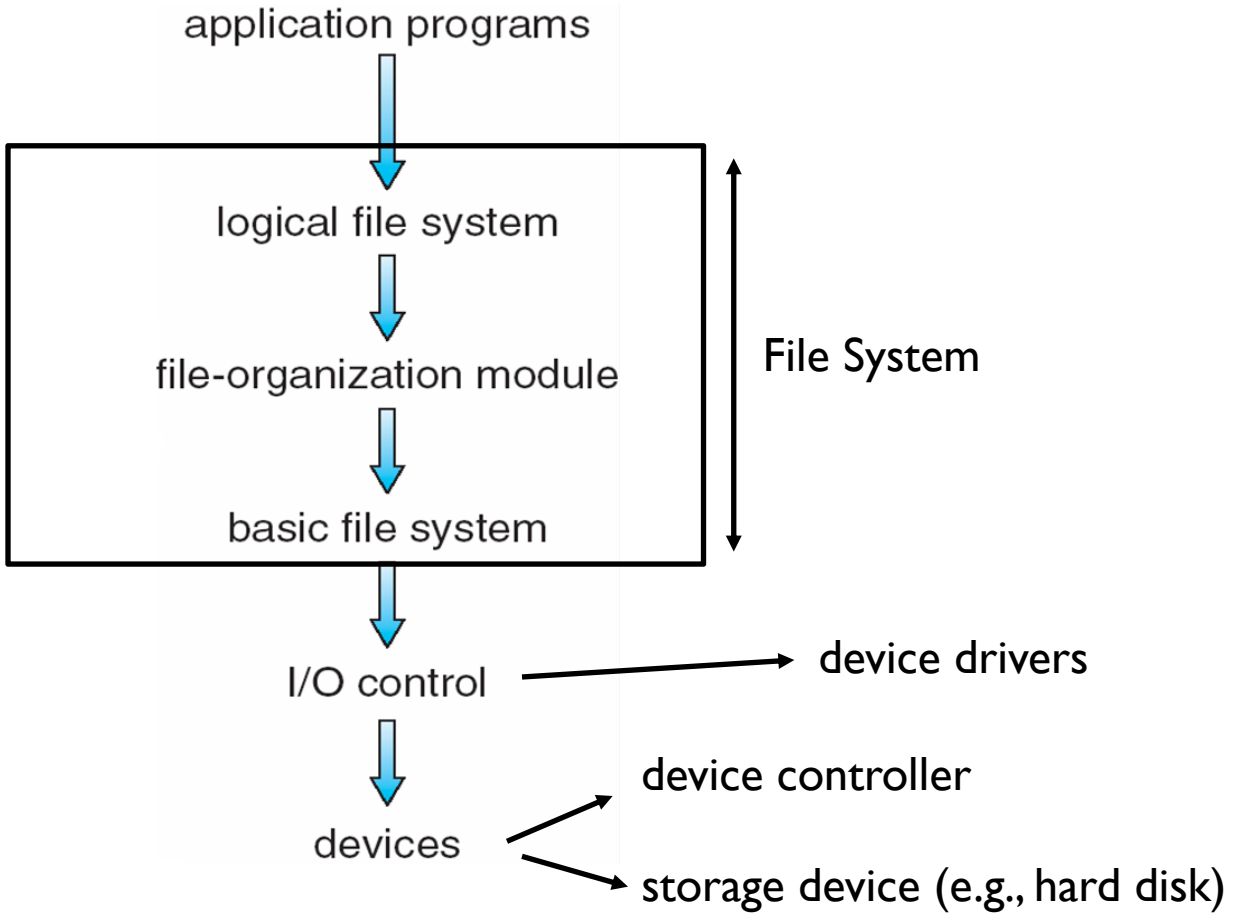
- File System Design/Implementation Involves
  - 1) Defining File System Interface.
    - How file system looks to the user.
    - What are the attributes.
    - What are the operations.
    - (logical) Directory structure.
  - 2) Designing and Implementing File System
    - Designing algorithms.
    - Designing data structures (in-memory and on-disk).
    - Mapping logical file system to physical storage device (disk, etc.).

File System Structure - Overview

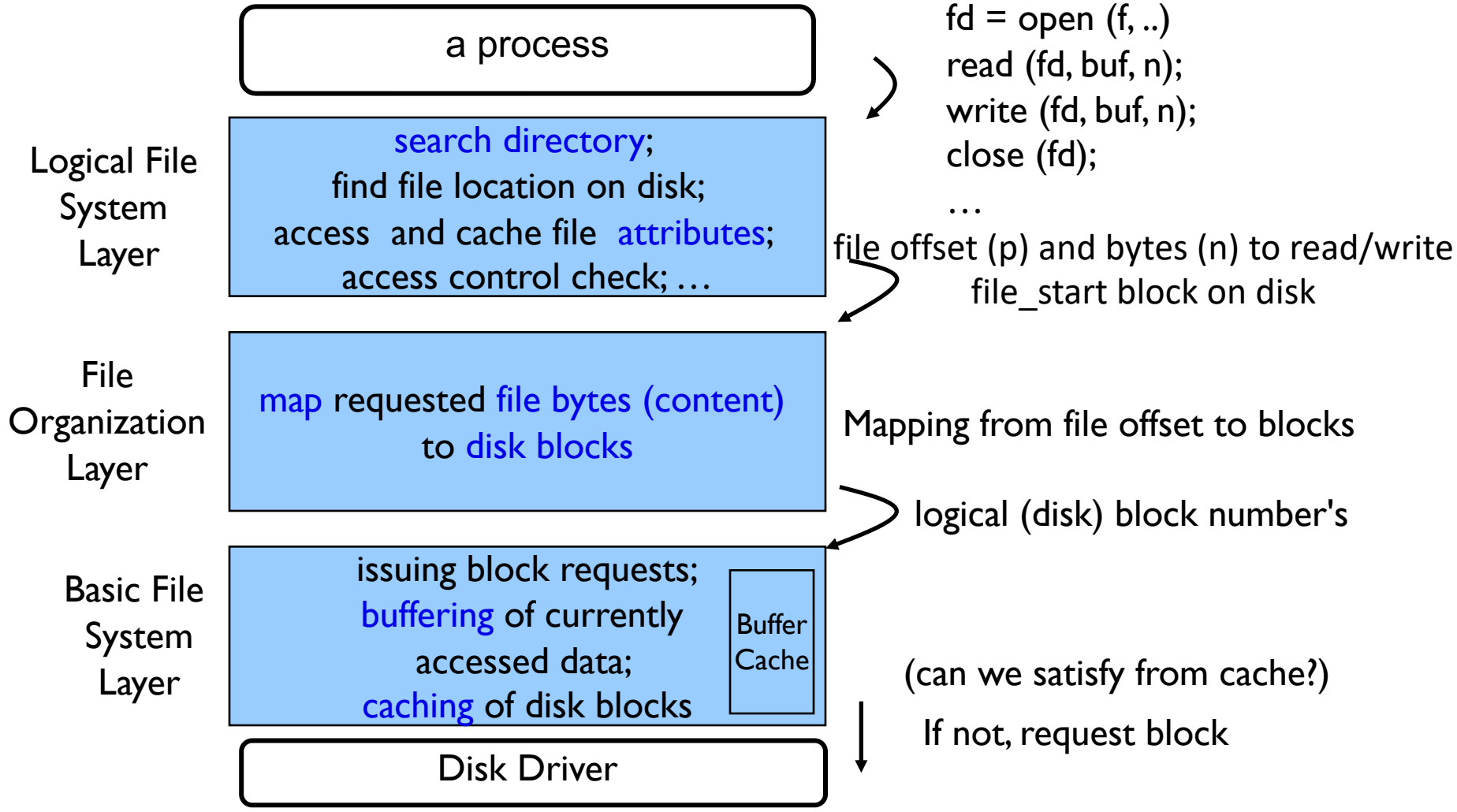
- File: logical storage unit (stores related information)
- File system structures and data reside on secondary storage (disks). Persistent.
  - FS provides efficient and convenient access to disk by allowing data to be stored, located and retrieved easily
  - Can sit on various storage media (HDD, SSD, USB disk, CD-ROM, etc.).
- File control block (inode): structure on disk that contains information about a file: file attributes.
- Device driver controls the physical device.

Layered File System

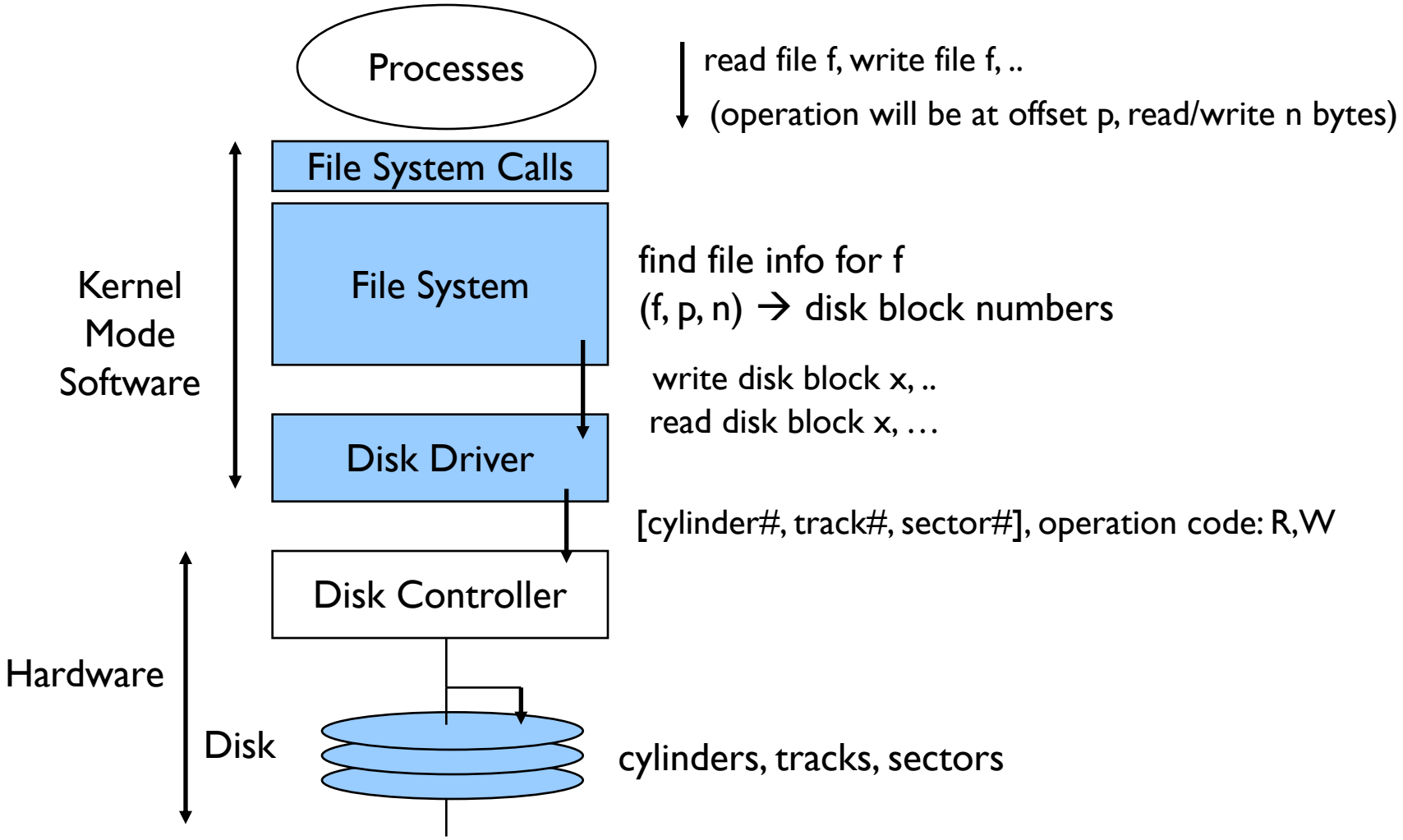
File system (FS) organized into logical layers



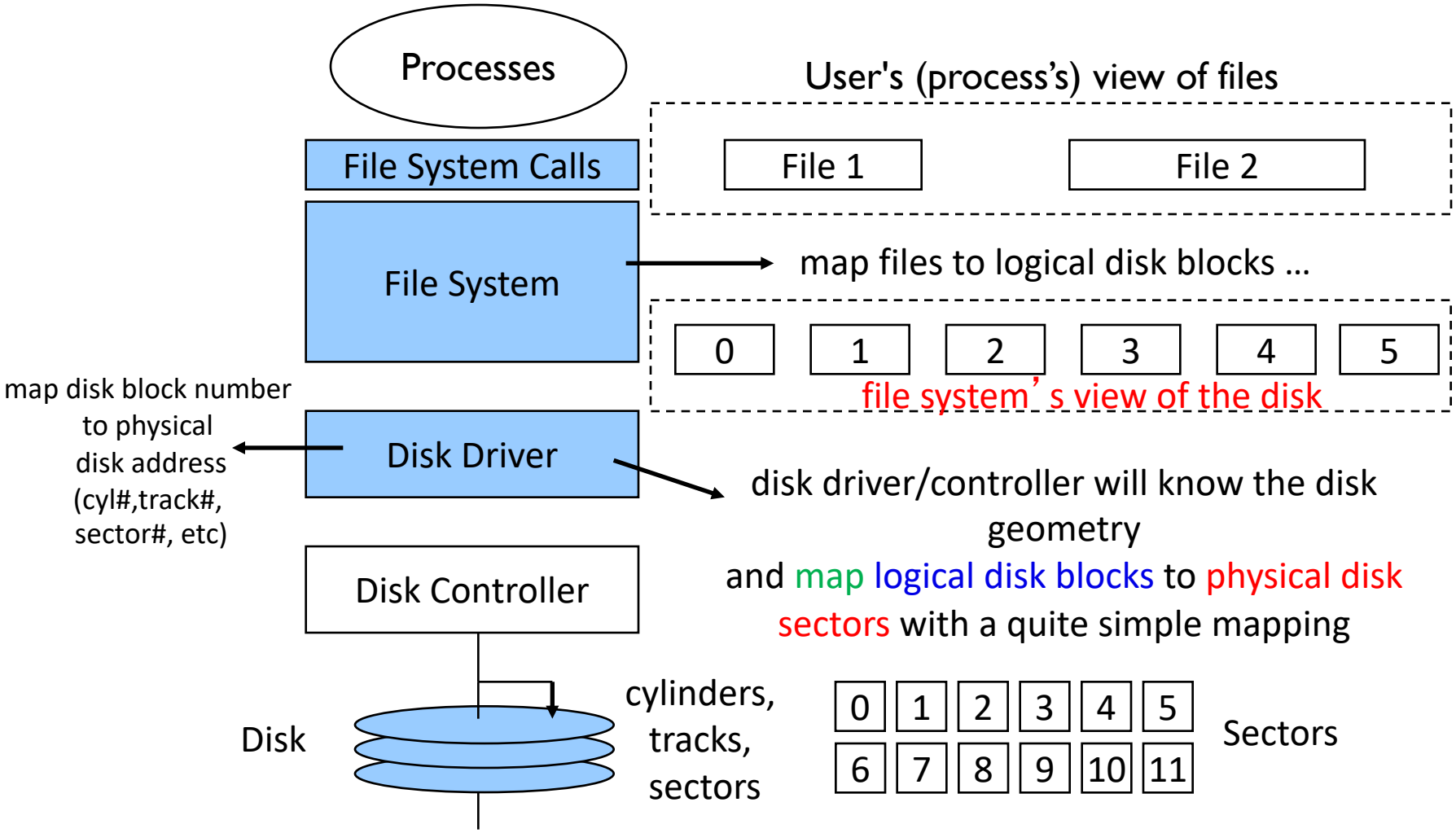
Layered File System



Layered Software



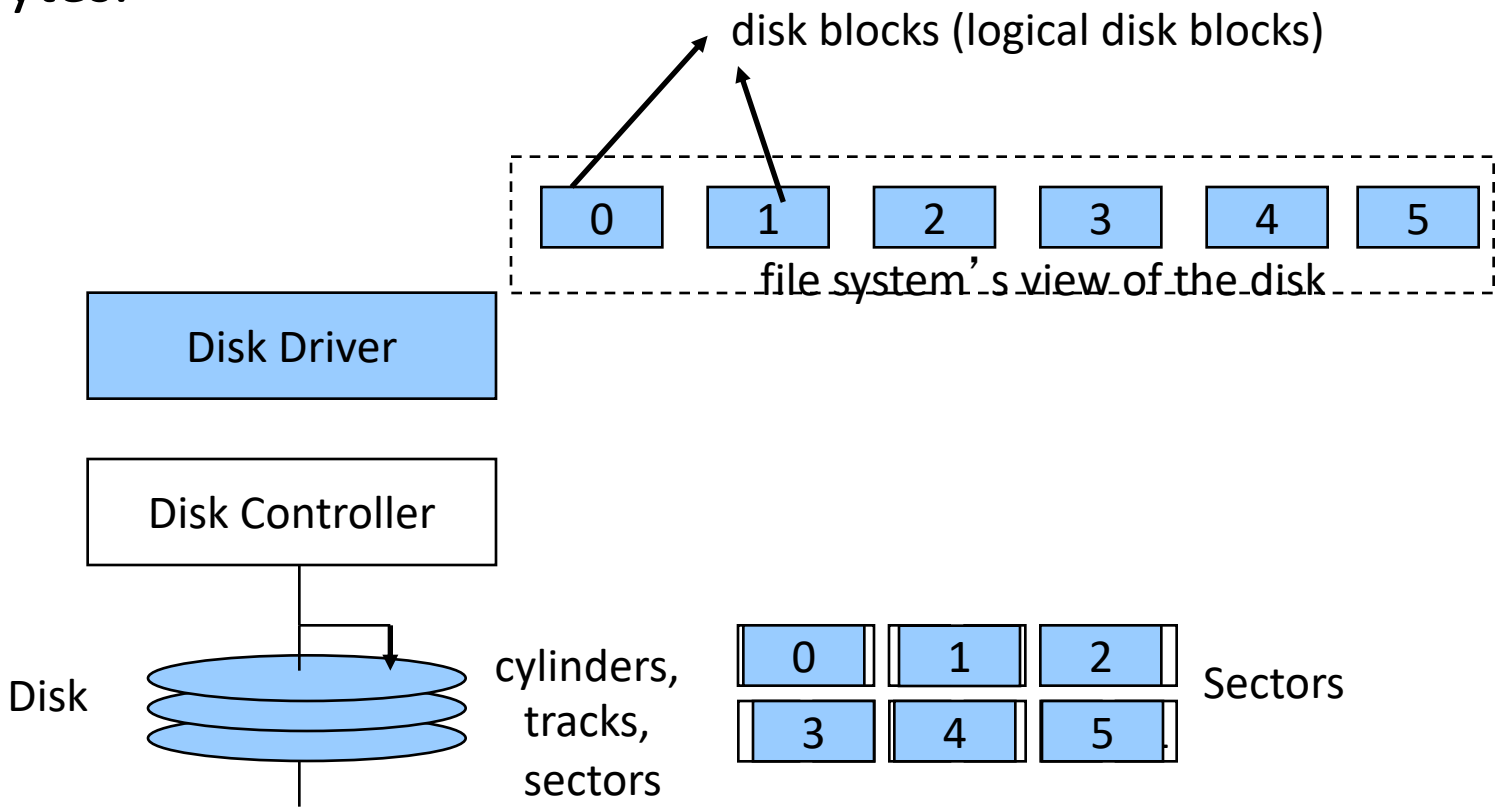
Layered Software



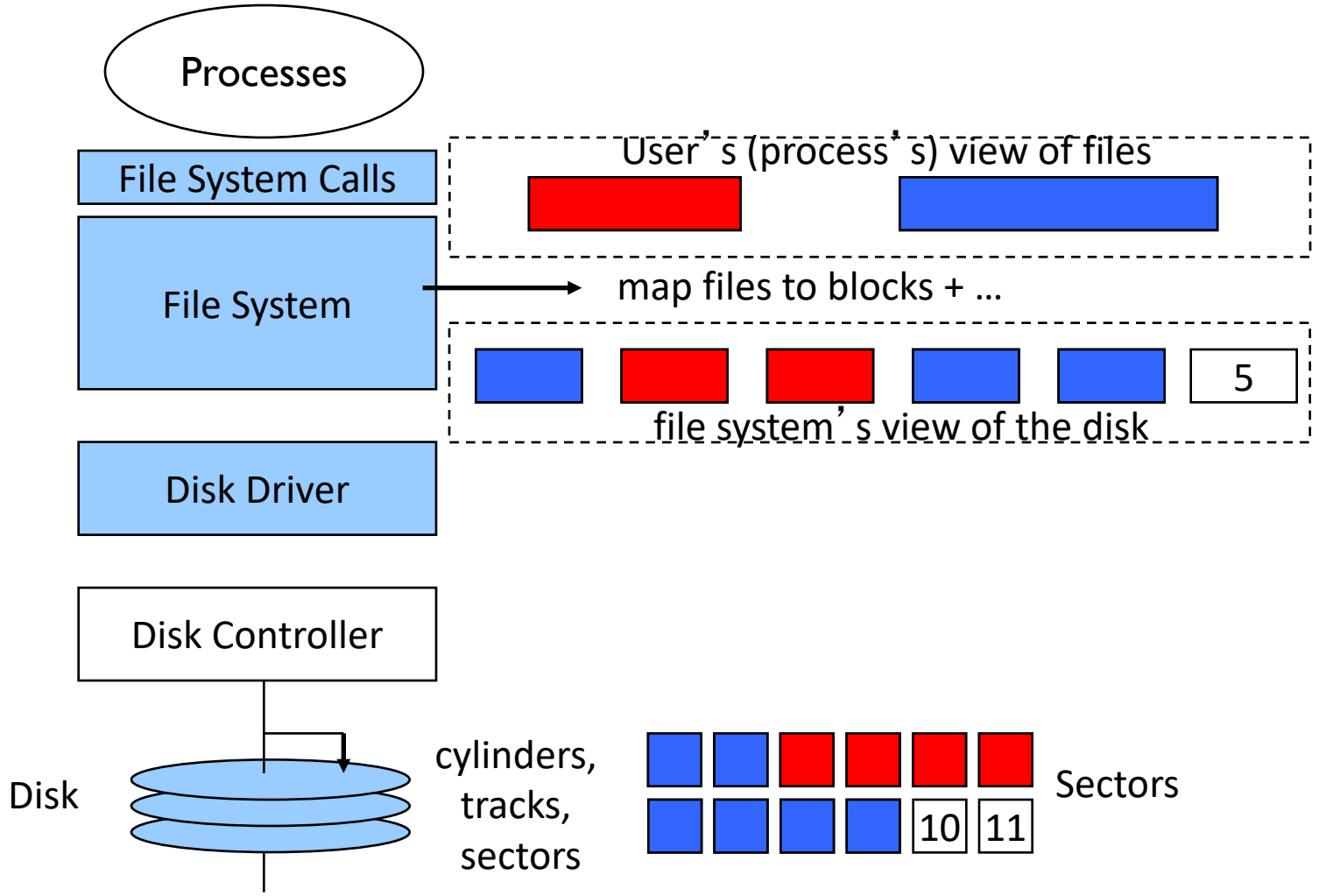
Disk Driver: Mapping disk blocks to physical disk sectors

Block size is a multiple of sector size.

Example: sector size can be 512 bytes; block size can be 1024 bytes, or 4096 bytes.

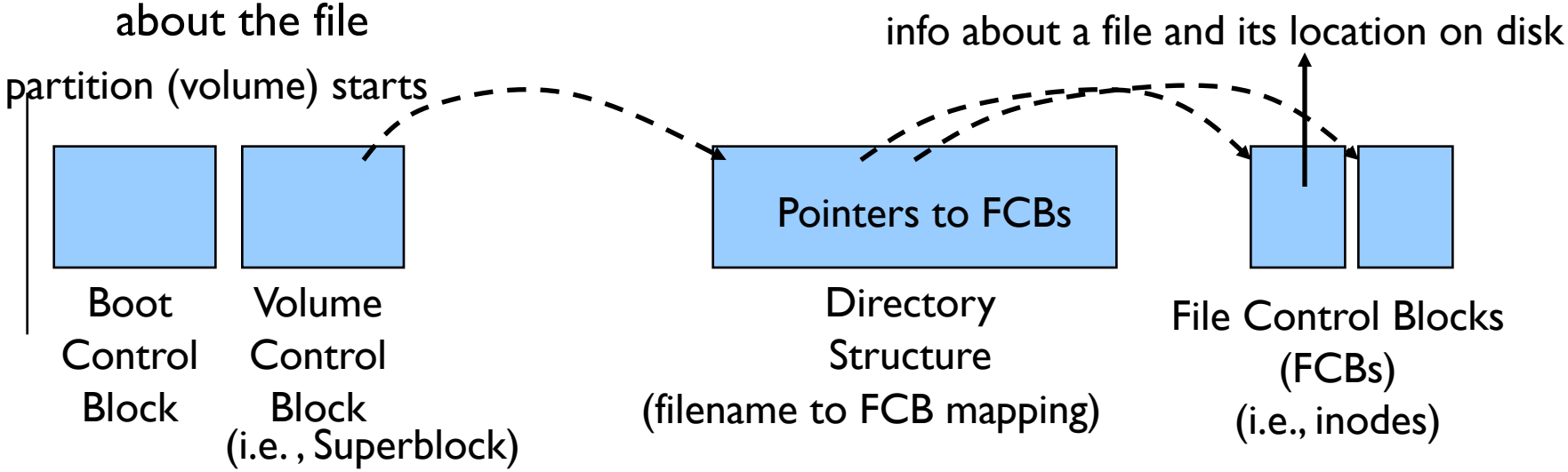


Example mapping files to blocks and sectors



File System Implementation

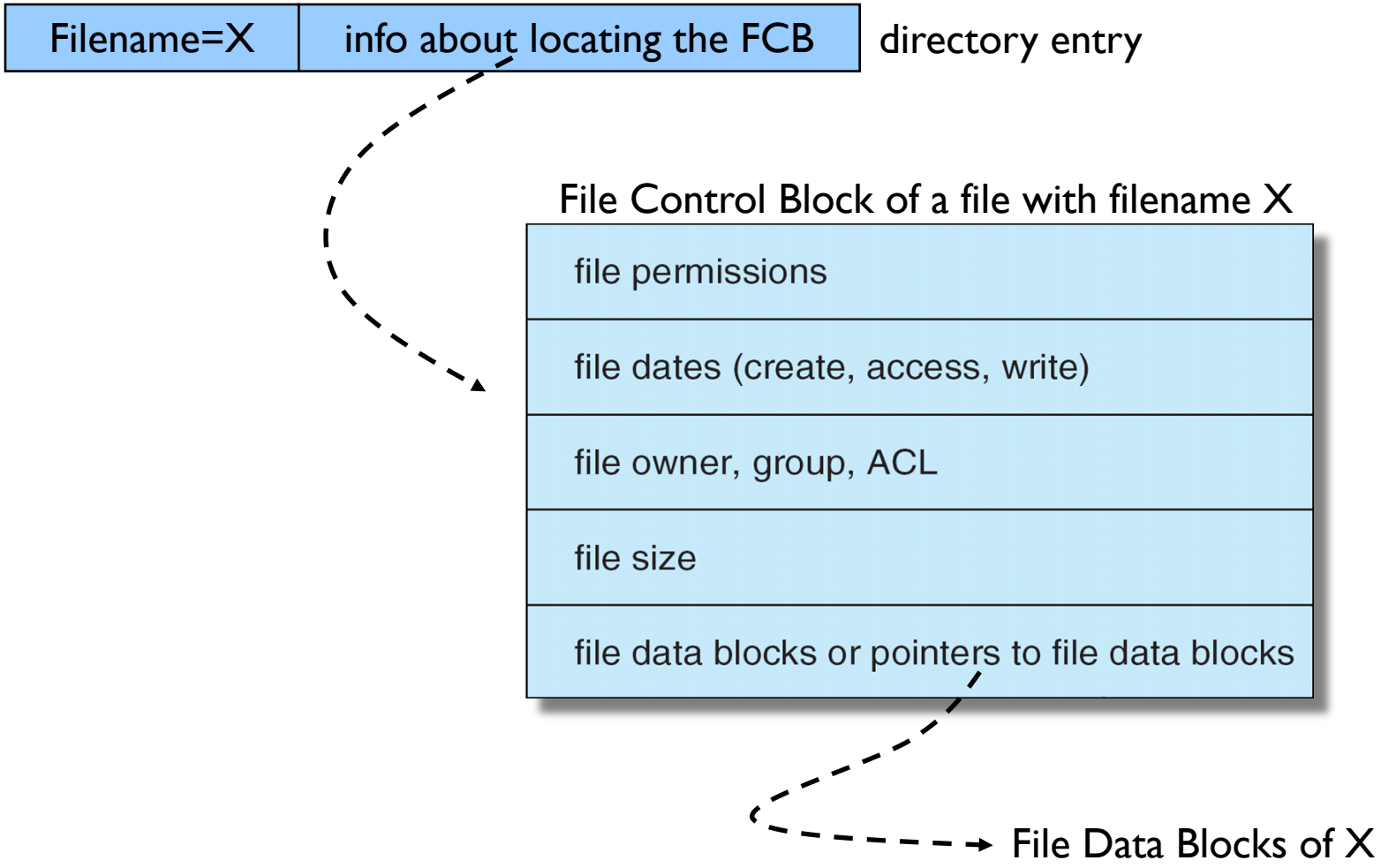
- Major On-disk Structures and Information
  - Boot control block contains info needed by system to boot OS from that volume
  - Volume control block contains volume details (superblock)
  - Directory structure organizes the files
  - Per-file File Control Block (called FCB or inode) contains many details about the file



Superblock in Linux Ext3 File System

```
/*
 * Structure of the super block
 */
struct ext3_super_block {
/*00*/   __le32  s_inodes_count;         /* Inodes count */
         __le32  s_blocks_count;         /* Blocks count */
         __le32  s_r_blocks_count;       /* Reserved blocks count */
         __le32  s_free_blocks_count;    /* Free blocks count */
/*10*/   __le32  s_free_inodes_count;    /* Free inodes count */
         __le32  s_first_data_block;     /* First Data Block */
         __le32  s_log_block_size;       /* Block size */
         __le32  s_log_frag_size;        /* Fragment size */
/*20*/   __le32  s_blocks_per_group;     /* # Blocks per group */
         __le32  s_frags_per_group;      /* # Fragments per group */
         __le32  s_inodes_per_group;     /* # Inodes per group */
         __le32  s_mtime;                /* Mount time */
…
…
}
```

A Typical File Control Block

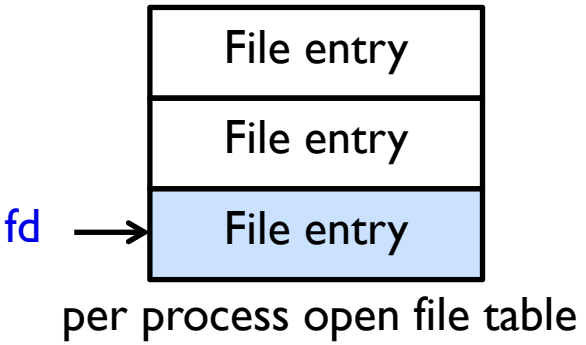


File Types

- Various things can be considered as files:
  - Regular files
    - The ascii files (text files) we use, binary files, .doc files, .pdf files, executable files, etc.
    - Some programs can look to them and understand what they are. They store content
  - Directories
    - A file can store directory information. Hence directories can be considered as special files (in some systems like Linux).
    - We will have a file control block for such a file as well.
  - Device files
    - We can refer to devices in the system with files.
    - Device file "/dev/sda5" may refer to a hard disk partition.
    - "/dev/fd0" may refer to floppy disk. "/dev/cdrom0" may refer to CDROM.
  - …

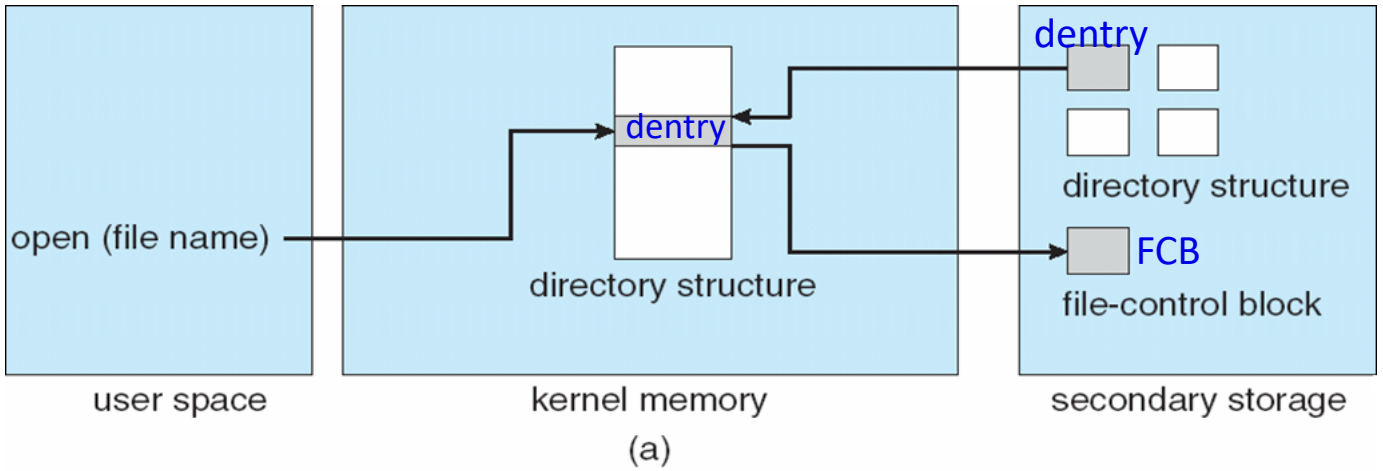
In Memory File System Structures

- There are also in-memory structures used by the file system
- While files are opened and used, file system (kernel) keeps information in memory
  - Information about file is brought from disk into memory
  - They are put into in-memory (data) structures
- Some of these structures are:
  - Directory entry
  - FCB (inode)
  - Per process open file table entry (file descriptor points to this)
    - fd = open(filename, …)
  - Data blocks cached
    - In "buffer cache" in memory



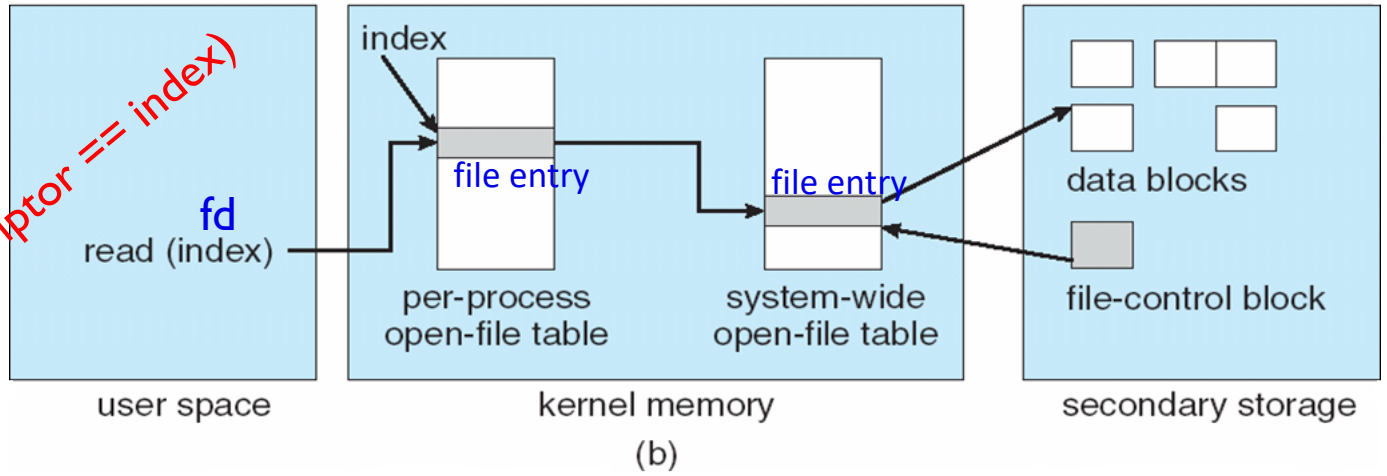
In-Memory File System Structures

Opening a file with filename. Returns a file descriptor (index)



reading a file with file descriptor

(file handle == file descriptor == index)



Directory Implementation

- How a directory (i.e., subdirectory) is implemented (stored in disk)?
- In Linux, a directory (i.e., a subdirectory) information is kept in a file.
  - Hence a folder (subdirectory) is represented in a file in Linux
  - However, it is a special file: not a regular Linux file storing application (user) information.
  - It is a file of type "DIR".
  - It has also an FCB (inode). Has inode number.
  - Is has a directory entry (in the parent directory).
- Each folder (subdirectory) is another file (of type DIR) inLinux FS.

Directory Implementation

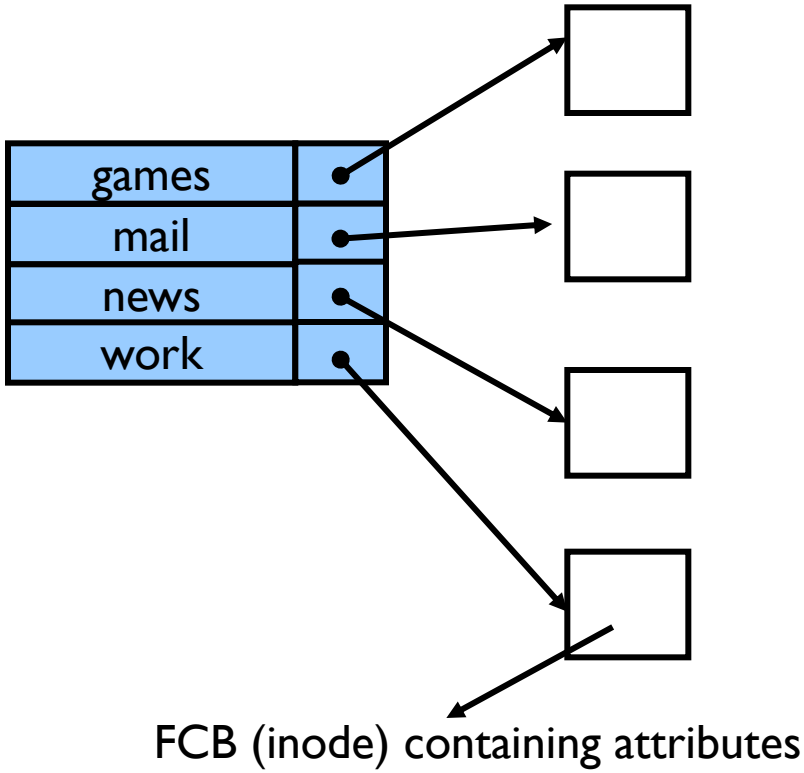
- Internal organization alternatives: 1) linear list; 2) hash table
- Linear list of file names with associated information (like inode number).
  (i.e, linear list of directory entries - dentries).
  - Simple to program
  - time-consuming to execute (search takes time)
- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - two file names may hash to the same location
  - fixed size

Directory Implementation: directory entries

games	attributes
mail	attributes
news	attributes
work	attributes

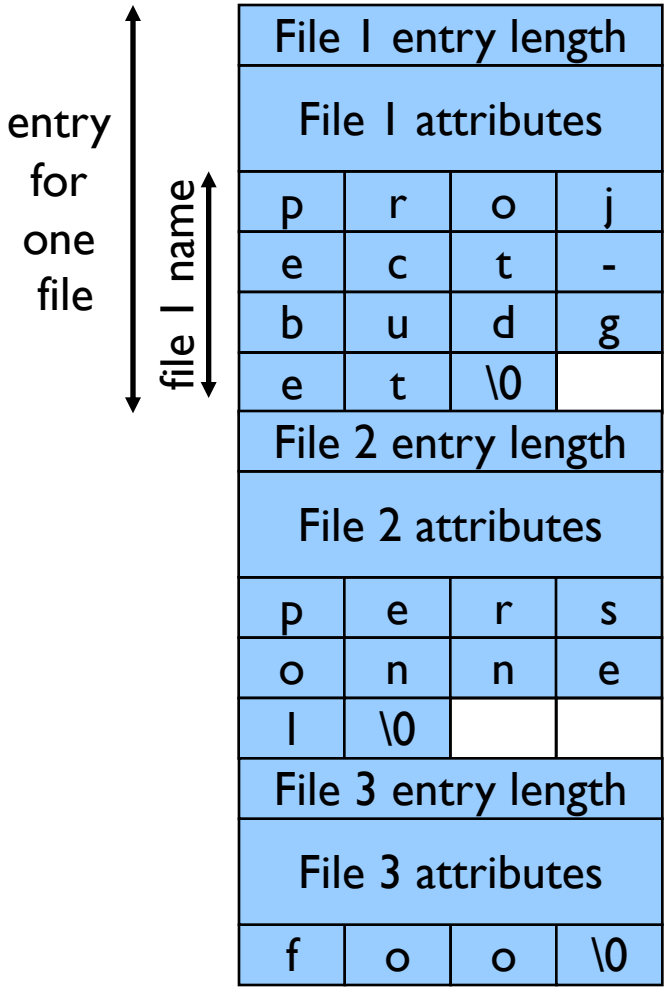
a directory with fixed sized entries

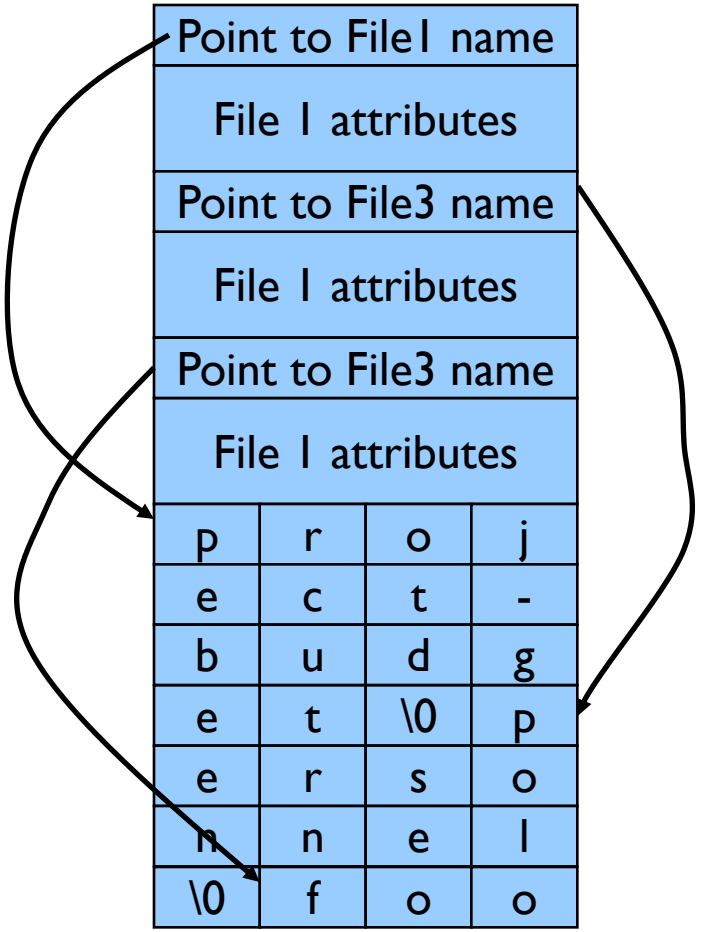
attributes include location info for data blocks of the file



Using fixed sized names

Directory Implementation: handling long filenames

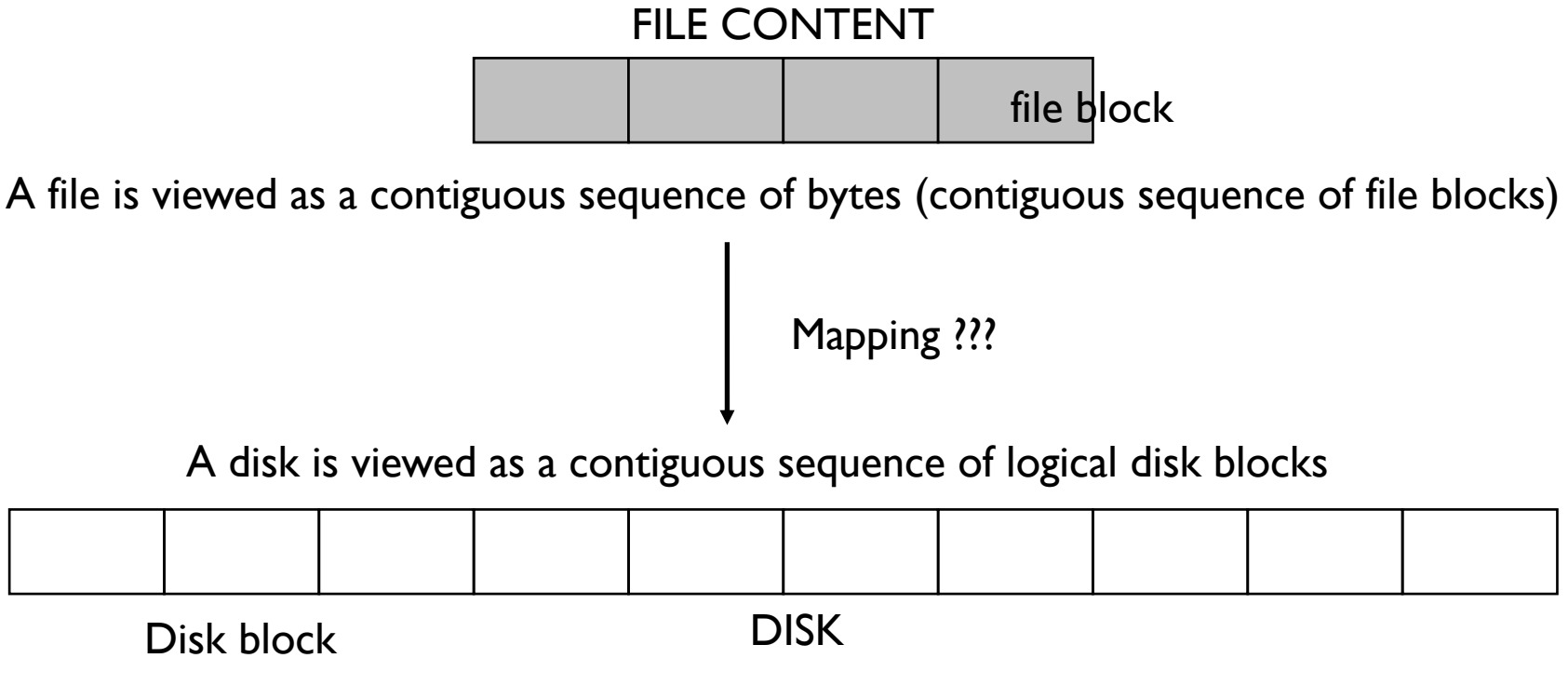




Using variable sized names

Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
  - Allocation of disk space to files
    - Deciding where the file content will sit on disk
  - Keeping track the disk blocks allocated to files



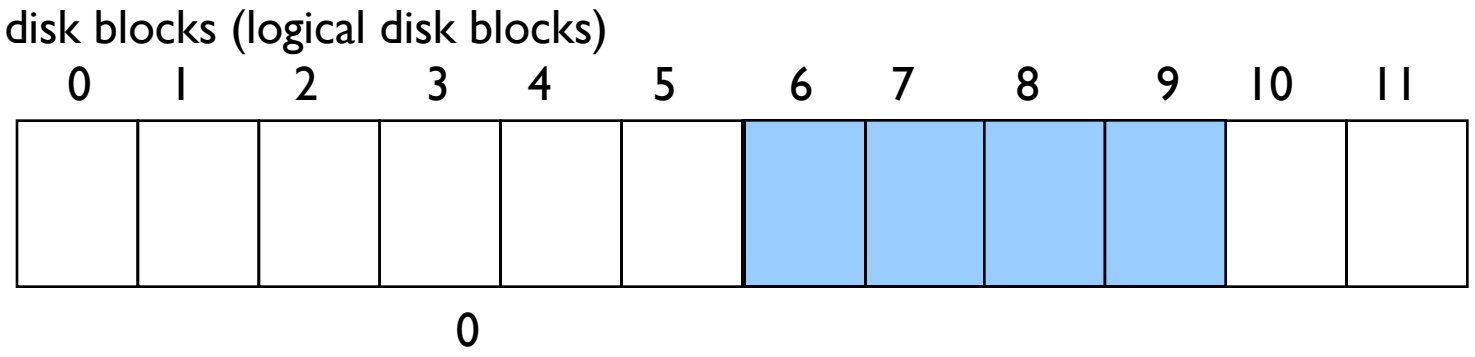
Allocation Methods

- There are 3 general methods.
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Contiguous Allocation

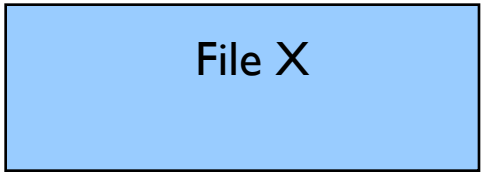
- Each file occupies a set of contiguous disk blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required to find out the disk blocks containing file data/content
- Random access is fast
- Wasteful of space (dynamic storage-allocation problem) (external frag.)
- Files cannot grow

Start address = 6
Number of blocks = 4

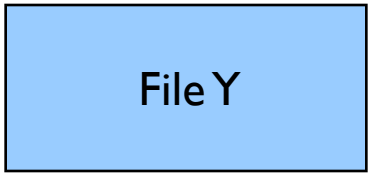


Example

offset 0



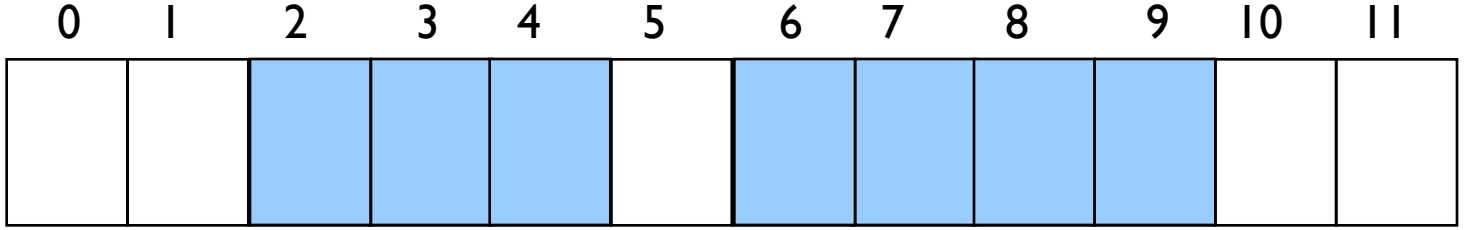
offset 0



File X: start=6, size_in_disk_blocks=4

File Y: start=2, size_in_disk_blocks=3

disk blocks (logical disk blocks)



Contiguous Allocation

file_offset: logical address into a file (i.e., offset of a byte in file)
(first byte has offset 0)

Mapping from file offset to disk address (mapping algorithm):

file_offset / disk_block_size

- Q = file_offset div disk_block_size
- R = file_offset mod disk_block_size

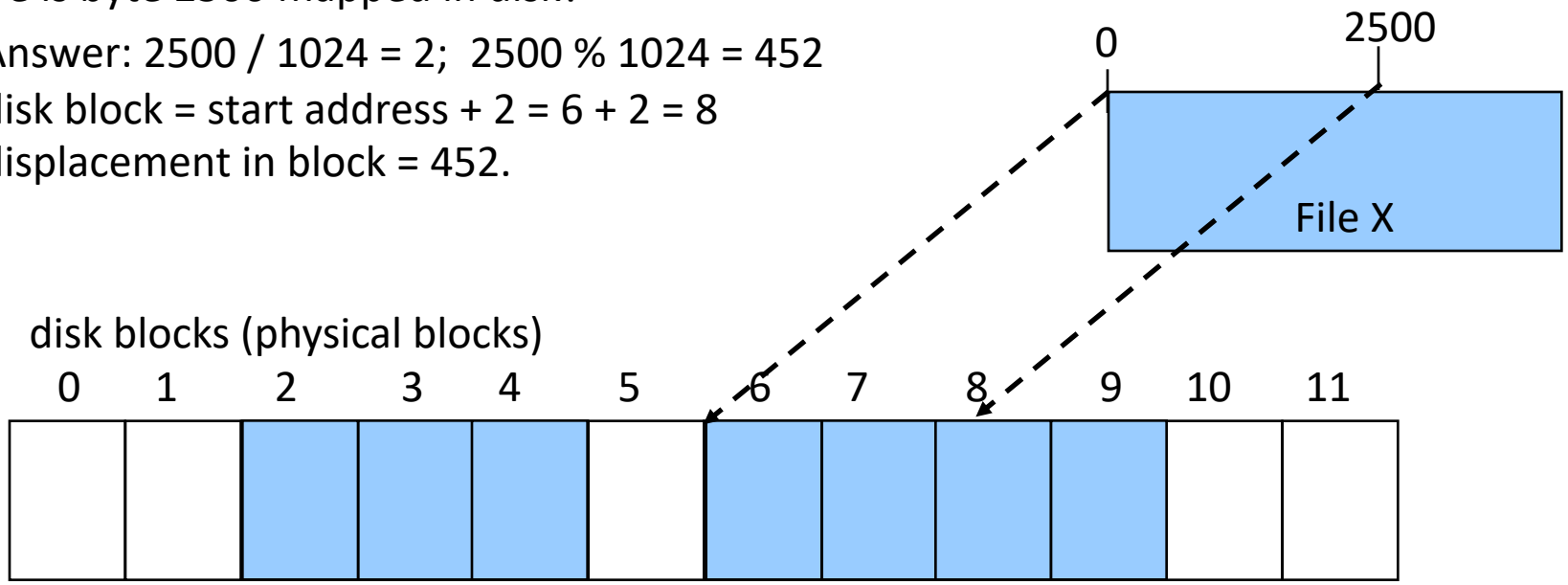
Disk Block to be accessed = Q + starting disk block number (address)
Displacement into disk block = R

Example

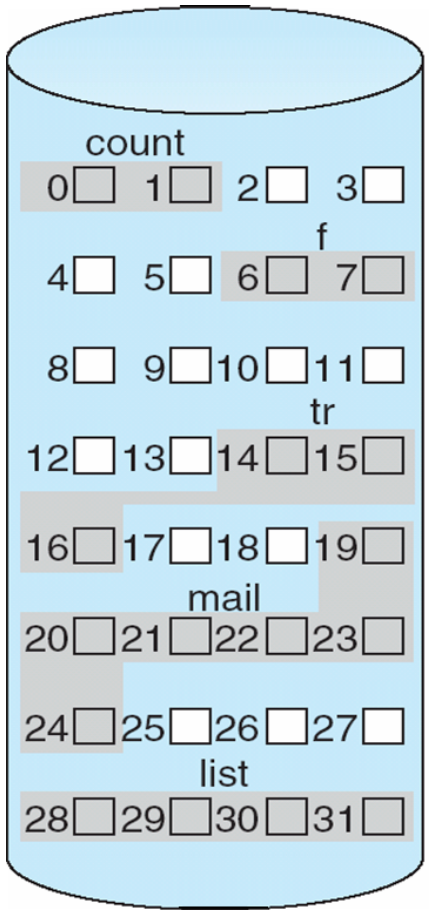
- Assume block size = 1024 bytes
- Which disk block contains the byte 0 of file X (file offset = 0)? What is the displacement inside that block?
  - Answer : disk block = 6, displacement (disk block offset) = 0
- Which disk block contains the byte at file offset 2500? In other words,

where is byte 2500 mapped in disk?

Answer: 2500 / 1024 = 2; 2500 % 1024 = 452
disk block = start address + 2 = 6 + 2 = 8
displacement in block = 452.



Contiguous Allocation of Disk Space



directory

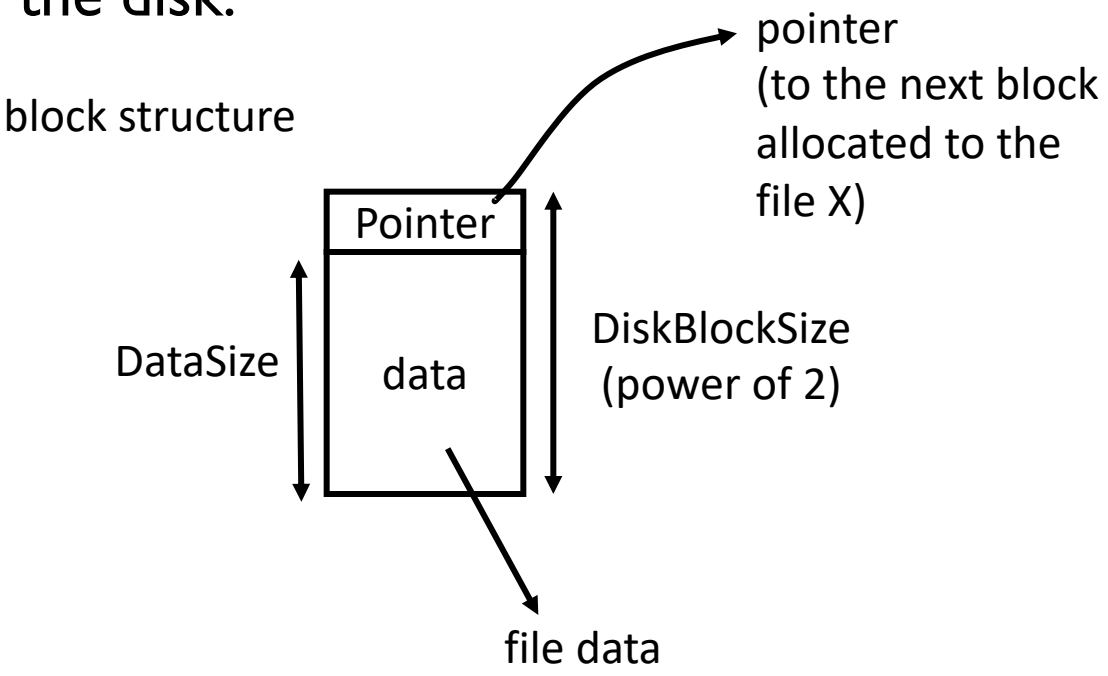
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

- Many newer file systems use a modified contiguous allocation scheme.
- Extent-based file systems allocate disk blocks in extents
- An extent is a contiguous blocks of disk
  - Extents are allocated for a file.
  - A file consists of one or more extents
  - A file may start with a single extent.
- Linux ext4 filesystem is also using extents.

Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.

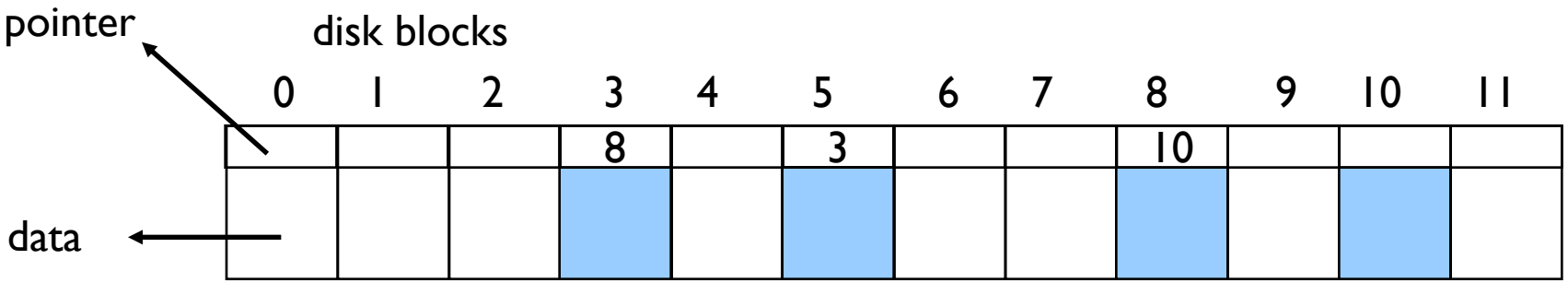


File data size in a disk block is no longer a power of 2

Linked Allocation

File X

File starts at disk block 5



Linked Allocation (Cont.)

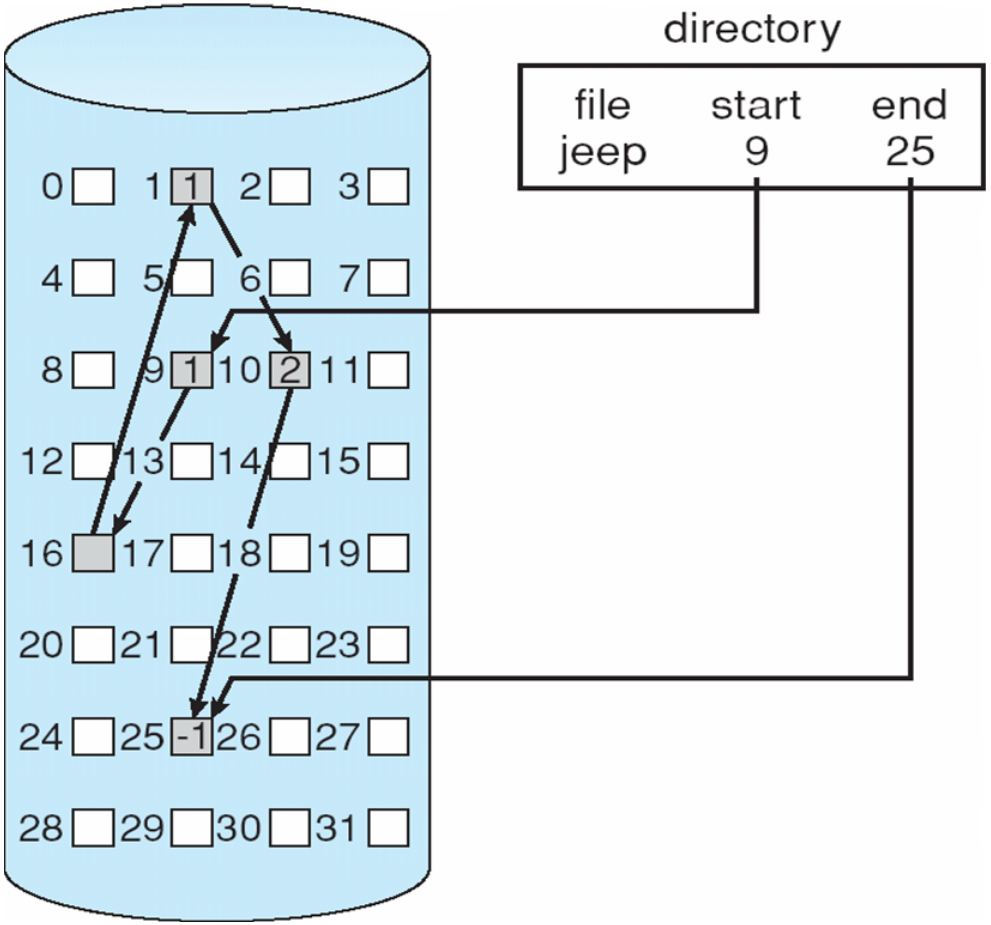
- Simple – need only starting address
- Disk space is used efficiently- no waste of space (no external fragmentation)
- No random access (random access is not easy)
- Mapping Algorithm:

file_offset / (BlockSize - PointerSize)

- Q (integer division result: quotient)
- R (remainder)

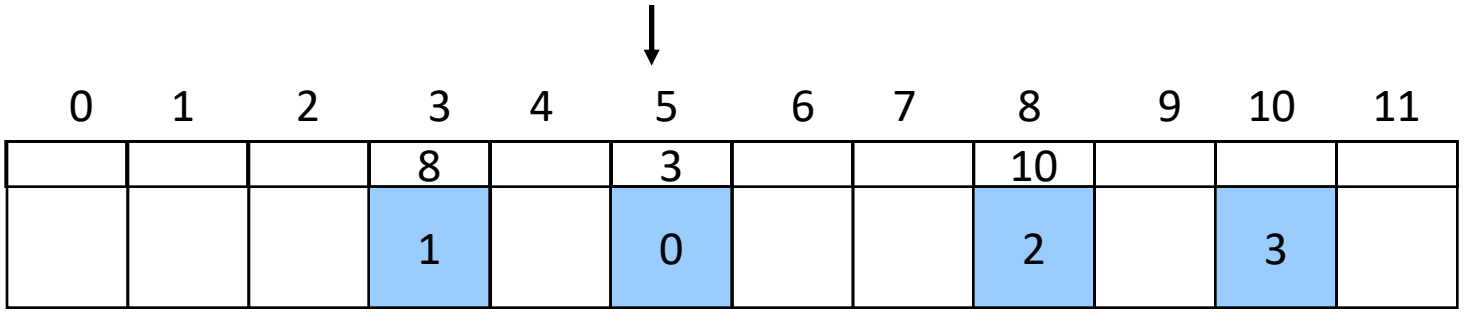
Block to be accessed = the Qth disk block in the linked chain of disk blocks storing file content.
Displacement in the disk block = R + PointerSize

Linked Allocation



Linked Allocation: Example

- Assume block size = 1024 bytes
- Pointer size if 4 bytes
- Assume we have a file that is 4000 bytes.
- File data is place as below to the disk blocks; file starts at disk block 5



Find out the disk location corresponding to file offset 2900?

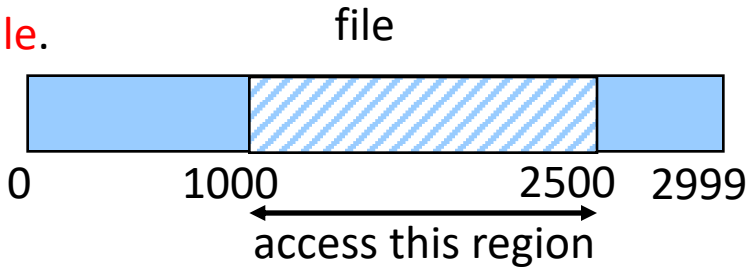
2900 / (1024-4) = 2
2900 % 1020 = 860

Go to the 2nd block in the chain

Second block in chain is disk block 8
Displacement is 860+4 = 864

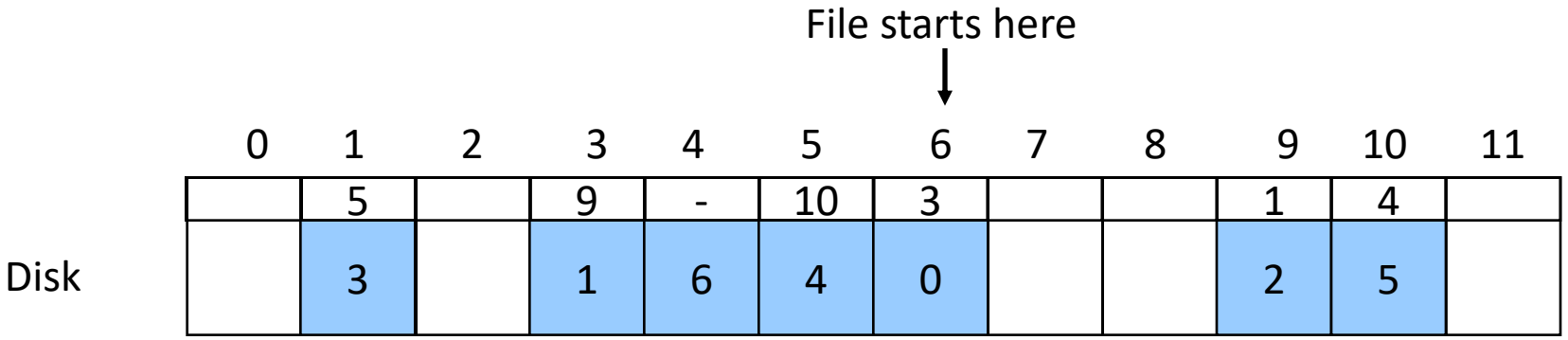
Linked Allocation: Another Example

We have a file that is 3000 bytes long.
Disk block size = 512 bytes; pointer size = 4 bytes.
We want to access bytes 1000 through 2500 of the file.
Which disk blocks should be retrieved?



1000/508=1; 1000%508=492
2500/508=4; 2500%508=468

File blocks to access: 1, 2, 3, 4

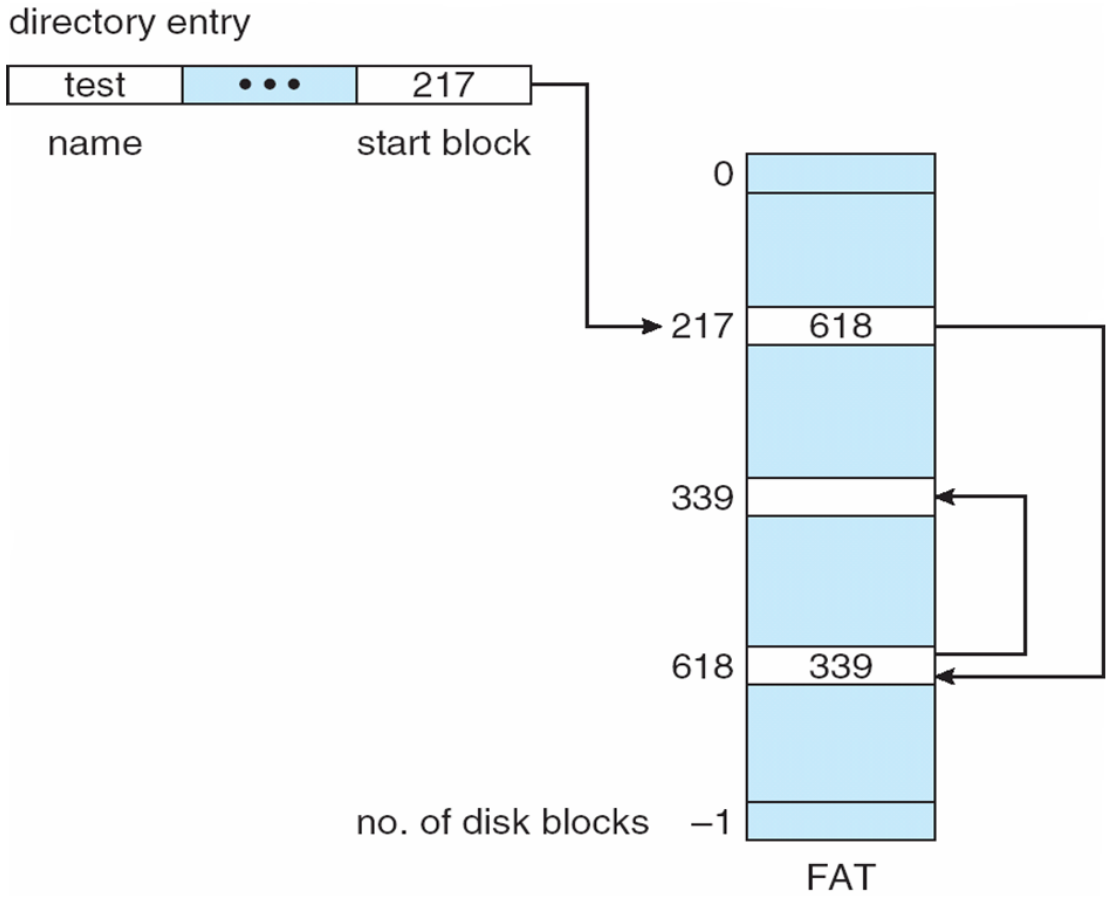


Answer: Disk Blocks to access 3, 9, 1, 5

File Allocation Table

- File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2 operating systems.
- Pointers (i.e., disk data block numbers) are kept in a table (FAT)
- Data block does not hold a pointer; hence data size in a disk block is a power of 2.
- FAT16, FAT32 file systems are using this.
- FAT is also used for free space information.

File-Allocation Table

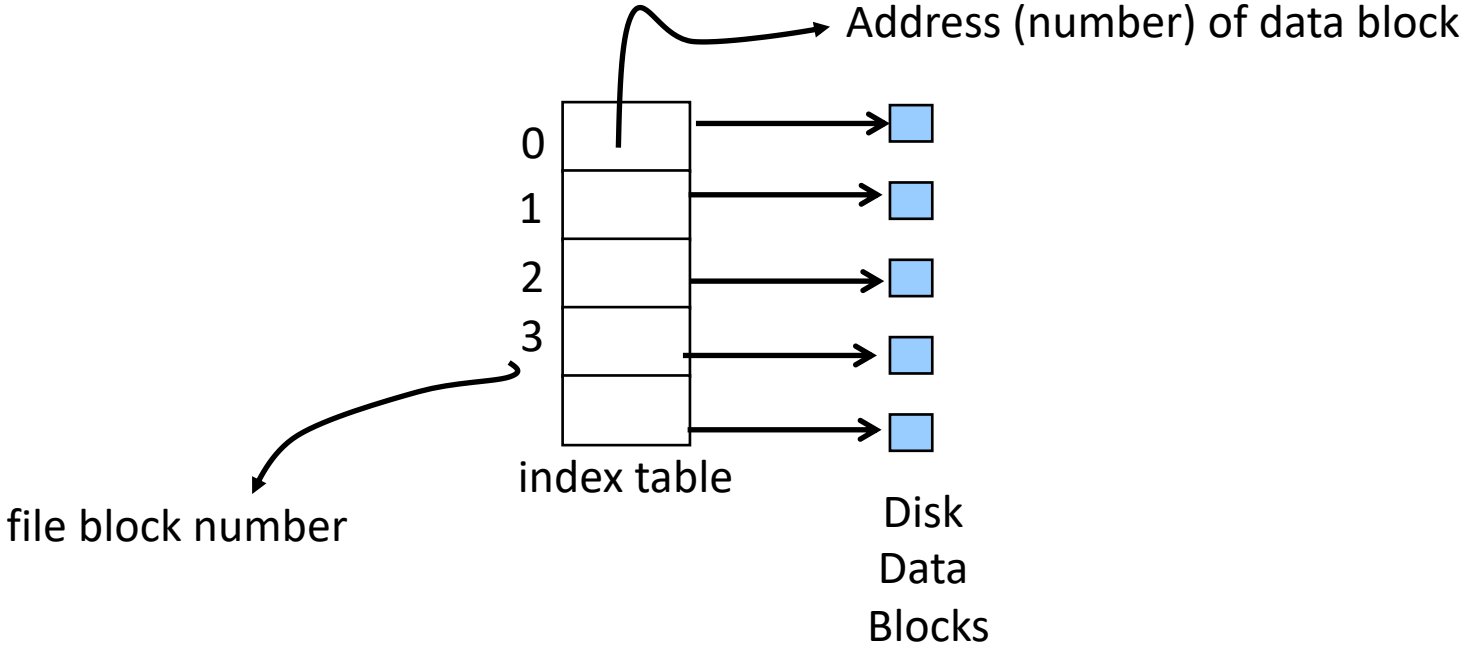


Linked Allocation: an enhancement

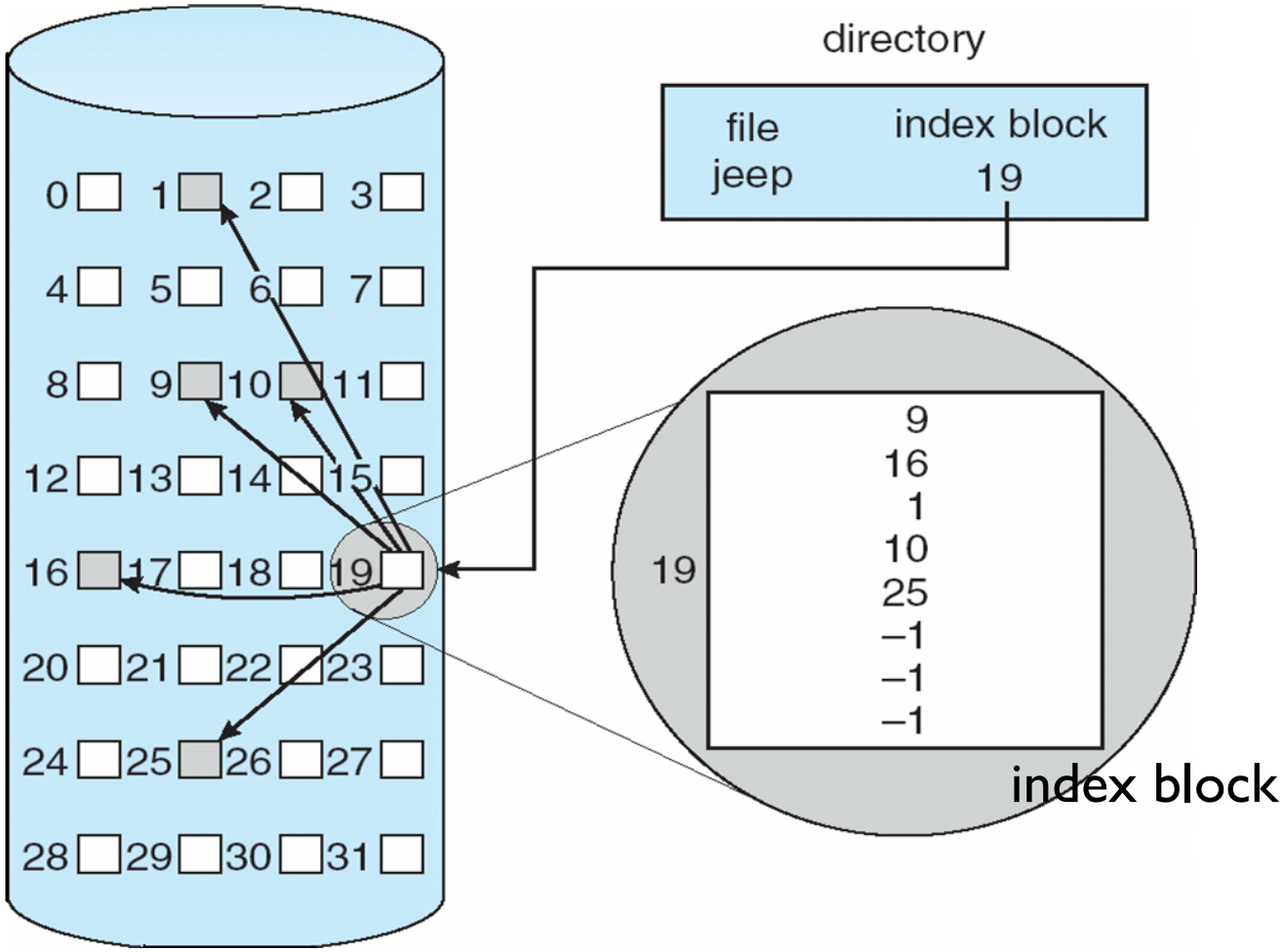
- A set of contiguous blocks can be considered together
  - Called cluster
- Allocation unit is now a cluster
- Less space wasted due to pointer in a block (in a cluster)
- Faster random access (more efficient)
- Beginning of a cluster has information about the next cluster
- Two clusters do not have to be next to each other.

Indexed Allocation

- For a file, keep the addresses of its (data) blocks in a table called index table.
- Index table can occupy one or more disk blocks called index blocks.
- An address is a disk block number



Example



In index table, address are stored in order, wrt file block number.

Indexed Allocation

- Index table stored in index blocks.
- Random access can be fast
- No external fragmentation, but have overhead of index blocks

Mapping Algorithm:

Q = displacement into index table (file block number)
R = displacement into block (block offset)

For larger files, we need many index blocks

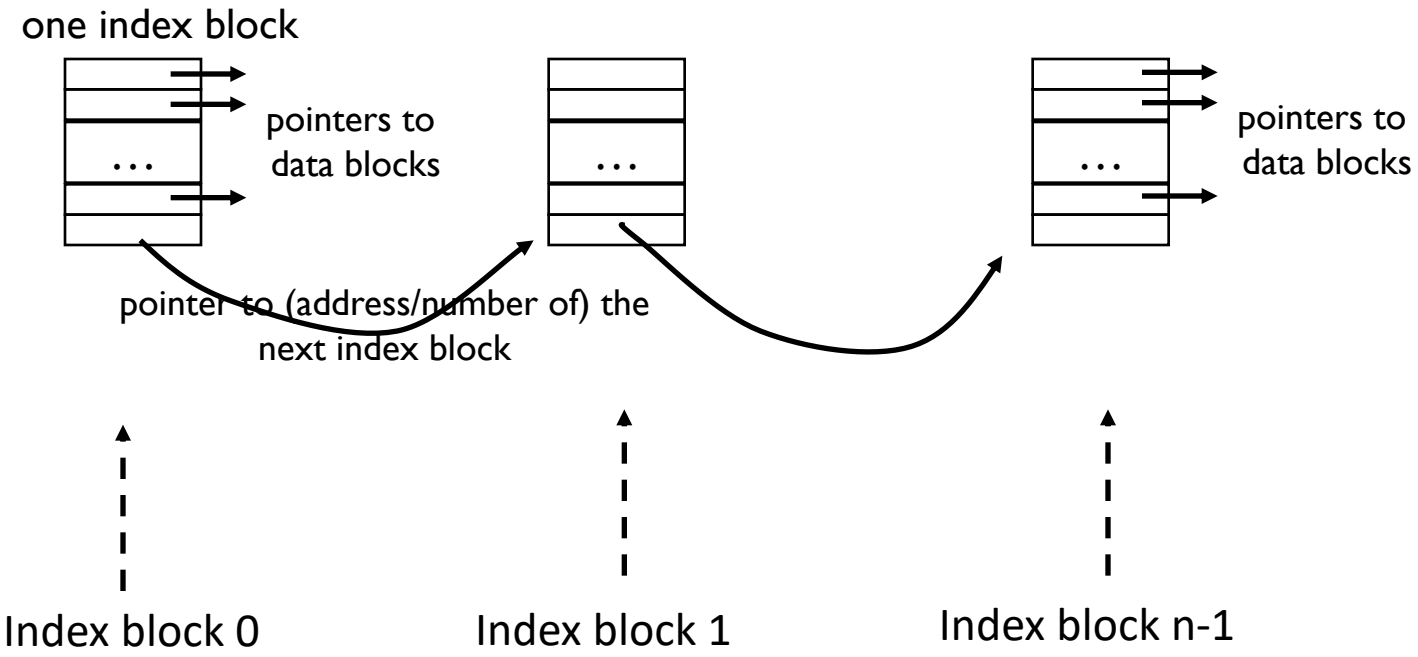
Indexed Allocation

- The index table size depends on:
  - Number of disk blocks are allocated for the data (contents) of the file (file size)
  - Disk pointer size (i.e., size of a disk block number)
- Example:
  - Assume block size is 4 KB.
  - Assume pointer size if 4 Bytes (a disk block address/number is 4 bytes)
  - A disk block can store an index table of size at most: 4 KB / 4 B = 1024 entries.

Indexed Allocation: Linked Scheme

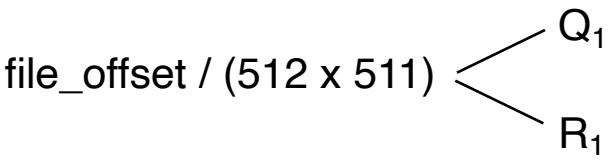
- If index table can not fit into a single block, we can use multiple index blocks and chain them together.

  Linked scheme – Link blocks of index table (no limit on file size)

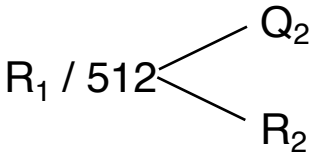


Indexed Allocation: Linked Scheme: Mapping

- Mapping from file offsets to disk block numbers in a file of unbounded length.
- Assuming: block size is 512 words and 1 pointer occupies 1 word).
- Mapping algorithm:

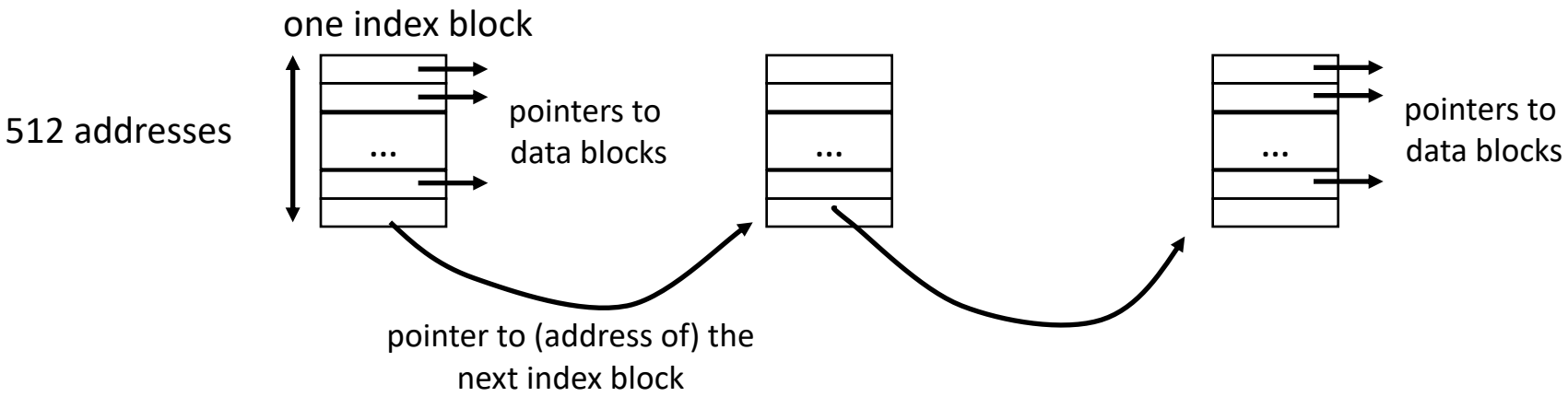


Q_1 = index block relative place
R_1 is used as follows:



Q_2 = displacement into the index block
R_2 displacement into block of file:

Indexed Allocation: Linked Scheme



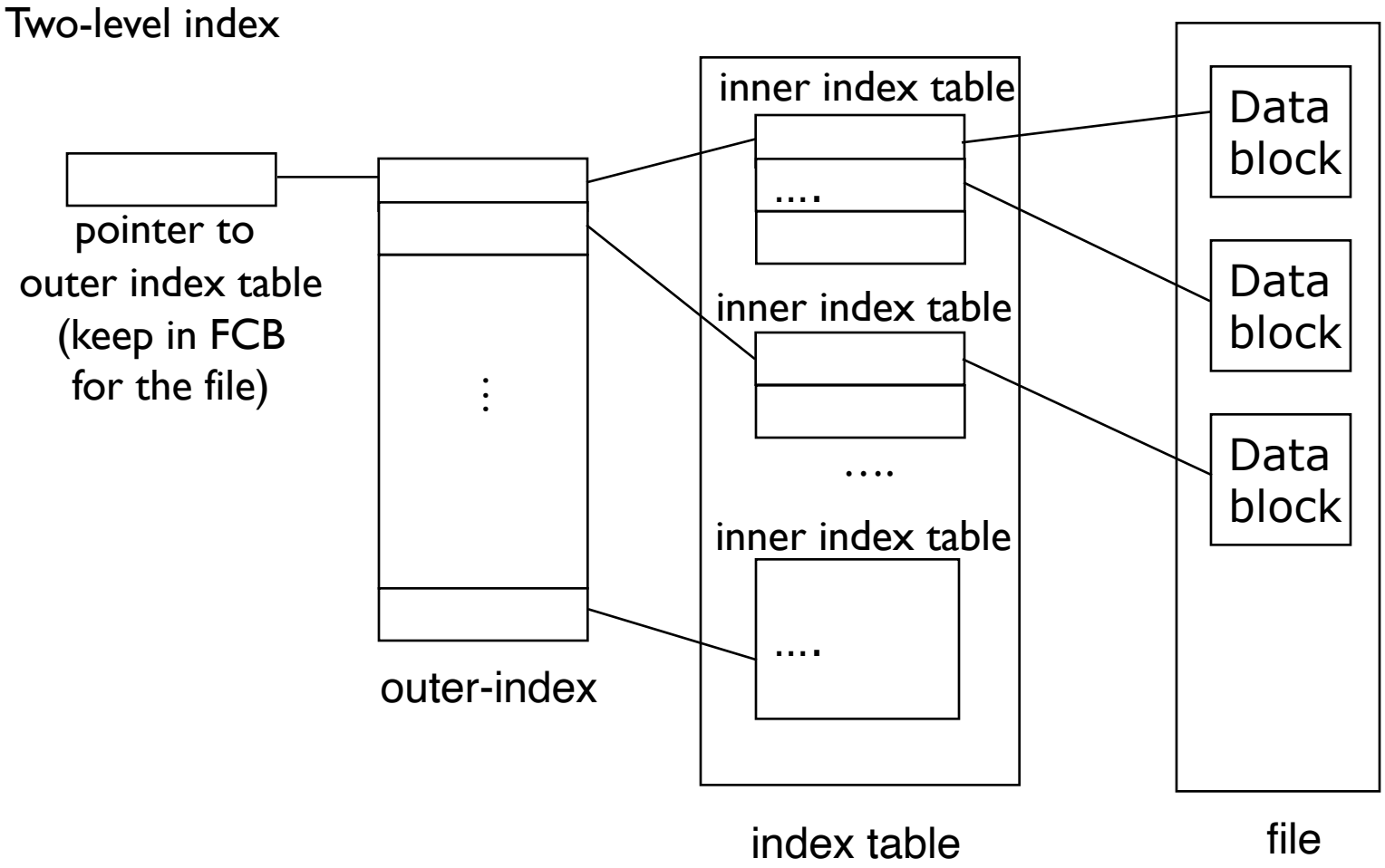
In an index block, 511 addresses are for data blocks.
Each data block is 512 words.

Hence, an index block can be used to map (511x512) words of a file

Indexed Allocation: Hierarchical Index (Multi-level Index)

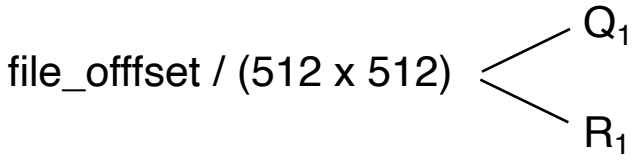
- Hierarchical index allocation
  - Use a hierarchy of index blocks.
  - Two-level, three-level hierarchies possible
  - Two level index; three-level index

Indexed Allocation – Two-level index

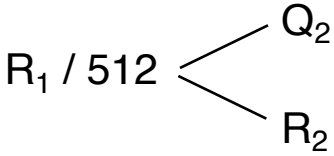


Indexed Allocation – Mapping (Cont.)

- Two-level index (maximum file size is 512^3 words assuming a block is 512 words and a pointer is 1 word).
- Mapping algorithm (assuming a block contains 512 pointers):



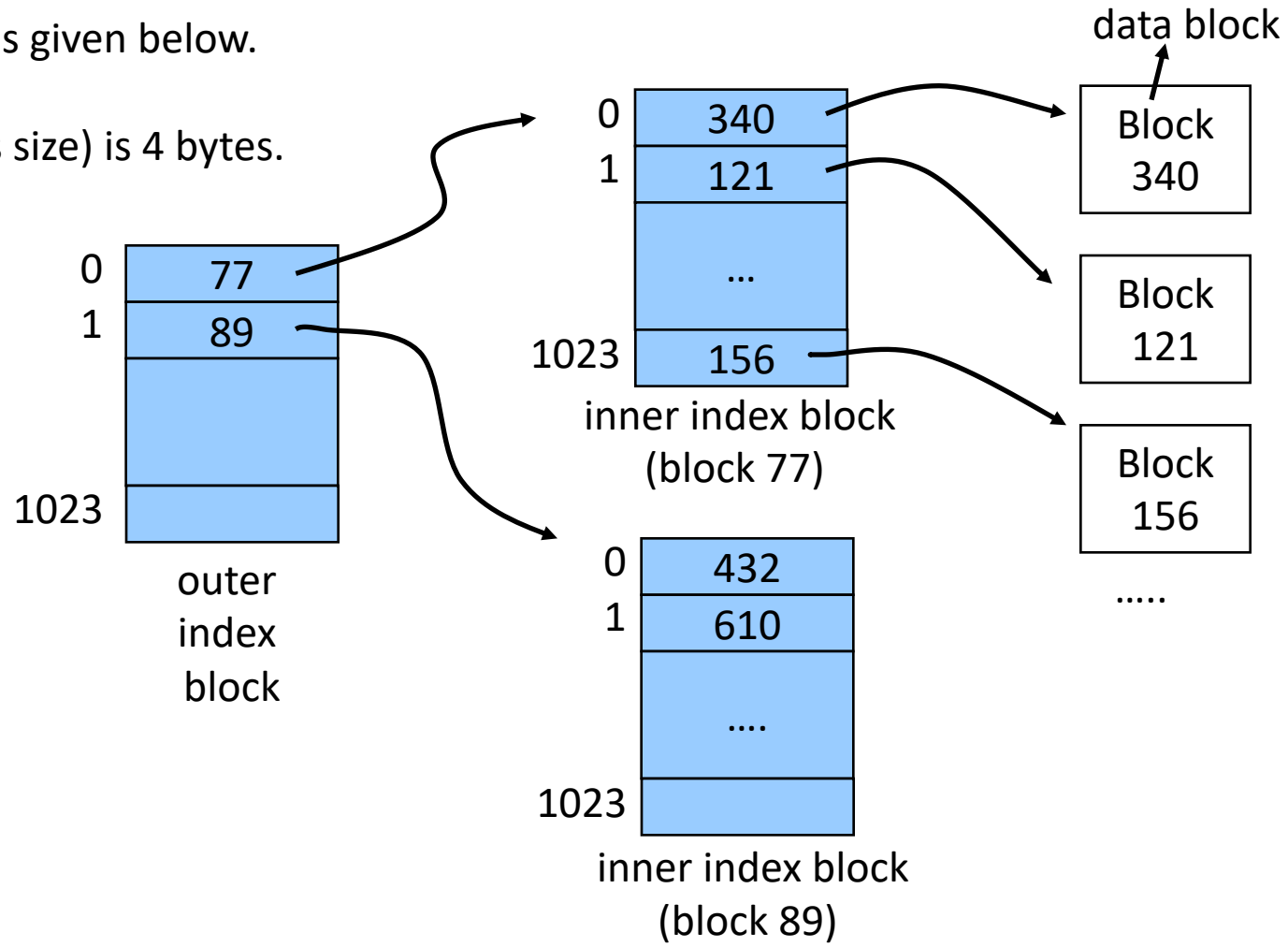
Q_1 = displacement into outer-index block
R_1 is used as follows:



Q_2 = displacement into an inner index block
R_2 = displacement into block of file:

Example

Index table for a file is given below.
Block size is 4KB.
Disk pointer (address size) is 4 bytes.



Example

- Where is file offset 5000?
  - 5000 / (1024 x 4096) = 0
  - 5000 % (1024 x 4096) = 5000
  - 5000 / 4096 = 1
  - 5000 % 4096 = 904
  - So it is on disk block 121 (follow outer table entry 0 and then inner table entry 1) and in that block displacement is 904.

Example

- Where is file offset 4198620?
  - 4198620 / (1024 x 4096) = 1.
  - Go to index 1 of outer table. That gives inner index table address: 89; Go to that inner index table (block).
  - 4198620 % (1024 x 4096) = 4316.
  - 4316 / 4096 = 1
  - Go to index 1 in the inner table. There is the data block address: 610.
  - Get that data block.
  - 4316 % 4096 = 220. displacement is 220

Indexed allocation: Combined Scheme

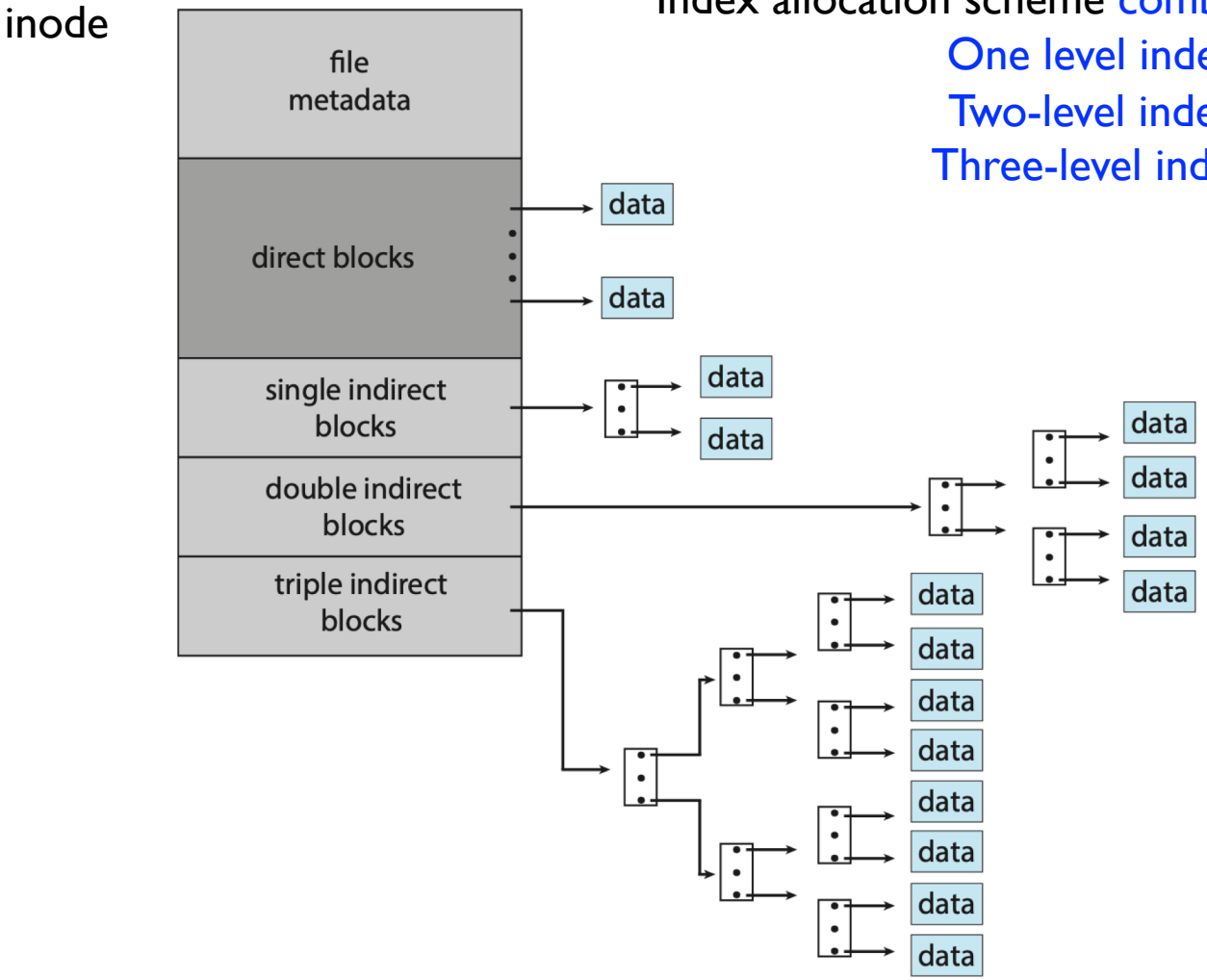
Linux

Index allocation scheme combining many levels

One level index

Two-level index

Three-level index



Indexed allocation: Combined Scheme Linux

- Max file size in combined scheme?
- Assume
  - Disk block size is 4 KB
  - Disk pointer (block number size) is 4 bytes
  - Assume we have 10 *direct* pointers kept in inode, 1 *single indirect* pointer, 1 *double indirect p*ointer, 1 *triple indirect pointer* kept in the inode.
- Then we have 4 KB / 4 = 1024 pointers in an index block.
- Then max file size is:
  - $4096 \times (10 + 1204 + 1024^2 + 1024^3)$ bytes
  - ~ around more than 4 TB.

Free Space Management

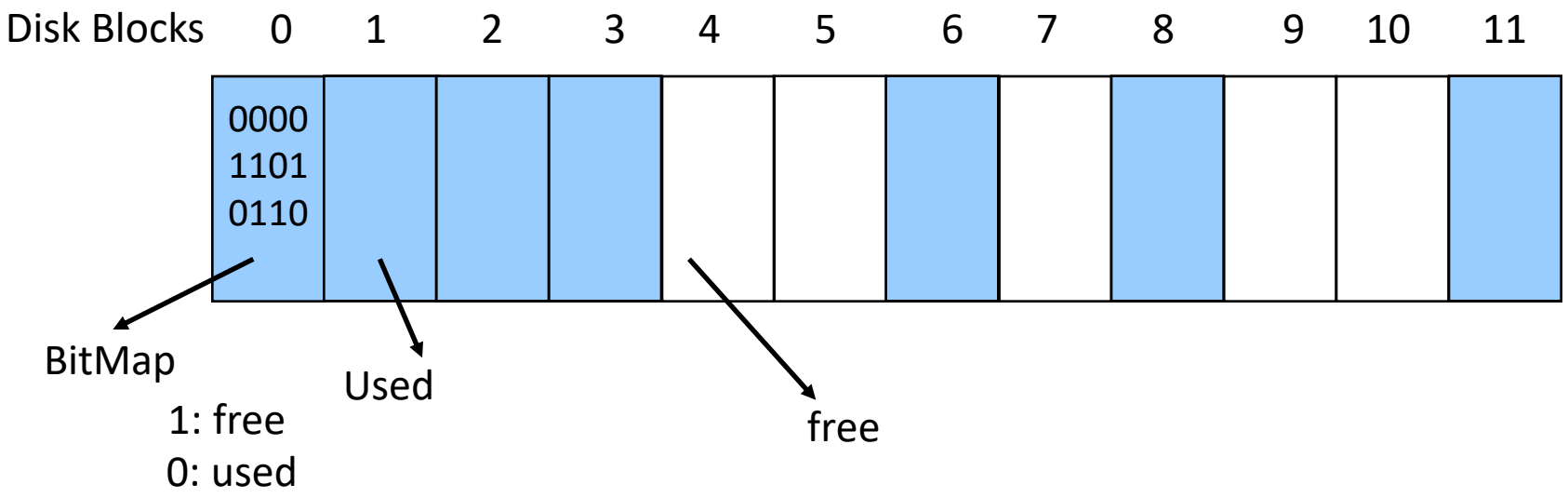
- How can we keep track of free blocks of the disk?
  - Which blocks are free?
- We need this information when we want to allocate a new block to a file:
  - allocate a block that is free.

- There are several methods to keep track of free blocks:
  - Bit vector (bitmap) method
  - Linked list method
  - Grouping
  - Counting
- FAT can itself show free blocks.

Free-Space Management: Bit Vector (Bit map)

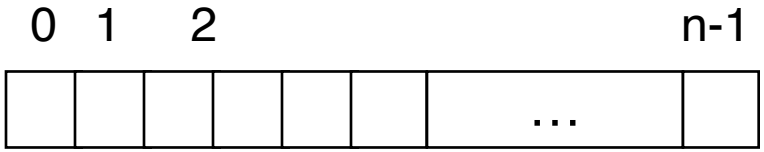
- We have a bit vector (bitmap) where we have one bit per block indicating if the block is used or free.
- If the block is free the corresponding bit can be 1, else it can be 0 (or vice versa).

Example:



Free-Space Management: Bit Vector (Bit map)

- Bit vector (n blocks in disk)



bit[i] = 0 ⇒ block[i] used
1 ⇒ block[i] free
(or vice versa)

Finding a free block (i.e. its number)

Start searching from the beginning of the bitmap:
Search for the first 1

First Free Block Number =
(number of 0-valued words) * (number of bits per word)
+ offset of first 1-valued-bit

0000000000000000
0000000000000000
0000000000000000
0000000010000000
0000000000000000
0000000000011000
0001100010000000
0000000011110000

3x16+8 = 56

Free-Space Management: Bit Vector (Bit map)

- Bitmap requires extra space
  - Example:

    block size = 2^{12} bytes

    disk size = 2^{30} bytes (1 gigabyte)

    n = $2^{30}/2^{12} = 2^{18}$ blocks exist on disk;

    Hence we need 2^{18} bits in the bitmap.

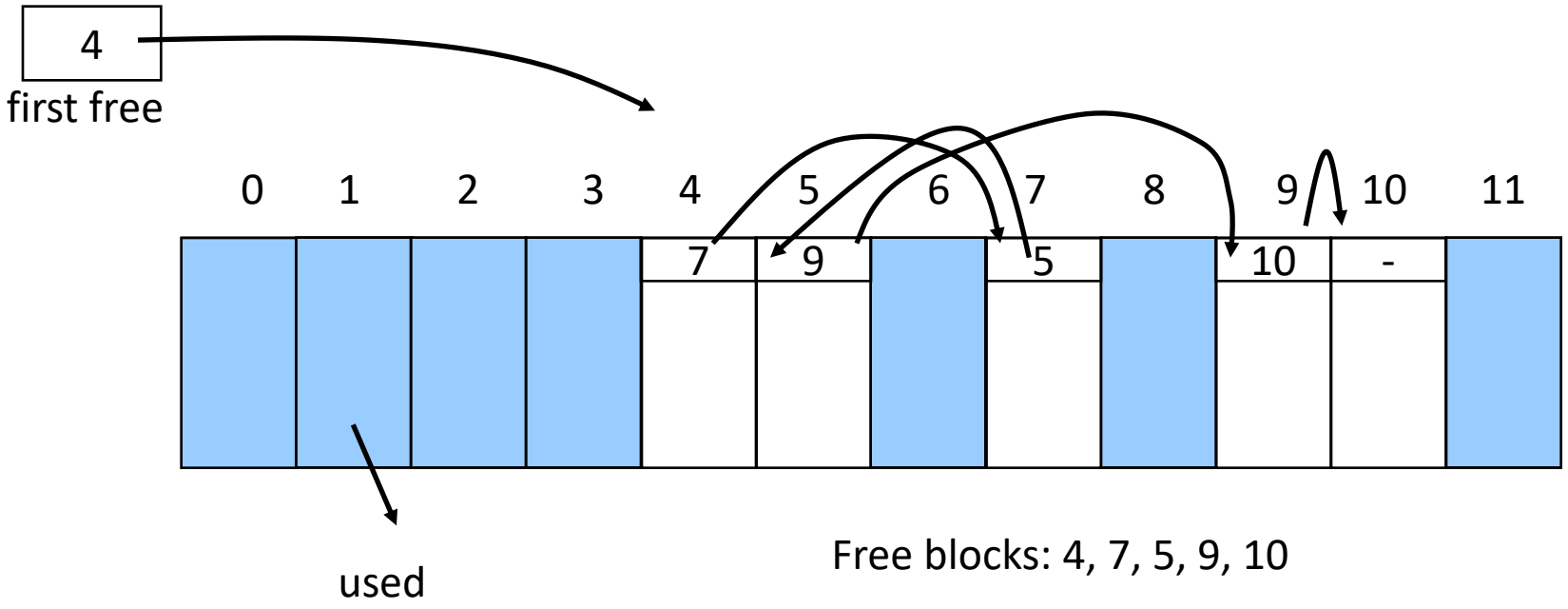
    That makes 2^{18} / 8 / 1024 = 32 Kbytes for bitmap.

    That means 32/4=8 disk blocks occupied by bitmap.

- Easy to get contiguous files
- Blocks of a file can be kept close to each other.

Free-Space Management: Linked List

- Each free block has pointer to the next free block
- We keep a pointer to the first free block somewhere (like superblock).

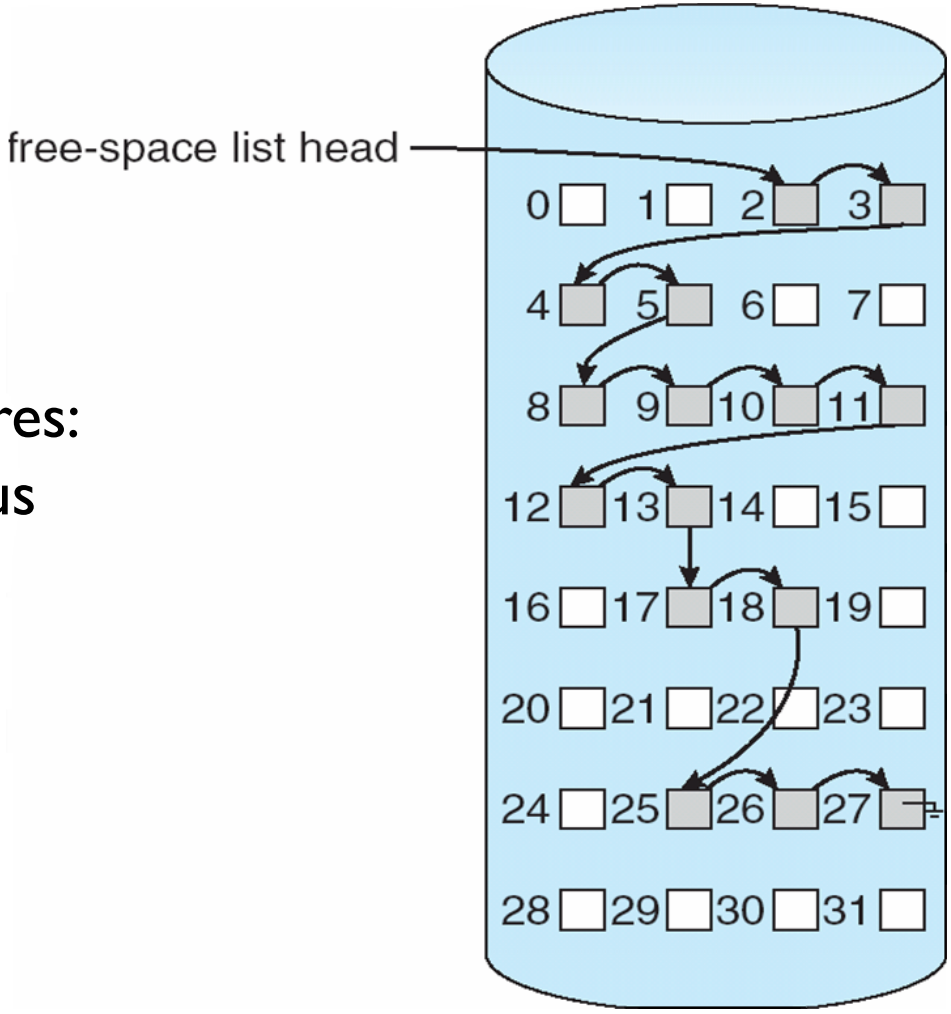


Free-Space Management: Linked List

Linked list (free list) features:

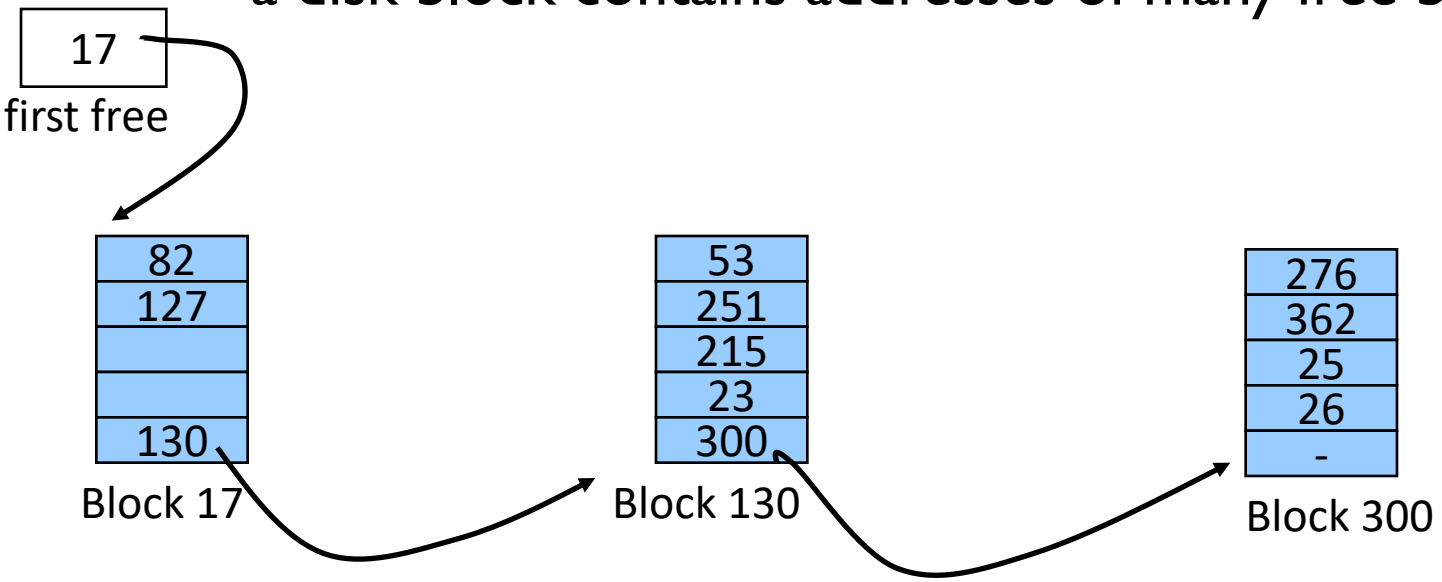
-Cannot get contiguous space easily

- No waste of space



Free-Space Management: Grouping

a disk block contains addresses of many free blocks

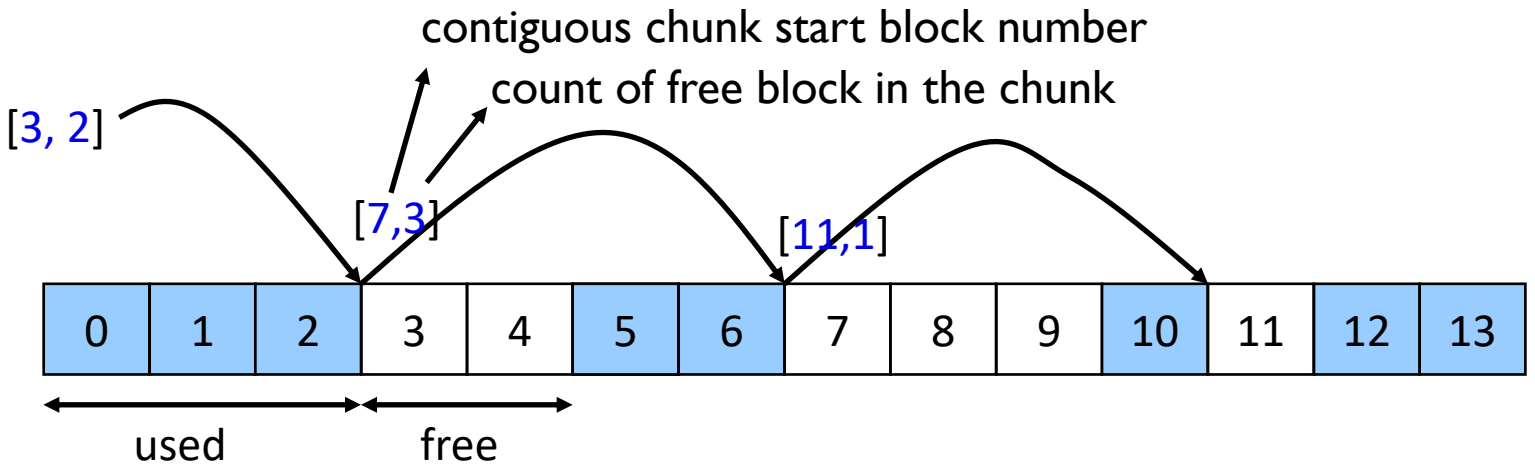


Free blocks are:
82 127 ... 53 251 215 23 ... 276 362 25 26

a block containing free block pointers will be free when those blocks are used.

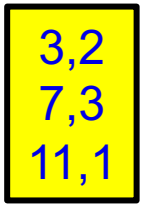
Free-Space Management: Counting

- Besides the free block pointer, keep a count saying how many blocks are free contiguously starting at that free block.



We can keep first_free_block# and count as above.

Or, keep all information together somewhere on the disk



Free-Space Management (Cont.)

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[i] to have a situation where bit[i] = 0 (allocated) in memory and bit[i] = 1 (free) on disk
  - Solution:
    - Set bit[i] = 0 in disk
    - Allocate block[i]
    - Set bit[i] = 0 in memory

Disk space efficiency

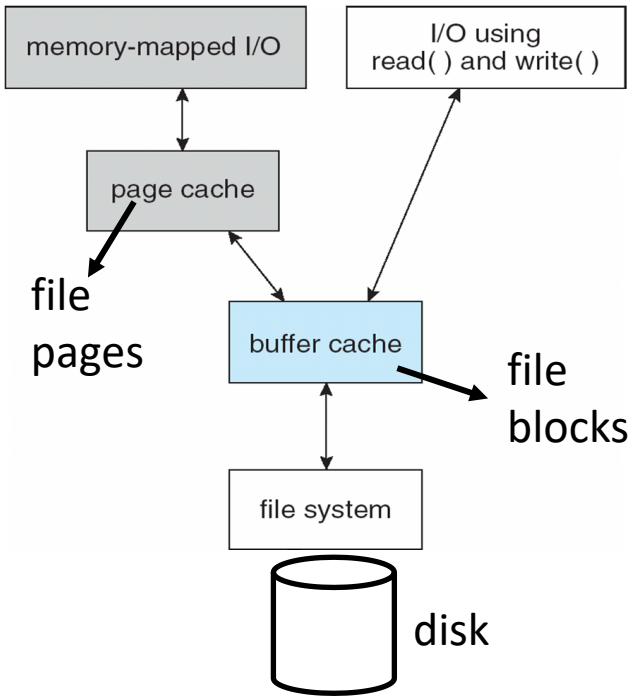
- Storage space efficiency depends on:
  - disk allocation and directory organization and algorithms
- Block allocation methods affect space efficiency
  - Contiguous allocation causes external fragmentation
  - Using a large block or cluster size increases internal fragmentation
- Meta-data also affects space efficiency.
  - All inodes are preallocated in Unix (before files created).
- Size of pointers affects used space for index blocks.
- Fixed sized or variable sized directory entries affect space usage.

Performance (access time)

- A separate section of memory can be used as disk cache.
  - To cache disk blocks
  - To cache pages of memory mapped files
- free-behind and read-ahead – techniques to optimize sequential access.
  - Free-behind: a block (page) accessed will not be accessed again in sequential reads (LRU is not the best)
  - Read-ahead: A requested block (or page) and several subsequent blocks or pages are read together.
- improve performance by dedicating section of memory as virtual disk, or RAM disk
  - /tmp file system is in RAM

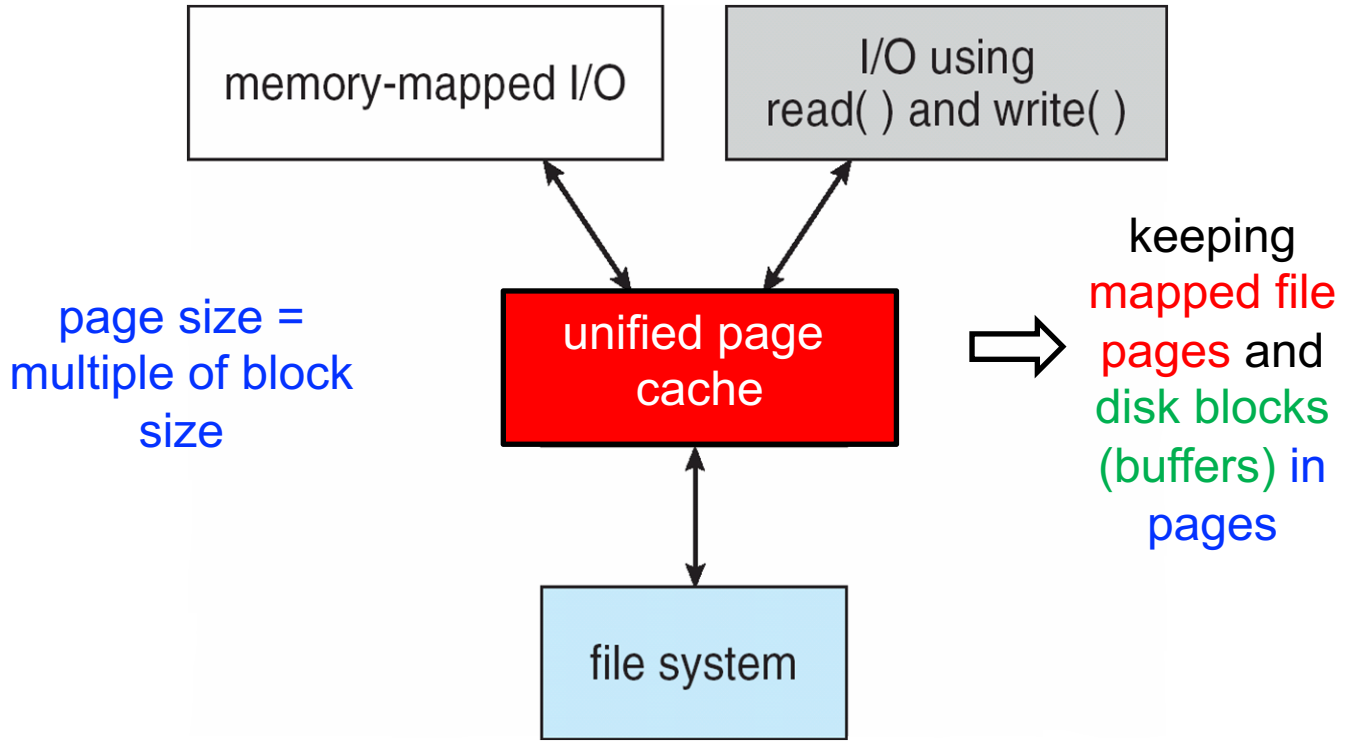
Page Cache

- A page cache caches pages rather than disk blocks using virtual memory techniques.
  - Memory-mapped file I/O uses a page cache.
  - Pages corresponding to file data cached here in pages.
  - A page may contain 1 or more disk blocks (depending on the size).
- Routine I/O through the file system uses the buffer (disk block) cache.



Unified Buffer/Page Cache

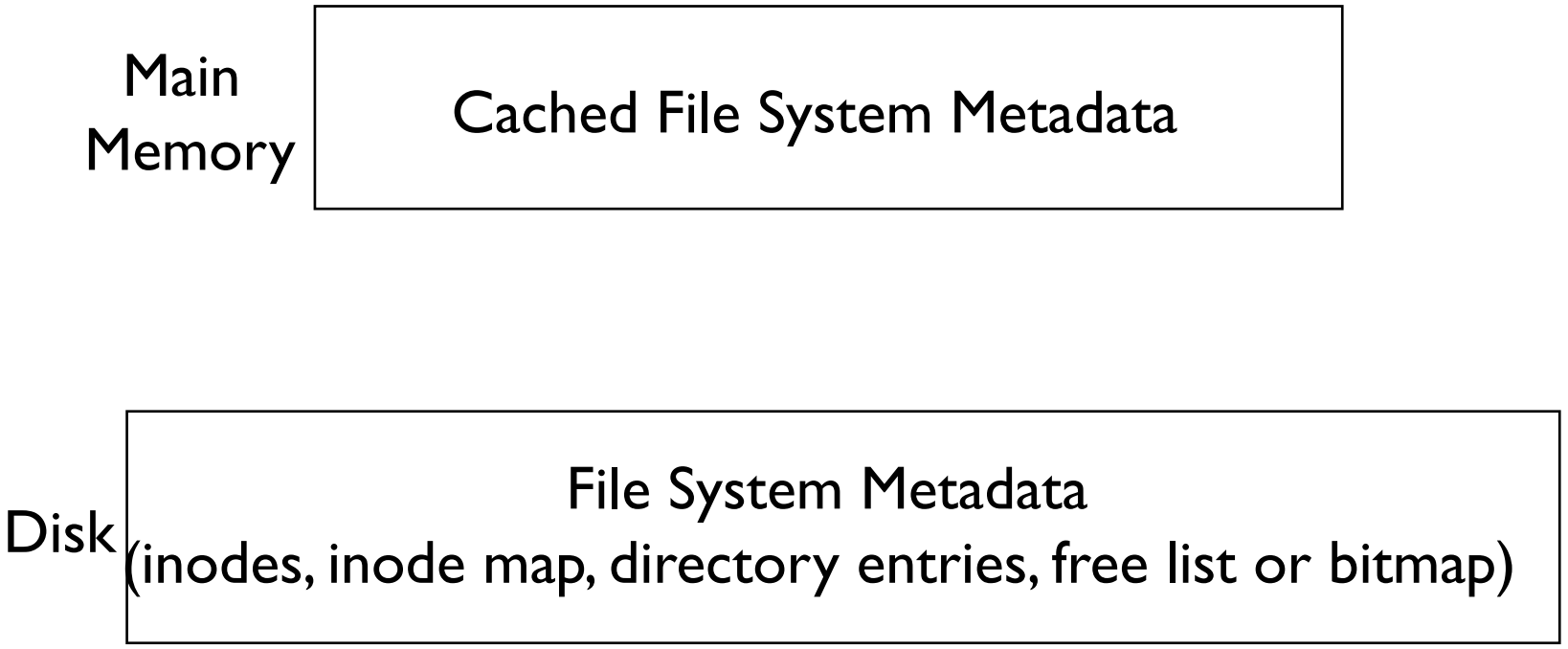
- A unified page/buffer cache uses the same cache to cache both memory-mapped file pages and ordinary file system I/O blocks (all in pages unit).



Recovery

- Power or system failure can happen ==> inconsistency in disk (i.e., file system metadata on disk)
  - Consistency checking – compares different meta-data and tries to fix the inconsistencies found (for a file: bitmap, directory entry, inode map, inode information need to be all consistent).
  - is invoked after a power or system failure.
- Permanent file/disk failure ==> file or disk lost
  - Solution: use system programs to back up data
  - Recover lost file or disk by restoring data from backup.

Journaling File Systems



Power failure, abrupt shutdown or system crash may happen at any time

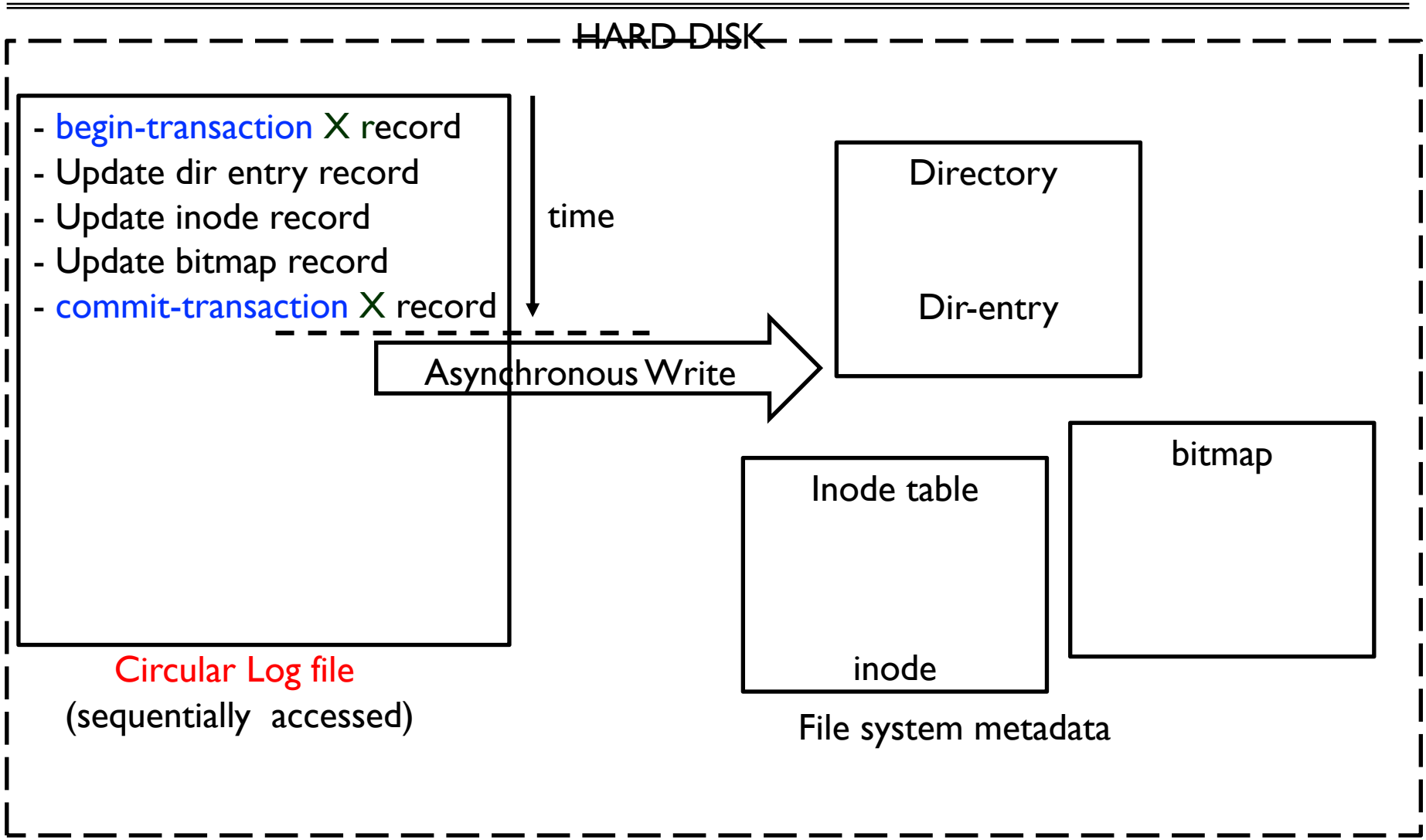
Journaling File Systems

- Example for a modification we can perform on the file system.
- To remove a file; following operations (updates have to be made)
  - 1) Directory entry should be removed (or marked unused)
    - Update directory entry on disk
  - 2) Inode must be marked as free (or removed)
    - Update inode (or inode map) on disk
  - 3) Blocks pointed by inode must be de-allocated
    - Added to the free list or bitmap
    - Update bitmap on disk (or free list).
- While doing these sequence of operations, a failure may happen and leave the disk structures in an inconsistent state. We need to recover from this kind of failures.
- Solution: consider these operations as a transaction.

Journaling File Systems

- A journaling file system records each file operation (modifying file system metadata) as a transaction:
  - a sequence of sub-operations to be done atomically: all or none
  - A transaction (updates or information about updates) is first written to a log (on disk)
- A transaction is considered committed once all its updates are written to the log (stable storage).
  - Then, in-place updates to metadata on disk starts. Operations recorded in the log is asynchronously executed on the file system.
  - When they finish, the transaction is removed from the log.
- If a crash happens, all remaining transactions in the log must still be performed.

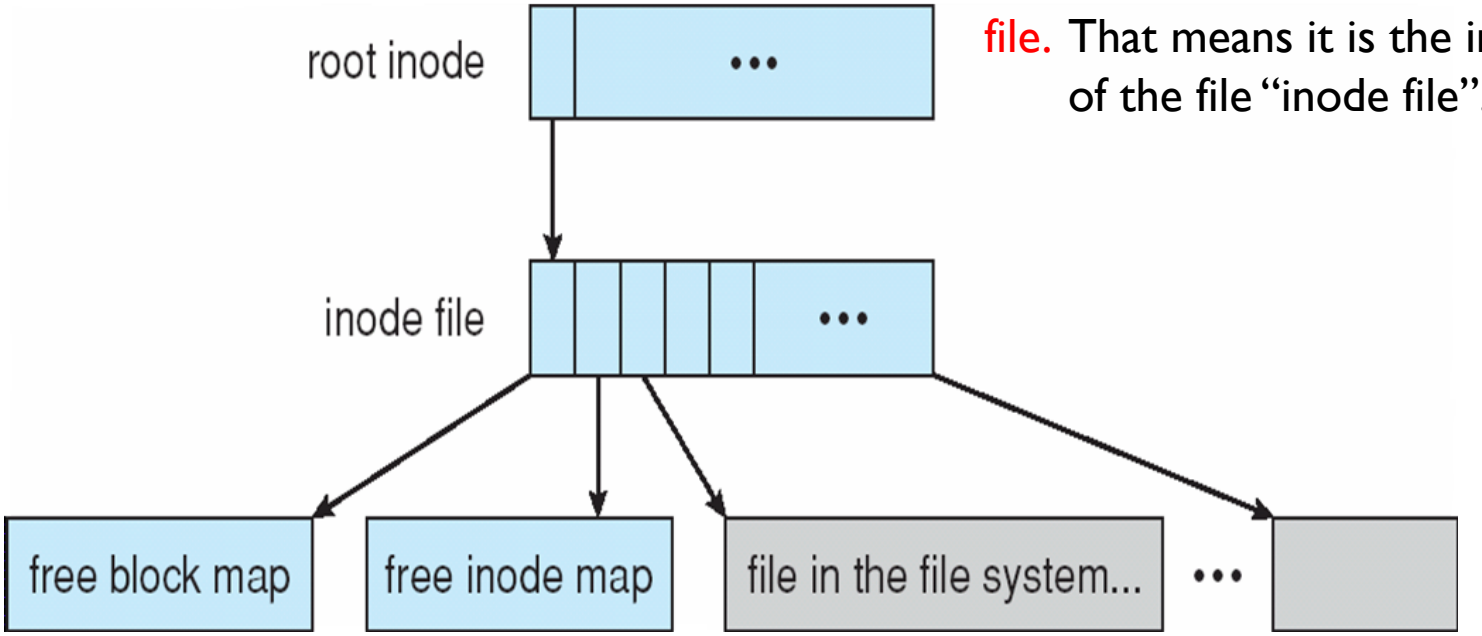
Transactions



Example: WAFL File System

- From a company called Network Appliance
  - Used on network file servers
  - Can be used together with a distributed file system
  - Can provide files to clients using NFS, CIFS, ftp, http.
- "Write-anywhere file layout"
- Server sees a lot of write() load to the file system
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Unix File System. Uses inodes.

The WAFL File Layout

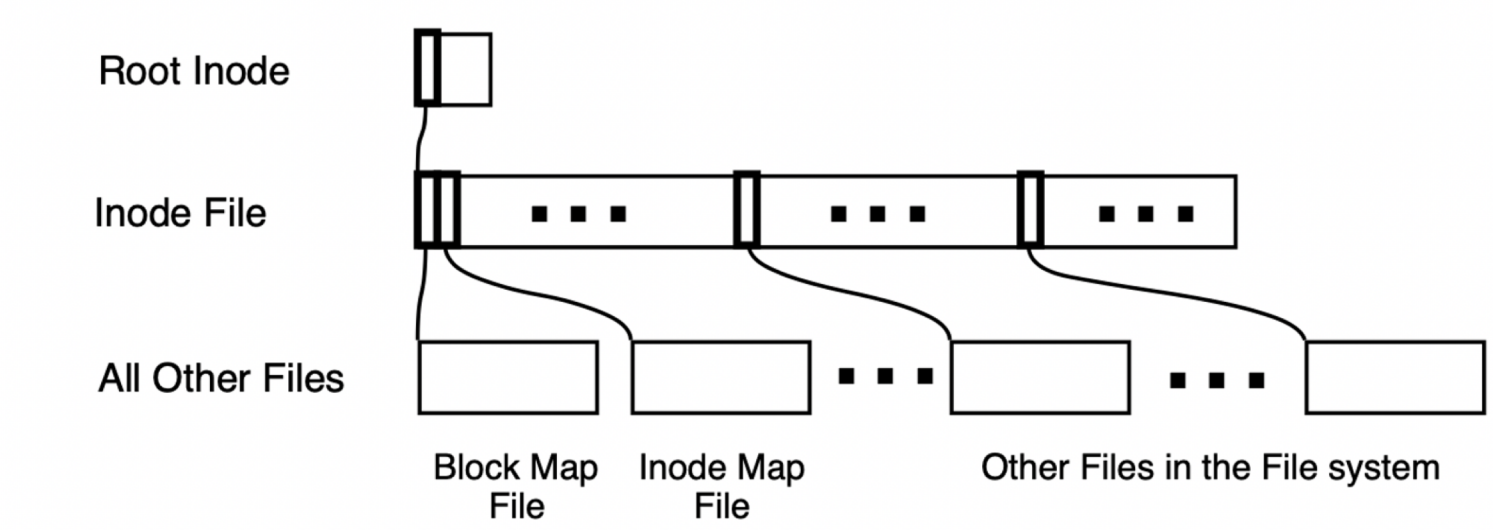


root inode describes the inode file. That means it is the inode of the file "inode file".

In WAFL, metadata lives in files. Therefore, blocks containing metadata do not have to have fixed locations on the disk.

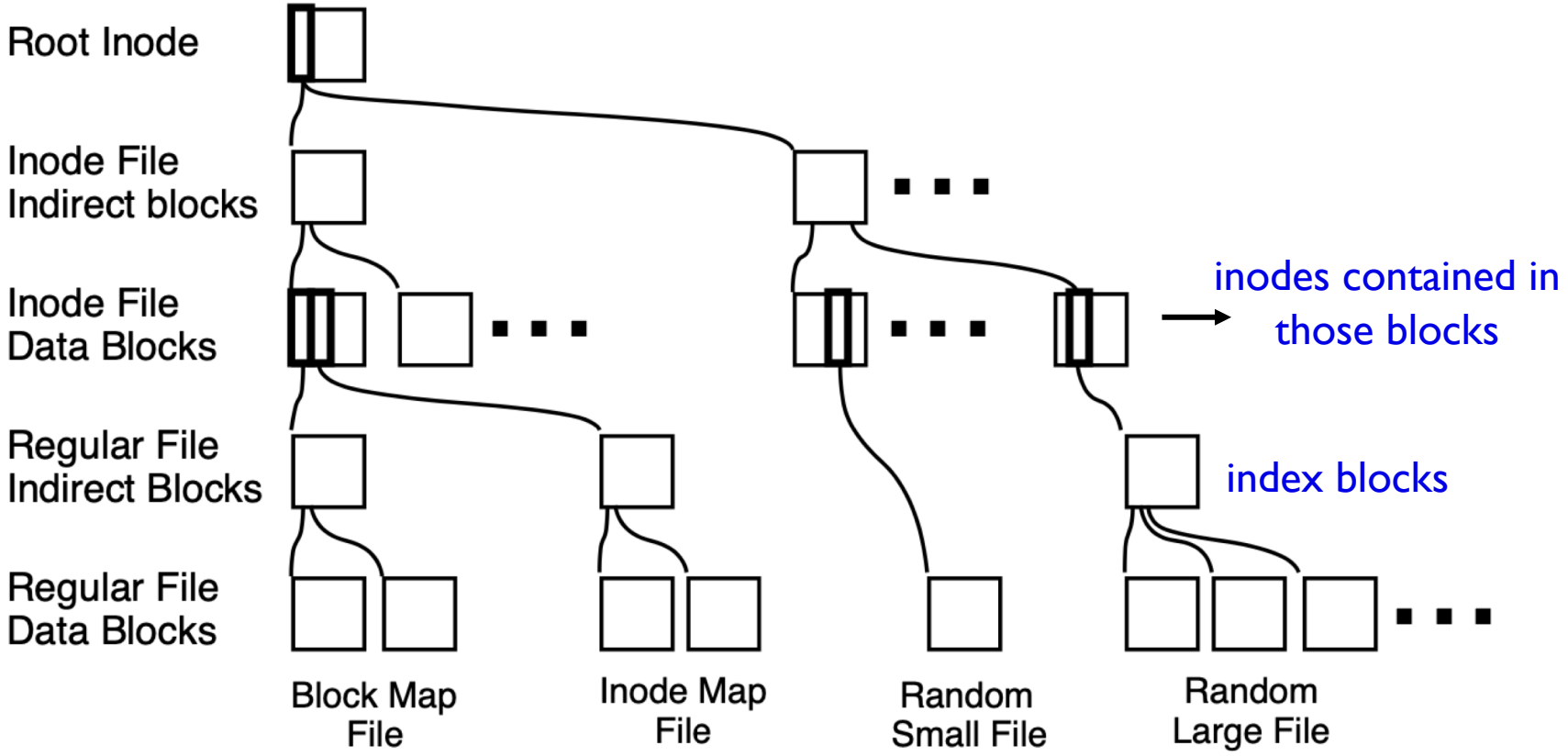
WAFL

- Only root inode's location is fixed on the disk. Other metadata can be anywhere on the disk (on any block).



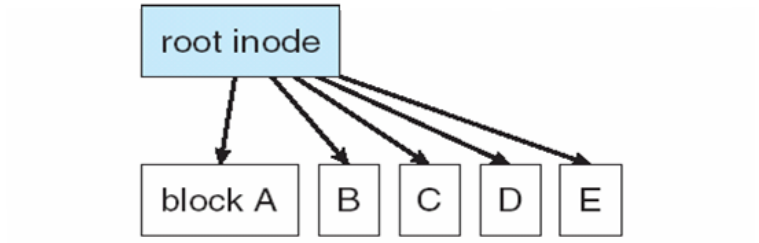
The WAFL file system is a tree of disk blocks with root inode, which describes the inode file, at the top, and metadata files and regular files underneath.

WAFL

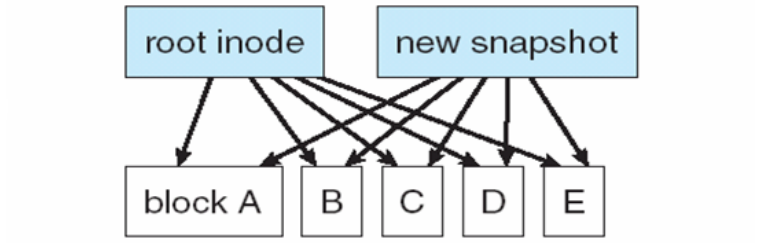


WAFL's tree of disk blocks.

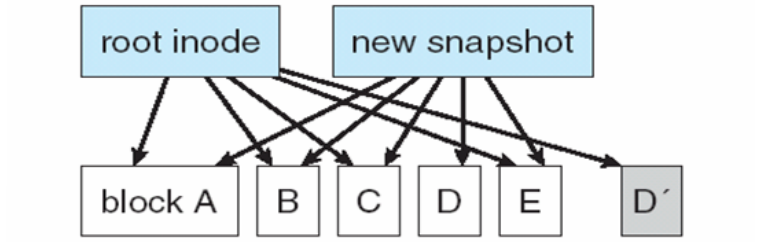
Snapshots in WAFL



(a) Before a snapshot.



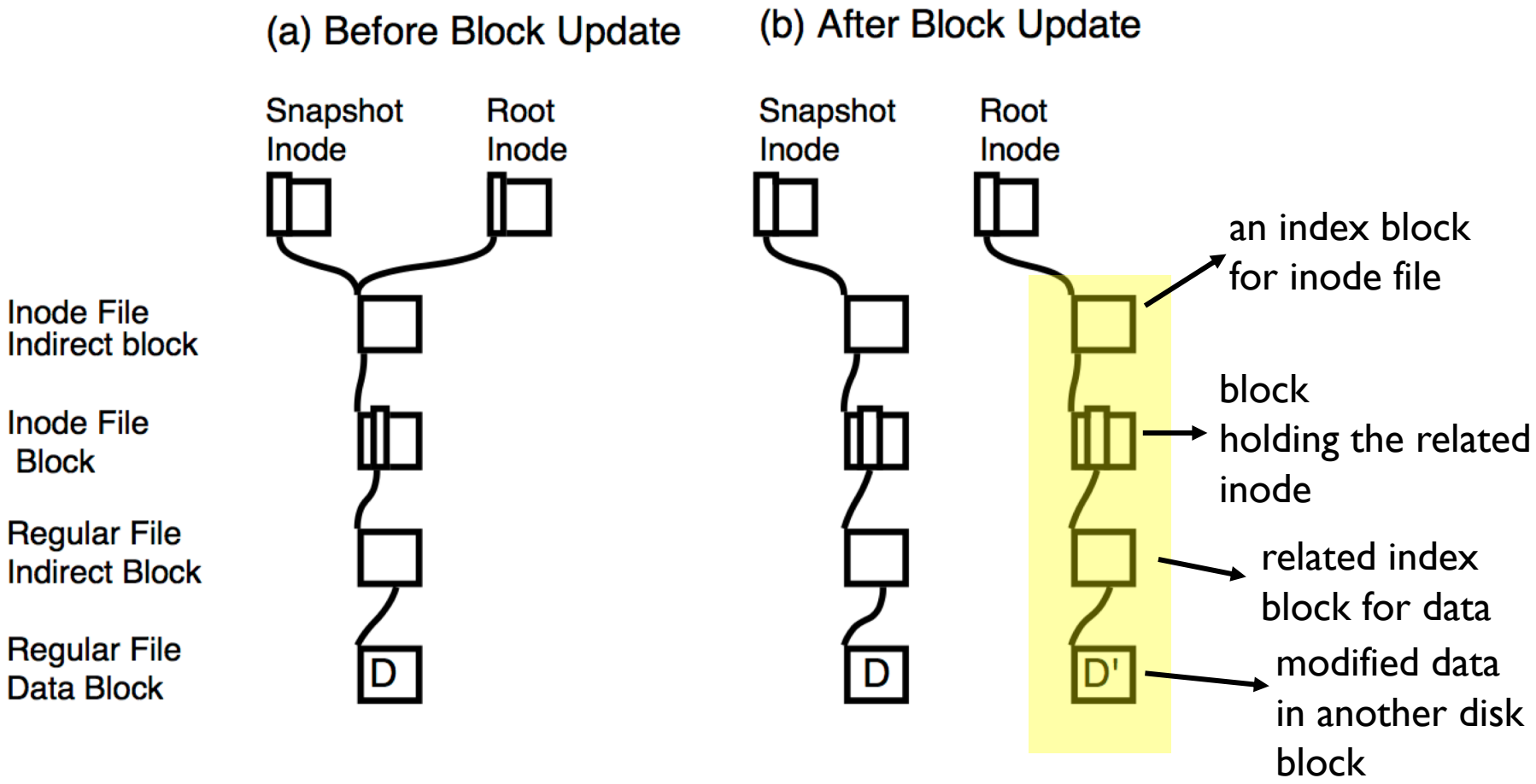
(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

CoW (Copy-on-Write technique is used

Snapshots in WAFL



WAFL

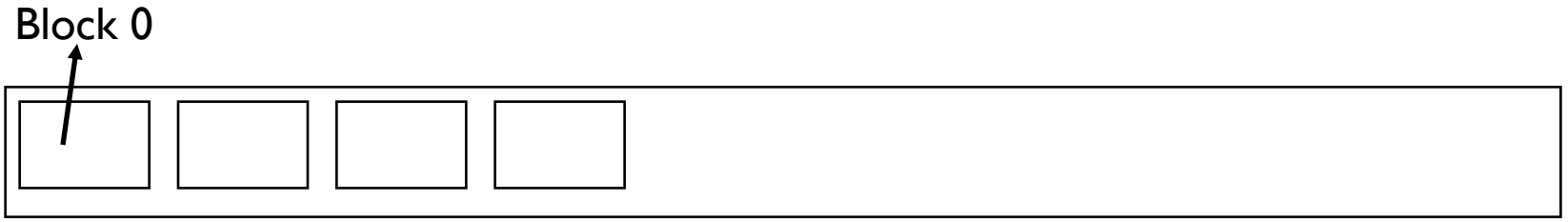
- WAFL is not writing those blocks immediateley to disk for each NFS request (would be inefficient).
- WAFL gathers hundreds of NFS requests before scheduling a write episode to disk.
- Requested modifications happen first in-memory (in NVRAM).
- Commonly modified blocks are modified many times in-memory.
- Then, modified blocks are written to disk to new locations (to new disk blocks) in the next write episode.
- The new disk blocks used may be close to each other in the disk (fast write performance).

Additional (Optional) Study Material

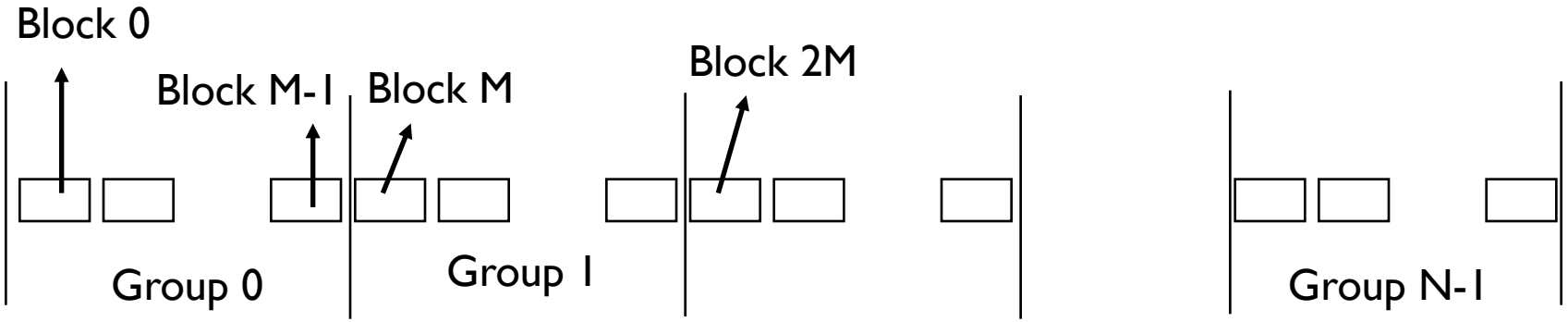
Example File System: Linux ext2/ext3 file system

- Linux ext2 file system is extended file system 2. Derived initially from Minix operating system. Linux is derived from Minix, an educational OS developed by A. Tanenbaum.
- The ext3 file system is fully compatible with ext2. The added new feature is Journaling. So it can recover better from failures.
- The disk data structures used by ext2 and ext3 are the same.

Partition Layout of ext3 (also ext2)



a disk partition before installing ext3 file system: just a sequence of blocks



Ext3 considers the partition to be divided into logical groups.
Each group has equal number of blocks; lets say M blocks/group

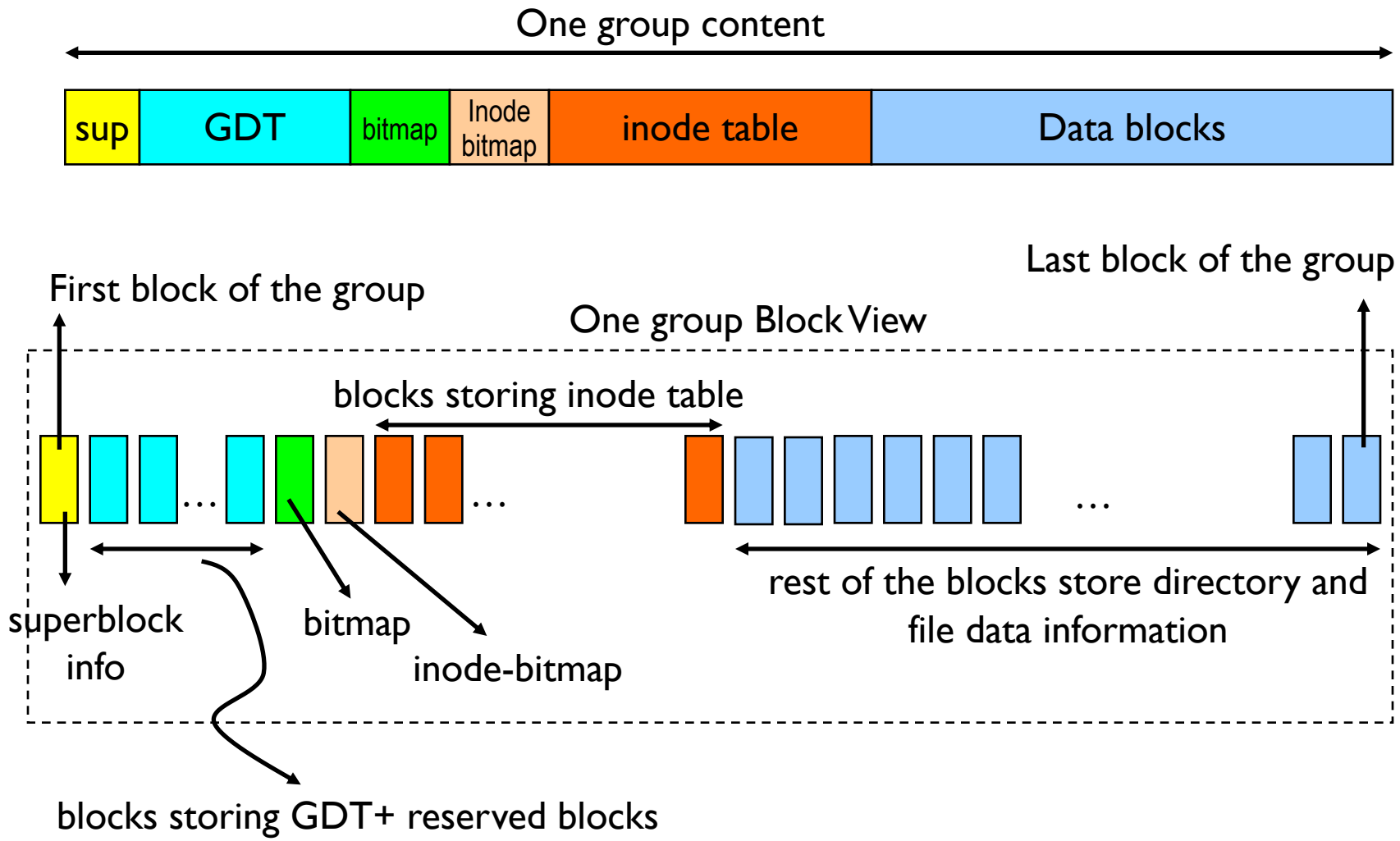
Ext3 file system

- What do we have in a group:
  - Assume block is size if 4 KB.
  - The first block of each group contains a superblock (i.e. superblock info) that is 1024 bytes long.
    - In group 0, superblock info starts at offset 1024 of the first block of the group (i.e. block 0 of the disk)
    - In all other groups, superblock info starts at offset 0 of the first block of the group.
  - Superblock keeps some general info about the filesystem
  - After the first block in a group, a few blocks keeps info about all groups. It means a few blocks store group descriptors table (GDT). Some of these block may be empty and reserved.

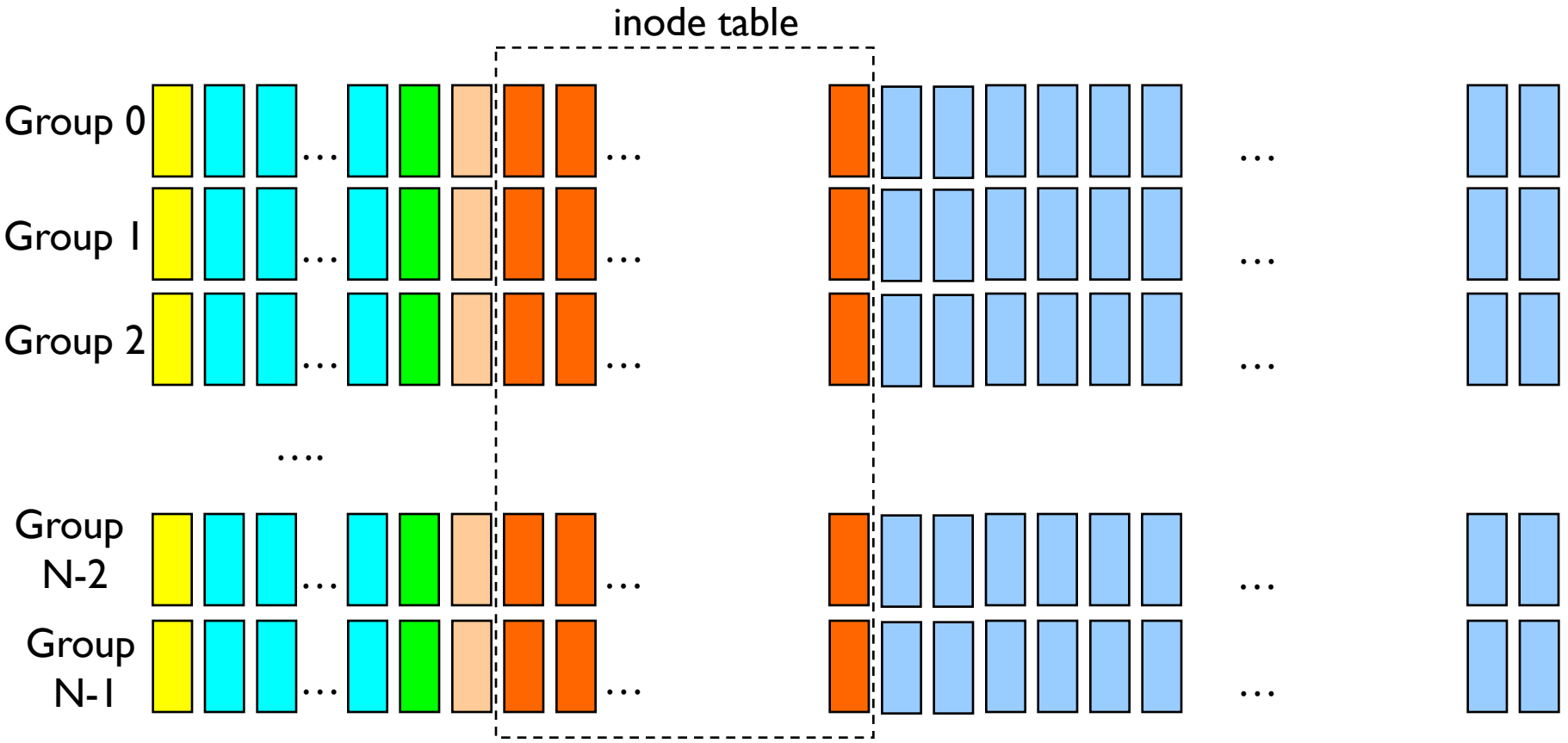
Ext3 file system

- What do we have in a group (continued)
  - After that comes bitmap that occupies one block. It is a bitmap for the group. Each group has it own bitmap.
  - Then comes an inode bitmap; showing which inodes are free.
  - After that comes the inodes (inode table). Each group stores a set of nodes. The number of inodes stored in a group is the same for all groups. So the inode table of the partition is divided into groups: each group stores a portion of the table.

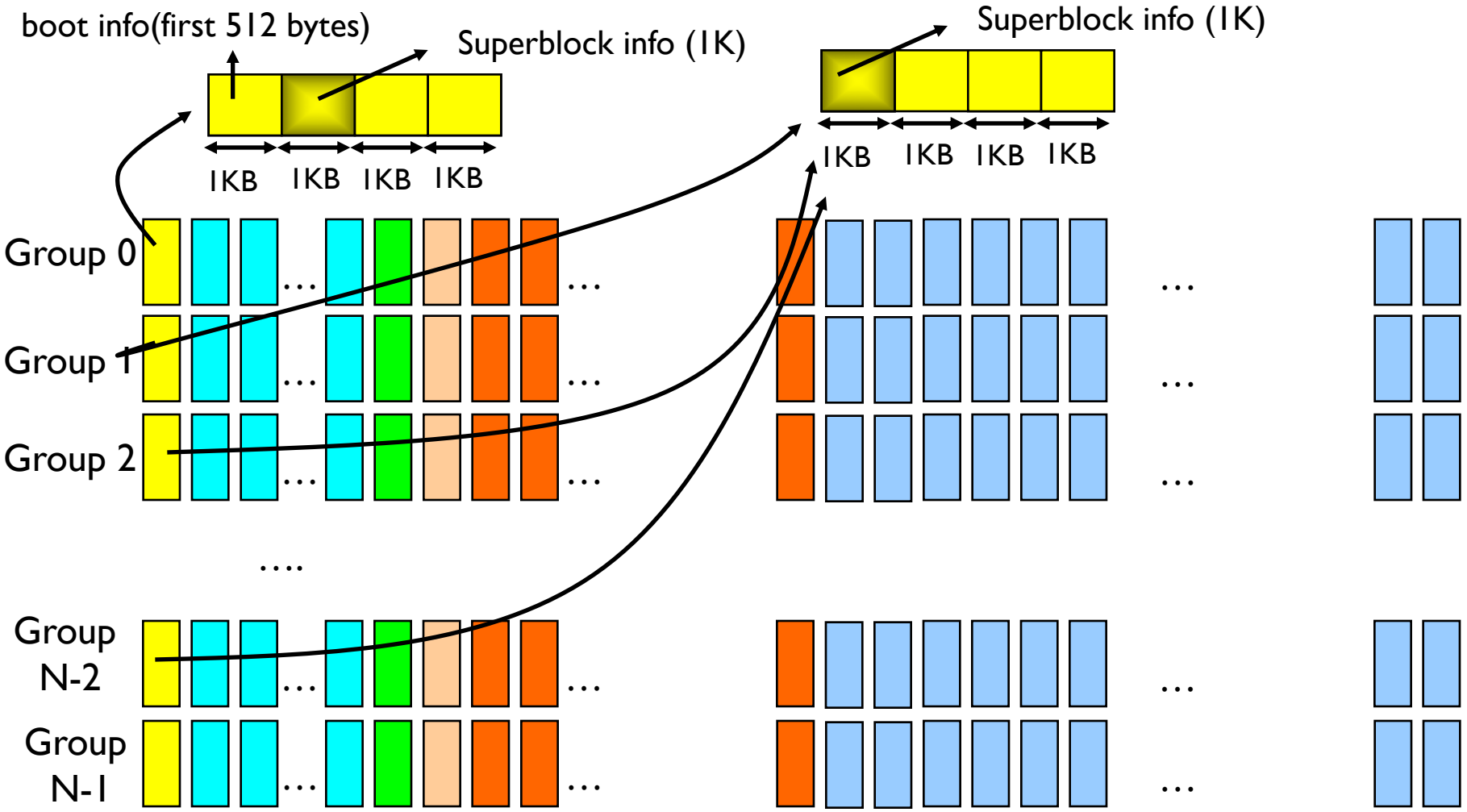
Ext3 file system: group structure



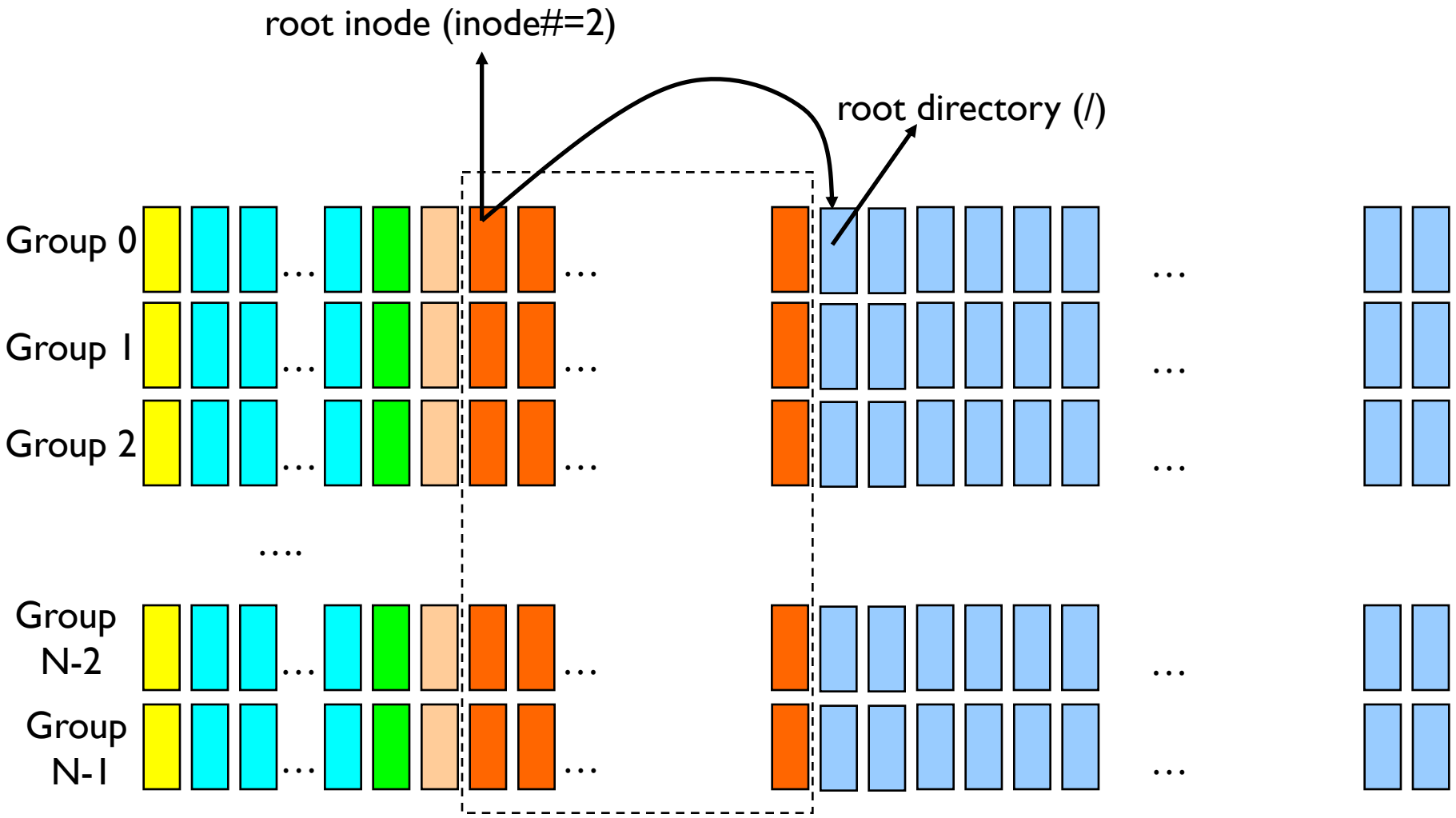
Ext3 file system: all groups and their structure



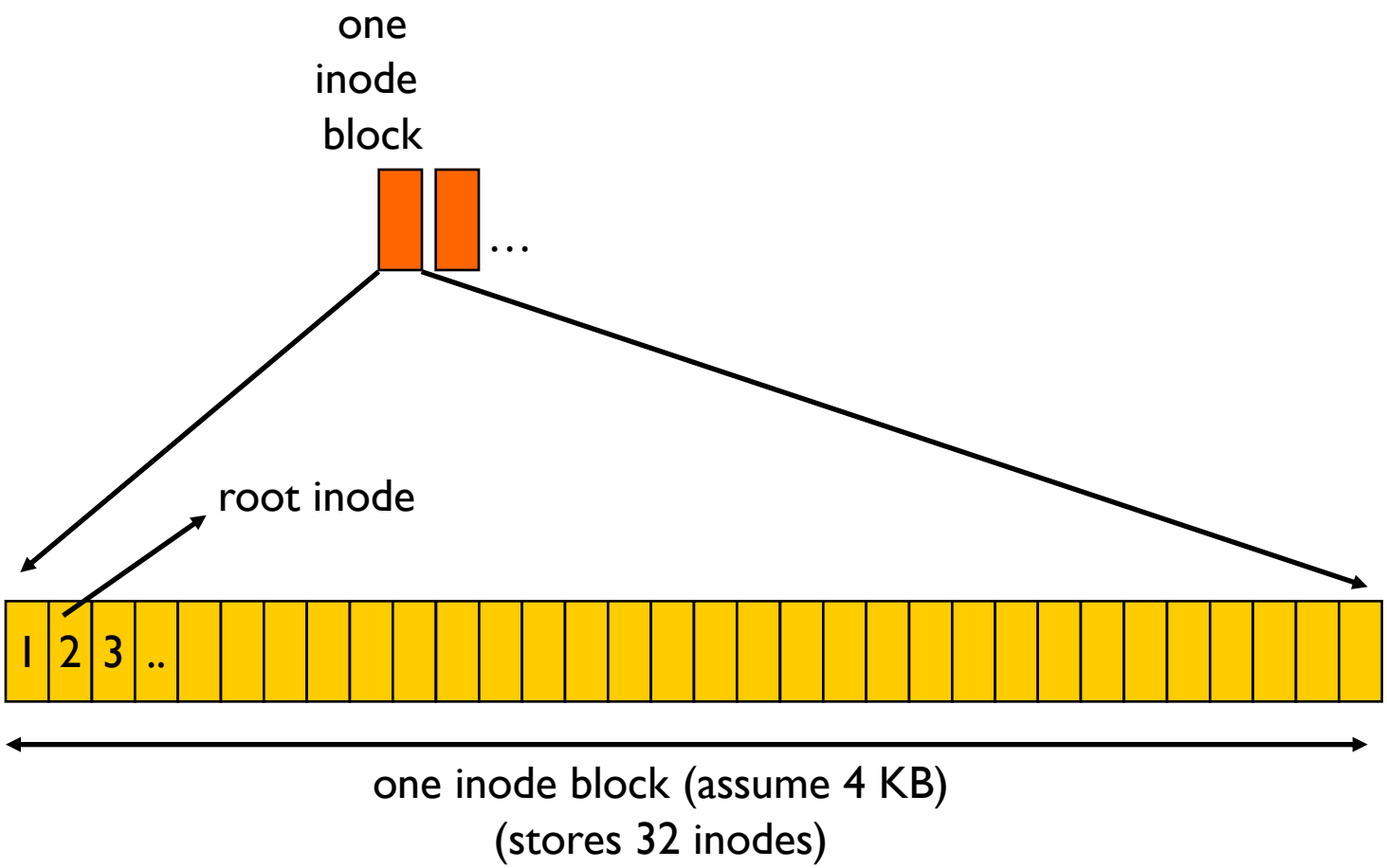
Ext3 file system: structure of the 1st block of each group



Ext3 file system: root inode and root directory



Ext3 file system: root inode



Ext3 file system: a real partition example

- We have a ~28 GB harddisk partition

number_of_groups =
block count / blocks_per_group
= 7323624 / 32768
= 223.49 => 224 groups
Groups from 0 to 223

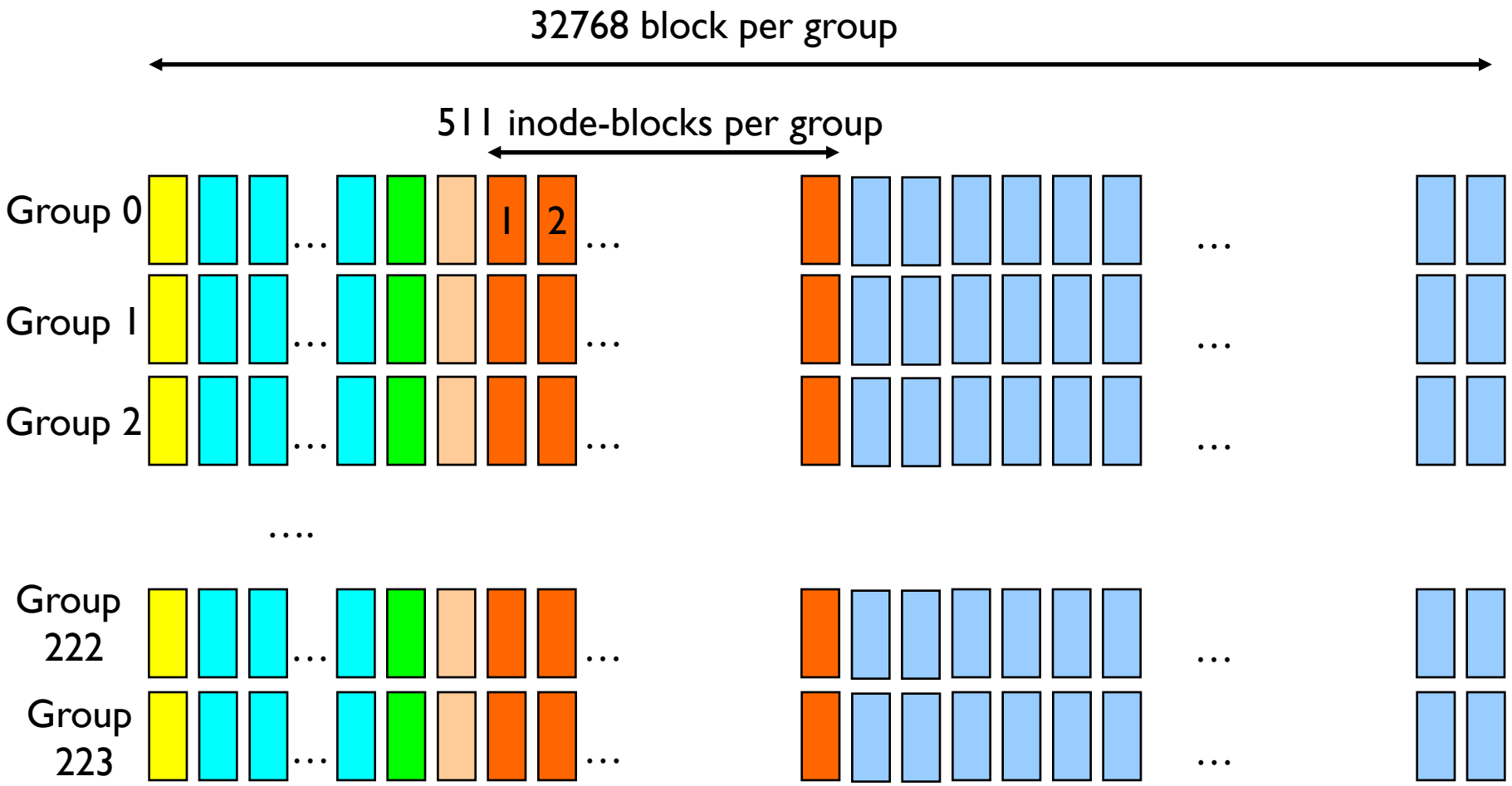
Inode size = 128 KB (can be 256 as well!)
Each block can contain 32 inodes
(4 KB / 128 bytes = 32)

There are 16352 inodes per group
16352/32 = 511 blocks required
to keep that many inodes in a group

superblock info

```
…
Filesystem OS type:       Linux
Inode count:              3662848
Block count:              7323624
Reserved block count:     366181
Free blocks:              4903592
Free inodes:              3288736
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      1024
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         16352
Inode blocks per group:   511
….
```

Ext3 file system: a real partition example



Ext3 file system: superblock structure

/usr/include/linux/ext3_fs.h

```
/*
 * Structure of the super block
 */
struct ext3_super_block {
/*00*/   __le32  s_inodes_count;         /* Inodes count */
         __le32  s_blocks_count;         /* Blocks count */
         __le32  s_r_blocks_count;       /* Reserved blocks count */
         __le32  s_free_blocks_count;    /* Free blocks count */
/*10*/   __le32  s_free_inodes_count;    /* Free inodes count */
         __le32  s_first_data_block;     /* First Data Block */
         __le32  s_log_block_size;       /* Block size */
         __le32  s_log_frag_size;        /* Fragment size */
/*20*/   __le32  s_blocks_per_group;     /* # Blocks per group */
         __le32  s_frags_per_group;      /* # Fragments per group */
         __le32  s_inodes_per_group;     /* # Inodes per group */
         __le32  s_mtime;                /* Mount time */
…
…
}
```

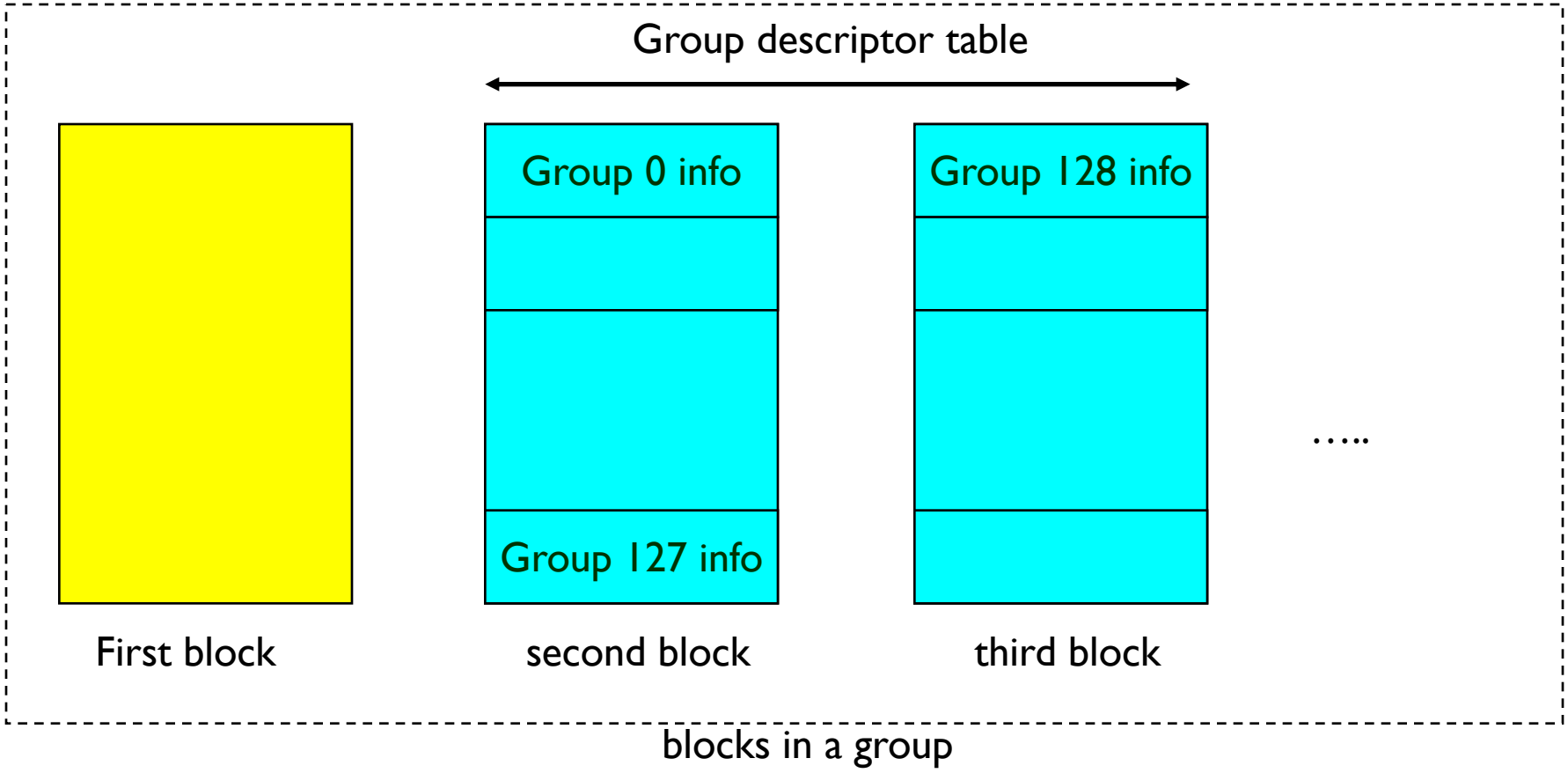
Ext3 file system group descriptors

- The number of blocks allocated for GDT table and reserved blocks may no be the name for each group. Therefore, the group decscriptor for a group tells where the inode bitmap and inode table starts.

```
struct ext3_group_desc
{
        __le32  bg_block_bitmap;        /* Blocks bitmap block */
        __le32  bg_inode_bitmap;        /* Inodes bitmap block */
        __le32  bg_inode_table;         /* Inodes table block */
        __le16  bg_free_blocks_count;   /* Free blocks count */
        __le16  bg_free_inodes_count;   /* Free inodes count */
        __le16  bg_used_dirs_count;     /* Directories count */
        __u16   bg_pad;
        __le32  bg_reserved[3];
};
```

Gives info about a group
Size of group descriptor is 32 bytes

Ext3 file system group descriptors



inodes

- Each inode keeps info about a file or directory
- Inode 2 is the inode for the root directory
- Inode numbers start with 1.

- Given inode number, it is east to compute on which group it is located.

inode structure

```
struct ext3_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
…
…
        __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[EXT3_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
…..
}
```

Directory entries

- A directory is a file that can occupies one or mode blocks.
- For example, root directory occupies one block.
- Directory is a sequence of entries.
- There is one entry per file
  - The entry points to the inode of the file. Namely it stores the inode number.
  - From the inode number (which is an index to the inode table), it is easy to compute the group# and index into the inode table in that group. The group descriptor also tells where the inode table in that group starts (disk blocks address). In this way we can reach to the disk block containing the inode.
- When we have the inode of a file, we can get further information about the file, like it data block addresses, attributed, etc.

Directory entry structure

/usr/iinclude/linux/ext3_fs.h

```
struct ext3_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT3_NAME_LEN];    /* File name */
};
```

Directory entry structure: file types

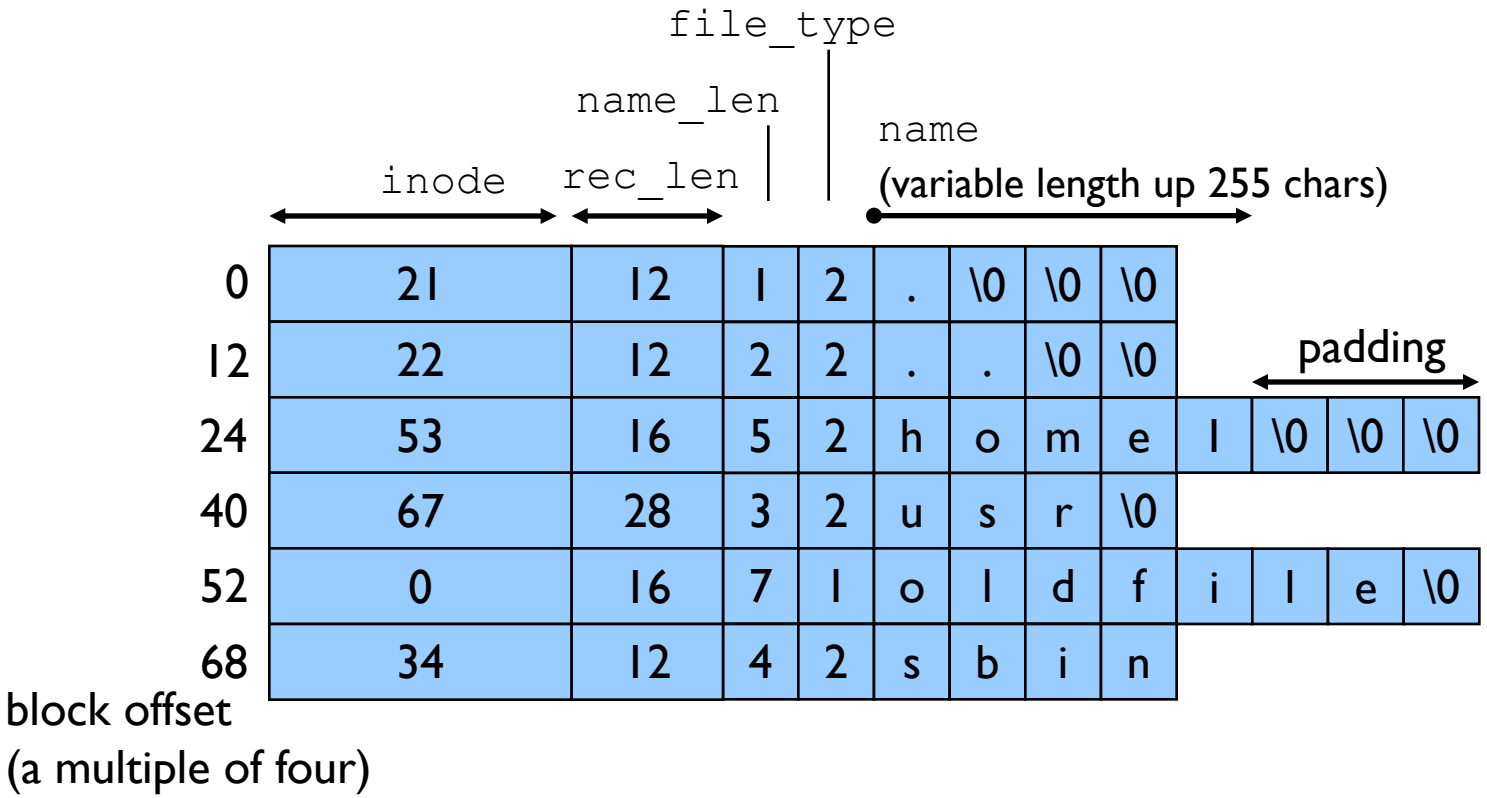
```
#define EXT3_FT_REG_FILE        1  /* regular file */
#define EXT3_FT_DIR             2  /* directory */
#define EXT3_FT_CHRDEV          3  /* char device file */
#define EXT3_FT_BLKDEV          4  /* block device file */
#define EXT3_FT_FIFO            5  /* fifo file */
#define EXT3_FT_SOCK            6  /* socket */
#define EXT3_FT_SYMLINK         7  /* symbolic link */
```

some file types have nothing to do with disk. They correspond to some other objects, like network connections, IPC objects, hardware devices, etc.

Example directory content

```
root directory (/) content
type=2 inode=2          name = .
type=2 inode=2          name = ..
type=2 inode=11         name = lost+found
type=2 inode=915713     name = etc
type=2 inode=1945889    name = proc
type=2 inode=2959713    name = sys
type=2 inode=2534561    name = dev
type=2 inode=1373569    name = var
type=2 inode=3008769    name = usr
type=2 inode=1586145    name = opt
type=2 inode=3270401    name = bin
type=2 inode=1177345    name = boot
type=2 inode=3482977    name = home
type=2 inode=130817     name = lib
type=2 inode=3057825    name = media
type=2 inode=2665377    name = mnt
…
```

Example directory content



There are 6 entries in this directory. Each entry starts at an offset that is multiple of 4.

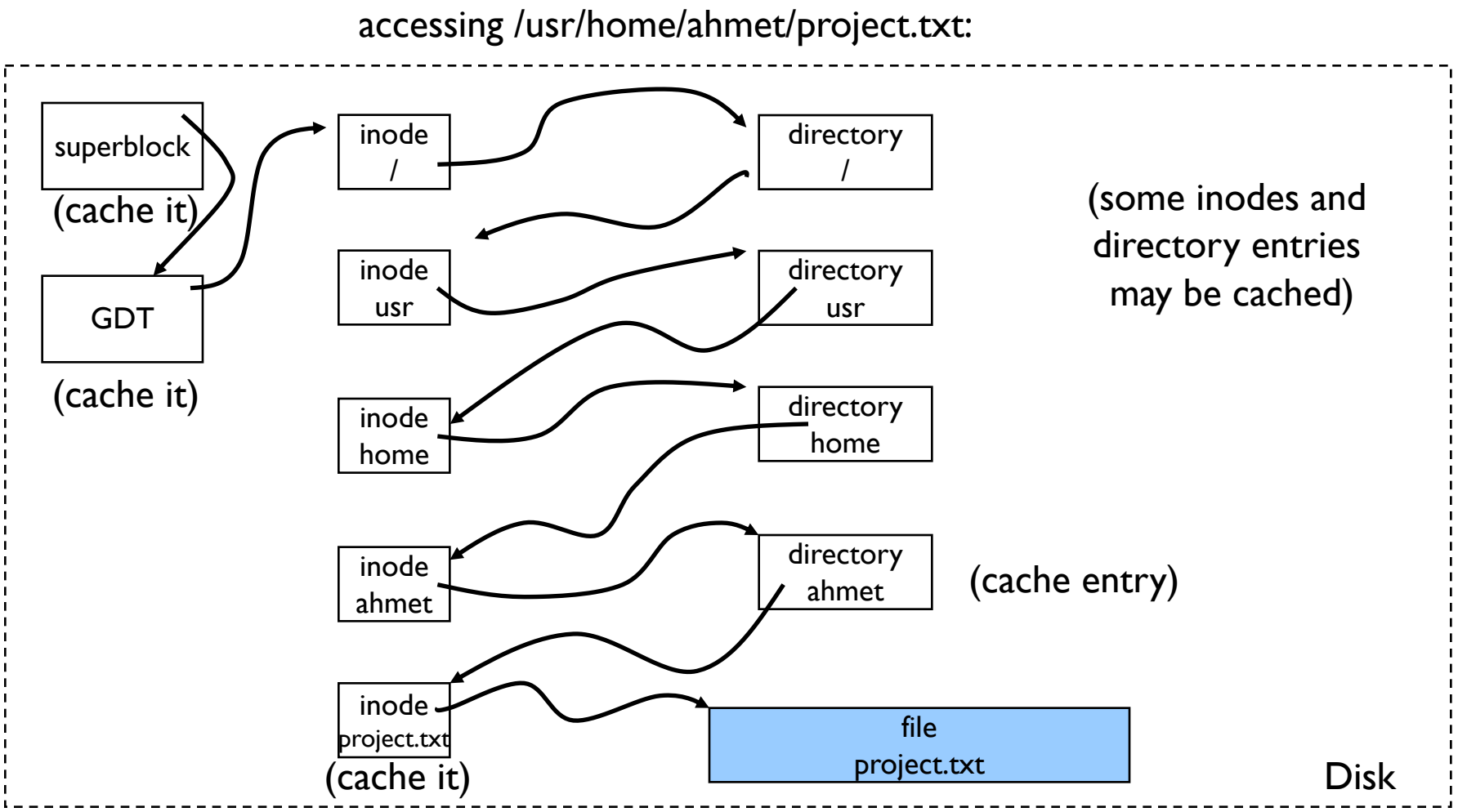
Searching for a file

- Given a filename, /usr/home/ahmet/project.txt, how can we locate it? For example while opening the file and then reading/writing the file.
- The filesystem may do the following steps:
  - Parse the pathname and divide into subdirectory names and file name:
    - /
    - usr
    - home
    - ahmet
    - project.txt
  - From root inode to go to the block that contains the root directory (hence go to the root directory).
  - Search there for an entry "usr". That entry will tell us the inode number of subdirectory usr (which is also considered a file; therefore has an inode number),
  - Access the inode for "usr" (we can compute the block number containing the inode quite easily).

Searching for a file

- The inode for “usr” will tell us which block(s) contains the “usr” directory.
- Go to that (those blocks) and access the “usr” directory information (a sequence of directory entries).
- There search for entry “home”. That entry will give us inode info for “home”.
- Access inode for “home” and obtain the block numbers containing the “home” directory information.
- Go to those blocks (i.e. to “home” directory).
- Search home directory entries for “ahmet”. The corresponding entry will tell the inode number for directory “ahmet”.
- Access inode for “ahmet” and then access directory information.
- In directory info “ahmet”, search for entry “project.txt”. The entry will tell where the inode for “project.txt” is.
- Access the inode for “project.txt”. It will tell the data block number for the file. Access those block to read/write the file.

Searching for a file



References

- Operating System Concepts, Silberschatz et al.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- OSTEP, Remzi Arpaci-Dusseau et al.
- File System Design for an NFS File Server Appliance (WAFL paper).