# CSE 330:
# Operating Systems

**Adil Ahmad**

**Lecture #19:** File system internals

# Project #5 introduction (next week)

# Expose the block abstraction to user-space

- **Block abstraction:** layer that translates different (strorage) device operations into a unified “block-by-block” access
  - The Linux kernel has an awesome implementation of block layer operations that is typically used by the file system

- **Your task:** write a kernel module that allows a program to access a (virtual) USB device using the block layer

- *Note:* There will be **less hand-holding** in this project’s document! Please start early and find things on your own.

# A (preliminary) set of APIs you must support

All APIs will be supported as “ioctl” calls to your kernel module

- WRITE → sequentially writes the provided data to the USB device from the “current” offset
  - Initially current offset is “0”
- READ → sequentially reads the USB device from the “current” offset and writes the data to a user space buffer
- READOFFSET/WRITEOFFSET → Instead of read/write to the “current” offset, you will read/write from a custom offset
- Perhaps 1-2 more (*check the final document!*)

# **Linux block abstraction:** *opening a block device*

- `lsblk` → provides you a list of block devices registered
  - The (virtual) USB will already be registered and given a name (e.g., `/dev/sdb`)

- `blkdev_get_by_path("device-name", permissions, …)`
  - Within the kernel module, you can use this API to open a device with certain permissions

- You must open the correct device within your kernel module to begin any operations

# **Linux block abstraction:** *communicating with a block device*

- `bio` → a structure used for communication
  - Simply called the “block IO” structure

- `bio_alloc(device-name, buffers, operation, flags)`
  - Allocates a new bio structure for the ”device” you opened
  - Within each bio structure, it creates “X” buffers (max is 256)
  - Operation (e.g., `REQ_OP_WRITE` → write)

- You must allocate a BIO correctly and initialize it based on which kind of operation (e.g., read/write) is required

# **Linux block abstraction:** *communicating with a block device*

- `bio→bi_iter.bi_sector` → which ”block” to write
  - This is also the “offset” within the disk

- `bio_add_page(bio, paddr-page, nbytes, pageoffset)`
  - **`paddr-page`**: physical address of page where data is to be read into or written from (depending on the operation)
  - **`nbytes`**: number of bytes (max = 512) to read/write
  - **`pageoffset`**: offset within the “page” to read/write from

- `submit_bio_wait(bio)`
  - Submits a BIO request and waits for it to be completed

# **Notes** on the Linux block abstraction

- `READ/WRITES` are performed at the granularity of 512 bytes from within the Linux block APIs

- You may be asked to perform operations larger than 512 bytes, and it is your job to make sure that you break them down correctly
  - 4096 → 8 operations of 512 bytes each
  - *Hint: you can use the "pageoffset"*

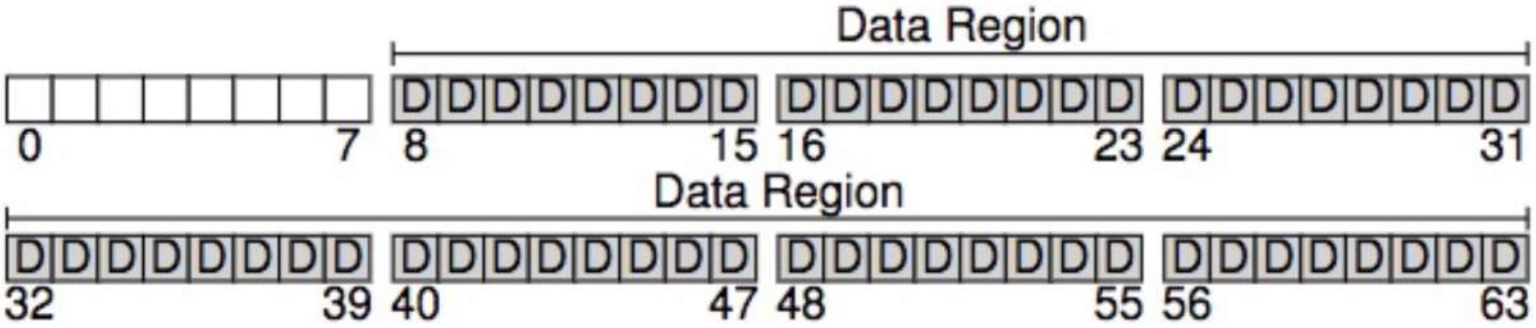
# Recap implementation of a filesystem

# What are the questions we need to answer to build an FS?

- Where to store the files in the disk?
- Where/how to store the information about existing files (e.g., inodes, path, blocks, etc.) in the disk?
- How to quickly find free locations in the disk for new files?

# Storing file **data blocks** within the disk

- Initially, the entire disk is empty
- We will save some space to store information
- Use the rest to store our “data blocks” (figure below)



# Now, we must answer:

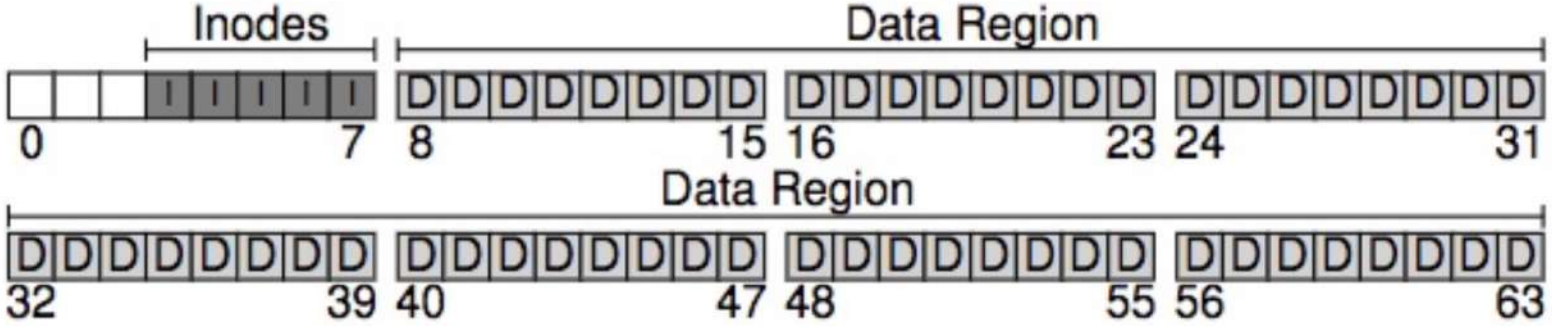
- Where to store the files in the disk?
- Where/how to store the information about existing files (e.g., inodes, path, blocks, etc.) in the disk?
- How to quickly find free locations in the disk for new files?

# An **inode** data structure

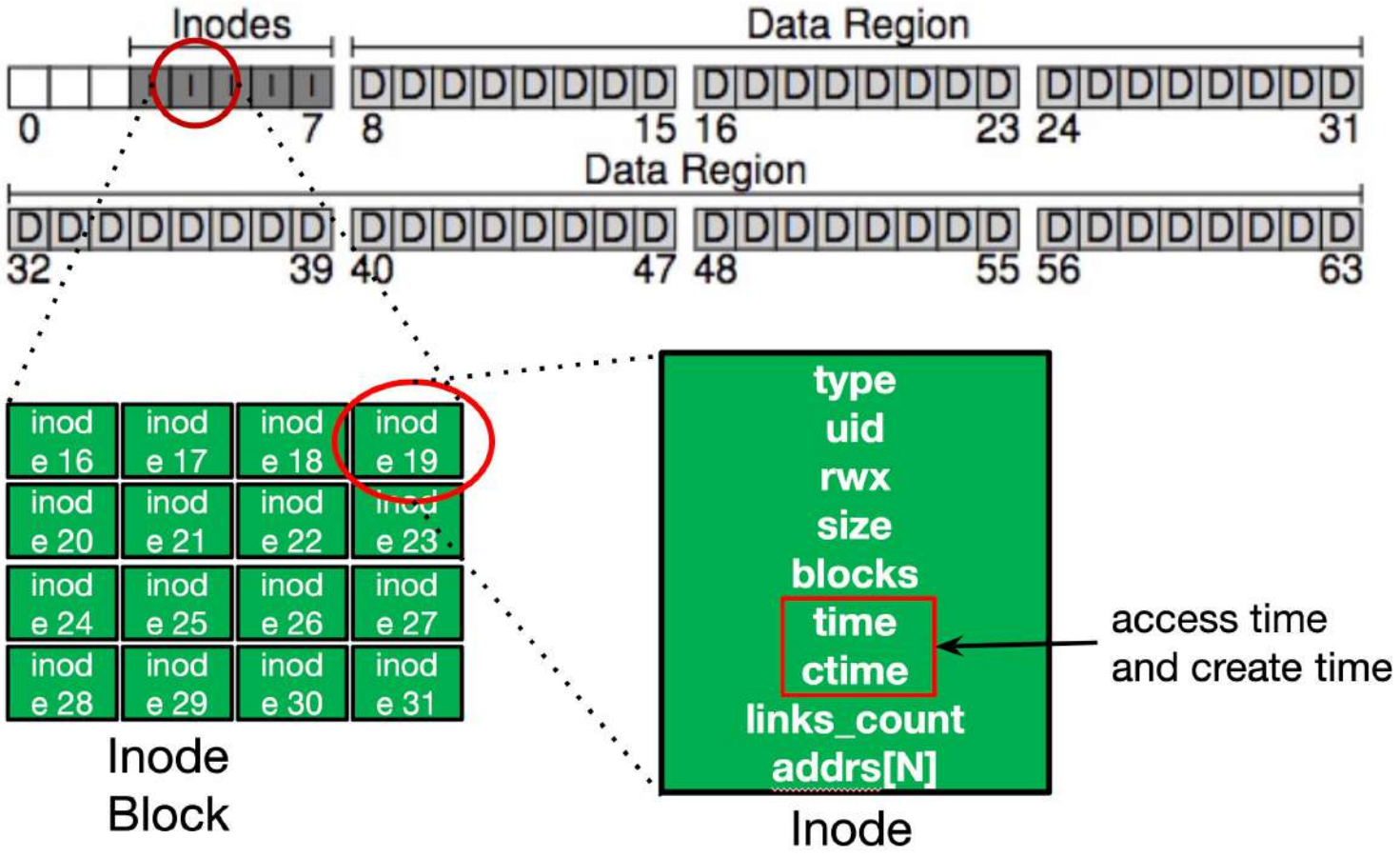
- Answers the *"how to store file information"* question
- Inode → The number corresponds to a data structure consisting of :
  - Type (e.g., file or directory)
  - Size
  - Number of data blocks
  - Address of data blocks
  - User permissions (e.g., root, etc.)
  - …
- Inodes are 128/256 bytes, depending on the FS implementation

# An **inode table**

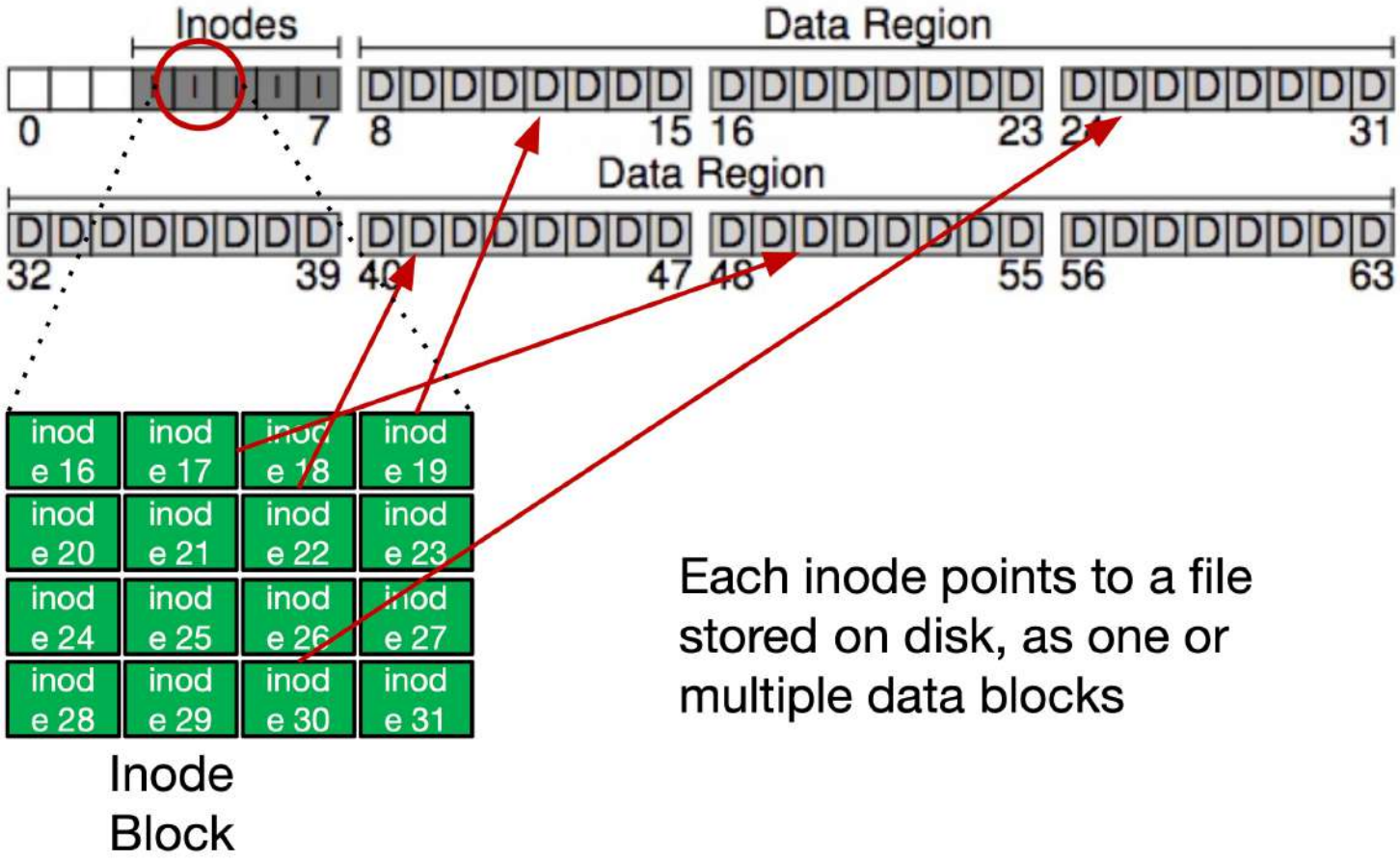
- Answers the *“where to store file information”* question
- In a part of the reserved disk space, as a table/array format



# Visualizing the **inode** and its **table** in the disk



# Visualizing the **inode** and its **table** in the disk



# Now, we must answer:

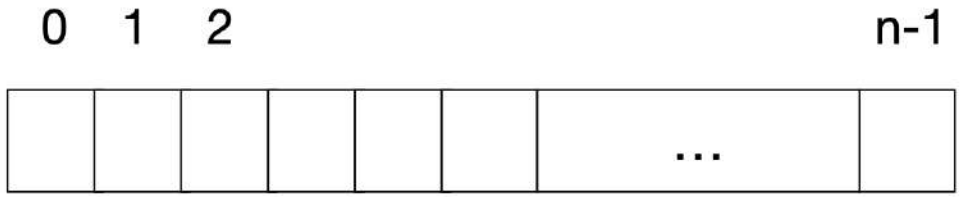
- Where to store the files in the disk?
- Where/how to store the information about existing files (e.g., inodes, path, blocks, etc.) in the disk?
- How to quickly find free locations in the disk for new files?

1) How to quickly find free “data blocks”?
2) How to quickly find free “inodes”?

# Using **bitmap(s)** to store free data blocks/inode

- **Can anyone tell me what's a bitmap?**

Each bit of the bitmap is used to indicate whether the corresponding object/block is **free** (0) or **in-use** (1)

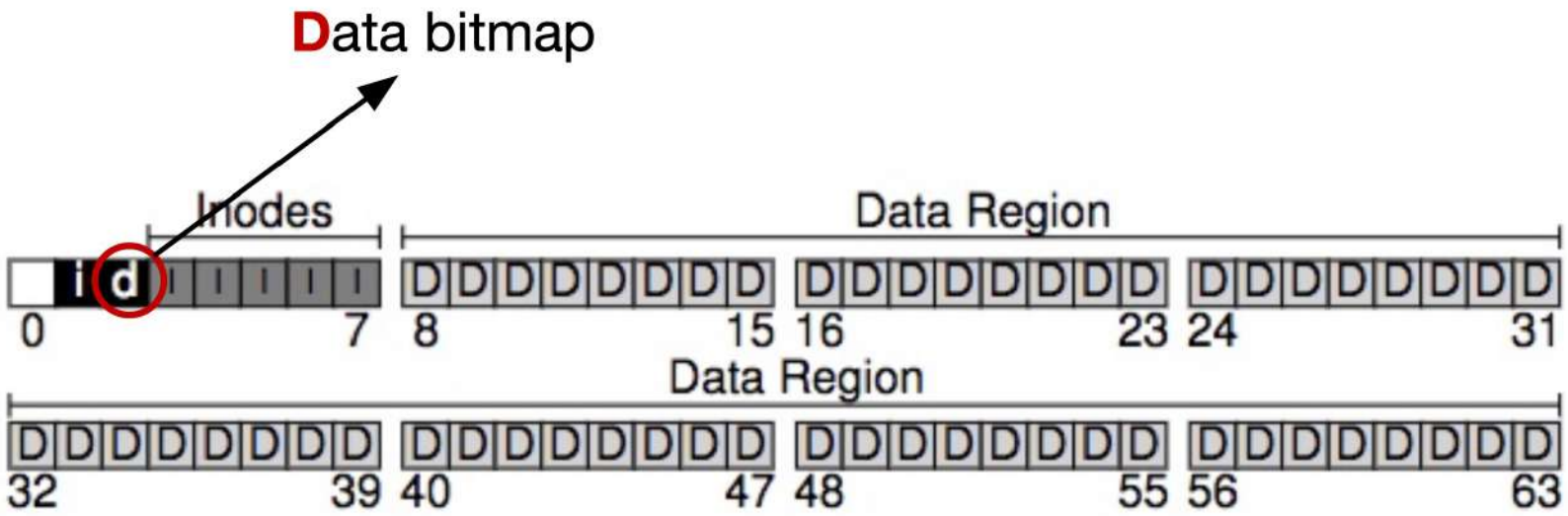



bit[i] = ?

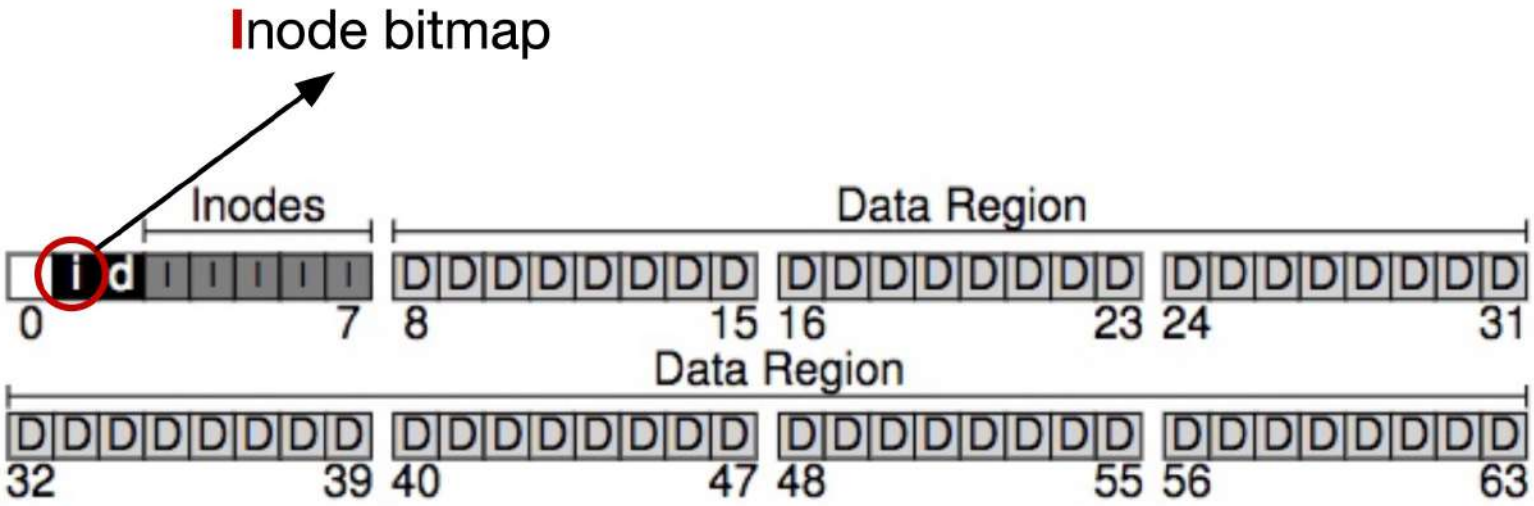
1 ⇒ object[i] in use

0 ⇒ object[i] free

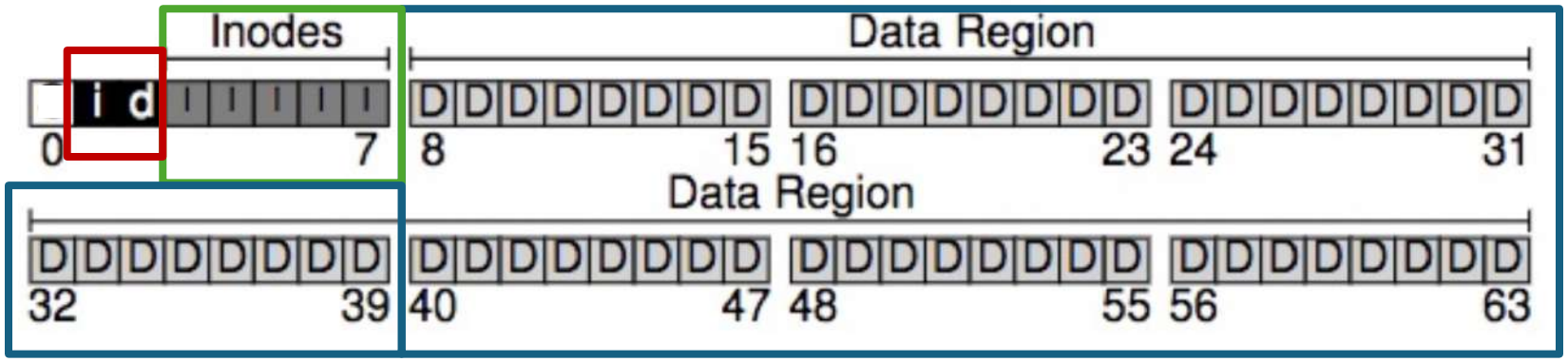
# **Bitmap to find free data blocks:** data bitmap



# **Bitmap to find free data blocks:** inode bitmap



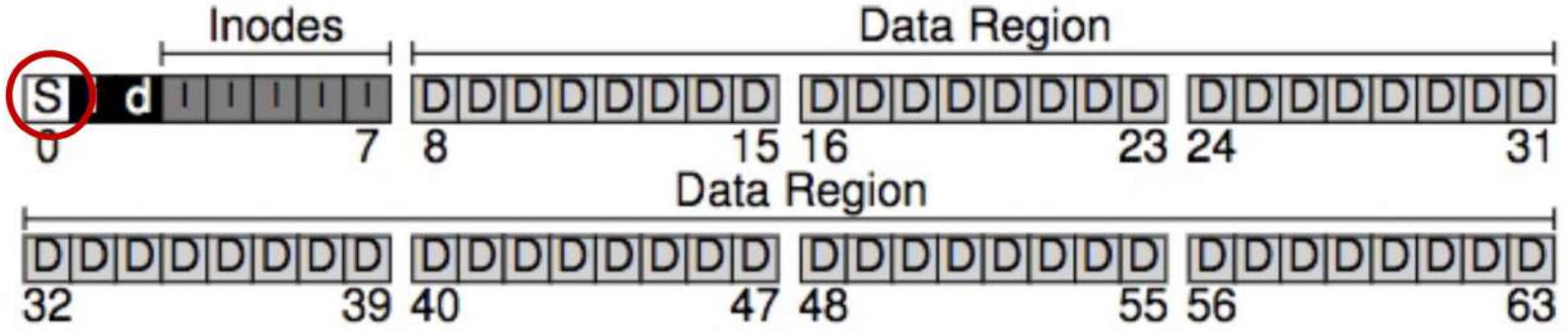
# Is there another question that we may have missed?



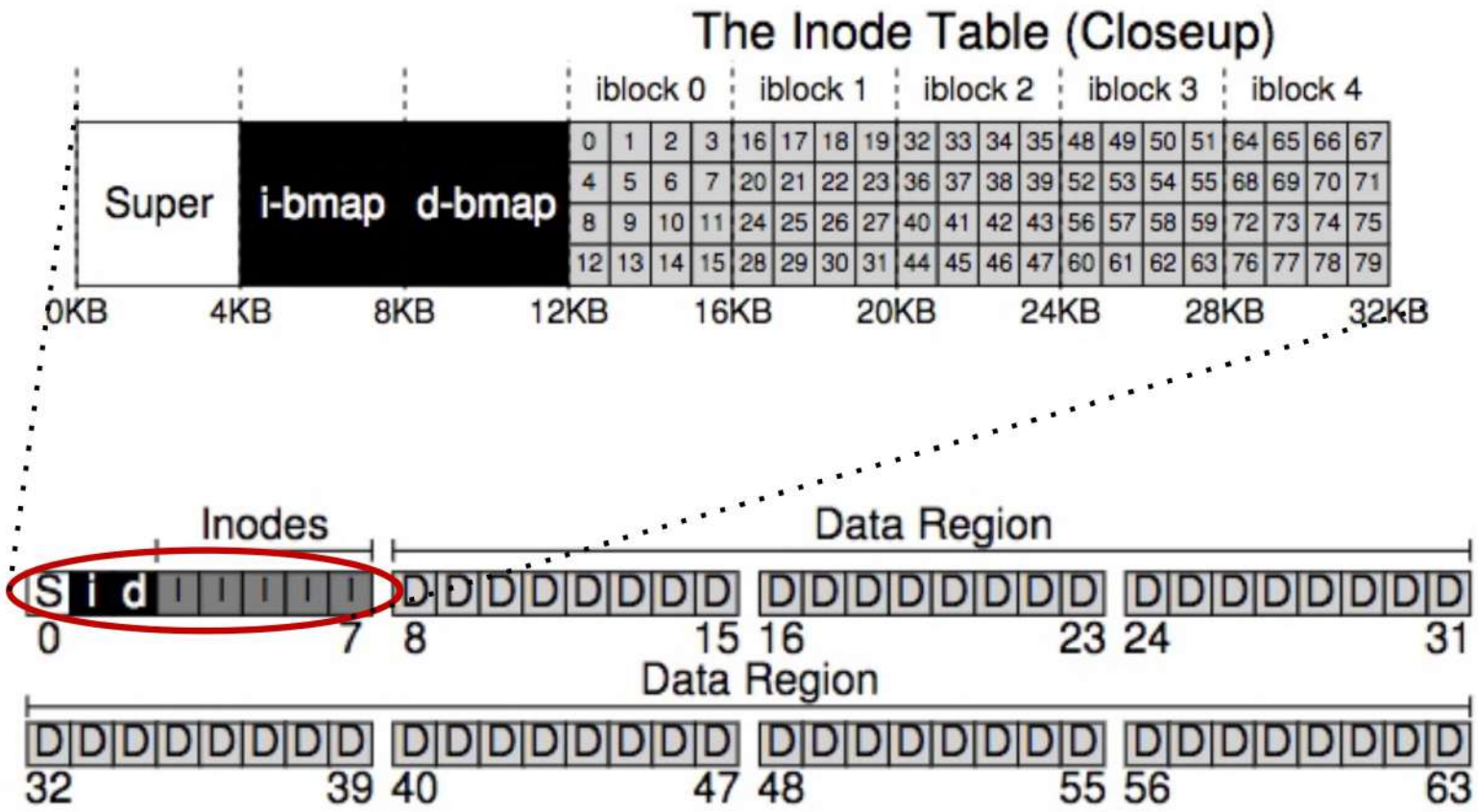

- How to persistently keep track of which regions contain all these disk structures (e.g., data blocks, inode table, bitmaps, etc.)?
  - Their size may be variable (e.g., depending on disk)

# The **superblock**

- Keeps track of basic filesystem information, like
  - Block size
  - How many inodes are there?
  - How much space is free?



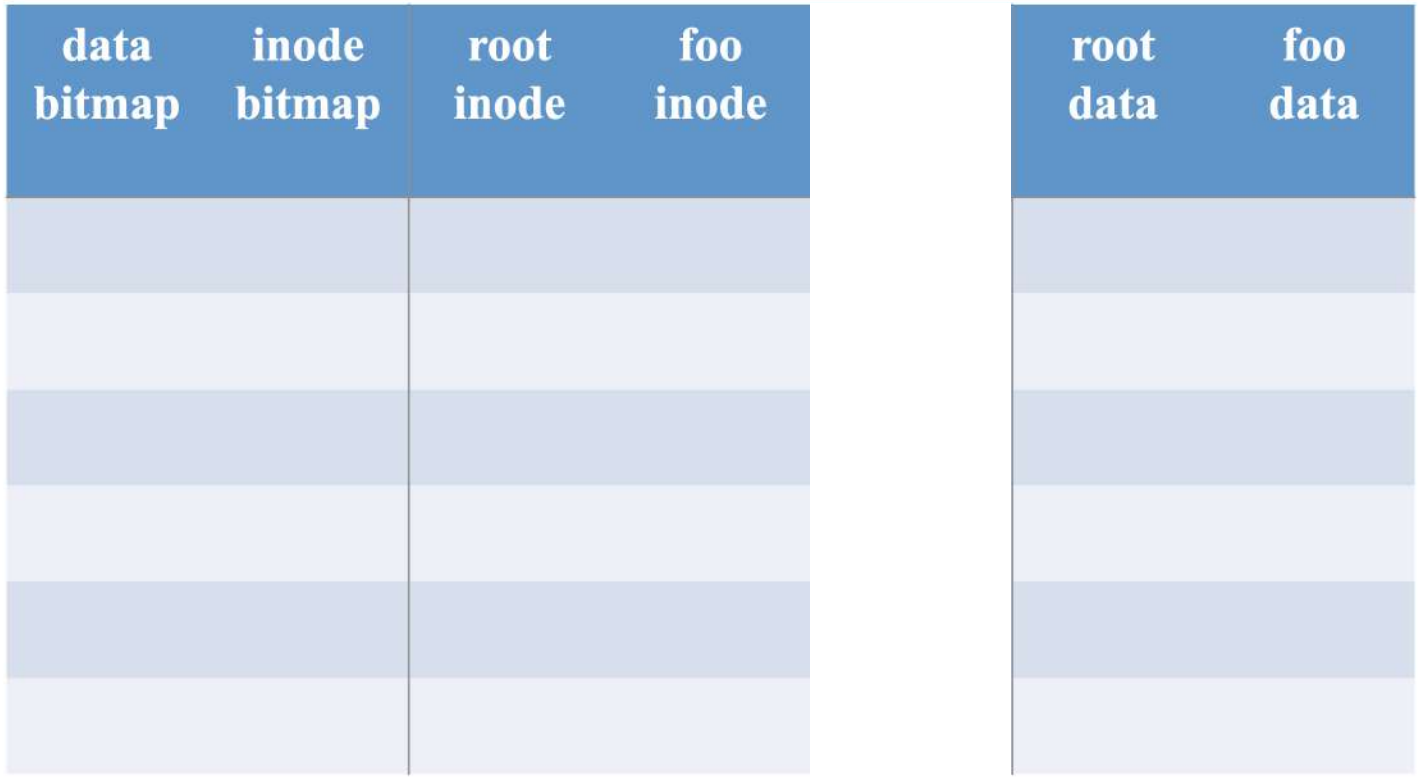
# Putting it all together



# Examples of internal FS operations

# Let’s start with this shape of the disk’s internals

create /foo/bar



# What set of operations do you think we must perform?

create /foo/bar

| data bitmap | inode bitmap | root inode | foo inode |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

| root data | foo data |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |

# Creating a file: *traversal*

create /foo/bar

[traverse]

| data bitmap | inode bitmap | root inode | foo inode |
|---|---|---|---|
| | | read | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

| root data | foo data |
|---|---|
| | |
| read | |
| | |
| | |
| | |
| | |

# Creating a file: *traversal*

create /foo/bar

[traverse]

| data bitmap | inode bitmap | root inode | foo inode | root data | foo data |
|---|---|---|---|---|---|
| | | read | | | |
| | | | | read | |
| | | | read | | |
| | | | | | read |
| | | | | | |
| | | | | | |

# What should we do first after traversal?

create /foo/bar

[traverse]

| data bitmap | inode bitmap | root inode | foo inode |
|---|---|---|---|
| | | read | |
| | | | |
| | | | read |
| | | | |
| | | | |
| | | | |

| root data | foo data |
|---|---|
| | |
| read | |
| | |
| | read |
| | |
| | |

# **Creating a file:** *permission and allocation check*

create /foo/bar

| data bitmap | inode bitmap | root inode | foo inode |
|---|---|---|---|
| | | read | |
| | | | |
| | | | read |
| | | | |
| | | | |
| | | | |

[traverse]

| root data | foo data |
|---|---|
| | |
| read | |
| | |
| | read |
| | |
| | |

- At ”foo”, check that we have permission for this folder. Then, find if “bar” file exists

# **Creating a file:** *allocate new inode*

create /foo/bar [allocate inode]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data |
|---|---|---|---|---|---|---|
| | | read | | | | |
| | | | | | read | |
| | | | read | | | |
| | | | | | | read |
| | read<br>write | | | | | |
| | | | | | | |

- Check the bitmap first to find a free inode for “bar”

# **Creating a file:** *populate new inode*

create /foo/bar [populate inode]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data |
|---|---|---|---|---|---|---|
| | | read | | | | |
| | | | | | read | |
| | | | read | | | |
| | | | | | | read |
| | read write | | | | | |
| | | | | read write | | |

# What should we do after populating the inode?

create /foo/bar [populate inode]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data |
|---|---|---|---|---|---|---|
| | | read | | | | |
| | | | | | read | |
| | | | read | | | |
| | | | | | | read |
| | read write | | | | | |
| | | | | read write | | |

# **Creating a file:** *update parent inode*

create /foo/bar [add bar to /foo]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data |
|---|---|---|---|---|---|---|
| | | read | | | | |
| | | | | | read | |
| | | | read | | | |
| | | | | | | read |
| | read<br>write | | | | | |
| | | | | read<br>write | | |
| | | | write | | | |
| | | | | | | write |

# Should we also allocate a data block for “bar”?

create /foo/bar [add bar to /foo]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data |
|---|---|---|---|---|---|---|
| | | read | | | | |
| | | | | | read | |
| | | | read | | | |
| | | | | | | read |
| | read write | | | | | |
| | | | | read write | | |
| | | | write | | | |
| | | | | | | write |

# **Writing to a file:** *check file block(s)*

write to /foo/bar [block full? yes]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data |
|---|---|---|---|---|---|---|
| | | | | read | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

# **Writing to a file:** *allocate block*

write to /foo/bar [allocate block]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data |
|---|---|---|---|---|---|---|---|
| | | | | read | | | |
| read<br>write | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Writing to a file: *update inode*

write to /foo/bar [point to block]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data |
|---|---|---|---|---|---|---|---|
| | | | | read | | | |
| read<br>write | | | | | | | |
| | | | | write | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# **Writing to a file:** *write data to block*

write to /foo/bar [point to block]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data |
|---|---|---|---|---|---|---|---|
| | | | | read | | | |
| read<br>write | | | | | | | |
| | | | | write | | | |
| | | | | | | | write |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

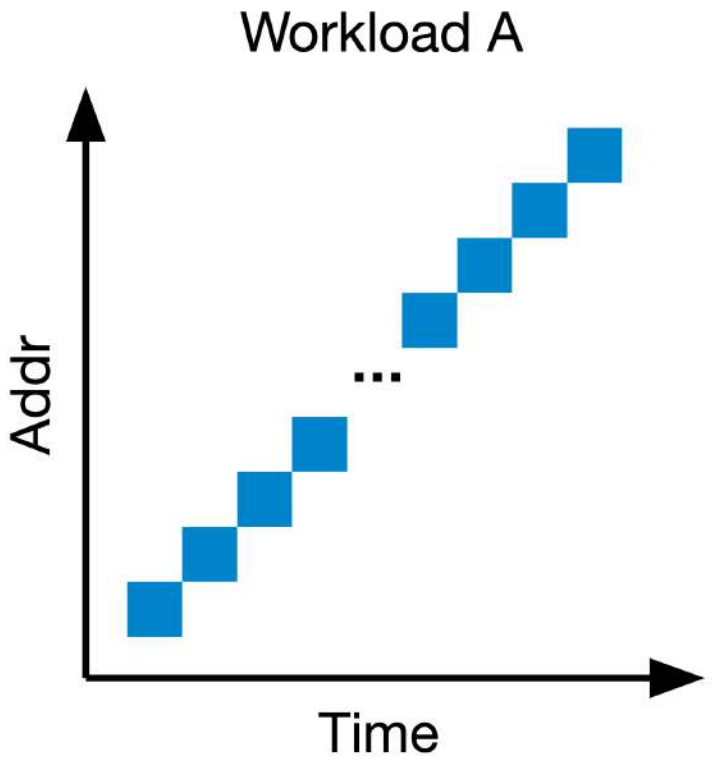
# Writing to a file **(summary)**

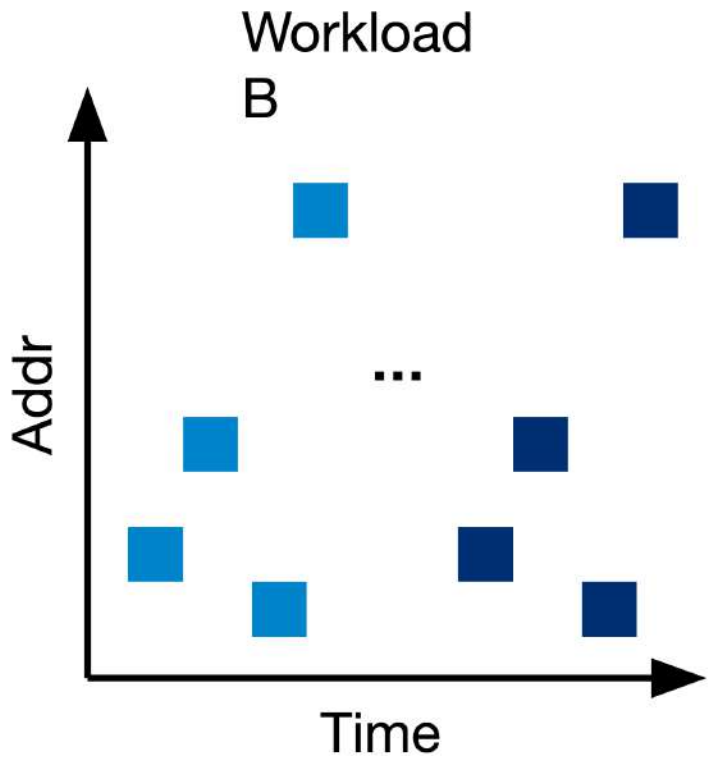
write to /foo/bar [point to block]

| data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data |
|---|---|---|---|---|---|---|---|
| | | | | | dir entries | | file |
| | | | | read | | | |
| read<br>write | | | | | | | |
| | | | | write | | | |
| | | | | | | | write |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# FS optimizations: data layouts in disks

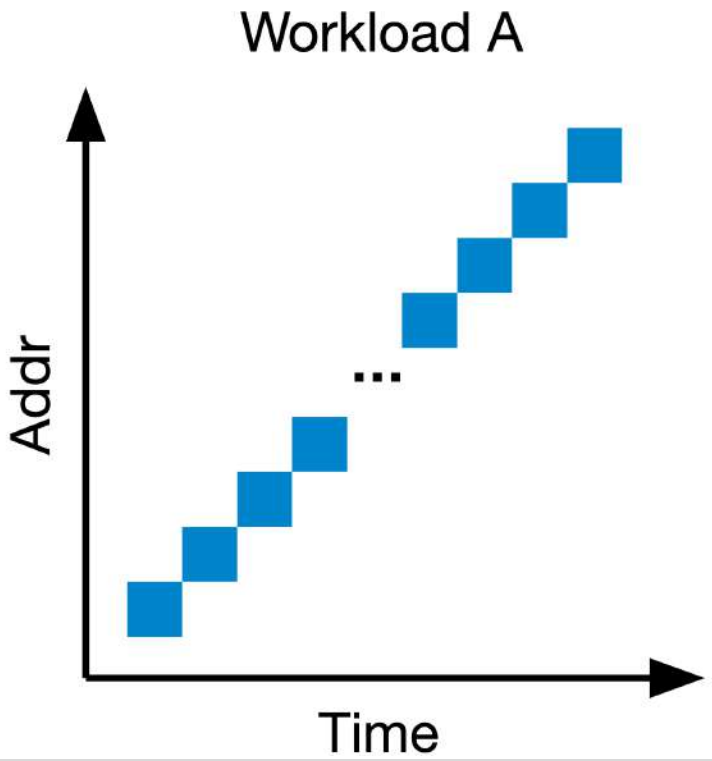
# What is the difference between the two workloads?

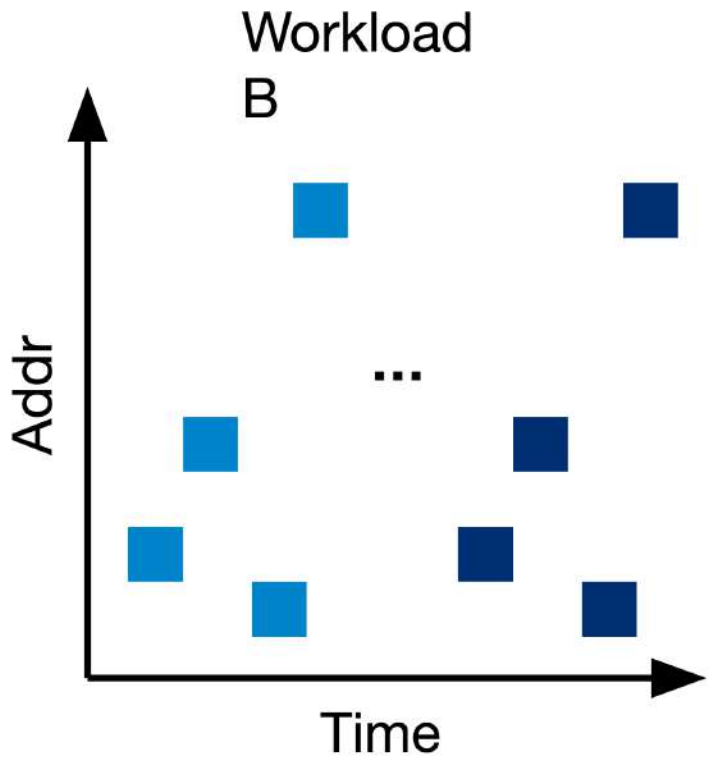



- **Spatial** locality: addresses are close to each other in “space”
- **Temporal** locality: addresses are close to each other in “time”

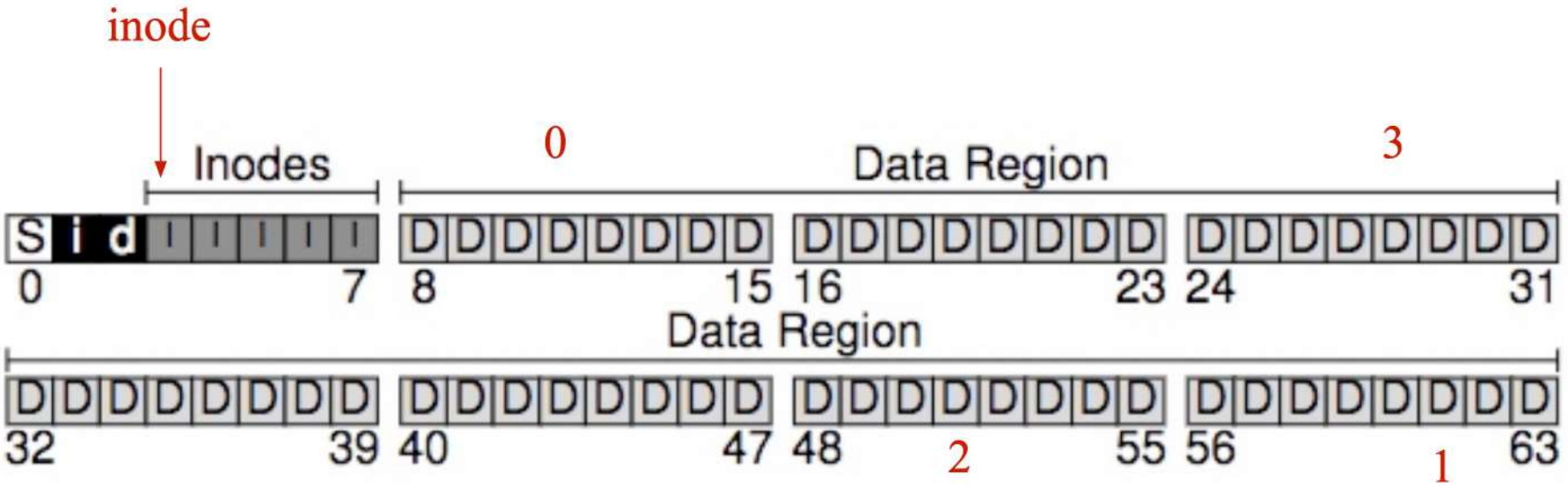
# Which of the following is actually better for disks?



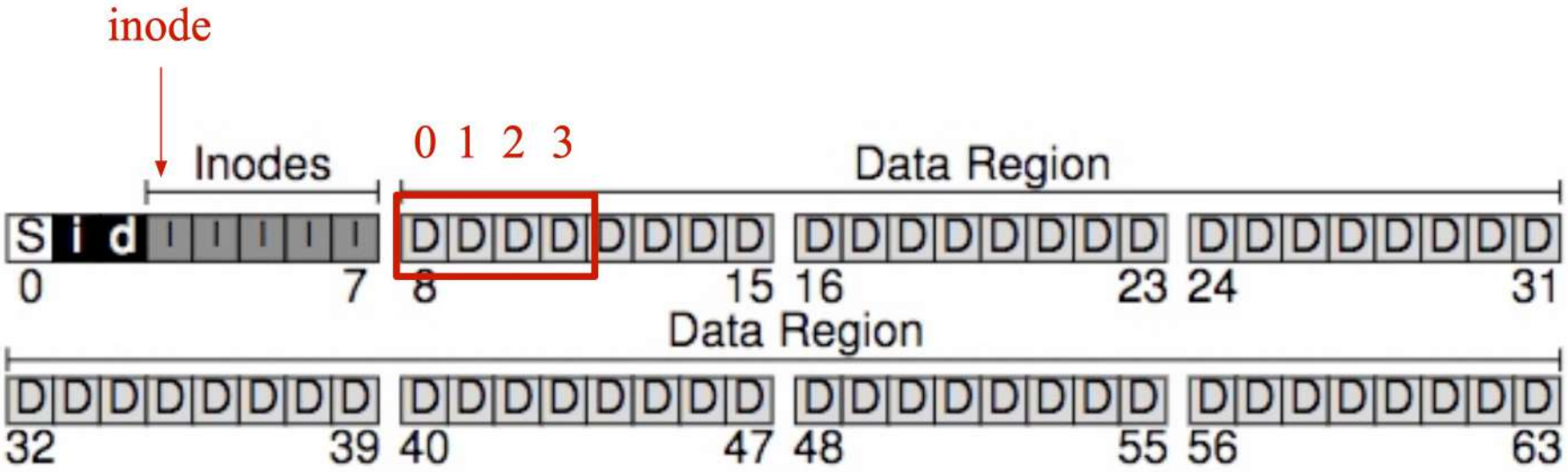

- **Spatial** locality: addresses are close to each other in “space”
- **Temporal** locality: addresses are close to each other in “time”

# Imagine that we must choose inode and data blocks



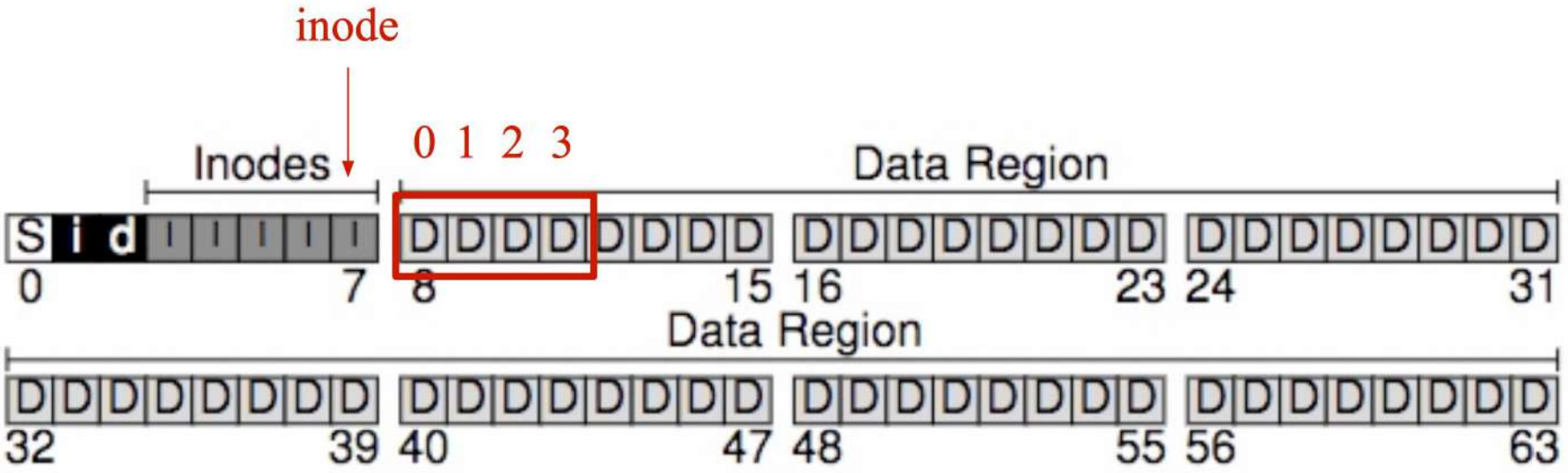

- **Is this a good or a bad policy to choose blocks?**
  - Bad policy; will require slow “random” access

# A better policy would be as follows




- **Is there something else that we can optimize?**
  - Location of the inode itself, in this particular case. ☺

# An even better policy



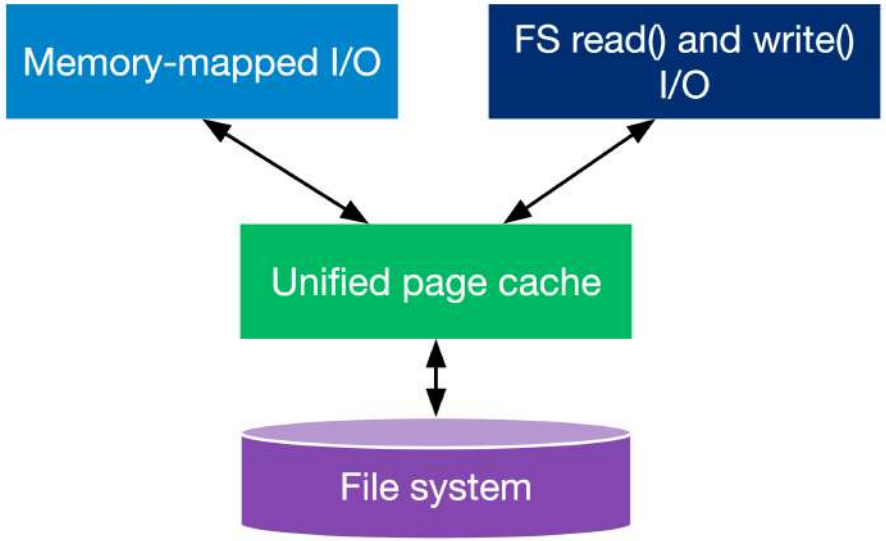
# FS optimizations: buffering in memory

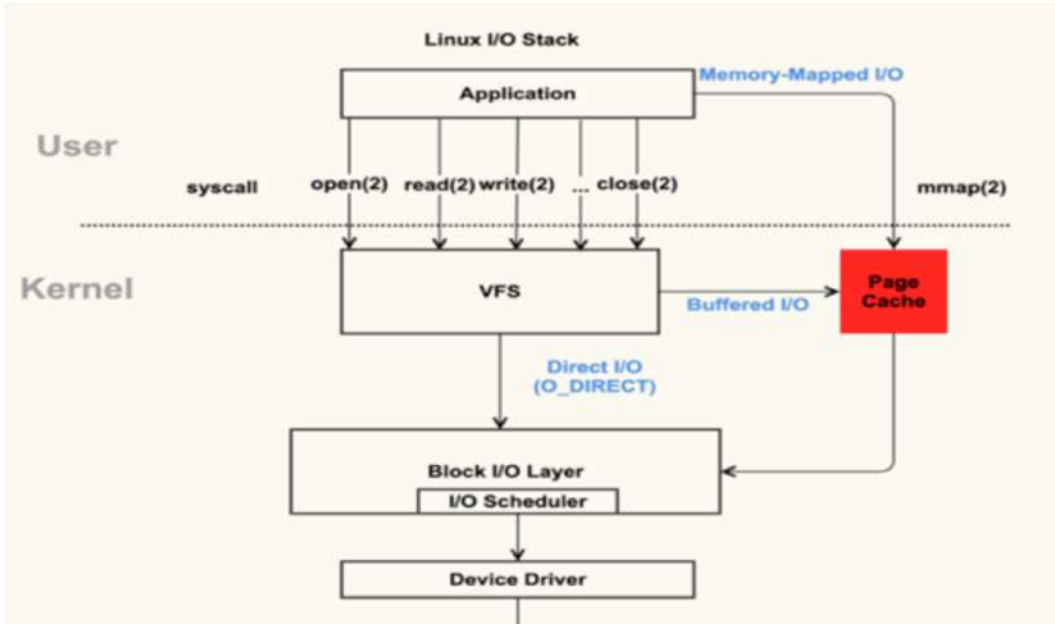
# Avoiding excessive I/O operations is important!

- Even with all possible disk optimizations, I/O remains slow and unpredictable depending on workloads
- File system will cache “files” in memory (recall how the xv6 block layer caches “blocks”) to improve performance
- To do so, the file system will:
  - Reserve a region for the **file cache**
  - Find out which files are **frequently read/written**
  - Keep those files within the cache

# Modern OSs take a **unified cache** approach

- Keep a single cache for (a) file pages and (b) program memory pages
- Avoids having multiple caches, and separately dealing with them



# Buffering creates a **tiny headache** for the file system..

- Imagine the following set of events from an FS:

  - Program writes to a new file (e.g., bar)
  - FS caches all writes to the file in-memory
  - FS updates the inodes within the disk
  - FS writes all changes to data blocks within disk (e.g., block-by-block using Linux's block I/O)

- **Can you spot potential problems?**

# Questions? Otherwise, see you next class!