DCS216 Operating Systems

Lecture 26
File Systems Implementation

Jun 12th, 2024

Instructor: Xiaoxi Zhang
Sun Yat-sen University

- **Content**
  - File-System Structure
    - File-System Layers
    - On-Storage Structures
    - In-Memory Structures
    - Partitions and Mounting
    - Virtual File Systems
  - The Very Simple File System (VSFS)
  - Directory Implementation
  - Allocation Methods
  - Free-Space Management
  - Efficiency and Performance
  - Recovery
  - Example: The WAFL File System

File System Structure

Overview

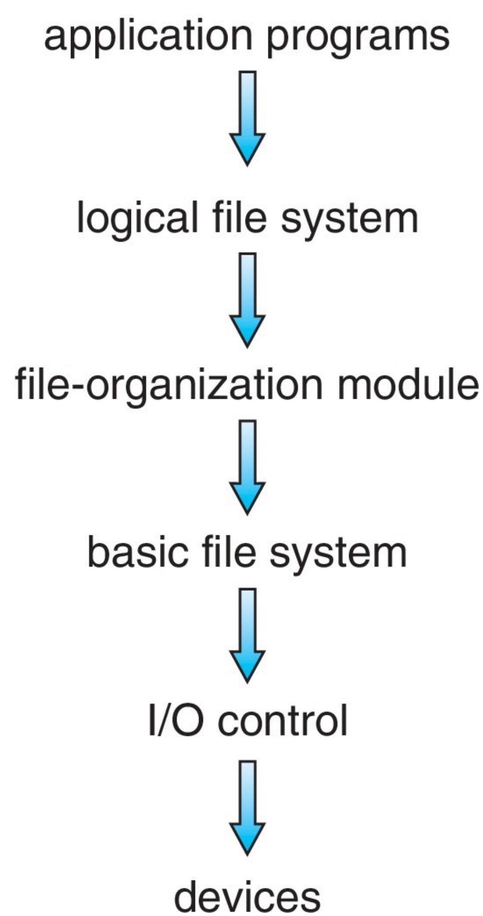
- File Structure
  - Logical storage unit
  - Collection of related information
- File System resides on **secondary storage** (**disks**)
  - Provides user interface to storage, mapping logical to physical (blocks)
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place **rewrite** (**原地重写**) and random access
  - I/O transfers performed in blocks of sectors (usually **512B** or **4096B**)
    - In Linux ext2/ext3, block size == 4096 Bytes by default.

```
$ sudo tune2fs -l /dev/sda1 | grep 'Block size'
Block size:                 4096
```

- **File Control Block** (**FCB**)
  - storage structure consisting of info about a file (or directory, a special type of file).
  - In UNIX-like systems, **FCB** is also called inode.

File System Layers

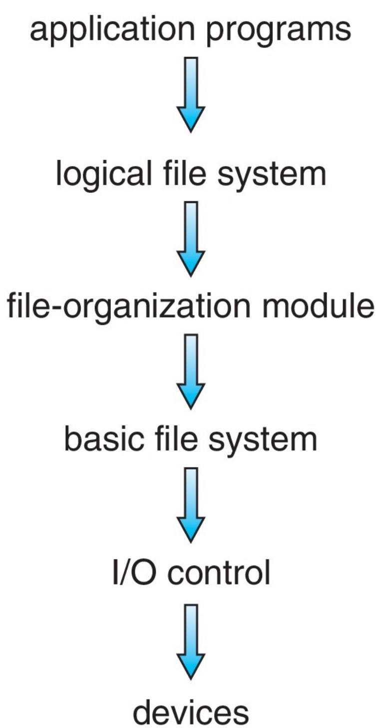
- File System is organized into **layers.**
- Each level uses the features of lower levels to create new features for use by higher levels.



File System Structure

File System Layers

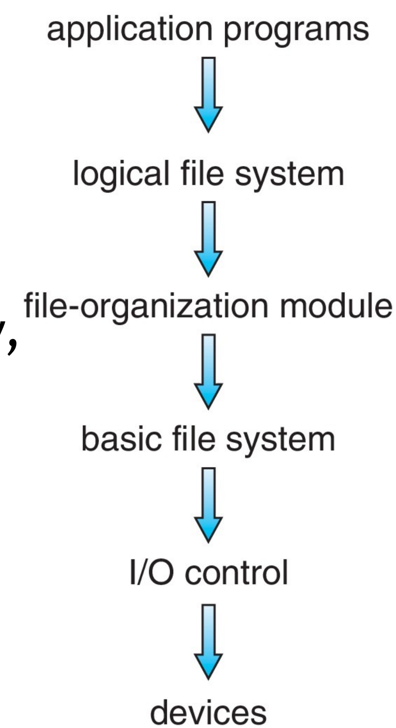
- **I/O Control** level consists of device drivers and interrupt handlers to transfer info between the main memory and disks.
  - **Device drivers** manage I/O devices, e.g., given commands like: **read** *drive 1, cylinder 72, track 2, sector 10* into **mem** location #1060, it outputs low-level hardware specific **commands** to hardware controller.
- **Basic File System** (called "block I/O subsystem" in Linux) needs only to issue generic commands to the appropriate device driver to **read** and **write** blocks on the storage device.
  - It issues commands to the drive based on logical block addresses like "retrieve **block** #123".
  - also manages memory buffers and caches (allocation, freeing, replacement)
    - **Buffers** hold data in transit.
    - **Caches** hold frequently used data.



File System Structure

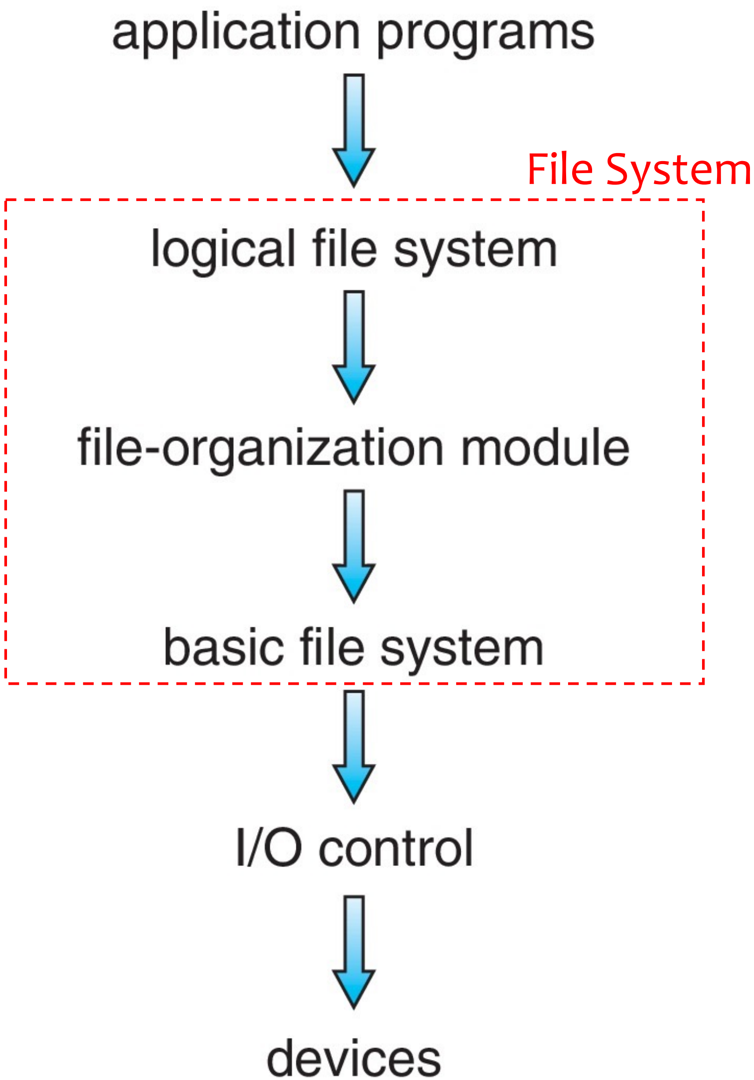
File System Layers

- **File Organization Module** understands files and their logical blocks.
  - Translates logical block # to physical block #.
  - Each file's logical blocks are numbered from `0` through `N`.
  - Also manages free space, disk allocation.
- **Logical File System** manages metadata information including all of the file-system structure except the actual data (contents of files).
  - Translates file name into file number, file handle, location by maintaining **file control blocks** (inodes in UNIX).
  - Directory management
  - Protection
- **Layering** useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- **Logical Layers** can be implemented by any coding method according to OS designer.



File System Structure

File System Layers



`fopen(), fread() in C Library`

`open(), read(), write(), close()`

`logical blocks ⇔ physical blocks`

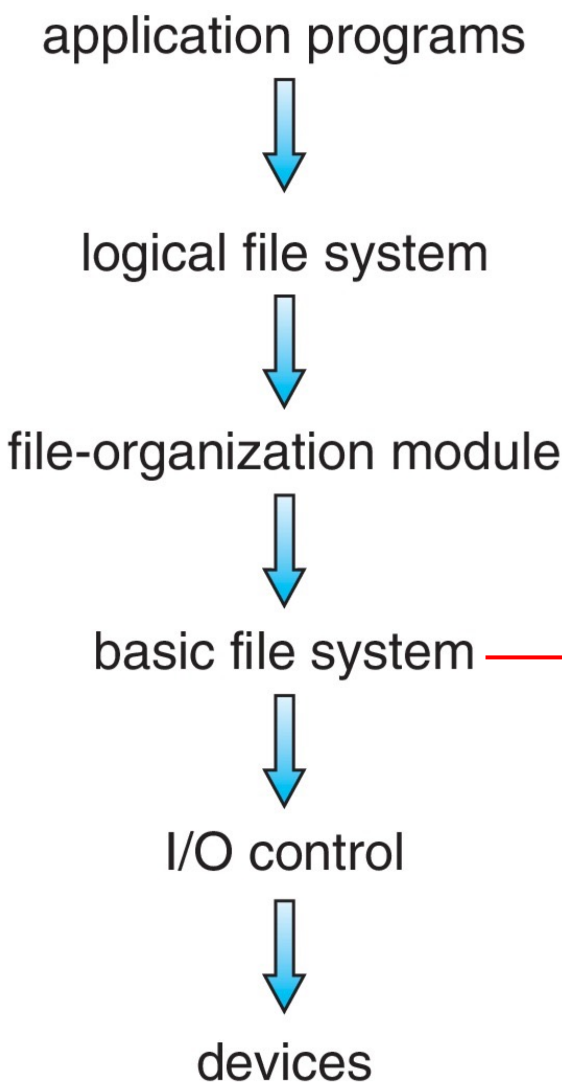
`struct block_device_operations{};`

`ioctl(), etc.`

`hardware-level commands`

File System Structure

File System Layers




```c
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    int (*release) (struct gendisk *, fmode_t);
    int (*locked_ioctl) (struct block_device *, fmode_t,
                         unsigned, unsigned long);
    int (*ioctl) (struct block_device *, fmode_t, unsigned,
                  unsigned long);
    int (*compat_ioctl) (struct block_device *, fmode_t,=
                         unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t,
                          void **, unsigned long *);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *,
                  struct hd_geometry *);
    blk_qc_t (*submit_bio) (struct bio *bio);
    struct module *owner;
}
```

File System Structure

File System Layers

- Great variety of file systems, sometimes different types of FS within an OS at the same time:
  - Each with its own format.
  - CD-ROM is **ISO 9660**
  - UNIX has **UFS** (UNIX File System), **FFS** (Fast File System, from Berkeley)
  - Windows has **FAT**, **FAT32**, **NTFS** as well as floppy, CD, DVD Blu-ray
  - Linux has more than 130 types, with extended file system **ext3** and **ext4** as the major type
  - New ones still arriving
    - **ZFS**
    - **GoogleFS**
    - Oracle **ASM**
    - **FUSE** (**F**ile System in **Use**rspace)

File System Structure

- Several **On-Storage** and **In-Memory** structures are used to implement a **file system.**
- **On-Storage**:
  - Boot Control Block
  - Volume Control Block
  - Directory Structure
  - File Control Block (**FCB**)
- **In-Memory**:
  - Mount Table
  - Directory Structure Cache
  - System-wide Open File Table
  - Per-process Open File Table
  - Buffers

On-Storage (On-Disk) Structures

- **Boot Control Block 引导控制块** (per volume) contains info needed by the system to boot OS from that volume.
  - Needed if volume contains OS, usually first block of volume, i.e., block `0`.
  - Also called the boot block (in Linux), or partition boot sector (in Window).
  - (*Traditionally*) When the system boots up, the **BIOS** or **UEFI firmware** loads the boot loader from the **Boot Control Block** to initialize the OS.

```
$ sudo fdisk -l /dev/sda
Disk /dev/sda: 256 GiB, 274877906944 bytes, 536870912 sectors
Disk model: QEMU HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 40AC494B-4E2E-4063-8AEF-46A00284C3C3

Device         Start       End   Sectors  Size Type
/dev/sda1       2048      4095      2048    1M BIOS boot
/dev/sda2       4096   4198399   4194304    2G Linux filesystem
/dev/sda3    4198400 536868863 532670464  254G Linux filesystem
```

The boot volume: /dev/sda1

On-Storage (On-Disk) Structures

- **Boot Control Block 引导控制块** (per volume) contains info needed by the system to boot OS from that volume.
  - Needed if volume contains OS, usually first block of volume, i.e., block `0`.
  - Also called the boot block (in Linux), or partition boot sector (in Window).
  - (*Traditionally*) When the system boots up, the **BIOS** or **UEFI firmware** loads the boot loader from the **Boot Control Block** to initialize the OS.

```
$ sudo dd if=/dev/sda1 bs=512 count=1 | hexdump -C
00000000  52 e8 28 01 74 08 56 be  33 81 e8 4c 01 5e bf f4  |R.(.t.V.3..L.^..|
00000010  81 66 8b 2d 83 7d 08 00  0f 84 e9 00 80 7c ff 00  |.f.-.}.......|..|
00000020  74 46 66 8b 1d 66 8b 4d  04 66 31 c0 b0 7f 39 45  |tFf..f.M.f1...9E|
00000030  08 7f 03 8b 45 08 29 45  08 66 01 05 66 83 55 04  |....E.)E.f..f.U.|
00000040  00 c7 04 10 00 89 44 02  66 89 5c 08 66 89 4c 0c  |......D.f.\.f.L.|
00000050  c7 44 06 00 70 50 c7 44  04 00 00 b4 42 cd 13 0f  |.D..pP.D....B...|
00000060  82 bb 00 bb 00 70 eb 68  66 8b 45 04 66 09 c0 0f  |.....p.hf.E.f...|
00000070  85 a3 00 66 8b 05 66 31  d2 66 f7 34 88 54 0a 66  |...f..f1.f.4.T.f|
00000080  31 d2 66 f7 74 04 88 54  0b 89 44 0c 3b 44 08 0f  |1.f.t..T..D.;D..|
00000090  8d 83 00 8b 04 2a 44 0a  39 45 08 7f 03 8b 45 08  |.....*D.9E....E.|
000000a0  29 45 08 66 01 05 66 83  55 04 00 8a 54 0d c0 e2  |)E.f..f.U...T...|
000000b0  06 8a 4c 0a fe c1 08 d1  8a 6c 0c 5a 52 8a 74 0b  |..L......l.ZR.t.|
...
```

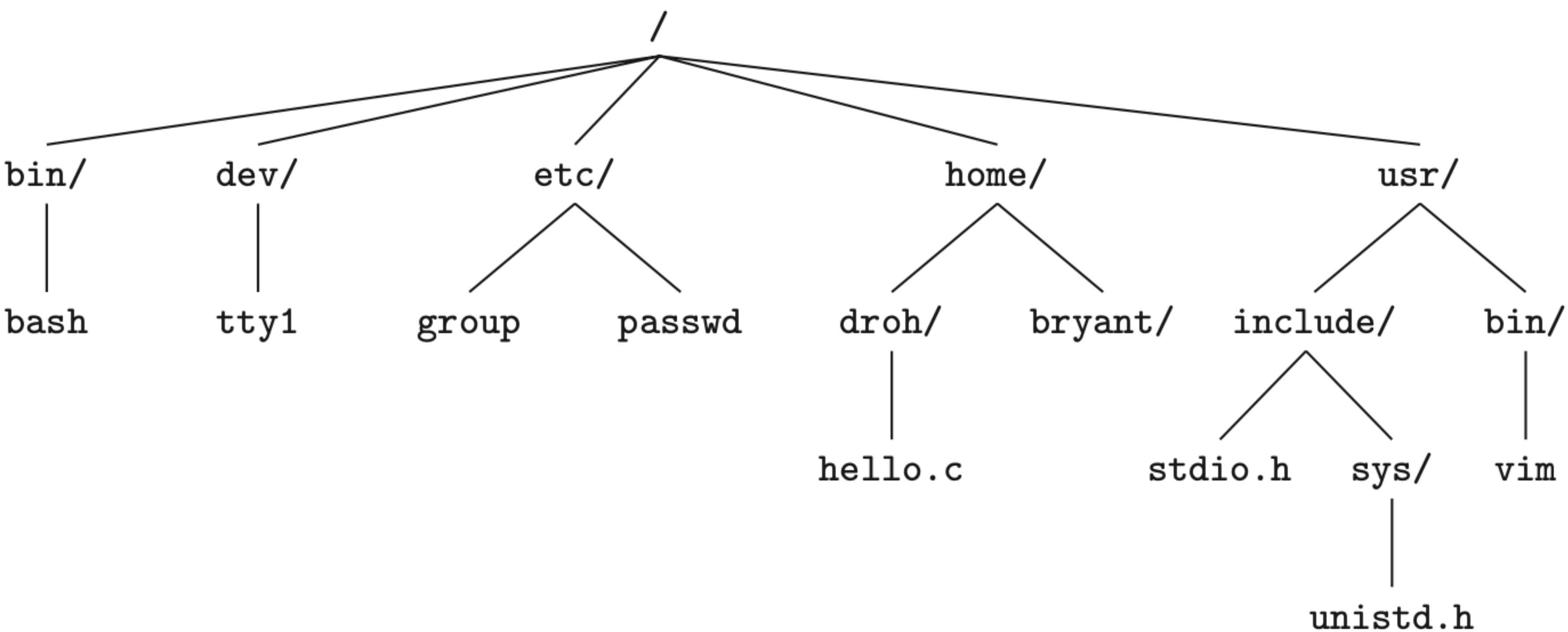
On-Storage (On-Disk) Structures

- **Volume Control Block 卷控制块** contains volume details.
  - Total number of blocks in the volume
  - **Block Size**.
  - Free-block count and pointers
  - Free FCB (inode) count and pointers
  - Also called superblock 超级块 (in Linux), master file table 主控文件表 (in Windows).
  - Typically in block 1 (the 2nd block after block `0`, the **Boot Control Block**), or block `0` (if the volume is not a boot volume)

```
$ sudo dumpe2fs /dev/sda2 | grep -i superblock
dumpe2fs 1.46.5 (30-Dec-2021)
  Primary superblock at 0, Group descriptors at 1-1
  Backup superblock at 32768, Group descriptors at 32769-32769
  Backup superblock at 98304, Group descriptors at 98305-98305
  Backup superblock at 163840, Group descriptors at 163841-163841
  Backup superblock at 229376, Group descriptors at 229377-229377
  Backup superblock at 294912, Group descriptors at 294913-294913
```

On-Storage (On-Disk) Structures

- **Directory Structure 目录结构** is used to organize the files.
  - In UNIX, this includes file names and associated inode numbers.
  - In Windows **NTFS**, it is stored in the master file table (i.e., **Volume Control Block**).



File System Structure

- **On-Storage (On-Disk) Structures**
  - **File Control Block 文件控制块** (per file) contains many details about the file.
    - It has a unique identifier number to allow association with a directory entry.
    - Also called inode In UNIX: inode number, permissions, size, dates, etc.
    - NTFS stores the **FCB** info in the master file table using relational DB structures.
    - A typical **File Control Block** (**FCB**):

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

- **In-Memory Structures**
  - The **in-memory** information is used for both file system management and performance improvement via caching. The data are loaded at mount time (加载时), updated during file system operations, and discarded at dismount (卸载时). Major **in-memory** structures:
    - **Mount Table (挂载表)** contains information about each mounted volume.
    - **System-wide Open File Table (系统打开文件表)** contains a copy of the FCB of each opened file, along with other information.
    - **Per-process Open File Table (单个进程的打开文件表)** contains **pointers** to the appropriate entries in the System-wide Open File Table, as well as other information, for all files that the current process has opened.

- **In-Memory Structures**
  - **Mount Table (挂载表)** contains information about each mounted volume.
    - We can use the `df` command to display mounted volumes.
    - For example, volume `/dev/sda3` is mounted at the root directory (`/`), while volume `/dev/sda2` is mounted at (`/boot`).
      - *Note that tmpfs is not actual physical volumes, but rather temporary file system in virtual memory.*

```
$ df -h
Filesystem                             Size  Used Avail Use% Mounted on
tmpfs                                   13G  1.9M   13G   1% /run
/dev/sda3                              249G   68G  170G  29% /
tmpfs                                   63G     0   63G   0% /dev/shm
tmpfs                                  5.0M     0  5.0M   0% /run/lock
tmpfs                                   63G     0   63G   0% /run/qemu
/dev/sda2                              2.0G  412M  1.4G  23% /boot
tmpfs                                   13G  4.0K   13G   1% /run/user/1000
```

File System Structure

In-Memory Structures

- **System-wide Open File Table** vs. **Per-process Open File Table**

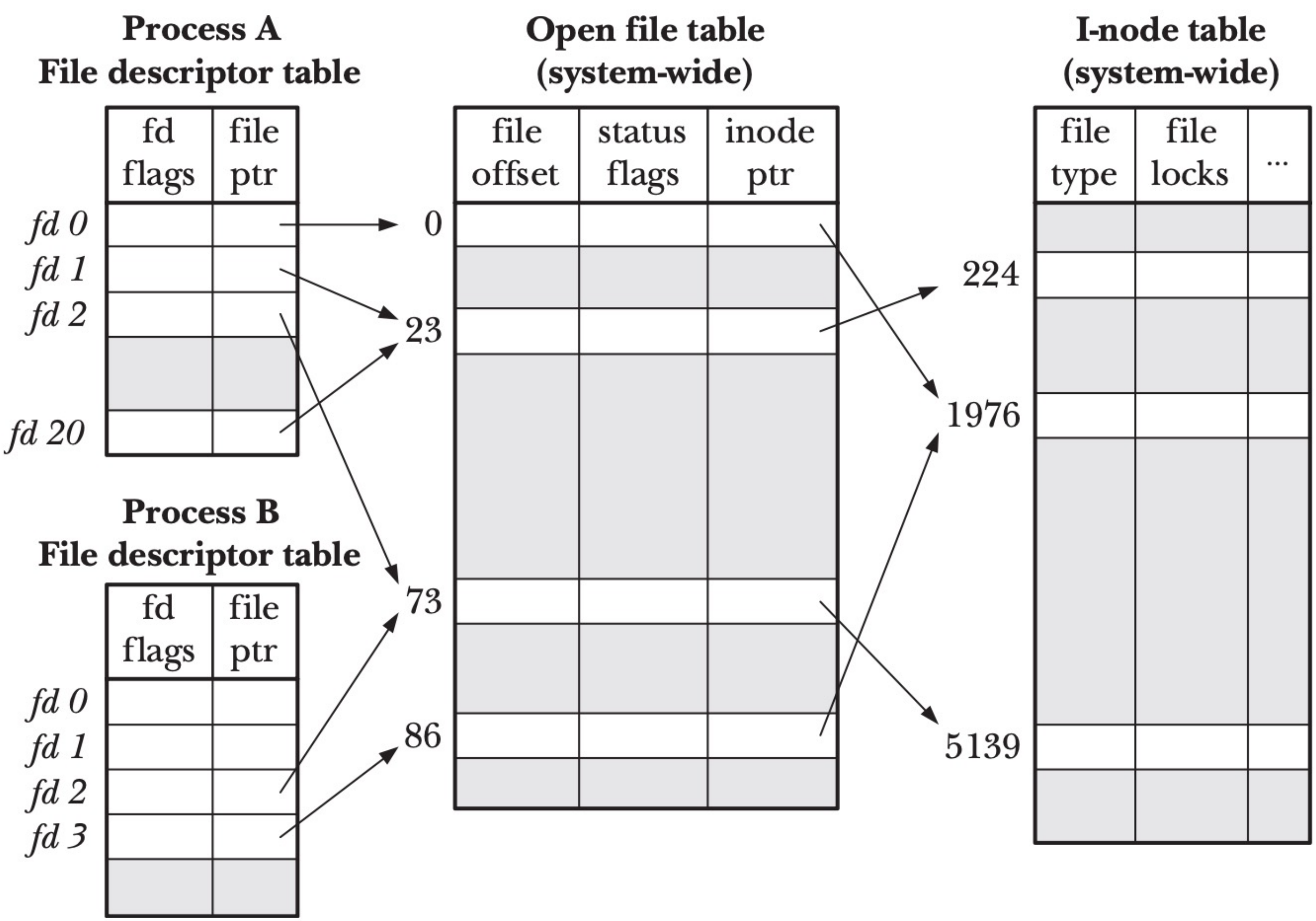
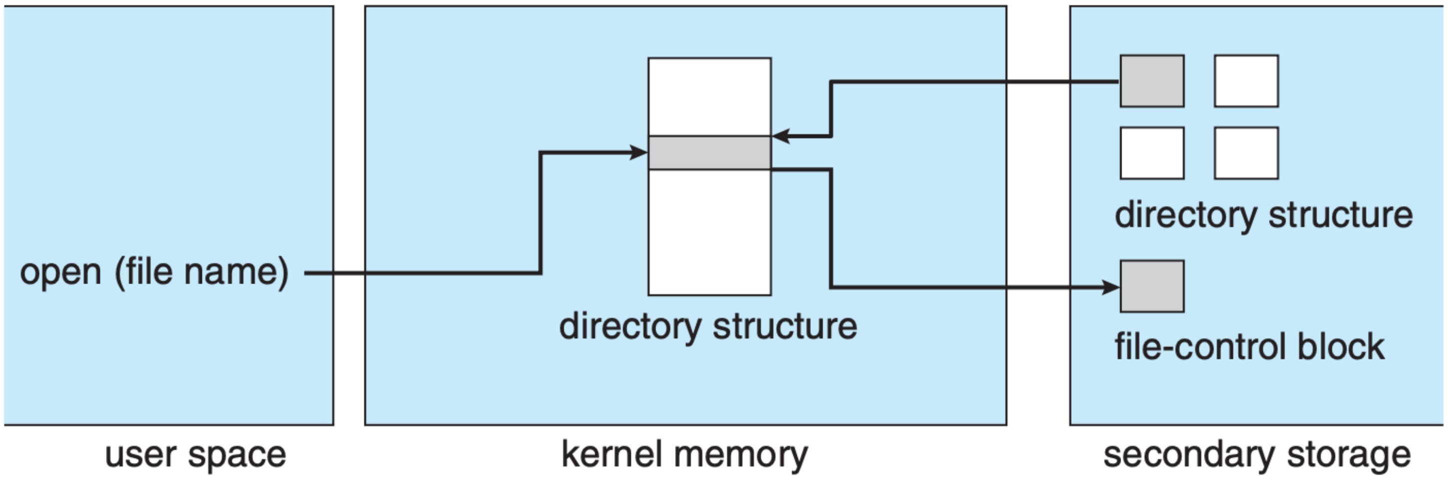


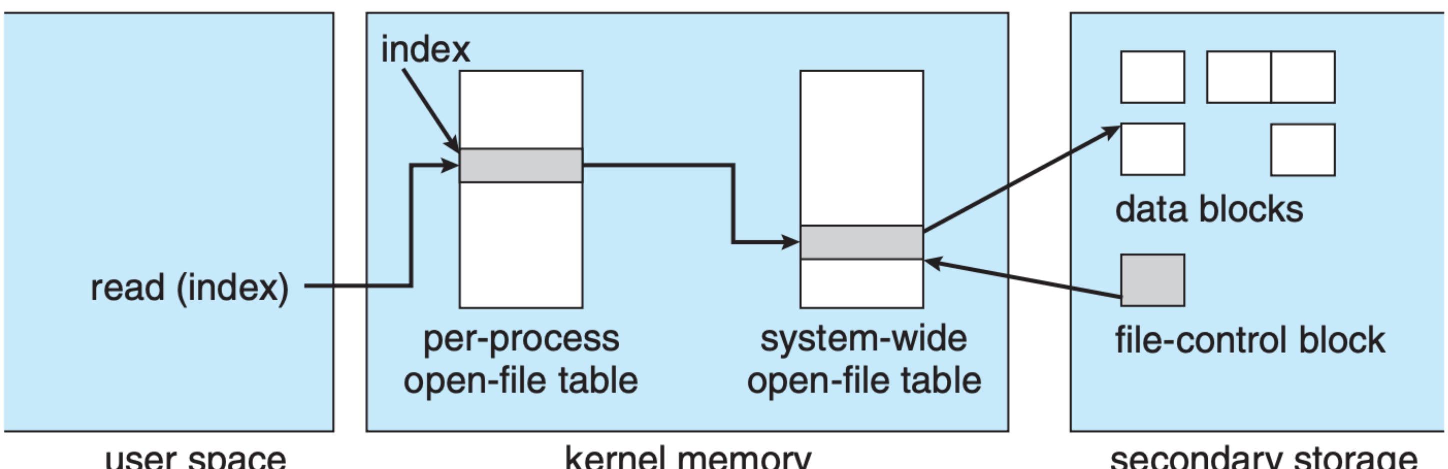
Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Usage of File System Structure

- necessary fs structs provided by OS when `open()`:



- necessary fs structs provided by OS when `read()`:

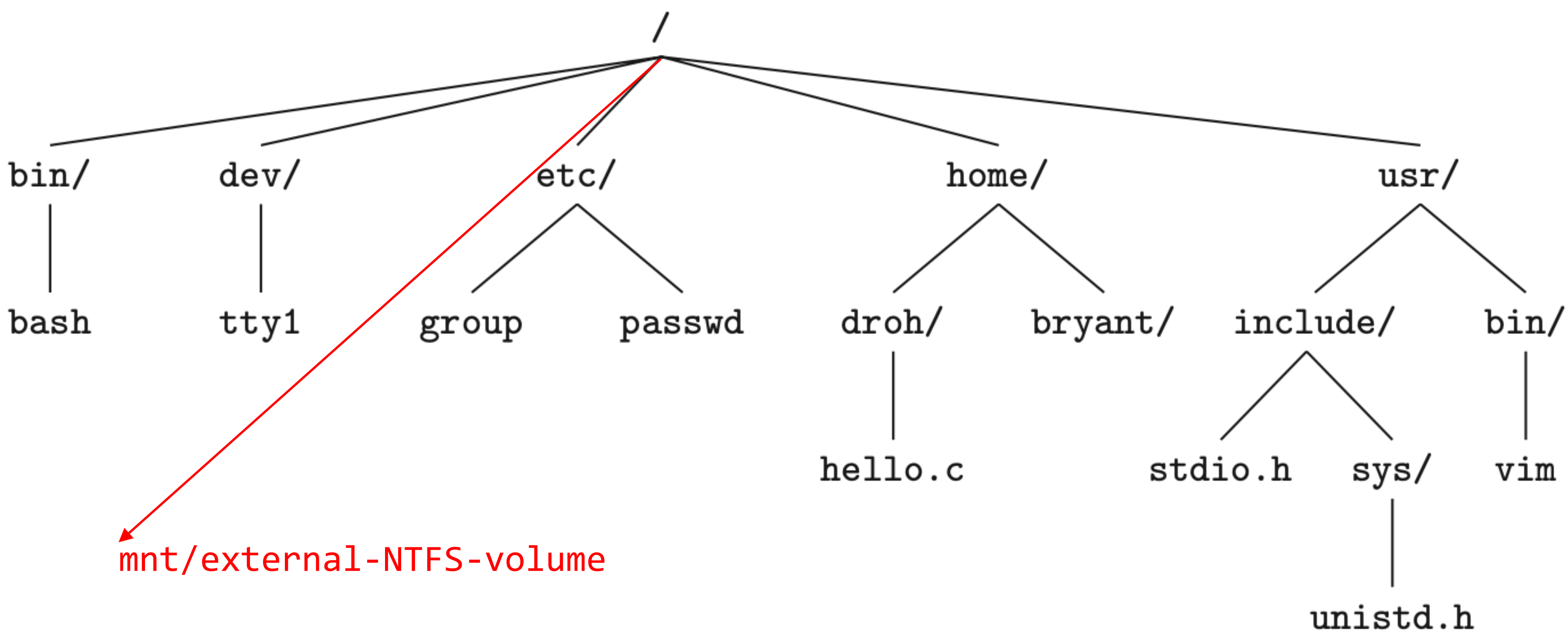


Partitions and Mounting

- **Partition (分区)** can be a volume containing a file system ("**cooked**") or **raw**.
  - **Raw** partition is just a sequence of blocks with no file system.
- **Boot Block** can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system.
  - It may also be a boot management program for multi-OS booting.
- **Root Partition (根分区)** contains the OS, other partitions can hold other OSes, other file systems, or just be raw.
  - Root partition is mounted at boot time.
  - Other partitions can be mounted automatically or manually.
- At **mount** (**挂载**) time, file system consistency is checked.
  - Is all metadata correct?
    - If not, fix it, and try again.
    - If yes, add to mount table, allow access.

Virtual File Systems

- Modern OSes must support multiple types of **File Systems.**
- But how does an OS allow multiple types of file systems (with different implementations, e.g., **ext3** and **NTFS**) to be integrated into a single directory structure?
  - different file and directory representations.

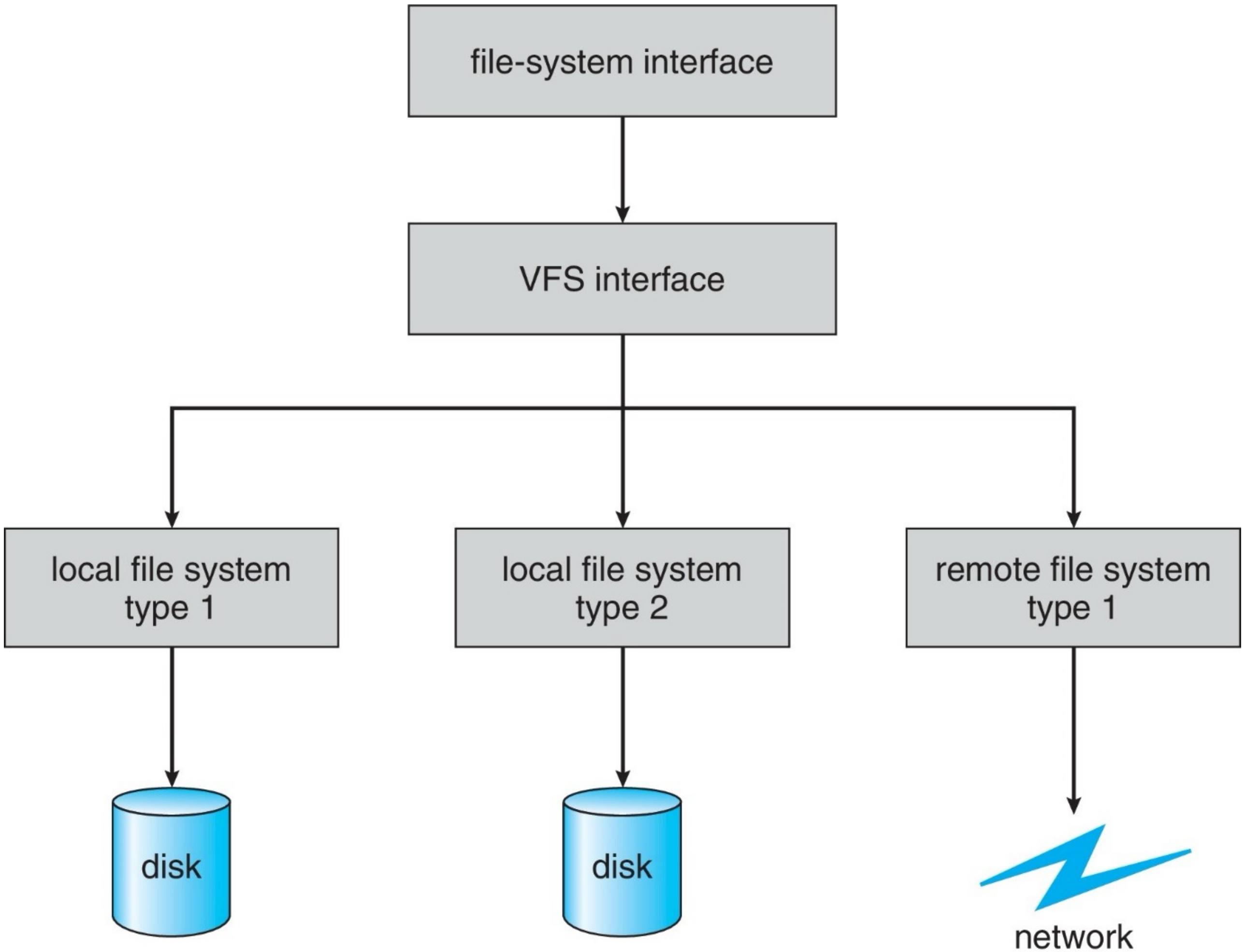


File System Structure

Virtual File Systems

- **Virtual File Systems 虚拟文件系统** (**VFS**) on UNIX provide an object-oriented way of implementing file systems.
- **VFS** allows the same system call interfaces (**APIs**, `e.g.`, `open`, `read`, `write`) to be used for **different types of file systems**.
  - **VFS** separates FS generic operations from implementation details.
  - Implementation can be one of many file system types, or network file system (**NFS**).
    - Implements vnodes which hold **inodes** or **network file details**.
  - Then dispatches operations to appropriate file system implementation routines
- The **API** is to the **VFS** interface, rather than any specific type of file system.

- **Virtual File Systems**
  - Schematic View of a Virtual File System.



File System Structure

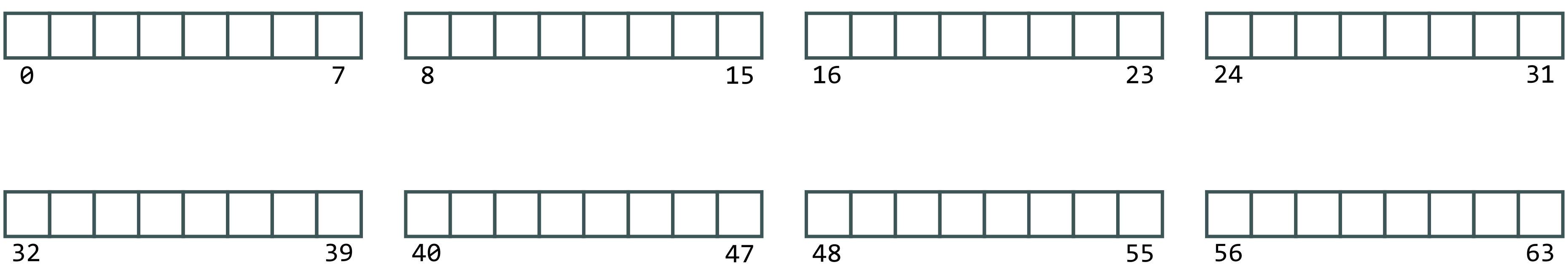
Virtual File Systems in Linux

- Four main object types defined by the Linux VFS:
  - The **inode** object ⇒ represents an individual file
  - The **file** object ⇒ represents an open file
  - The **superblock** object ⇒ represents an entire file system
  - The **dentry** object ⇒ represents an individual directory entry.
- Every object of one of these types contains a pointer to a function table, which has the addresses of the actual functions that **implement** the defined operations for that particular object.
  - For example, for the **file** object:

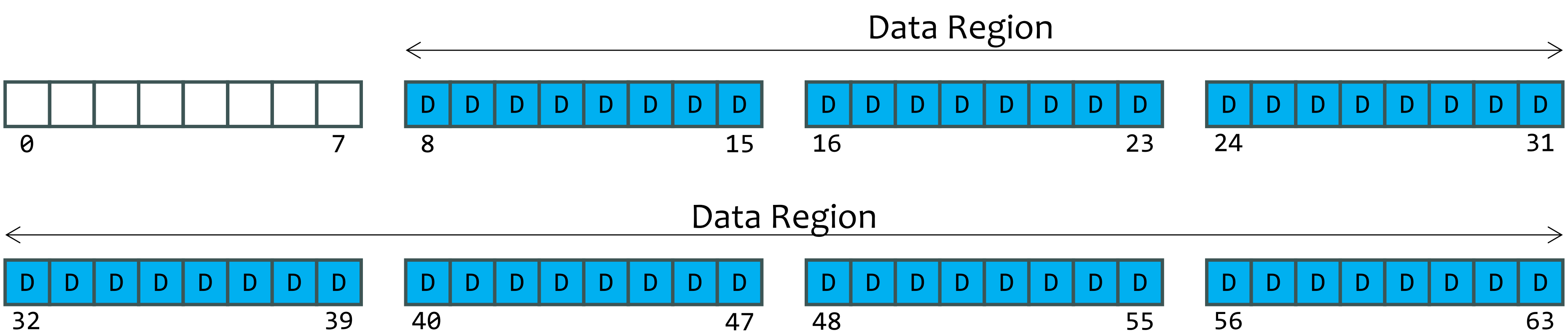
```
int open()              // open a file
int close()             // close a file
ssize_t read()          // read from a file
ssize_t write()         // write to a file
int mmap()              // memory-map a file
```

- **VSFS (Very Simple File System)**
  - Here, we introduce a simple file system implementation, known as **VSFS** (**V**ery **S**imple **F**ile **S**ystem).
    - a simplified version of a typical **UNIX** file system.
    - well-suited to introduce some of the basic on-disk structures, access methods, and various policies that are found in many modern file systems today.
  - The core problems:
    - How can we build a simple file system?
    - What structures are needed on the disk?
    - What do they need to track?
    - How are they accessed?

- **VSFS (Very Simple File System)**
  - On-disk organization:
    - block size: **4KB** (the most common case)
    - assume we have a really small disk, with just **64** blocks.
  - What do we need to **store** in these blocks **to build a file system**?

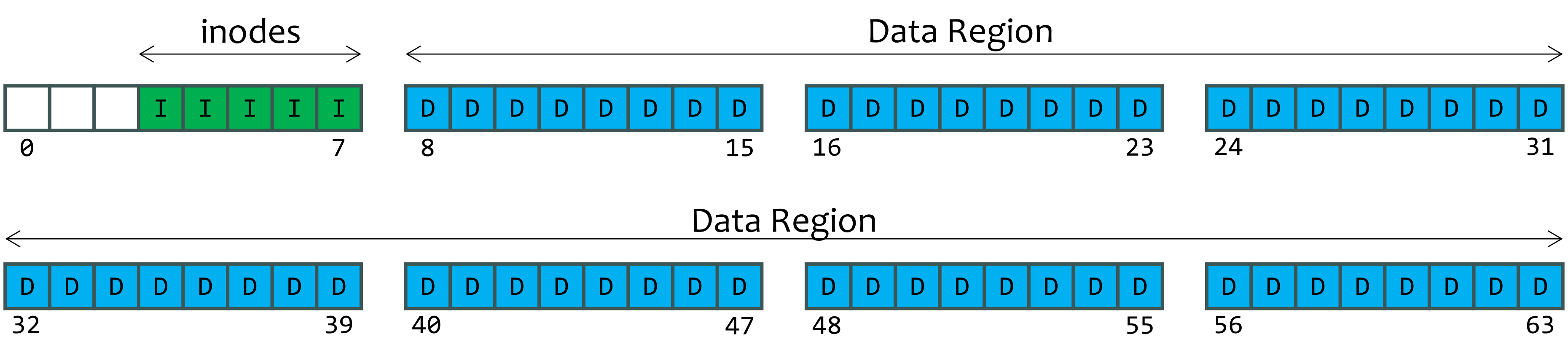


- **VSFS (Very Simple File System)**
  - Of course, the first thing that comes to mind is **user data.**
  - In fact, **most of the space** in any file system is (should be) **user data.**
    - Let's call the region of the disk we use for **user data** the **data region**.
    - for simplicity, we reserve a fixed portion of the disk for these blocks, e.g., the last **56** of **64** blocks on the disk:
    - The **data region** stores the actual **content** of all the files in this file system.



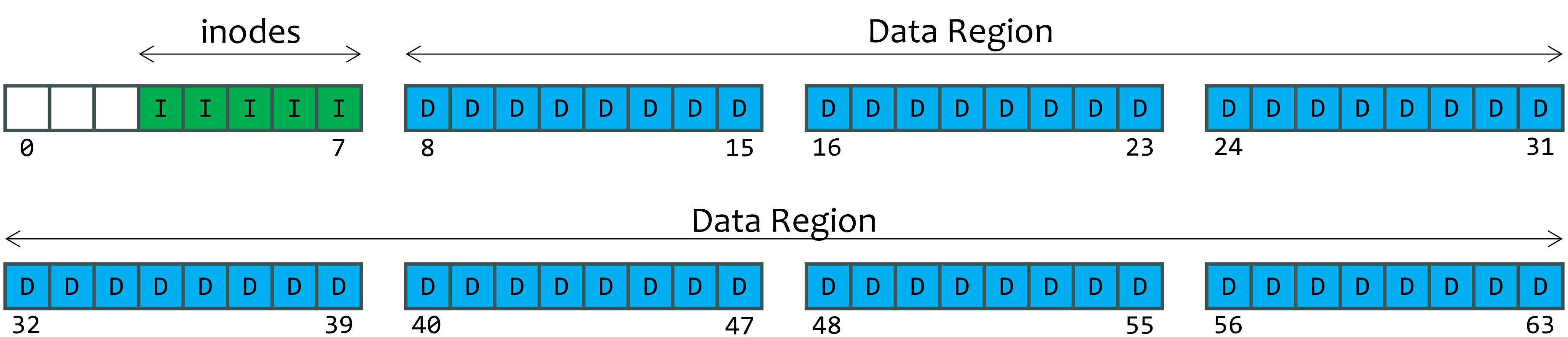
VSFS (Very Simple File System)

- The file system also has to track information about each file.
  - This information is a key piece of **metadata**, and tracks things such as:
    - which data blocks (in the data region) comprise a file
    - the size of the file
    - its owner and access rights
    - access and modify times
    - other information.
  - To store this information, file systems usually have a structure called an **inode** (or the more general name: **FCB**, **File Control Block**)



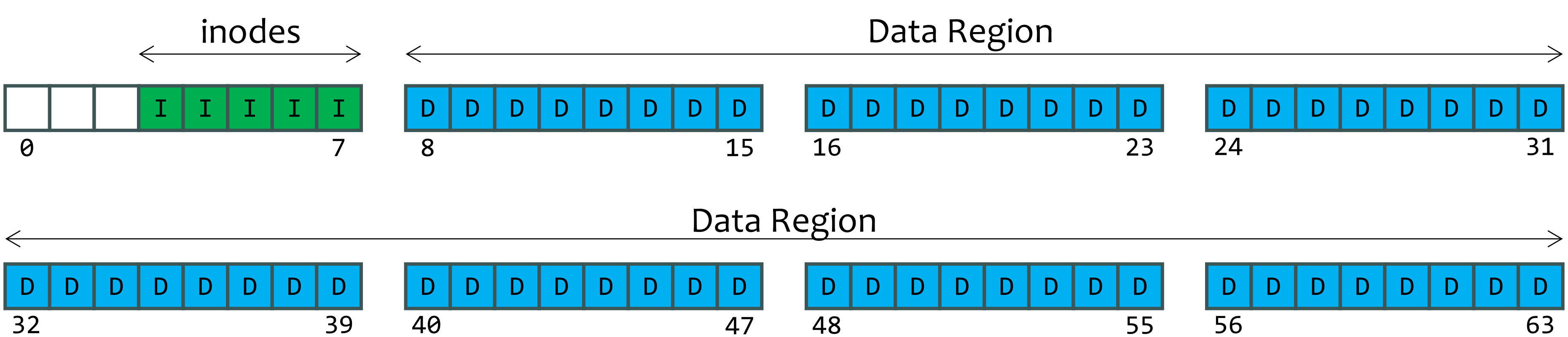
VSFS (Very Simple File System)

- To accommodate **inodes**, we'll need to reserve some space on the disk for them as well. We call this portion of the disk the **inode table**, which simply holds an array of on-disk **inodes**.
  - In our simple example, we use **5** of the 64 blocks for inodes (denoted by **`I`**'s in the diagram).
  - Note that **inodes** are typically not very big. We assume 256 bytes per **inode** $\Rightarrow$ a 4KB block can hold 16 **inodes**.
    - Thus, our file system contains up to $\frac{4KB}{256B} \times 5 = \frac{4096B}{256B} \times 5 = 80$ **inodes** in total, which represents the **total number of files** possible.



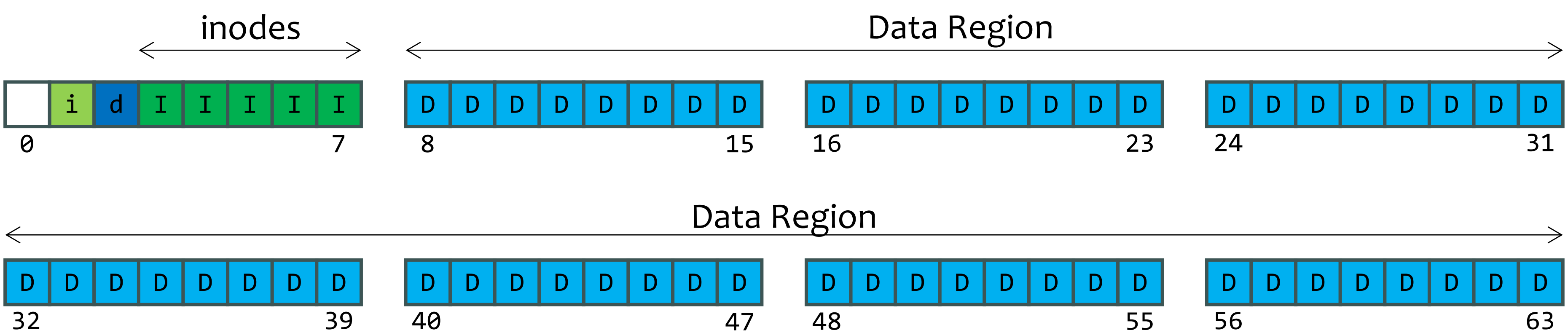
VSFS (Very Simple File System)

- Our file system thus far has **data blocks** (**D**) and **inodes** (**I**). But a few things are still missing.
  - One major concern is, how do we track which **inodes** or **data blocks** are free or allocated.
  - Such **allocation structures** are thus a requisite element in any file system.
  - Many **allocation-tracking** methods are possible. For example, we could use a **free list** that points to the first free block, which then points to the next free block, and so forth.

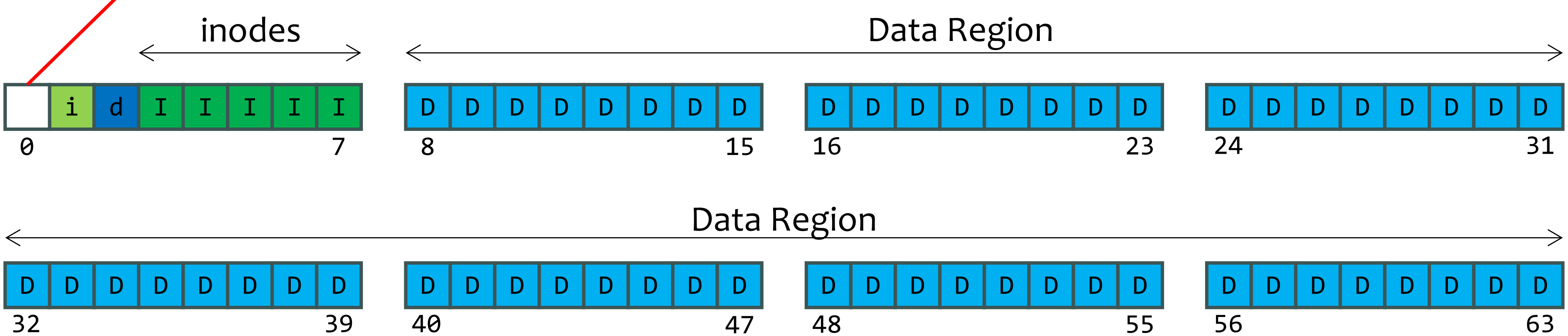


VSFS (Very Simple File System)

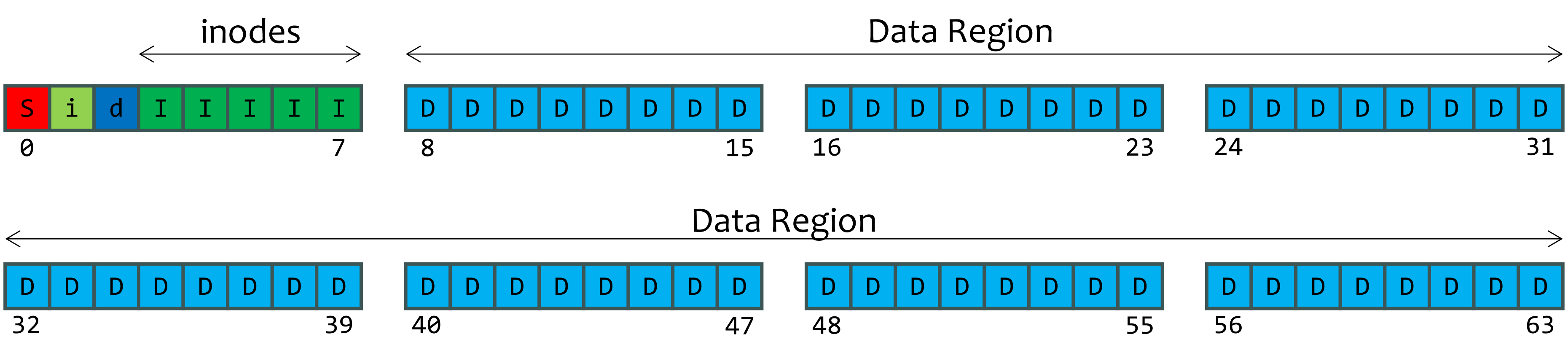
- In **VSFS**, we simply choose a simple and popular structure known as a **bitmap**.
  - One for the **data region** (the data bitmap, or d-bmap)
  - One for the **inode table** (the inode bitmap, or i-bmap)
- A **bitmap** is a simple structure: each **bit** is used to indicate whether the corresponding object/block is free (`0`) or in-use (`1`).
  - Notice that it is a bit of overkill to use an entire **4KB** block for these bitmaps: such a bitmap can track whether 32768 objects are allocated, and yet we only have 80 inodes and 56 data blocks. However, we just use an entire **4KB** block for each of these bitmaps for **simplicity**.



- **VSFS (Very Simple File System)**
  - Notice that there is still **one block** left unallocated.

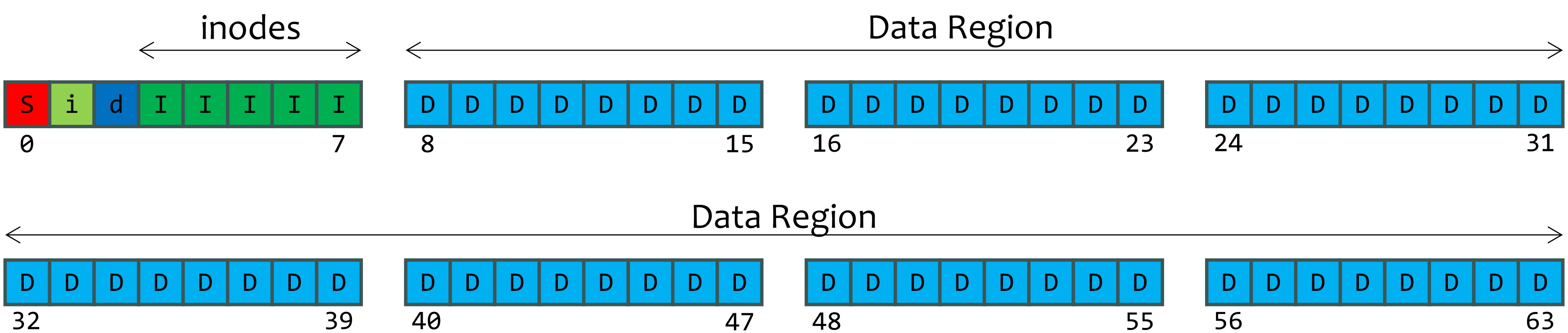


- **VSFS (Very Simple File System)**
  - We reserve this for the **superblock**, denoted by **S**.
  - The **superblock** contains information about this particular file system (**VSFS**), including, for example:
    - how many **inodes** and **data blocks** are in the file system?
      - 80 and 56, respectively in this instance
    - where the **inode table** begins?
      - block 3 in this instance
    - where the **data blocks** begin?
    - a magic number to identify the file system type (**VSFS**).
    - ...and so on.

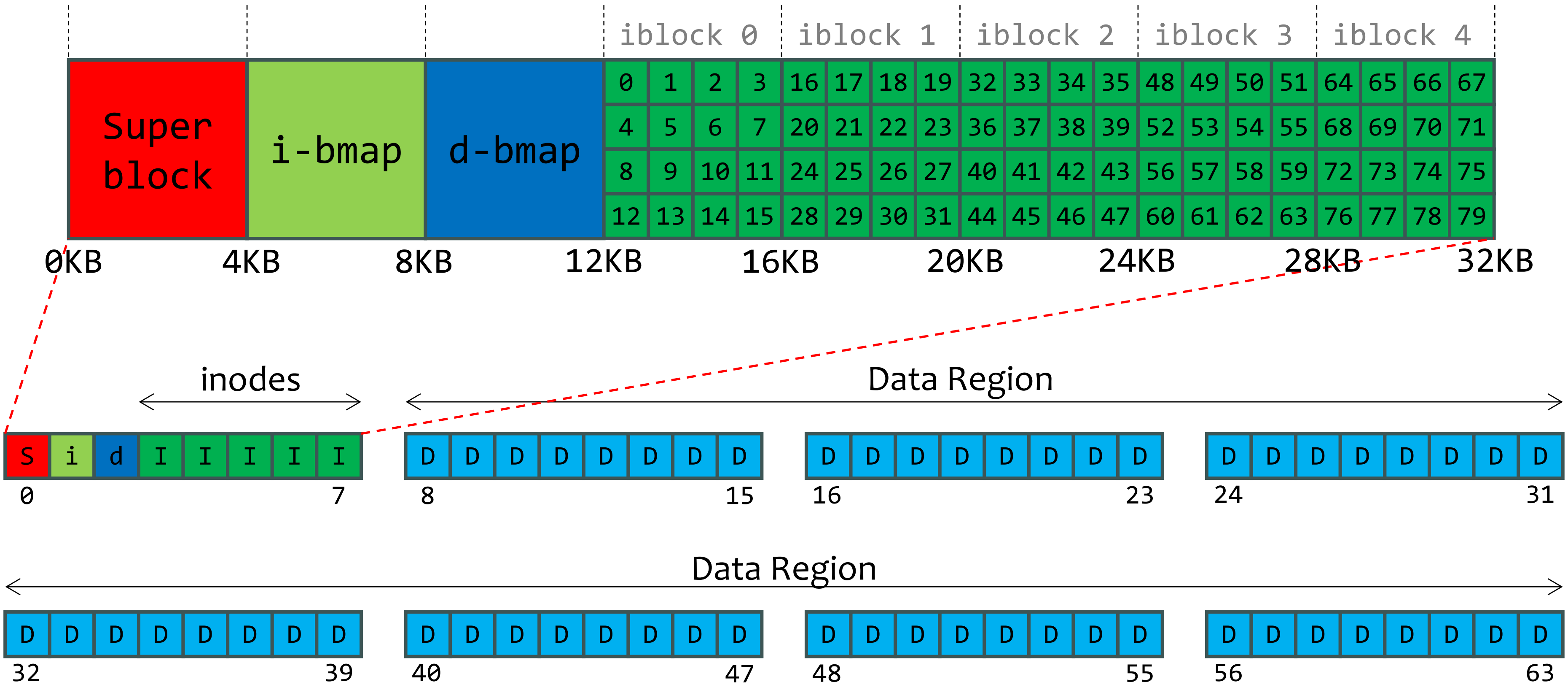


File System Implementation

- **VSFS (Very Simple File System)**
  - Thus, when **mounting** a file system, the OS will read the **superblock** first, to initialize varioius parameters, and then attach the volume to the file-system tree.
  - When files within the volume are accessed, the system will thus know exactly where to look for the needed **on-disk** structures.

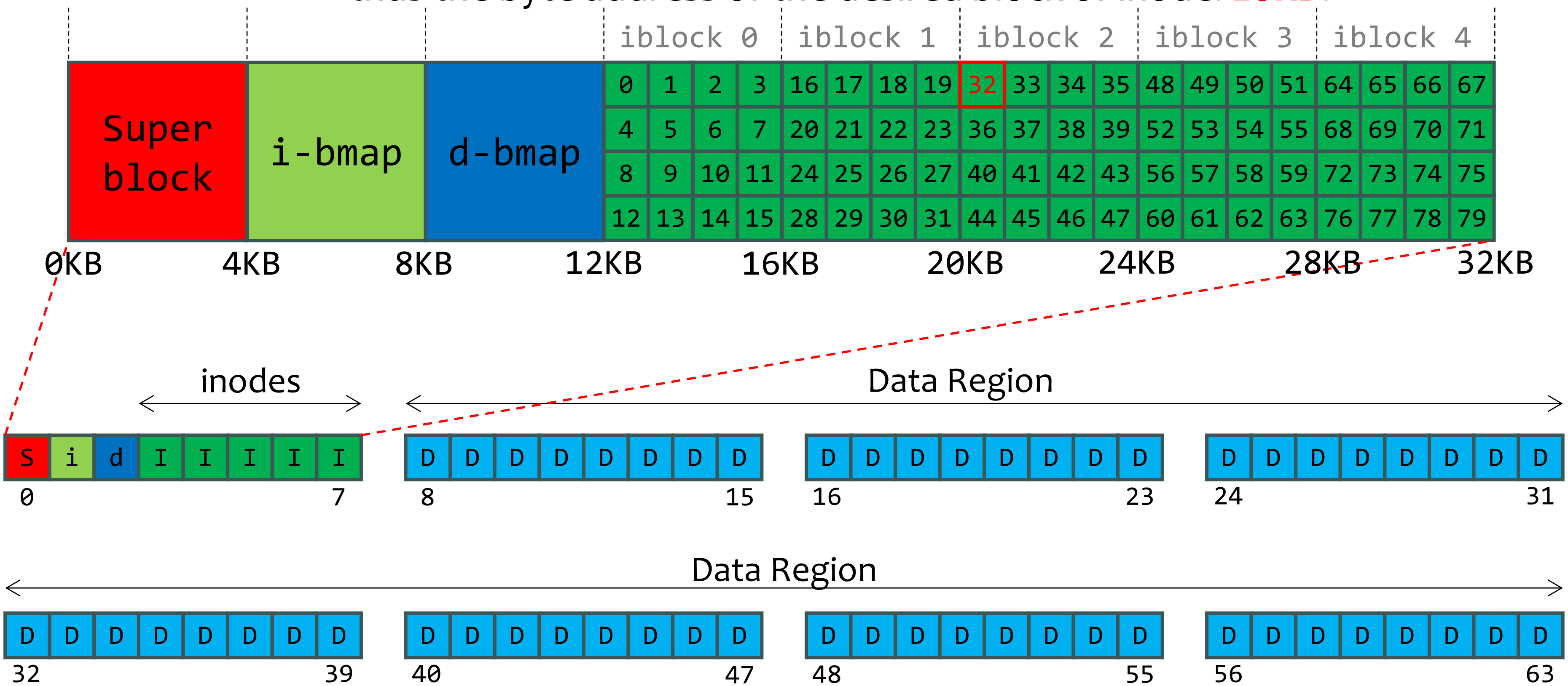


- **VSFS (Very Simple File System)**
  - The **inode table** (*a closer look*)



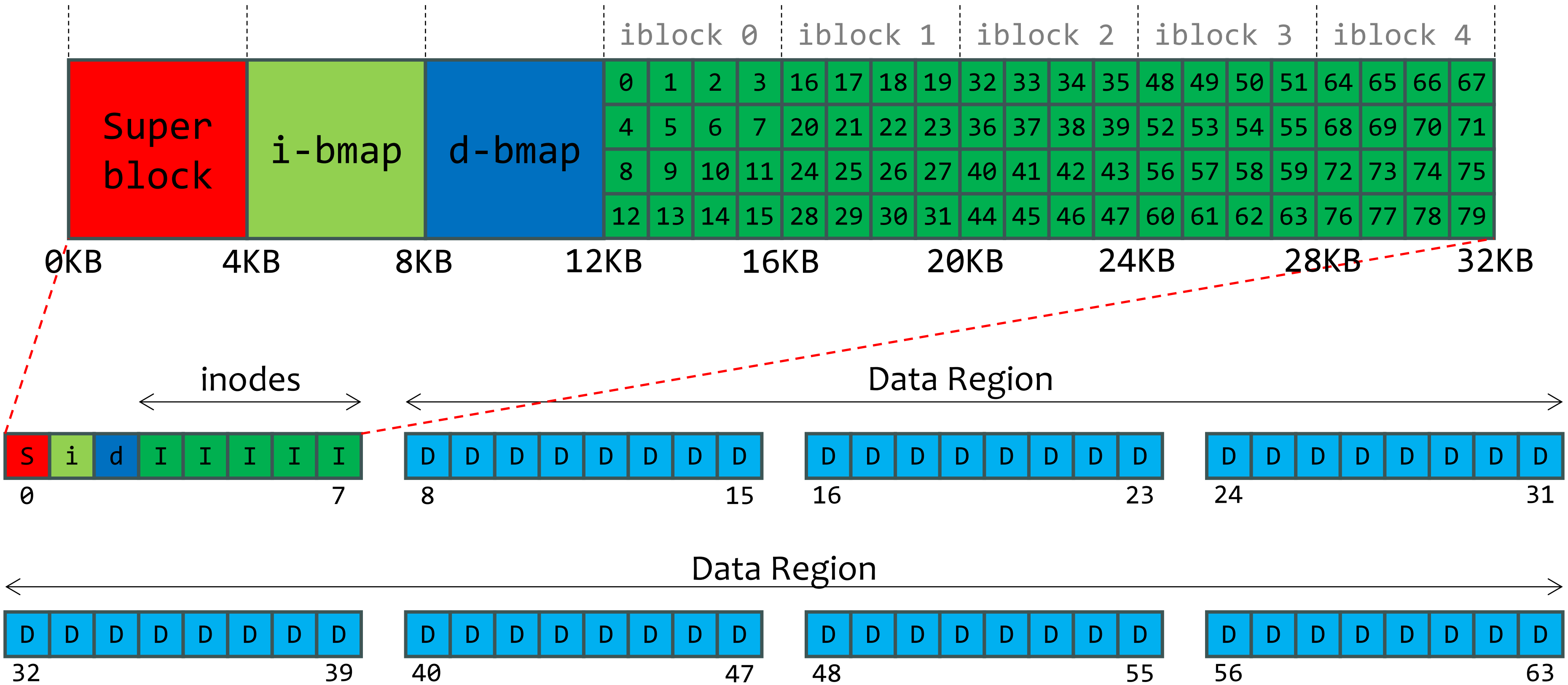
VSFS (Very Simple File System)

- The **inode table** (*a closer look*)
  - For example, to read inode number 32, the file system would first calculate the offset into the inode region ($32 \times sizeof(inode) = 8192 = 8KB$), plus the start address of the inode table on disk (12KB).
    - thus the byte address of the desired block of inode: **20KB**.



File System Implementation

- **VSFS (Very Simple File System)**
  - Generally, the sector address of the inode block can be calculated:
    - `blk = (inum * sizeof(inode_t)) / blockSize;`
    - `sector = ((blk*blockSize)+inodeStartAddr) / sectorSize;`

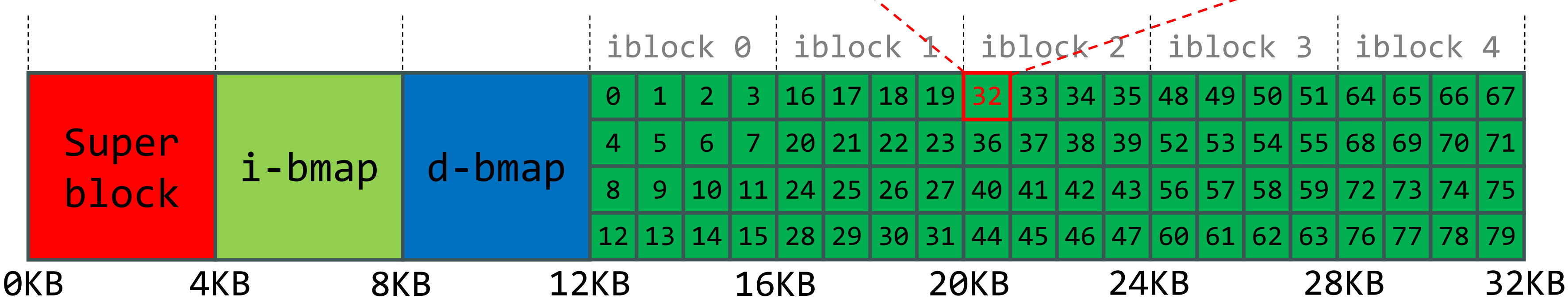


File System Implementation

- **VSFS (Very Simple File System)**
  - Inside each **inode** is virtually all the info about a file:
    - called ***metadata***.

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Simplified ext2 inode

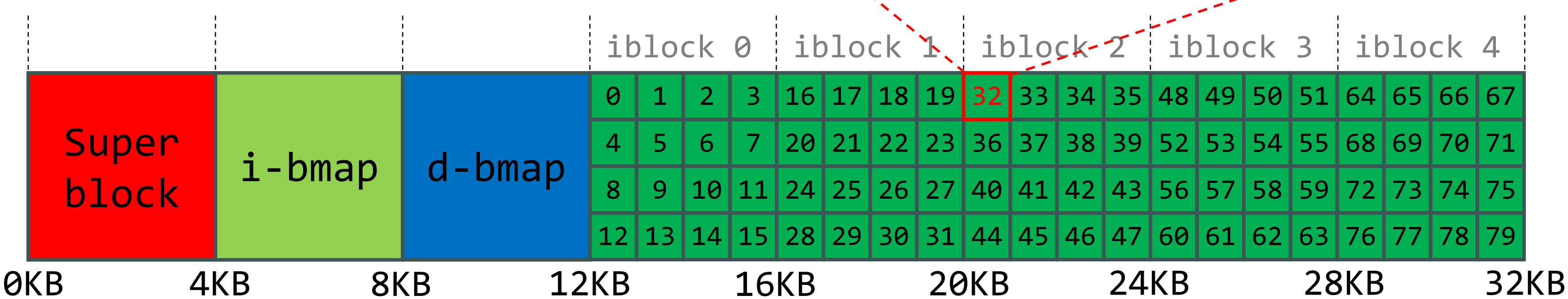


VSFS (Very Simple File System)

- Inside each **inode** is virtually all the info about a file:
  - called ***metadata***.
- One of the most important decisions in the design of **inode** is how it refers to where **data blocks** are.

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Simplified ext2 inode

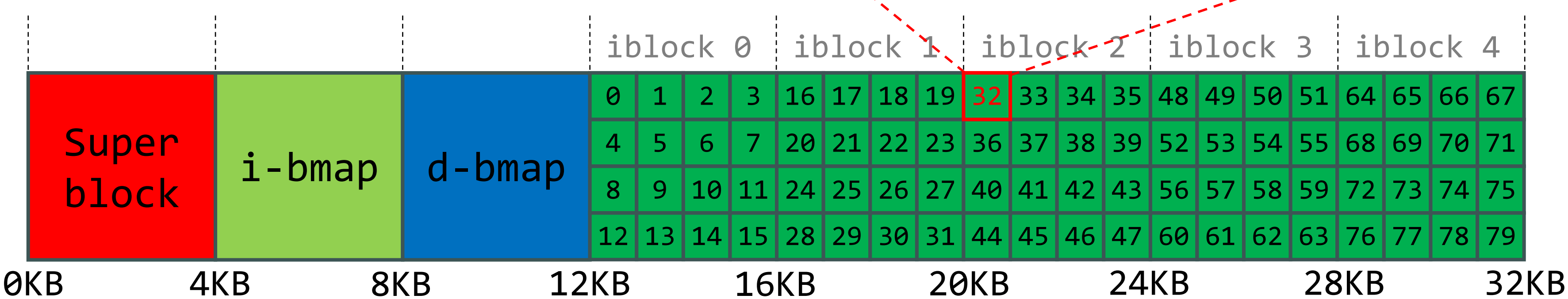


VSFS (Very Simple File System)

- Inside each **inode** is virtually all the info about a file:
  - called ***metadata***.
- One of the most important decisions in the design of **inode** is how it refers to where **data blocks** are.

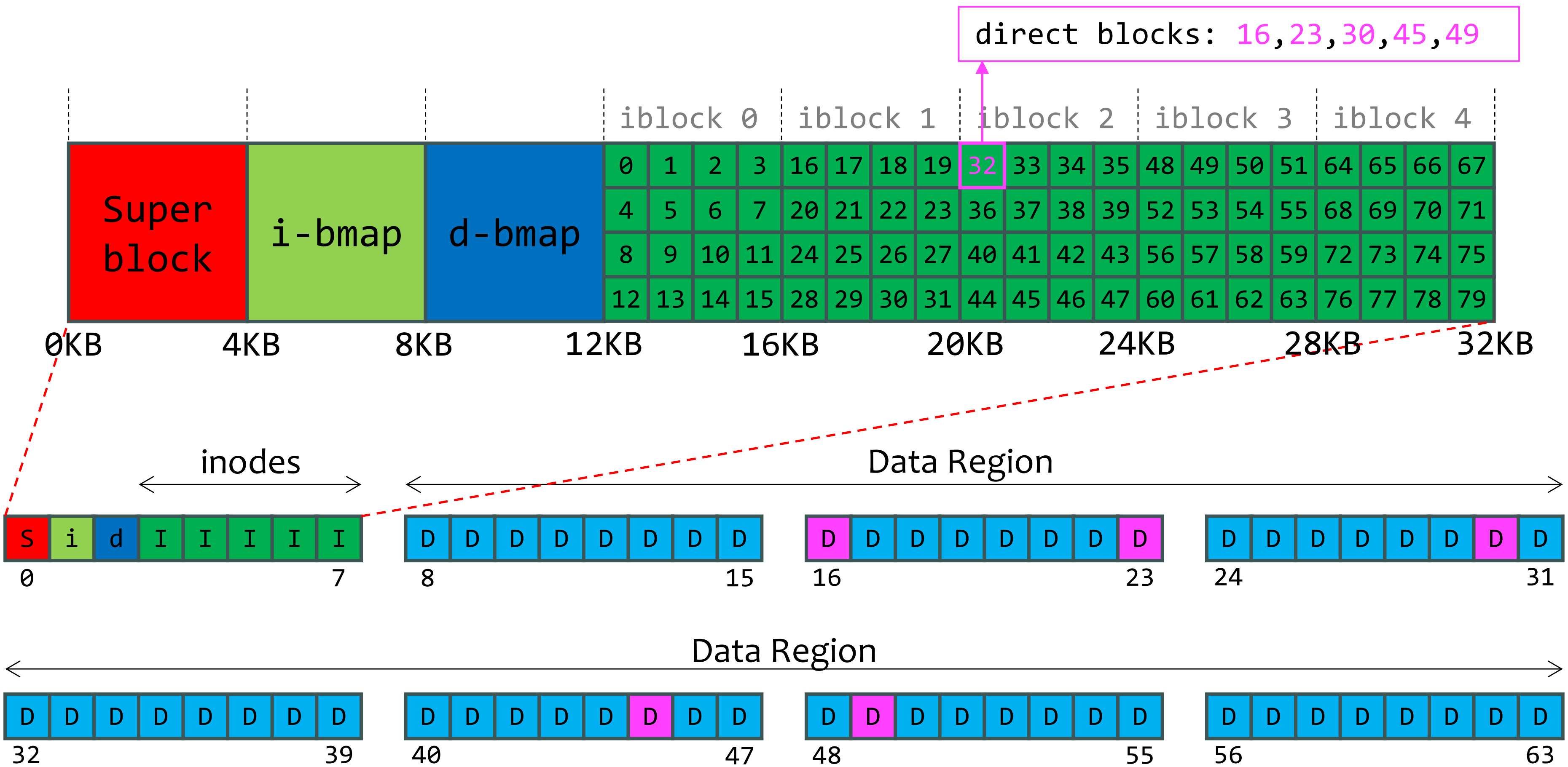
Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Simplified ext2 inode

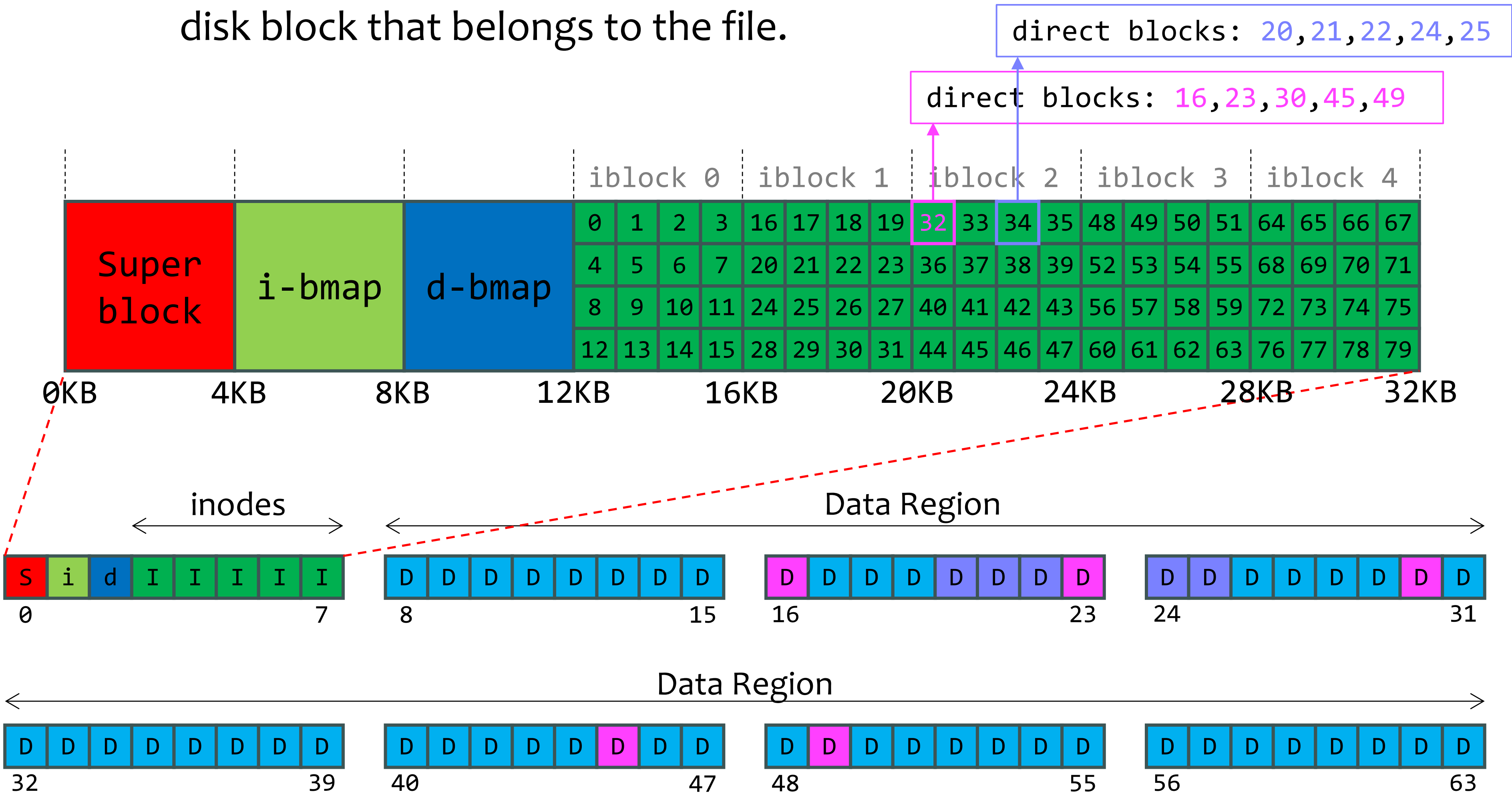


VSFS (Very Simple File System)

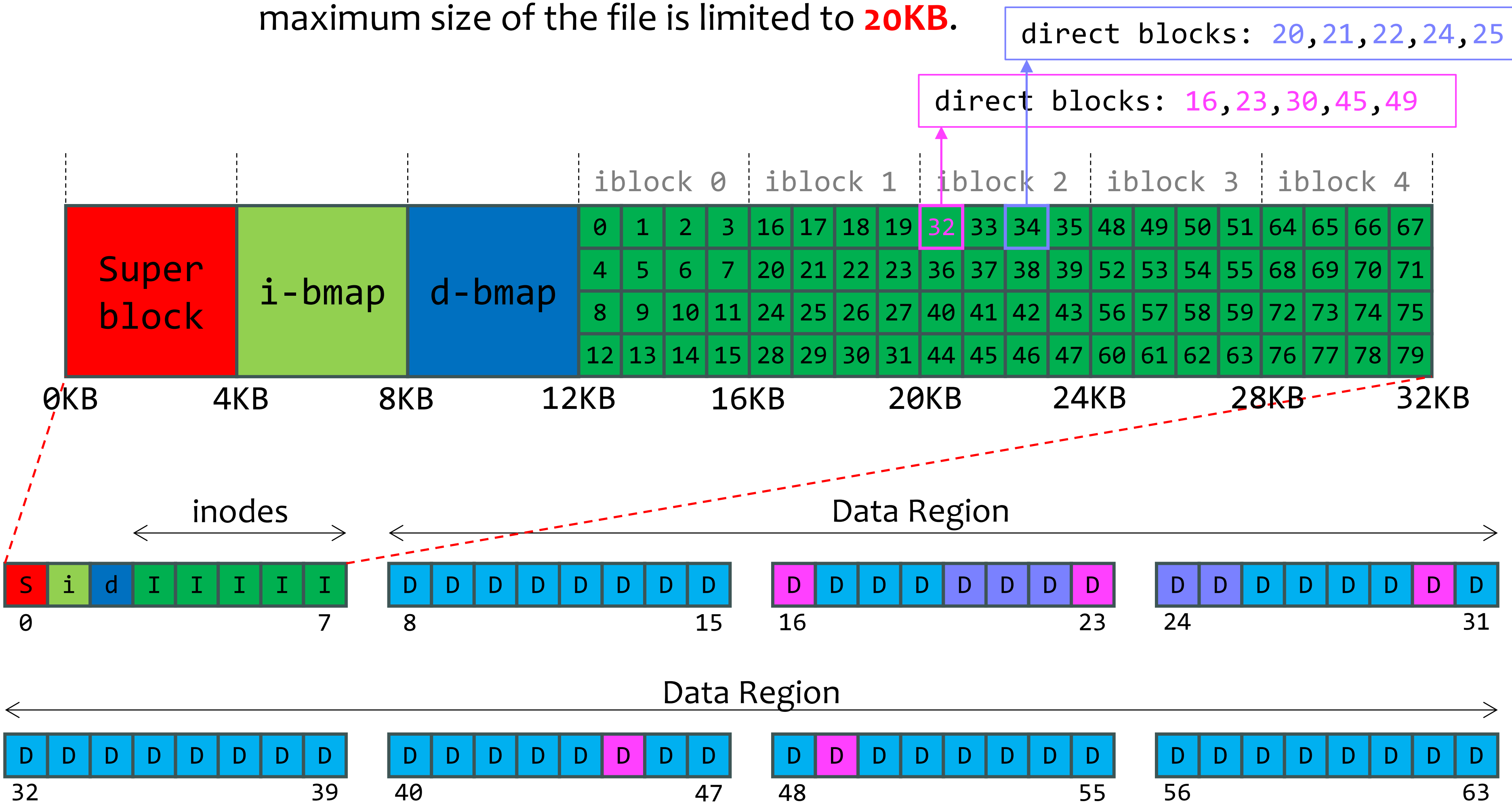
- **Direct Pointers**: One simple approach is to have one or more **direct pointers** (disk address) inside the inode; each pointer refers to one disk block that belongs to the file.



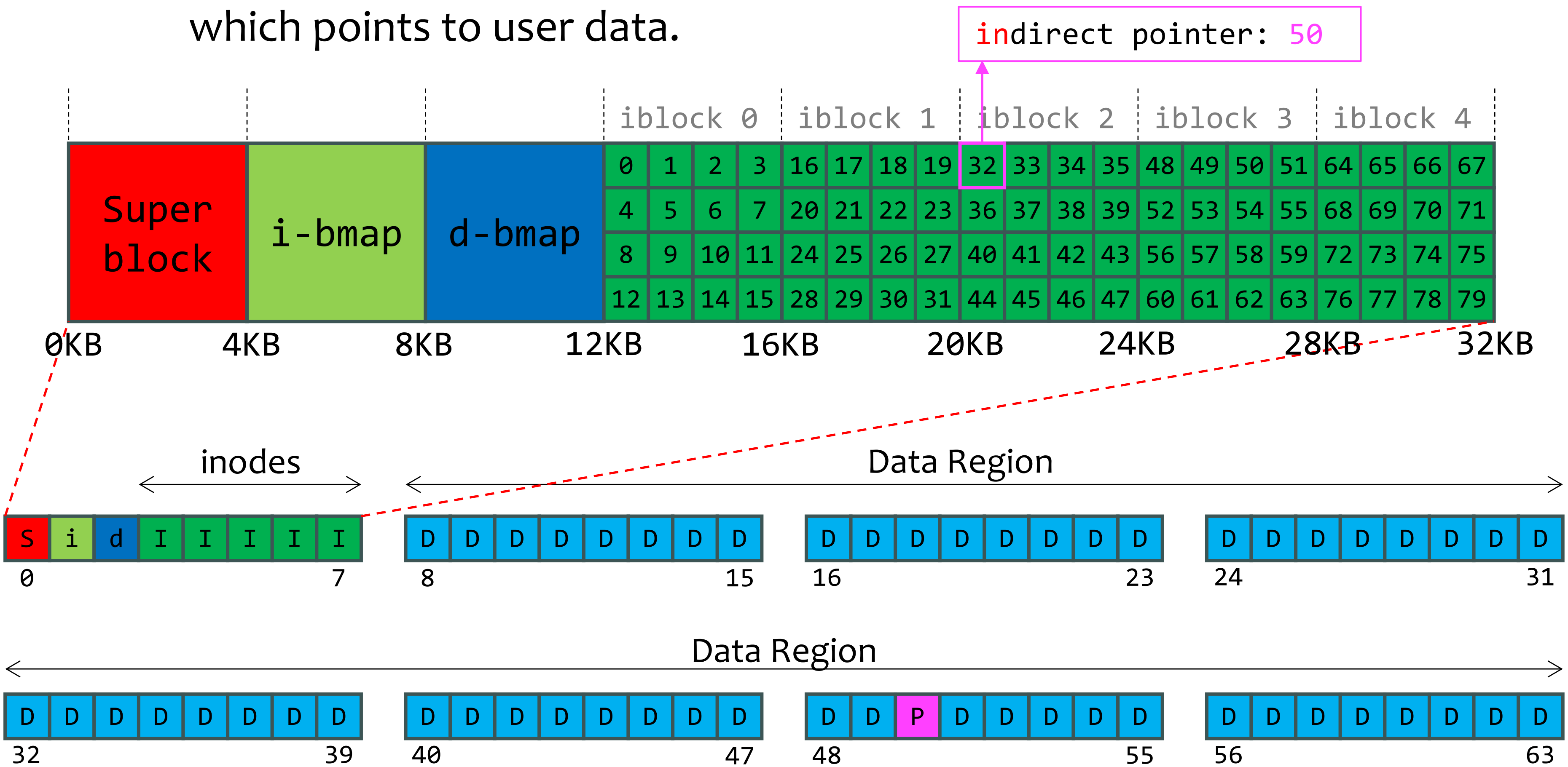
- **VSFS (Very Simple File System)**
  - **Direct Pointers**: One simple approach is to have one or more **direct pointers** (disk address) inside the inode; each pointer refers to one disk block that belongs to the file.



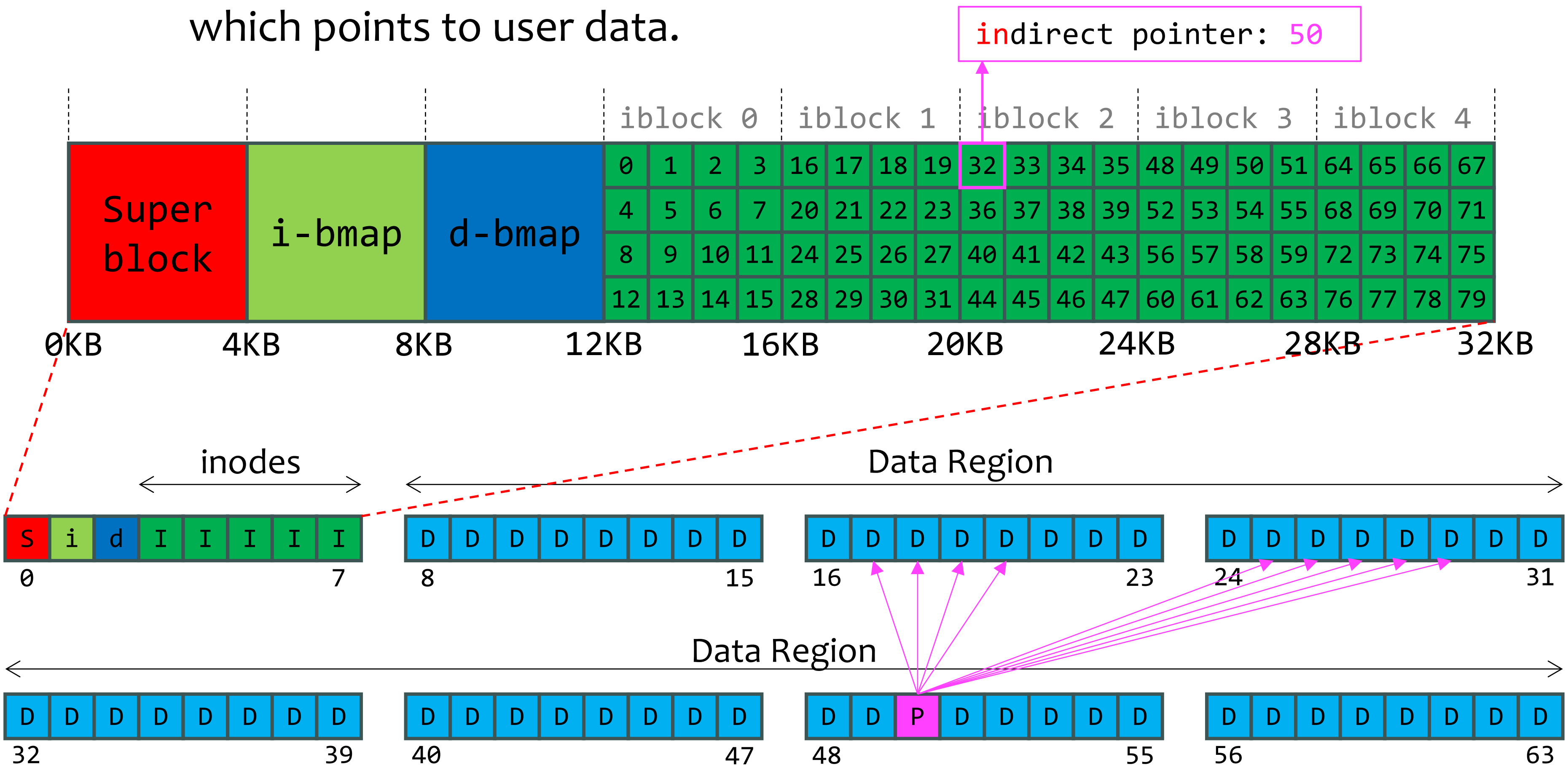
- **VSFS (Very Simple File System)**
  - **Disadvantage of Direct Pointers**: limited file size.
    - For example, if the number of **direct blocks** in an inode is **5**, then the maximum size of the file is limited to **20KB**.



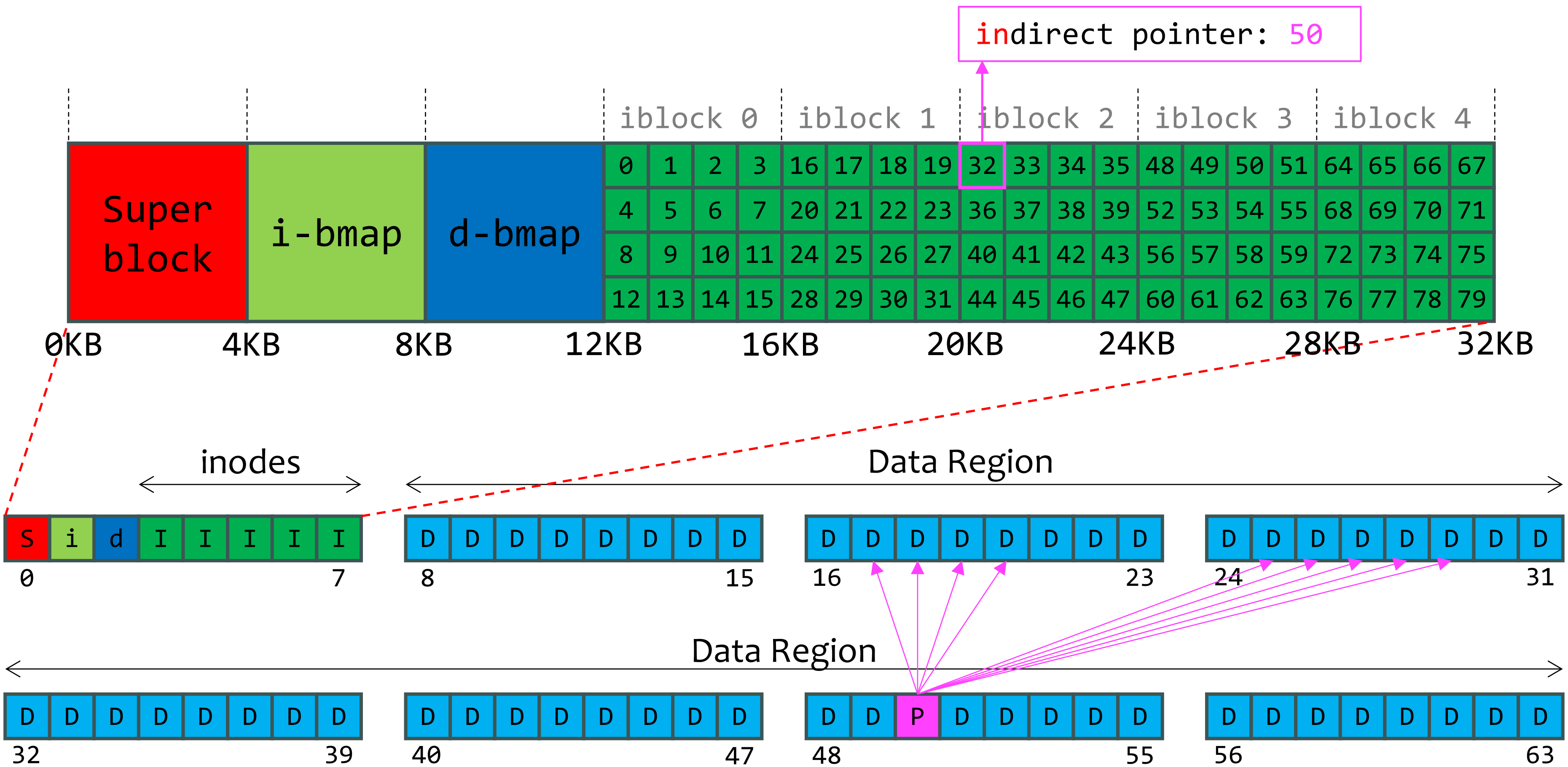
- **VSFS (Very Simple File System)**
  - To support bigger file size, one common approach is to support **Indirect Pointers**: instead of pointing to a block that contains user data, it points to a block that contains more (*direct*) pointers, each of which points to user data.



- **VSFS (Very Simple File System)**
  - To support bigger file size, one common approach is to support **Indirect Pointers**: instead of pointing to a block that contains user data, it points to a block that contains more (*direct*) pointers, each of which points to user data.

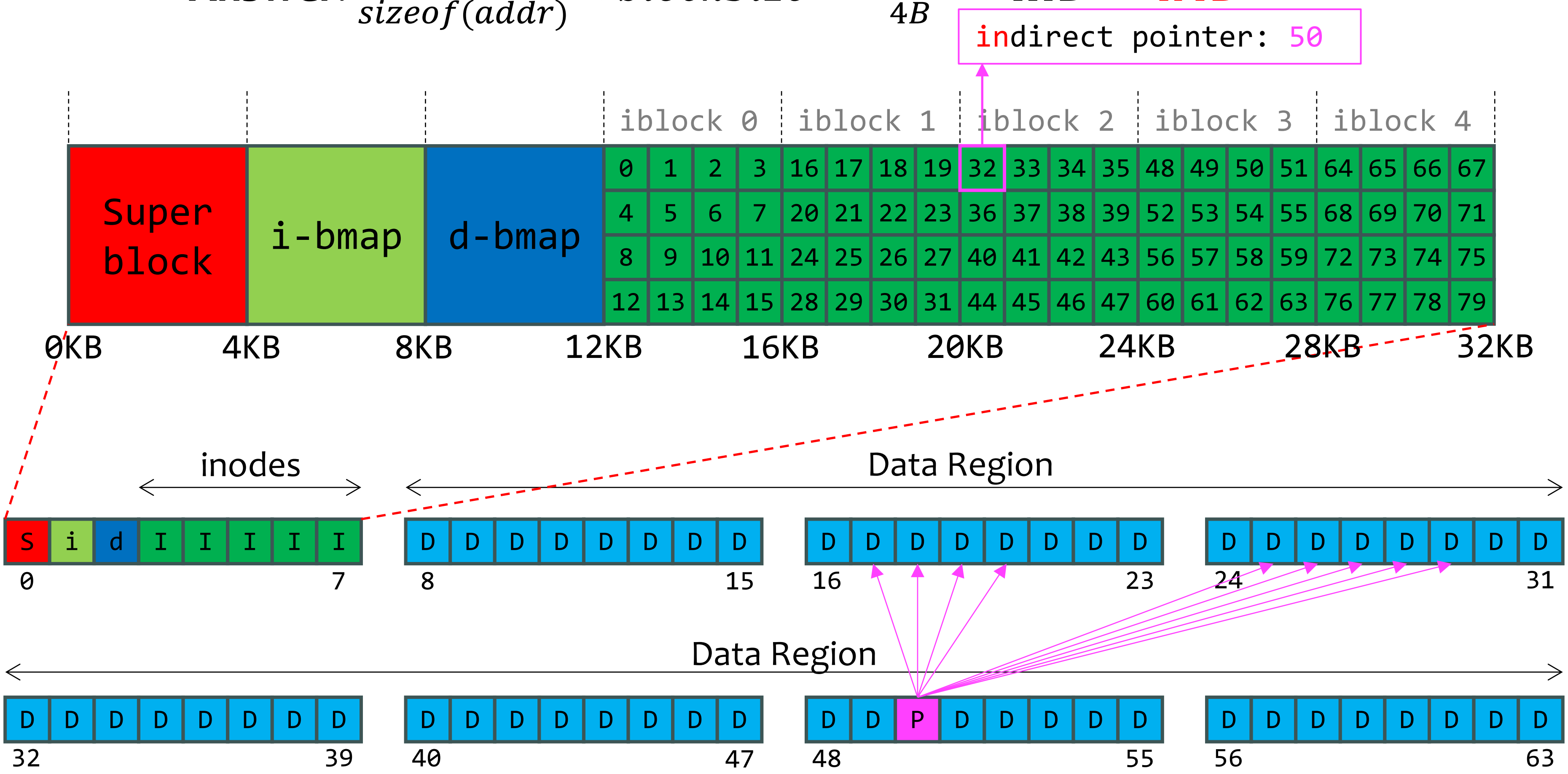


- **VSFS (Very Simple File System)**
  - **Question:** With a (one-level) **indirect pointer**, what's the maximum supported file size (assume a 4-byte disk address)?

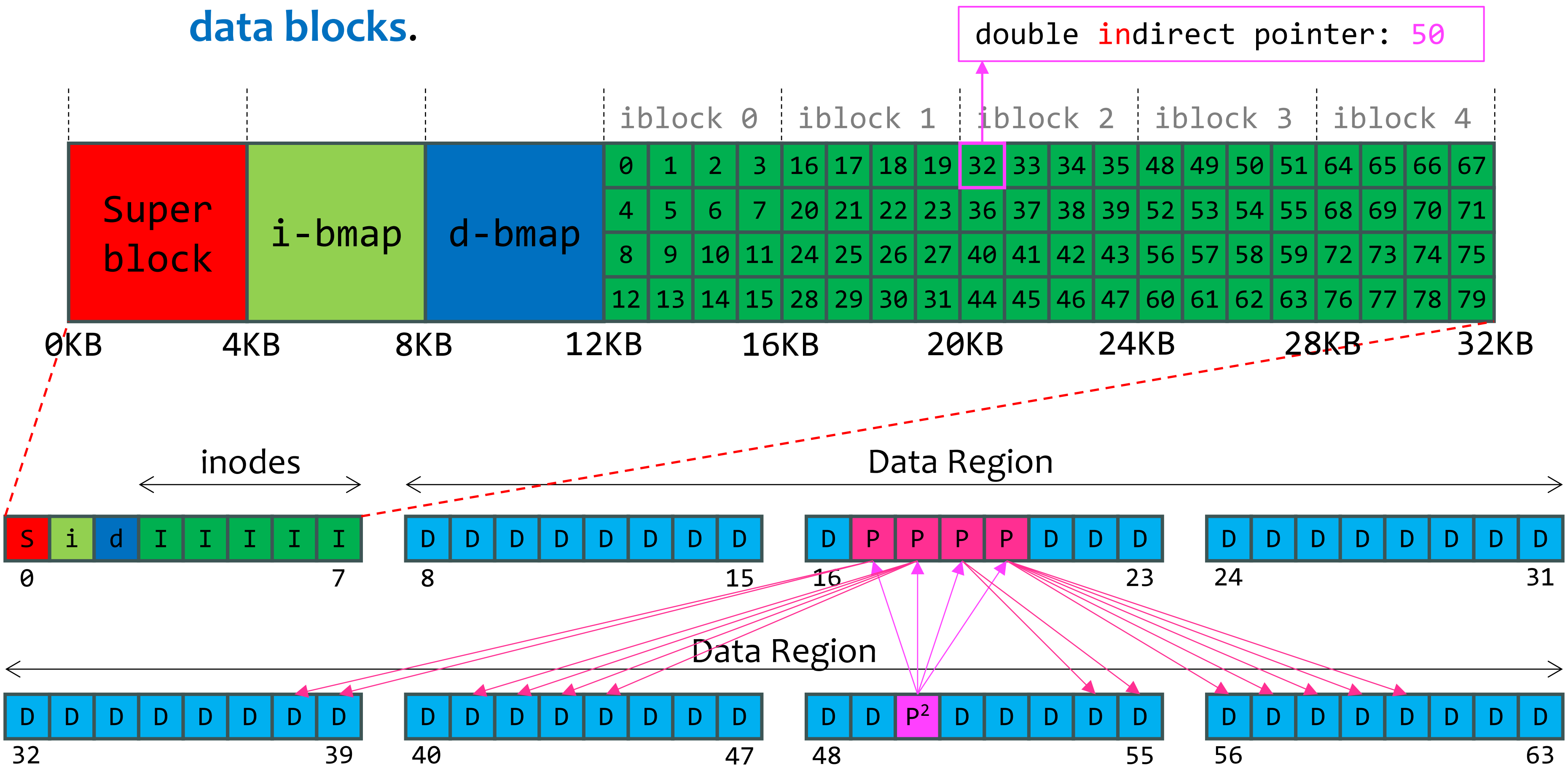


File System Implementation

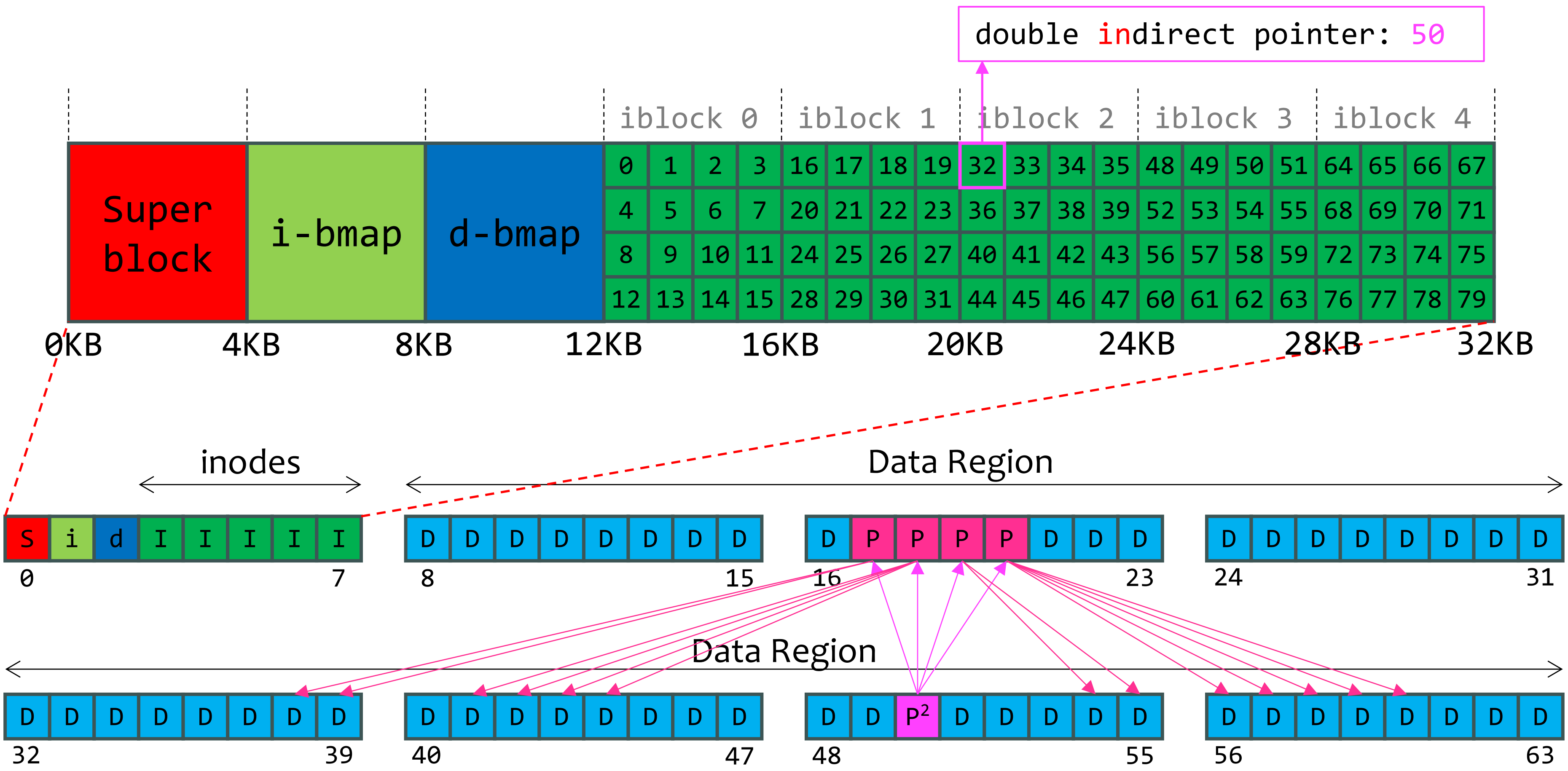
- **VSFS (Very Simple File System)**
  - **Question:** With a (one-level) **indirect pointer**, what's the maximum supported file size (assume a 4-byte disk address)?
  - **Answer:** $\frac{blockSize}{sizeof(addr)} \times blockSize = \frac{4KB}{4B} \times 4KB = 4MB$



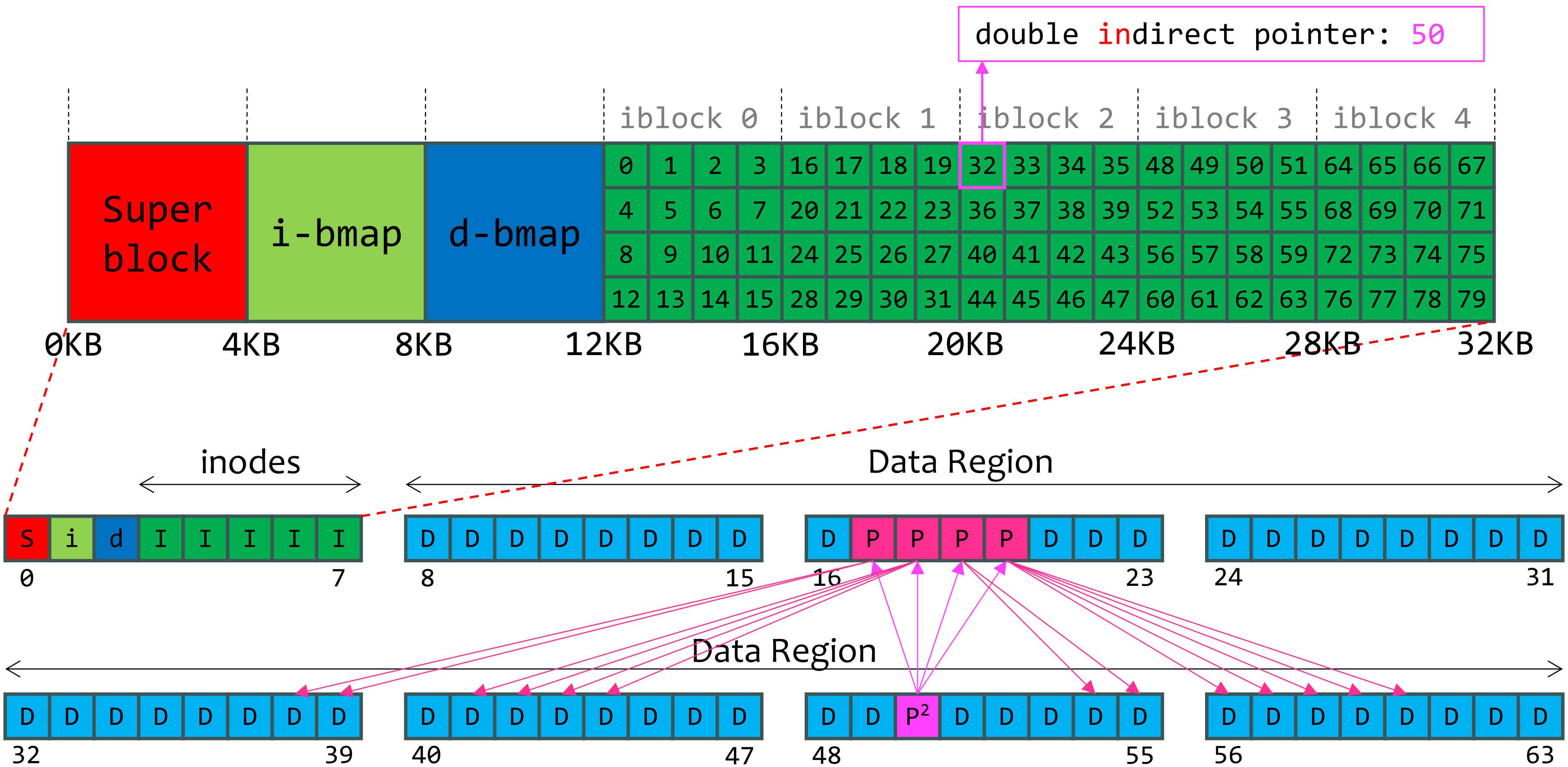
- **VSFS (Very Simple File System)**
  - **Multi-Level Index**: To support even bigger file size, we can use the **double indirect pointer**, which points to a block that contains pointers to indirect blocks, each of which contains pointers to actual **data blocks**.



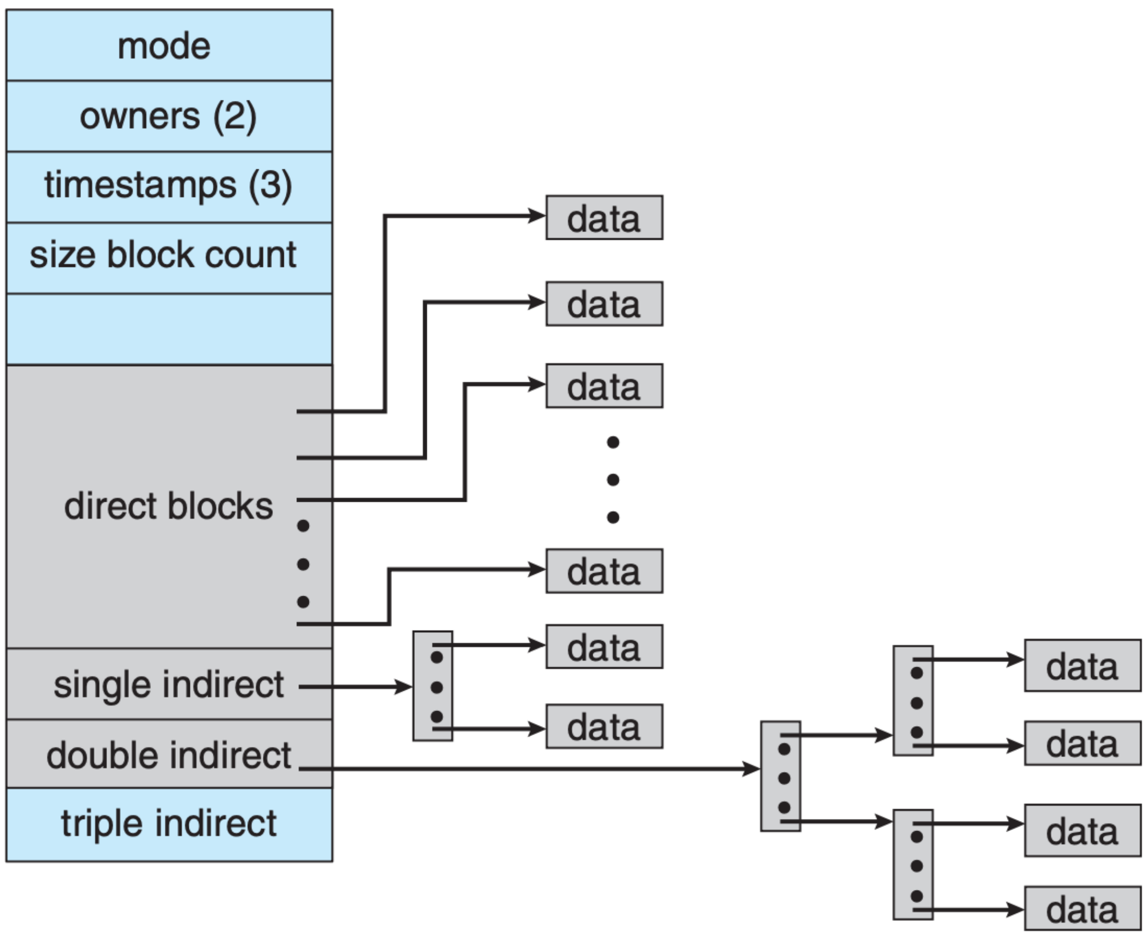
- **VSFS (Very Simple File System)**
  - **Question:** With a (two-level) **indirect pointer**, what's the maximum supported file size (assume a 4-byte disk address)?



- **VSFS (Very Simple File System)**
  - **Question:** With a (two-level) **indirect pointer**, what's the maximum supported file size (assume a 4-byte disk address)?
  - **Answer:** $1024^2 \times 4KB = 4GB$



- **VSFS (Very Simple File System)**
  - In practice, many file systems adopt a combination of direct pointers, one-level indirect pointers, two-level indirect pointers, or even more.



Characteristics of Files

A Five-Year Study of File-System Metadata

Authors:

Nitin Agrawal, *University of Wisconsin, Madison;* William J. Bolosky, John R. Douceur, and Jacob R. Lorch, *Microsoft Research*

Abstract:

For five years, we collected annual snapshots of filesystem metadata from over 60,000 Windows PC file systems in a large corporation. In this paper, we use these snapshots to study temporal changes in file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity and consumption, and degree of file modification. We present a generative model that explains the namespace structure and the distribution of directory sizes. We find significant temporal trends relating to the popularity of certain file types, the origin of file content, the way the namespace is used, and the degree of variation among file systems, as well as more pedestrian changes in sizes and capacities. We give examples of consequent lessons for designers of file systems and related software.

Published in FAST 2007

Link: https://dl.acm.org/doi/10.1145/1288783.1288788

Characteristics of Files

Observations:

- **#1:** Most files are small (< **48KB**)
  - the **12 direct pointers** in traditional UNIX inodes are sufficient to refer to most (*small*) files.

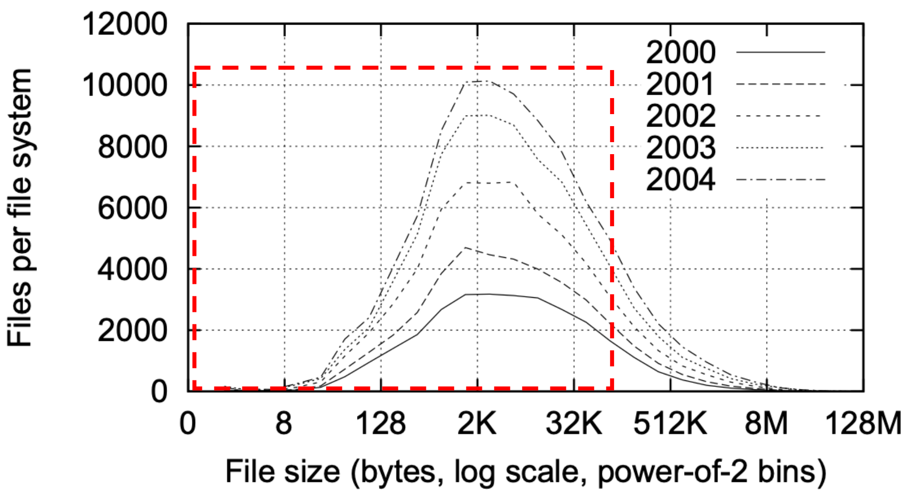


Figure 2: Histograms of files by size

Characteristics of Files

Observations:

- **#1**: Most files are small (< **48KB**)
  - the **12 direct pointers** in traditional UNIX inodes are sufficient to refer to most (*small*) files.
- **#2**: Most bytes are in medium-large files.
  - use single indirect, double indirect for medium-large files (<**4GB**);
  - triple indirect pointers only necessaray for very larger files.

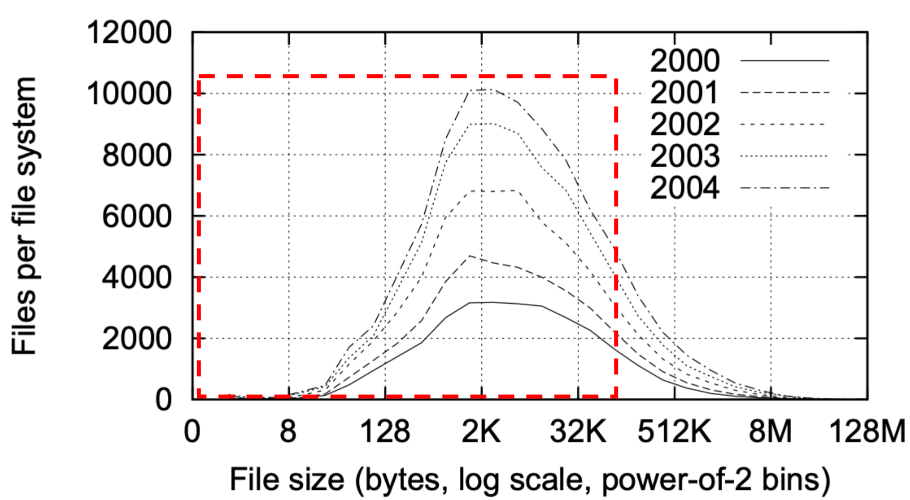


Figure 2: Histograms of files by size

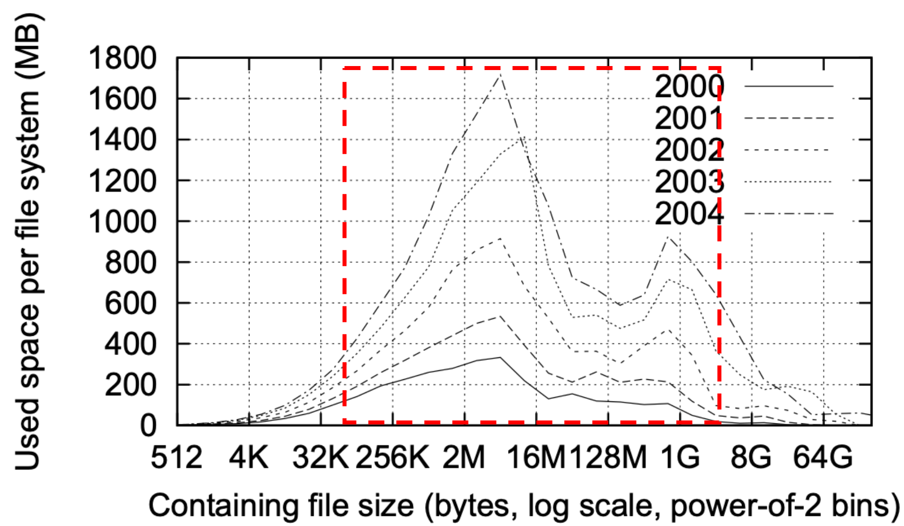
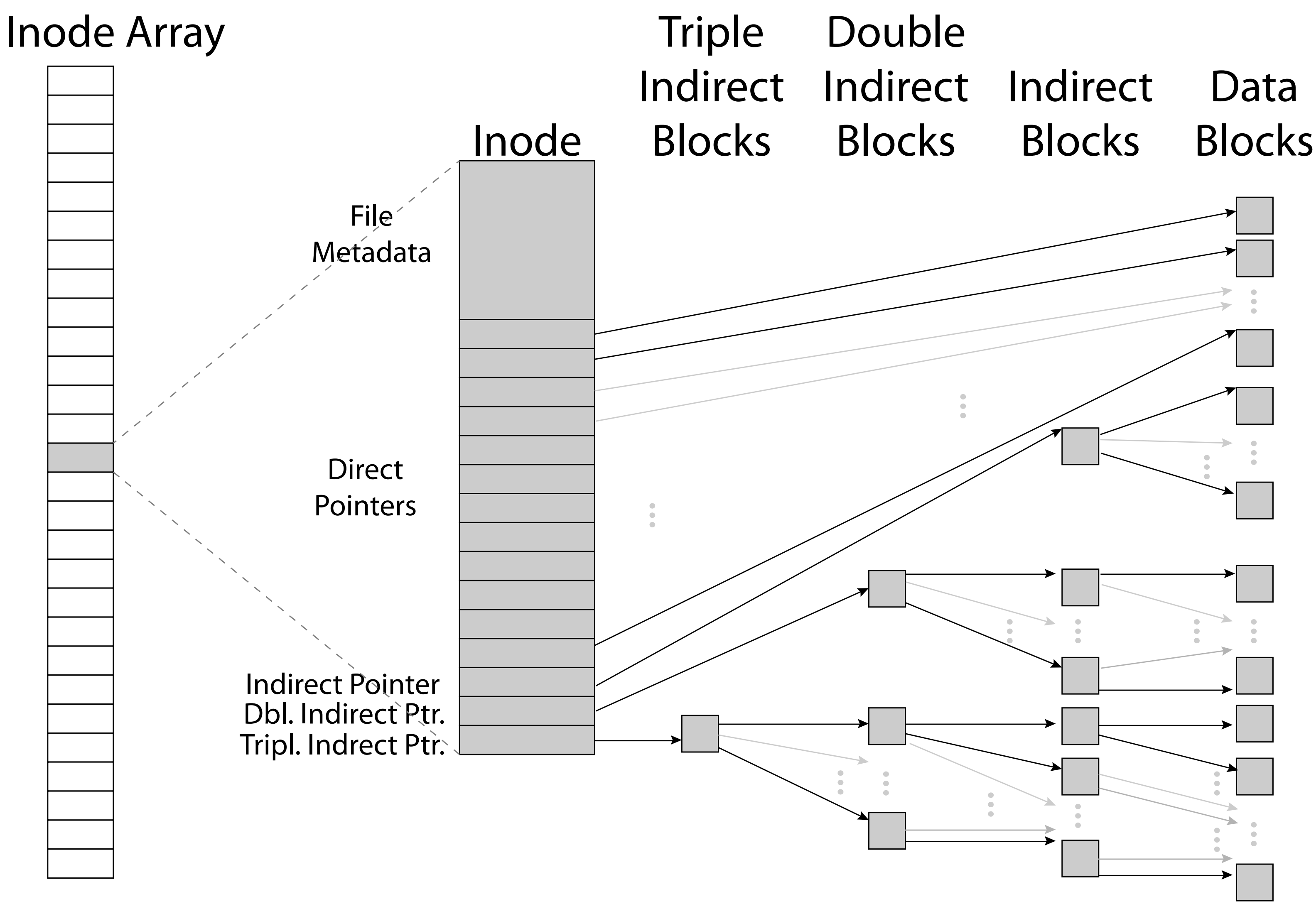
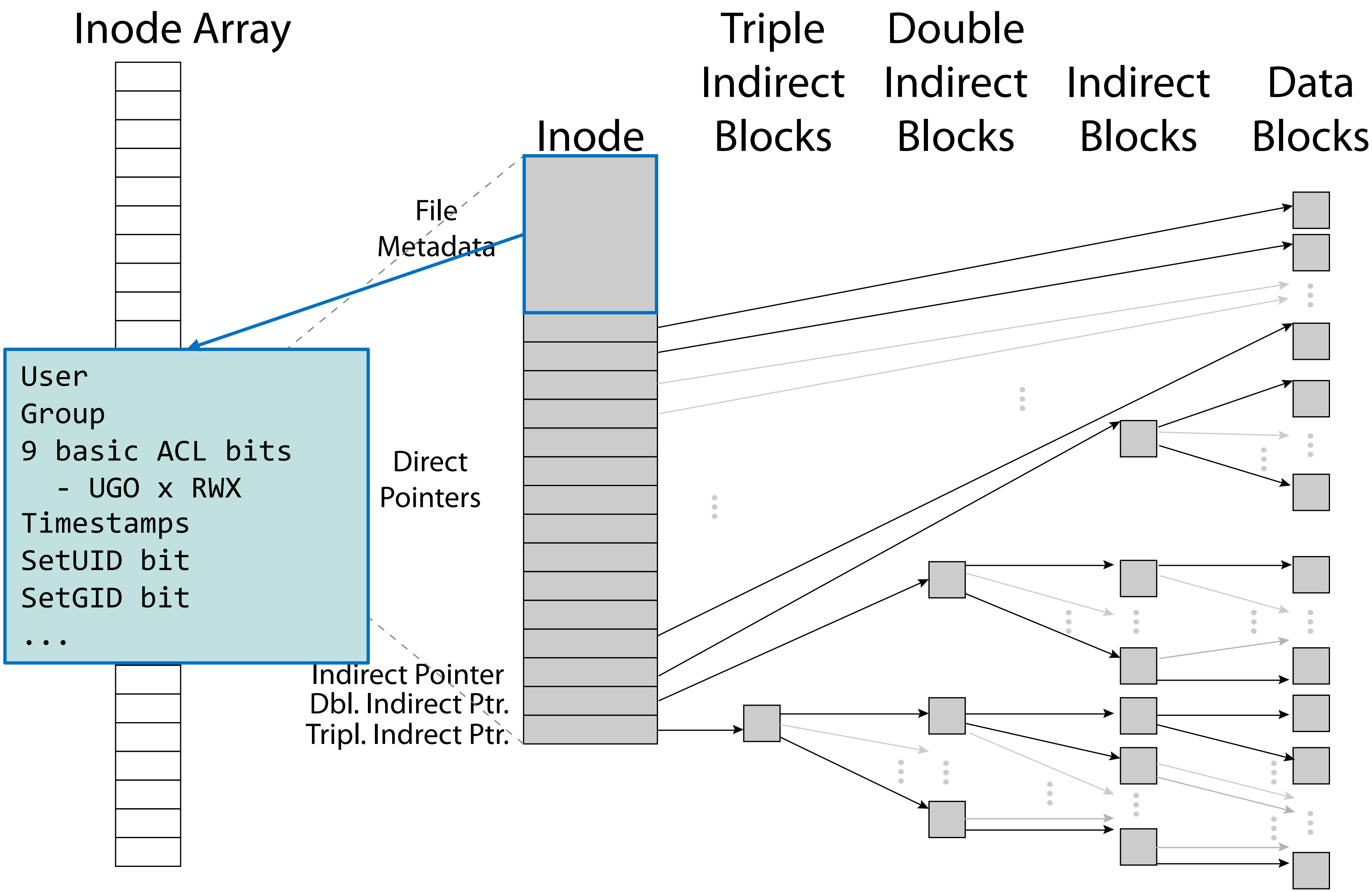


Figure 4: Histograms of bytes by containing file size

- **UNIX inode Structure (15 Pointers to blocks)**



UNIX inode Structure



UNIX inode Structure

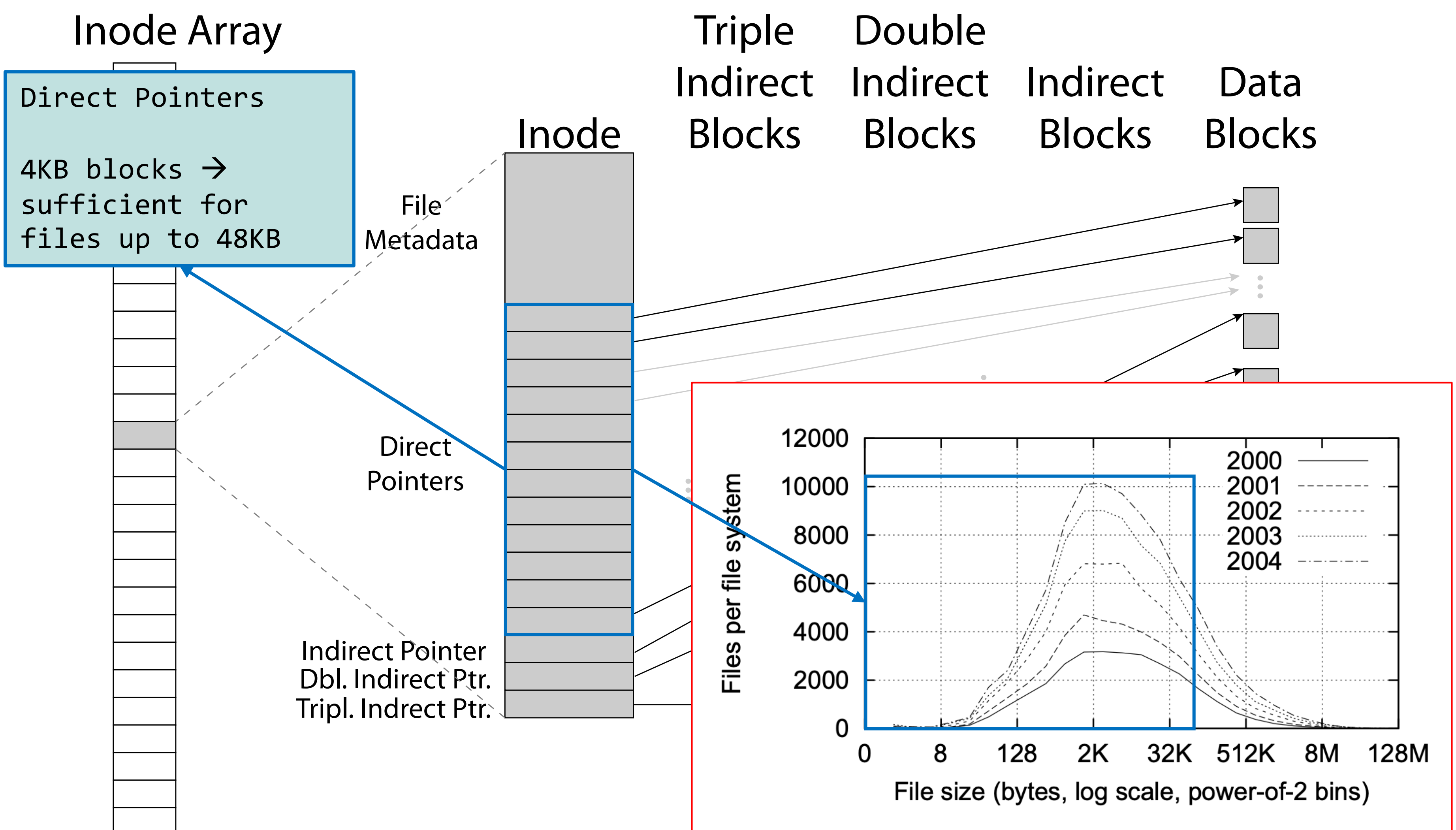
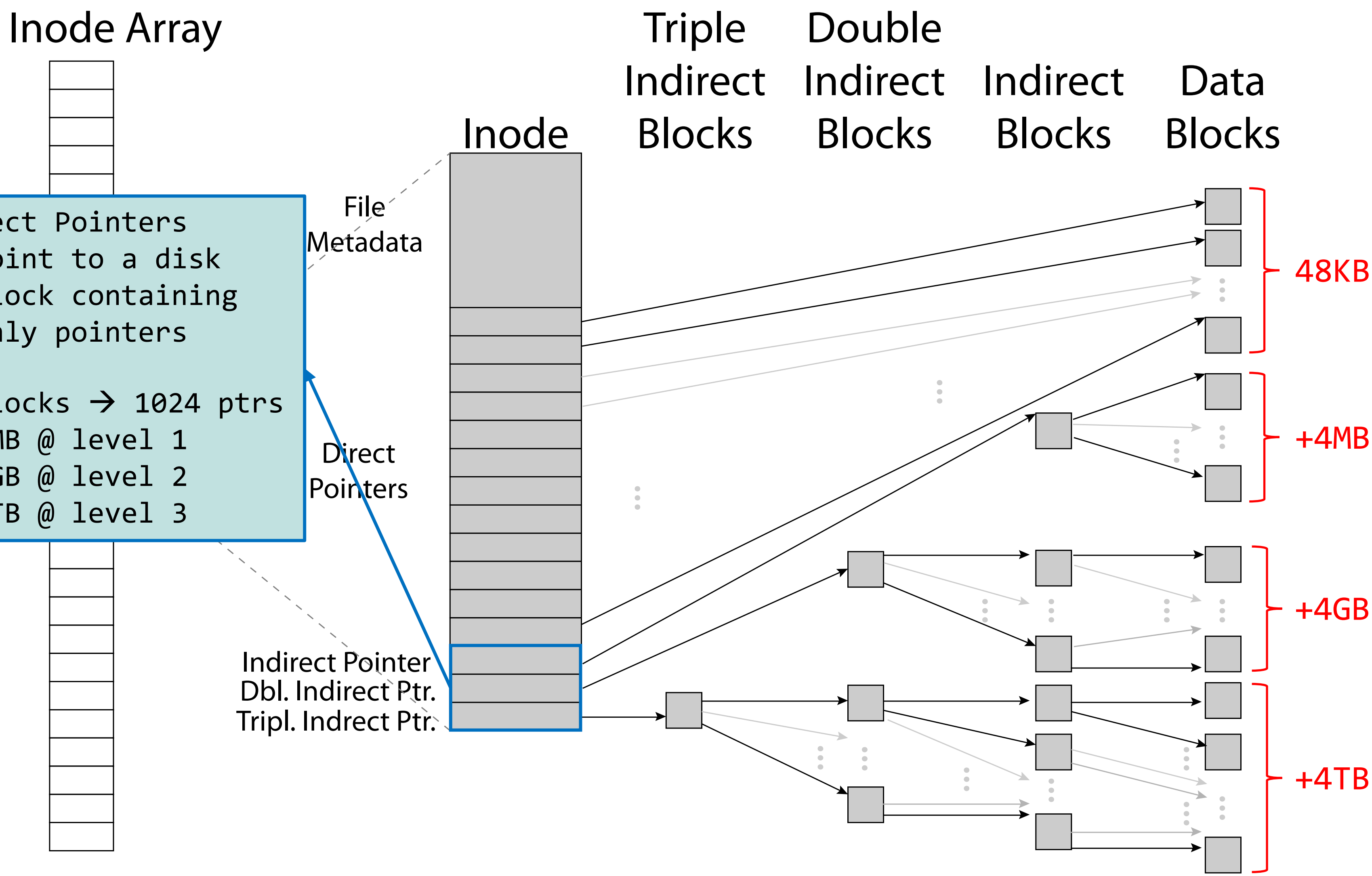


Figure 2: Histograms of files by size

UNIX inode Structure

```
Indirect Pointers
  - point to a disk
    block containing
    only pointers

4KB blocks → 1024 ptrs
  - 4MB @ level 1
  - 4GB @ level 2
  - 4TB @ level 3
```



Thank you!