

Week 11

Persistent Storage: Basic File System Implementation

(Part 1)

Max Kopinsky
18 March 2025

State of the Course

- As mentioned, I've had to step back from constantly monitoring Discord...
 - You might notice my title on Minerva/official documents is *Course* Lecturer, not *Faculty* lecturer. I haven't technically finished my Master's yet, because I spend too much time focused on trying to make my courses good 😊.
 - Given McGill's current financial situation, finishing and properly joining the faculty is now a top priority. This is why you've seen less of me and issues have been fixed/handled slower. I've also delegated official grading for assignments to a TA, and the turnaround time to communicate & fix issues is therefore slower.
- We are close with assignment grades. Once A1 is done, A2 + MP should follow quickly.

State of the Course

- There was some trouble with the MP autograder...
 - My understanding is that this was mostly about the indentation check
 - I think we fixed the issue, but I respect that this caused a lot of stress.
 - To alleviate some of that, we will **not do a style check for MP**. Any issues you had with the autograder are moot 😊.
- **I will do an emoji poll on Discord to check how many people requested a late day in response to autograder feedback about style.**
 - If the number is small, then I will handle it on a case-by-case basis.
 - If many students did this, then I may need to have you fill out a form.
- Ultimately I want your grade to reflect your understanding. We'll make that happen 😊.

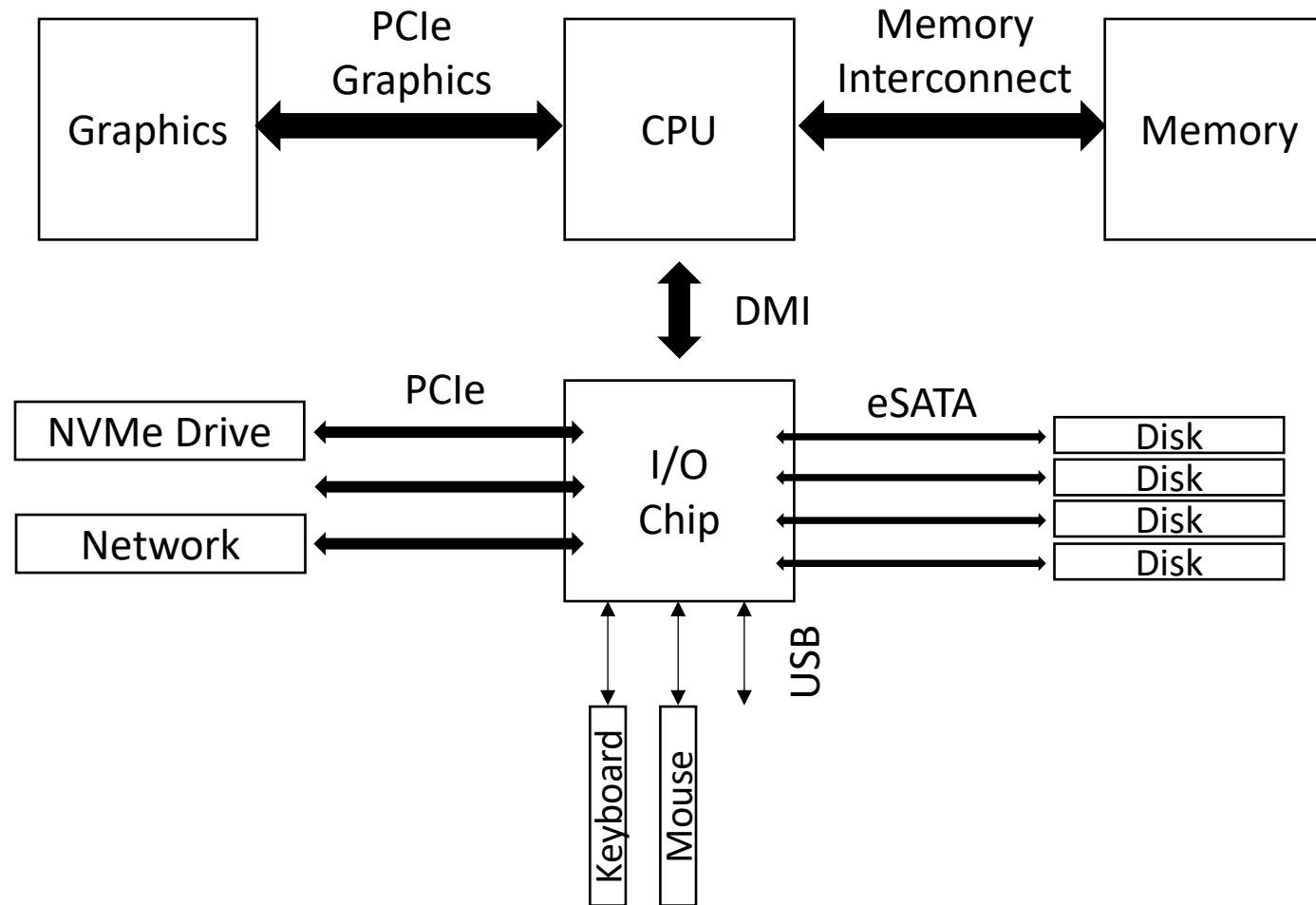
State of the Course

- With all that out of the way....
- **Course evaluations are available.**
 - Please think about your feedback, both positive and negative, about the course for the next couple of days.
 - As stated, I am not a research professor. My job is to teach. Your feedback, positive and especially negative, help me improve my teaching and the course year over year.
- I will give you time in-class to do course evaluations at the start of the lecture on Thursday. **I will leave the room during this time.**
- For logistical reasons, I am considering a small blanket extension to A3. However, if course evaluations get a $\geq 70\%$ response rate, **I will immediately extend the deadline to the last day of class (11 Apr).**

Key Concepts

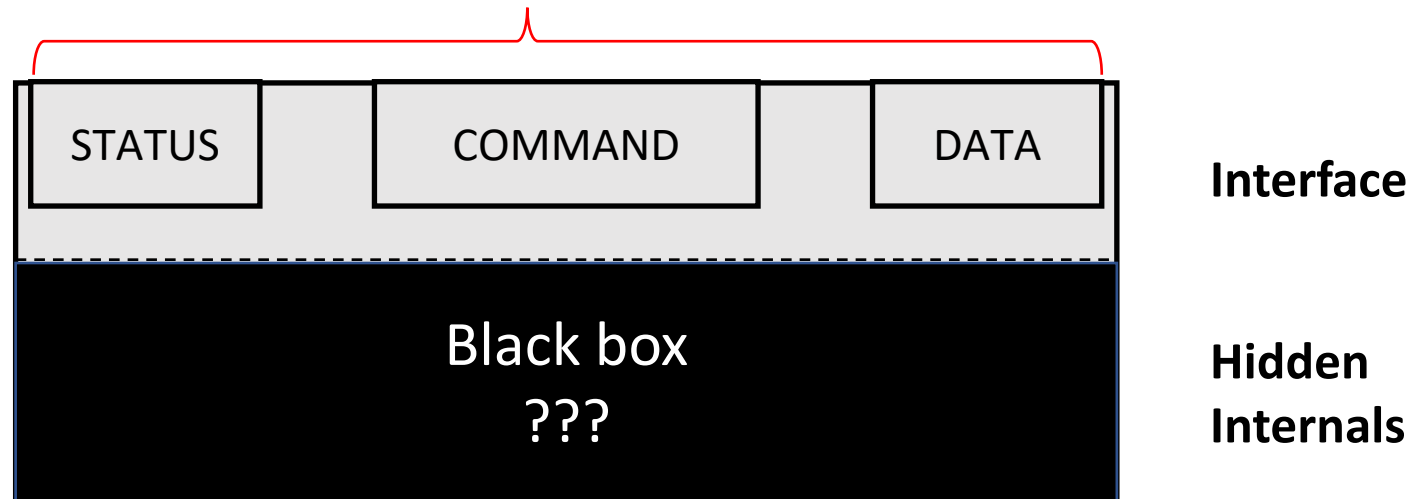
- File system “mental model”
 - Data structures : on disk, in memory
 - File data allocation methods
 - Contiguous, extent-based, linked, FAT, indexed, indirect blocks
 - File access methods
 - Create, open, write, read, close

Recap Week 10 - I/O System Architecture



Recap Week 10 - How does OS use devices?

Canonical device interface: OS reads/writes to these to control device behavior



- OS accesses device interface through: polling, interrupts,
- Direct memory access (DMA)

Recap Week 10 - File System Interface

Recap Week 10 - File

- Un-interpreted un-typed sequence of bytes
- Identified by a globally unique *uid*

Recap Week 10 - Open File

- File instance accessed by a process
- Identified by a per-process unique *tid* or *fd*

Recap Week 10 - Directory

- Set of mappings (string \rightarrow uid)

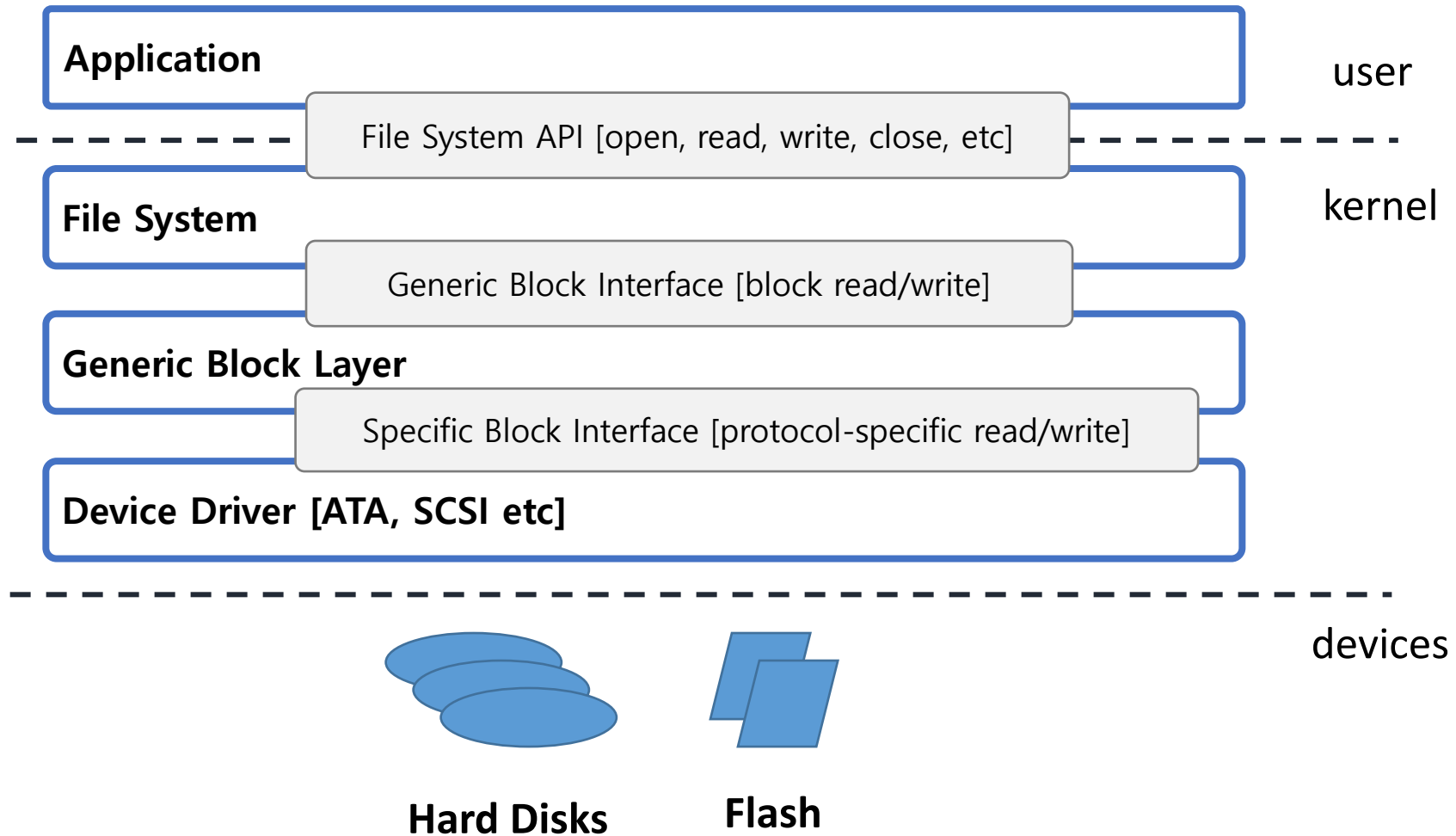
Recap Week 10 - File System Primitives

- Access: Create(), Delete(), Read(), Write()
- Random vs. Sequential and Seek()
- Concurrency: Open(), Close()
- Naming

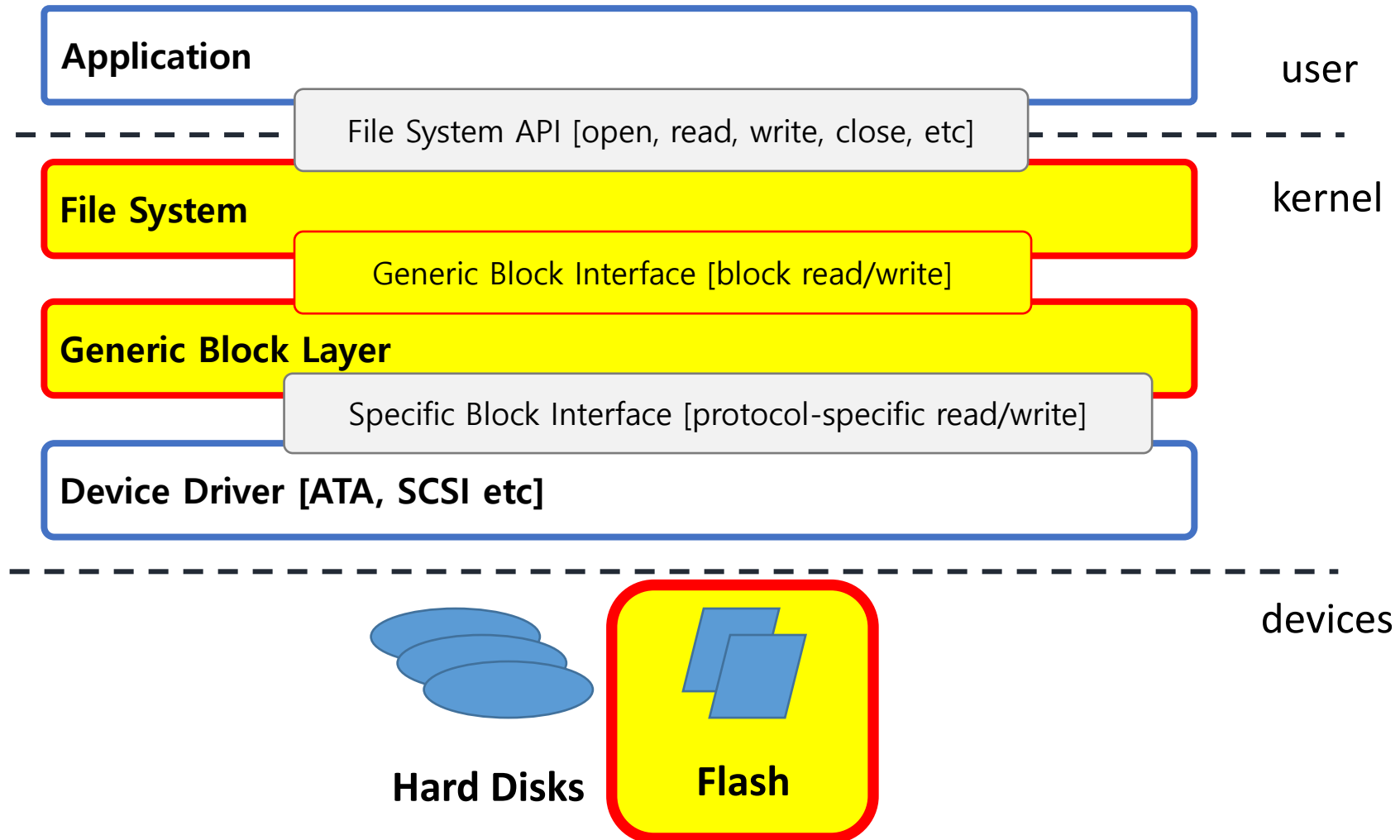
Recap Week 10 - Hierarchical Directory Structures

- Tree
- (Acyclic) Graph
 - Allows sharing of two *uids* under different names
- Two additional primitives
 - HardLink() and SoftLink()

Overall Picture



This week and next week Lectures



File System Implementation

File System Role

The main task of the file system is to **translate**

From **user interface** functions

```
Read(uid, buffer, bytes)
```

To **disk interface** functions

```
ReadSector(logical_sector_number, buffer)
```

File System Implementation

Key aspects of the system:

1. Data structures

- On disk
- In memory

2. Access methods

- How do we open(), read(), write() ?

Data Structures

Disk vs in-memory simple but golden rules:

- **If it is not on disk and you crash, it is gone!**
- **If you need it after a crash, it must be on disk.**

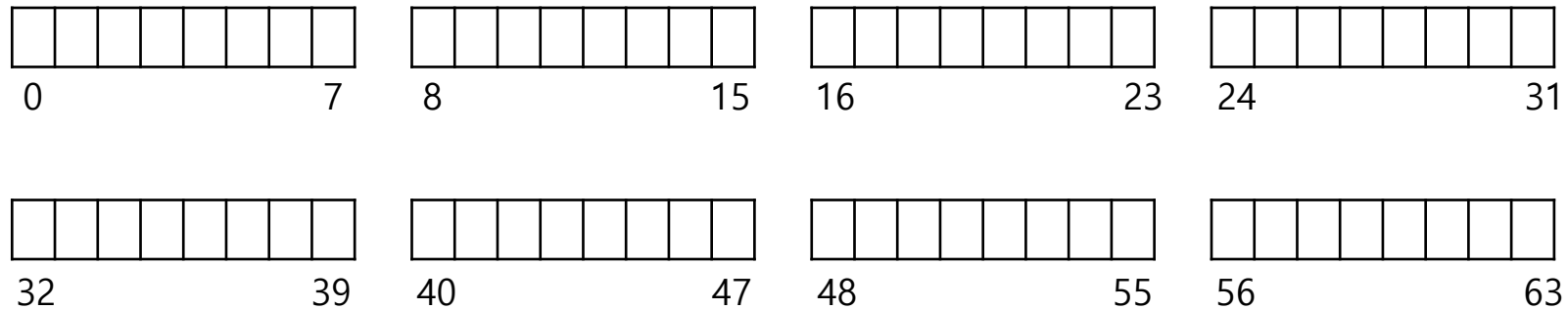
Disk Data Structures

Disk Data Structures

- **Data Region** ← occupies most space in FS
 - User data
 - Free space
- **Metadata**
 - Inodes
 - Free space management
 - Superblock

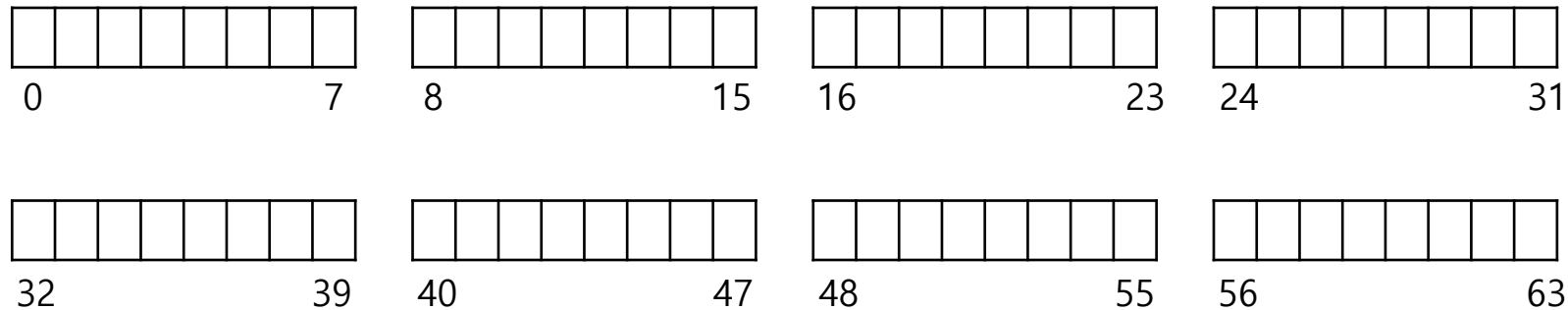
Simple Example

- Disk with 64 blocks.
- 4KB block size.



Simple Example

- Disk with 64 blocks.
- 4KB block size.



Remember: Want translation between user FS interface to disk interface
→ Need some structure to map files to disk blocks

Remember: Similarity to Memory?

Same principle: map logical abstraction to physical resource

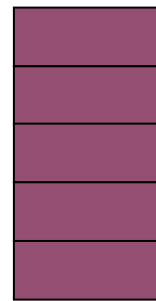
Logical View: Address Spaces



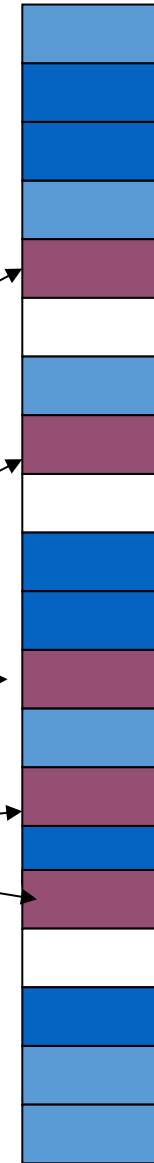
Process 1



Process 2



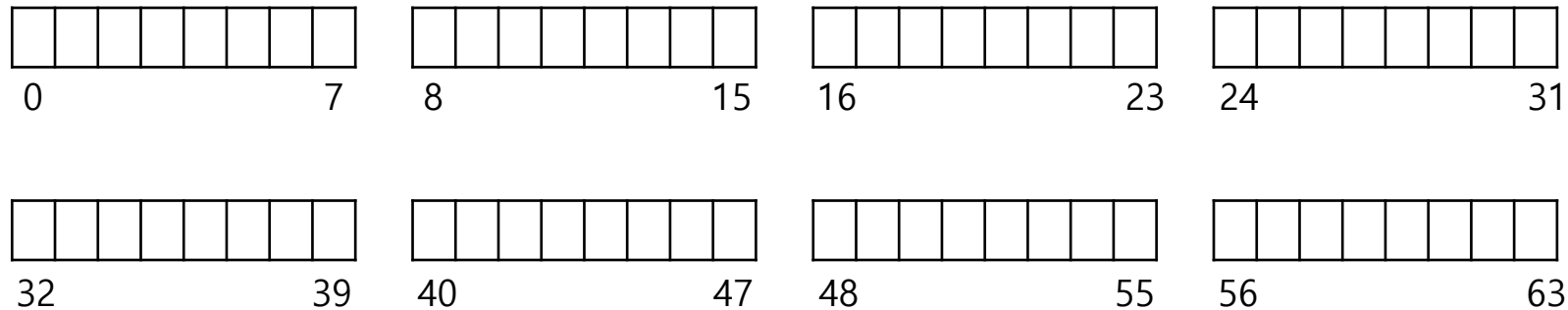
Process 3



Physical View: RAM

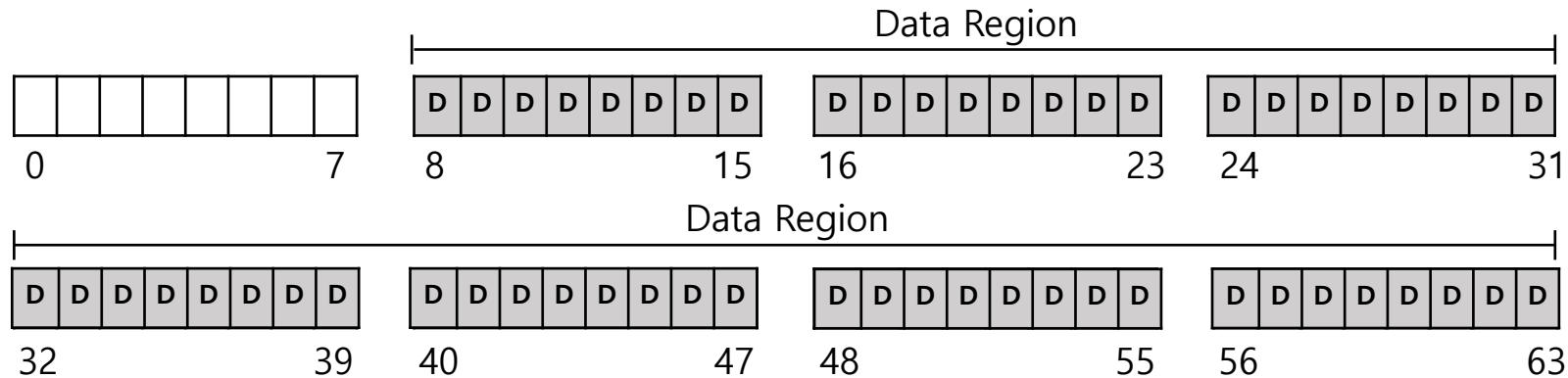
Need structure to map files to disk blocks

- Disk with 64 blocks.
- 4KB block size.



Reserve Data Region to Store User Data

- Disk with 64 blocks.
- 4KB block size.



- Data region contains user data and free space.

Reserve Data Region to Store User Data

- Disk with 64 blocks.
- 4KB block size.



- Data region contains **user data (files)** and **free space**.
 - How do we track files ?
 - How do we track free space ?

How do we track files?

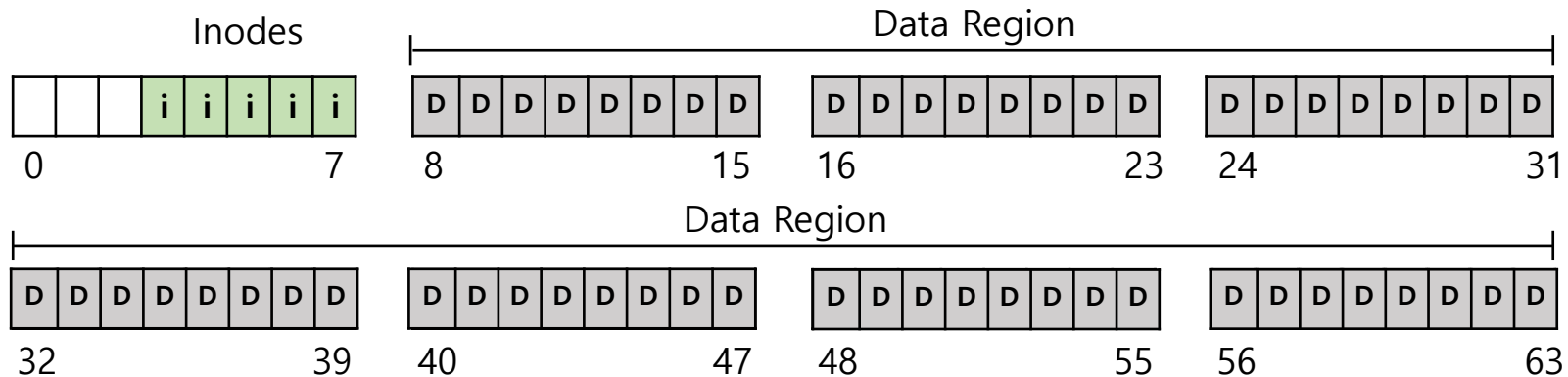
Inode

- Short from “index node”
- Tracks file metadata:
 - type (file or dir?)
 - uid (owner)
 - rwx (permissions)
 - size (in bytes)
 - blocks occupied by file
 - timing information (creation time, last access time)
 - links_count (# paths aka # hard links)

Reserve Inode Table to Track Files

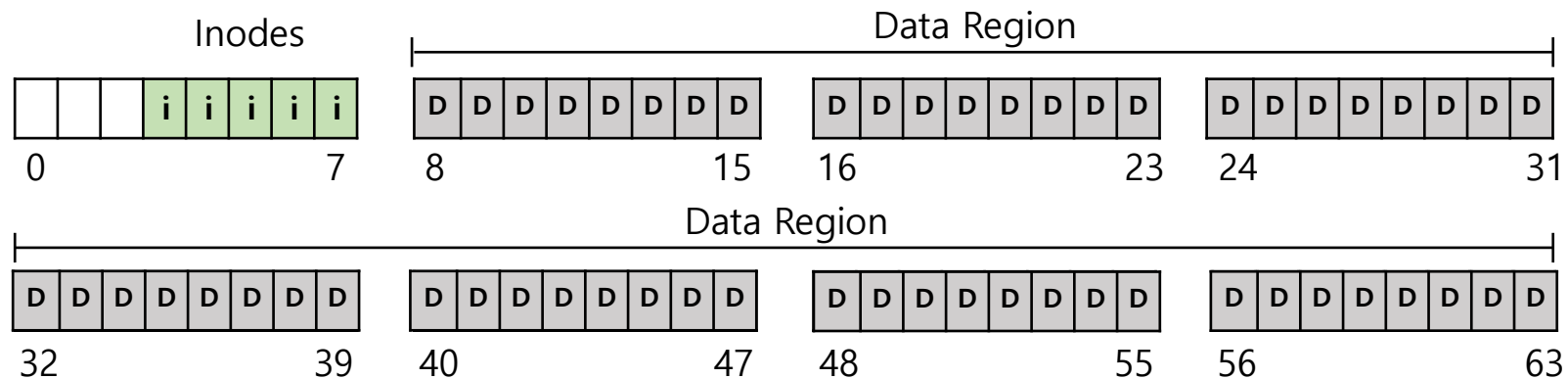
Inode table: holds an array of on-disk inodes

- Inodes are not too large (128 or 256 bytes per inode)



How many inodes?

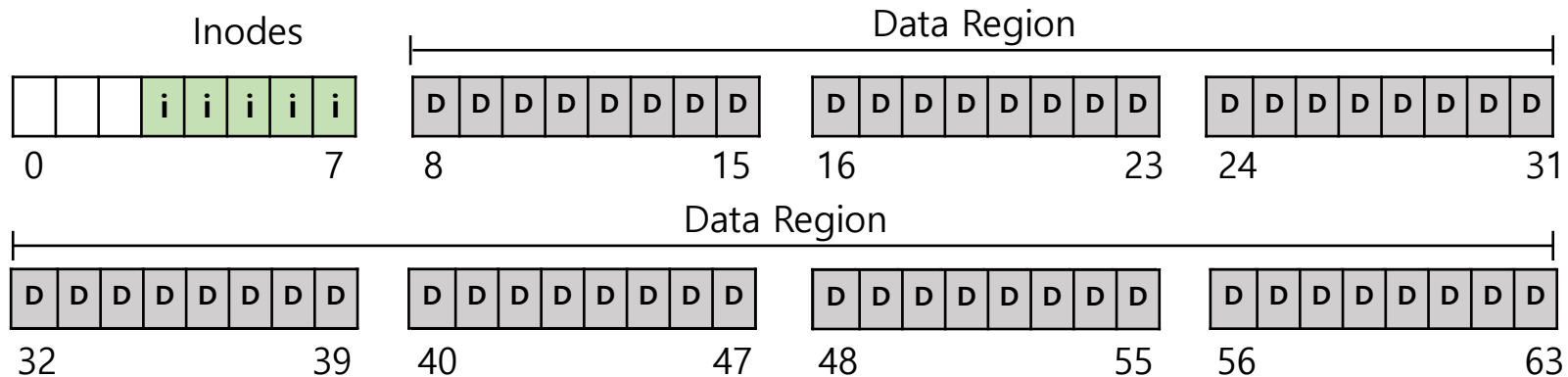
Question: Assuming 256B per inode, how many inodes in our file system with 4KB blocks?



How many inodes?

Question: Assuming 256B per inode, how many inodes in our file system with 4KB blocks?

4KB = 4096 B \rightarrow $4096 / 256 = 16$ inodes per block \rightarrow $16 * 5 = \mathbf{80}$ inodes in total in FS



Close-up on Inodes Table

Inodes Table																							
				iblock 0				iblock 1				iblock 2				iblock 3				iblock 4			
				0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
				4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
				8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
				12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79
0KB	4KB	8KB	12KB	16KB				20KB				24KB				28KB				32KB			

How do we track free space?

Allocation structures:

- For data
- For inodes

Allocation Structures Implementation

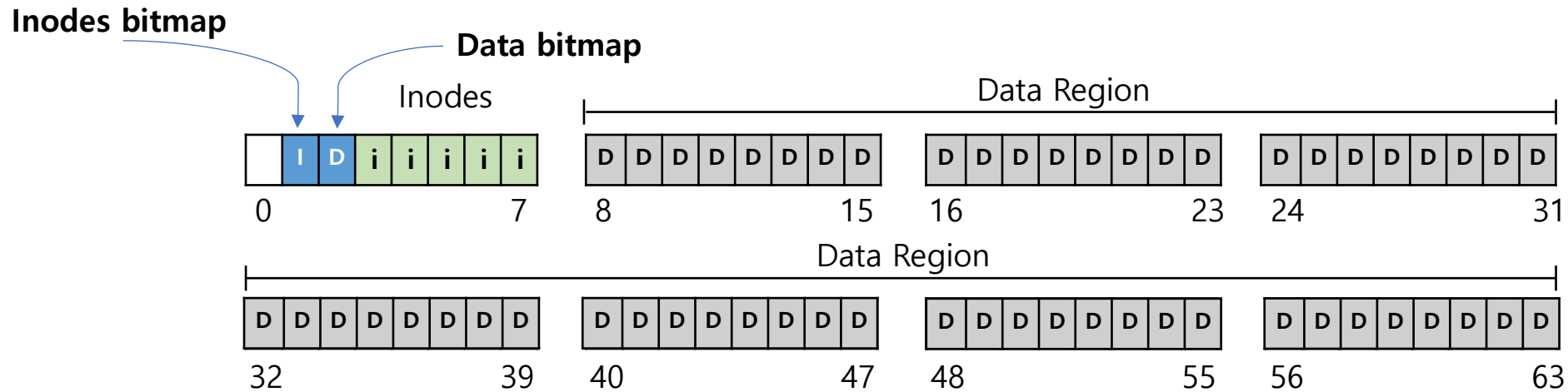
- Free-lists (Remember from memory management module)
- Bitmaps
 - Data structure where each bit indicates if corresponding object is free or in use
 - 1 = in use
 - 0 = free

Allocation Structures Implementation

We will use bitmaps:

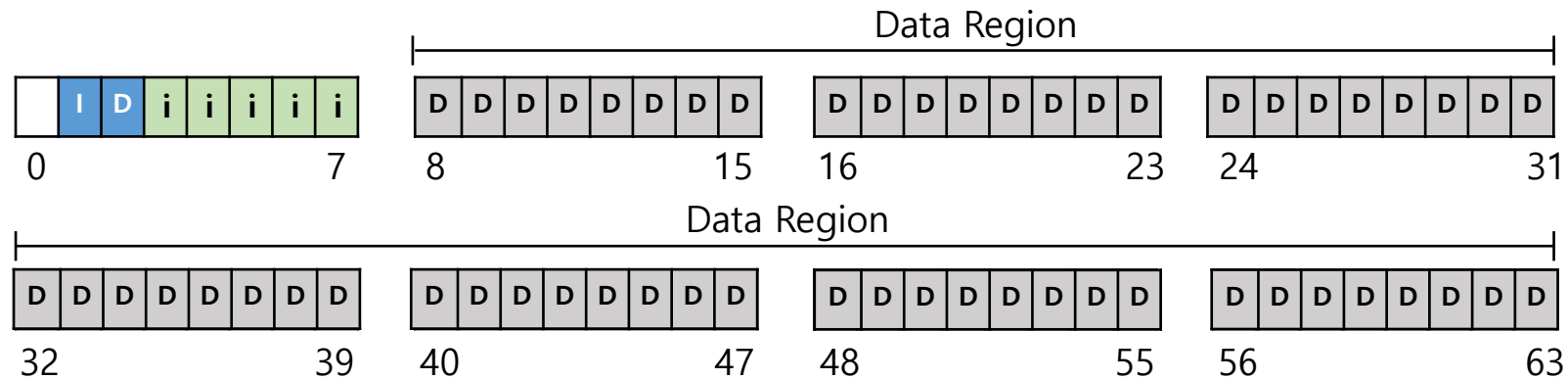
- Data bitmap
- Inodes bitmap

Reserve Blocks for Allocation Structures



Bitmap capacity?

Question: Assuming we use one 4KB block per bitmap, how many inodes and data blocks can we track?

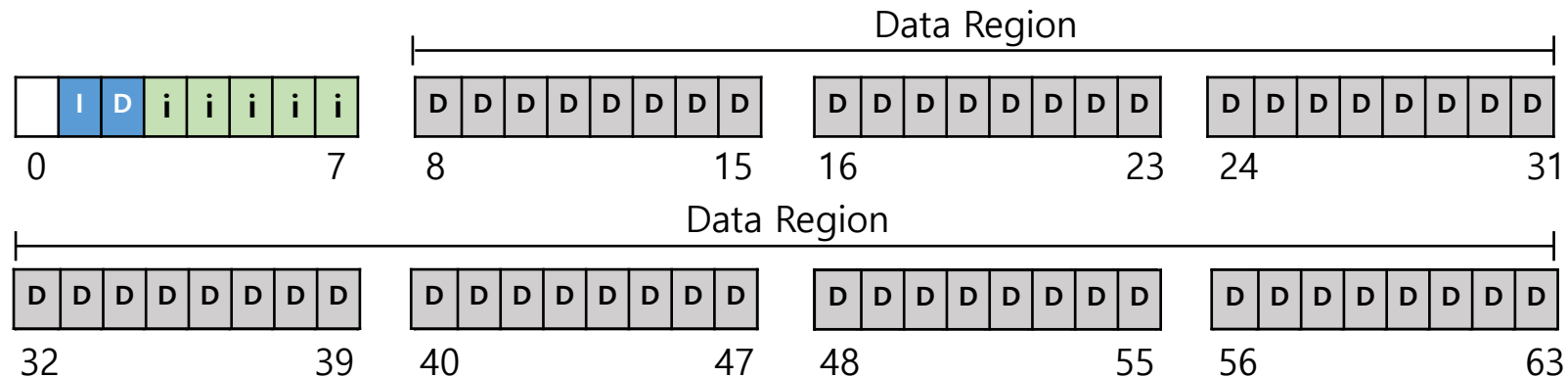


Bitmap capacity?

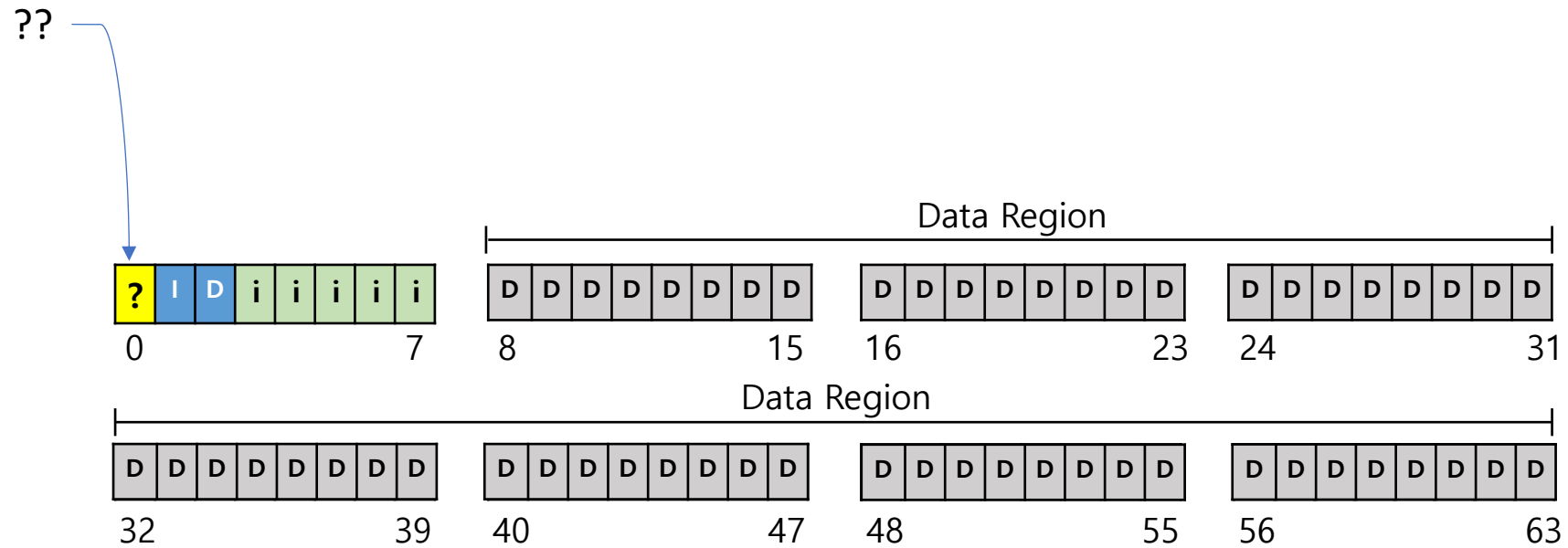
Question: Assuming we use one 4KB block per bitmap, how many inodes and data blocks can we track?

$4096 \text{ B} * 8 \text{ bits/B} = 32768 \text{ bits}$ in each bitmap

→ Can track ~32K inodes and ~32K blocks (a bit excessive for our system with max 80 inodes and 56 data blocks, but a real 1 TB drive has $1\text{TB}/4\text{KB}=250$ million data blocks, so this is low!)



One block left



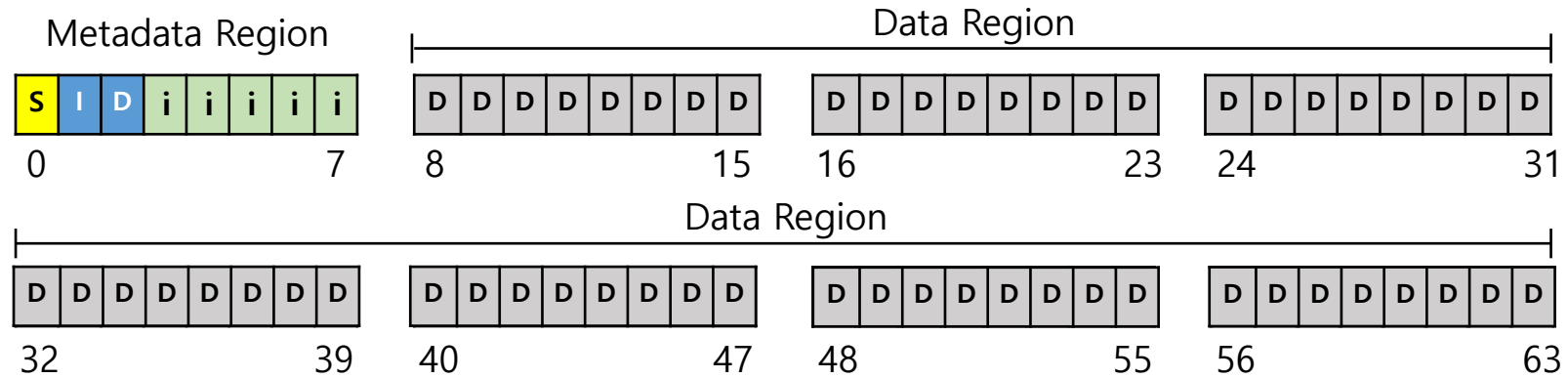
Superblock

Contains **file system metadata**

- #inodes
- #data blocks
- Start of inodes table
- What kind of FS is this?
- ...



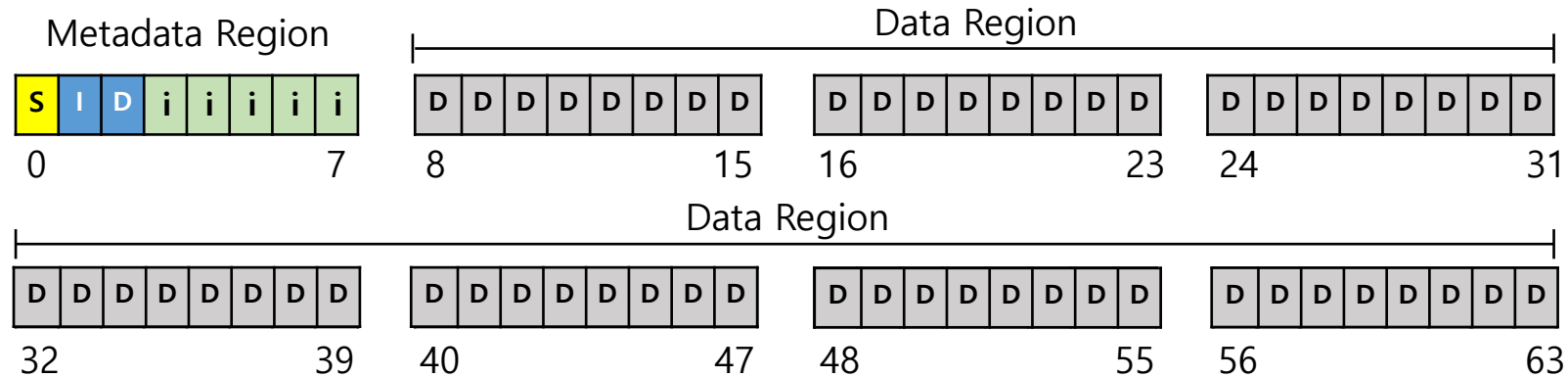
Disk Data Structures



Disk Data Structures

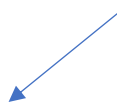
One important question remains:

How do we allocate files to blocks?



(and the converse: How do we handle free space?)

Remember: Inode

- Tracks file metadata:
 - type (file or dir?)
 - uid (owner)
 - rwx (permissions)
 - size (in bytes)
 - **blocks occupied by file**
 - timing information (creation time, last access time)
 - links_count (# paths)
- We want to know “good” strategies to map files to blocks.
- 

User Data Allocation

Strategies

- Contiguous
- Extent-based
- Linked list
- File-Allocation Tables (FAT)
- Indexed
- Multi-level Indexed

User Data Allocation

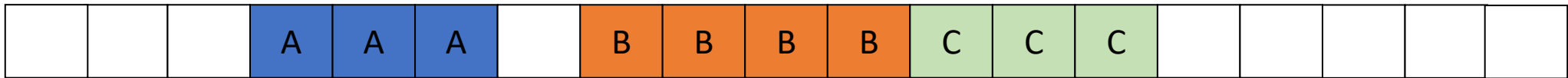
Questions

- Amount of fragmentation (external and internal)
- Ability to grow file over time
- Sequential access performance
- Random access performance
- Metadata overhead
 - i.e., “wasted space” to persistently store metadata

Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk

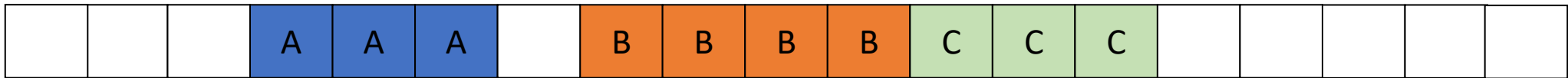
Metadata: Starting block + size of file



Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk

Metadata: Starting block + size of file



Fragmentation

Growing files

Sequential access

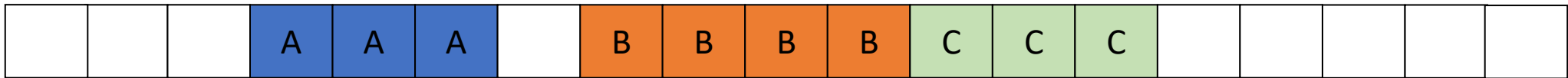
Random access

Metadata overhead

Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk

Metadata: Starting block + size of file



Fragmentation

High fragmentation (many unusable holes) 😞 😞

Growing files

May not be able to grow without moving file 😞 😞

Sequential access

Excellent 😊 😊

Random access

Simple calculation 😊

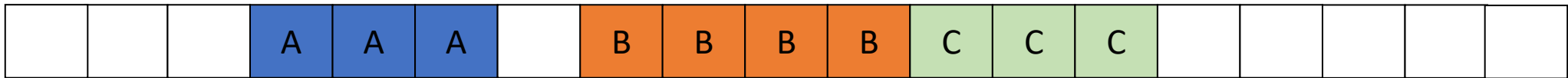
Metadata overhead

Low 😊

Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk

Metadata: Starting block + size of file



Fragmentation

High fragmentation (many unusable holes) ☹️ ☹️

Growing files

May not be able to grow without moving file ☹️ ☹️

Sequential access

Excellent 😊 😊

Random access

Simple calculation 😊

Metadata overhead

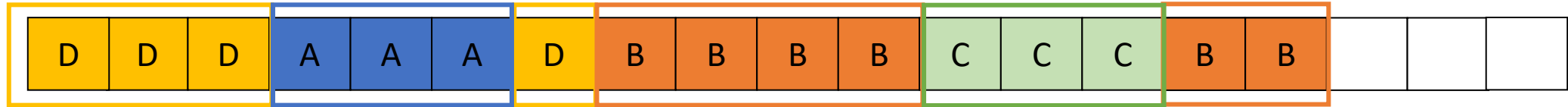
Low 😊

Impractical

Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file

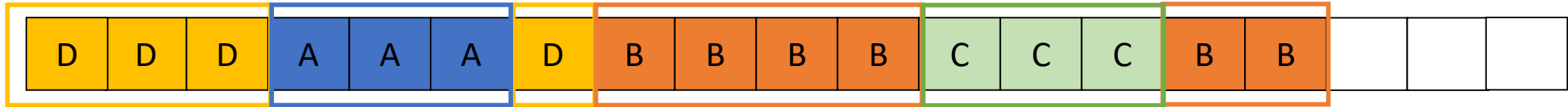
Metadata: Array of extents. Each entry contains extent starting block and size



Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file

Metadata: Array of extents. Each entry contains extent starting block and size



Fragmentation

Growing files

Sequential access

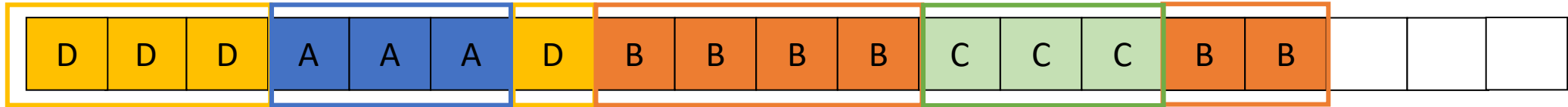
Random access

Metadata overhead

Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file

Metadata: Array of extents. Each entry contains extent starting block and size



Fragmentation

Helps external fragmentation

Growing files

Can grow 😊

Sequential access

Good performance 😊

Random access

Simple calculation 😊

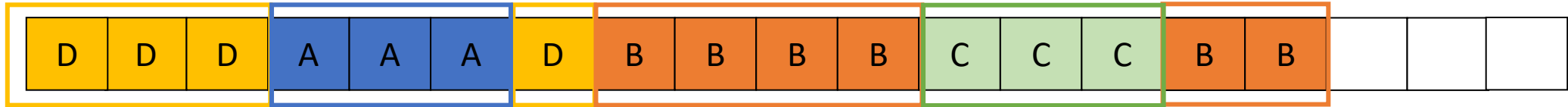
Metadata overhead

Can get large if many extents. 😞

Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file

Metadata: Array of extents. Each entry contains extent starting block and size



Fragmentation

Helps external fragmentation

Growing files

Can grow 😊

Sequential access

Good performance 😊

Random access

Simple calculation 😊

Metadata overhead

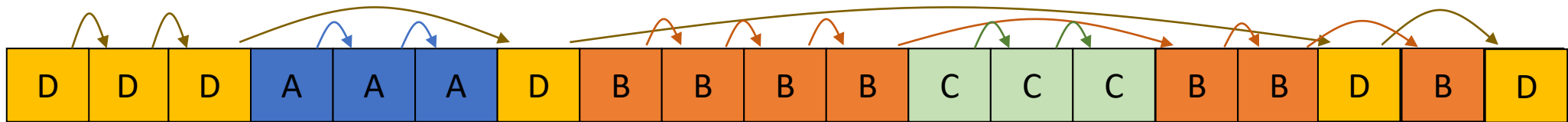
Can get large if many extents. 😞

Common practice in Linux

Linked-List Allocation

Strategy: Allocate linked-list of blocks

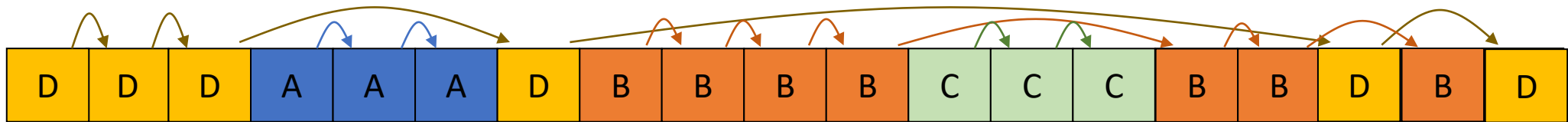
Metadata: Location of first block of file, plus each block contains pointer to next block.



Linked-List Allocation

Strategy: Allocate linked-list of blocks

Metadata: Location of first block of file, plus each block contains pointer to next block.



Fragmentation

Growing files

Sequential access

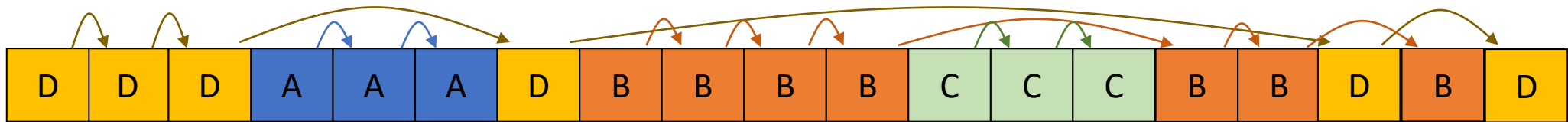
Random access

Metadata overhead

Linked-List Allocation

Strategy: Allocate linked-list of blocks

Metadata: Location of first block of file, plus each block contains pointer to next block.



Fragmentation

No external fragmentation, low internal fragmentation 😊

Growing files

Can grow easily 😊

Sequential access

Depends on data layout

Random access

Slow 😞😞

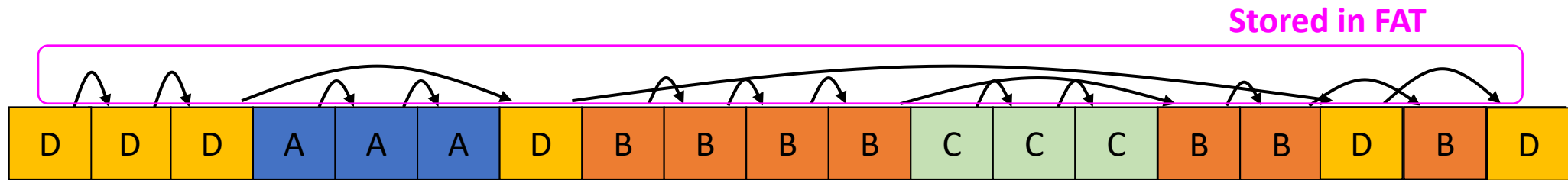
Metadata overhead

Waste space on pointers in data blocks. 😞

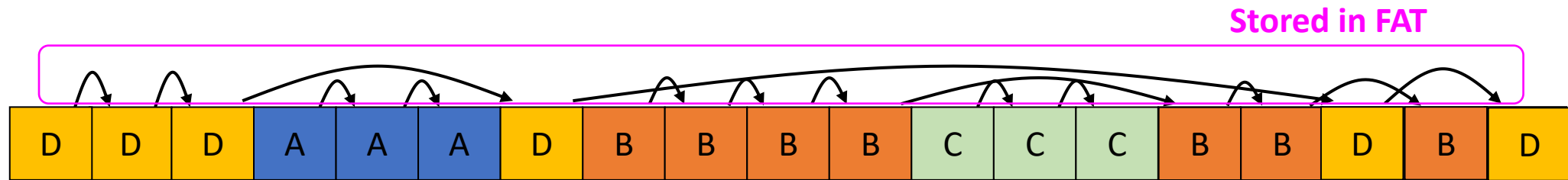
File-Allocation Tables (FAT)

Strategy: Keep links information for all files in a data structure on disk, called the **file allocation table (FAT)**. *Optimization: the FAT can be cached in memory.*

Metadata: Location of first block of file, plus FAT table itself.

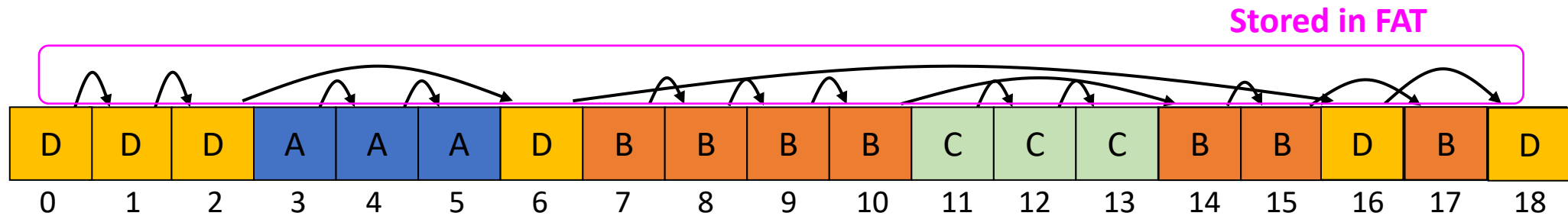


File-Allocation Tables (FAT)



Draw the FAT, assuming a linked-list structure.

File-Allocation Tables (FAT)



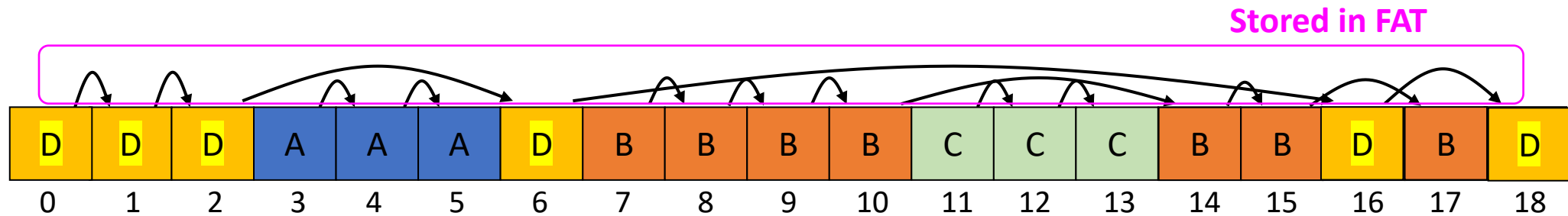
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	
1	
2	
3	
4	
5	
6	

Block index	Next block
7	
8	
9	
10	
11	
12	

Block index	Next block
13	
14	
15	
16	
17	
18	

File-Allocation Tables (FAT)



Draw the FAT, assuming a linked-list structure.

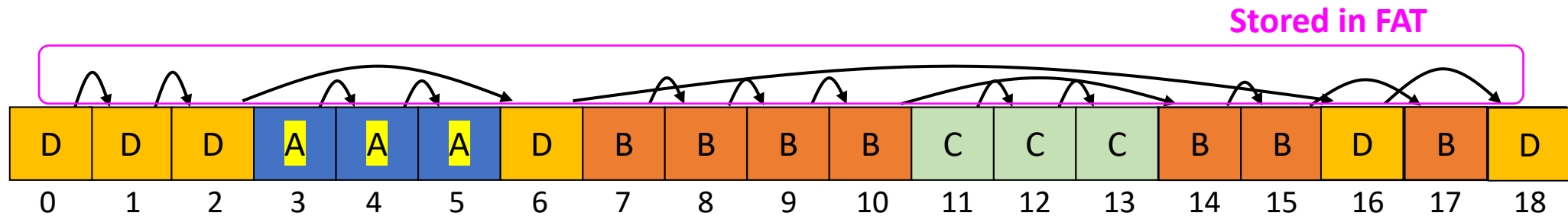
Block index	Next block
0	1
1	2
2	6
3	
4	
5	
6	16

Block index	Next block
7	
8	
9	
10	
11	
12	

Block index	Next block
13	
14	
15	
16	18
17	
18	-1

convention to
signal end of file

File-Allocation Tables (FAT)



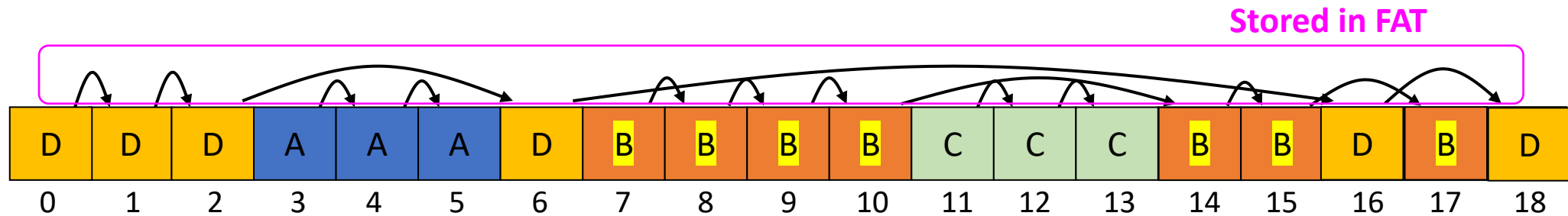
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Block index	Next block
7	
8	
9	
10	
11	
12	

Block index	Next block
13	
14	
15	
16	18
17	
18	-1

File-Allocation Tables (FAT)



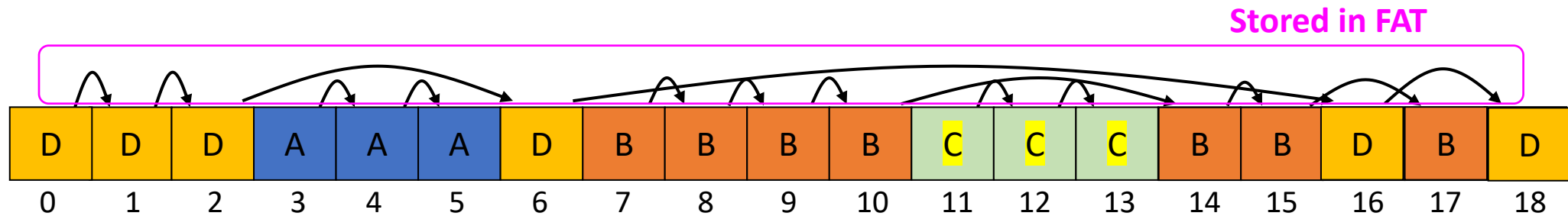
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Block index	Next block
7	8
8	9
9	10
10	14
11	
12	

Block index	Next block
13	
14	15
15	17
16	18
17	-1
18	-1

File-Allocation Tables (FAT)



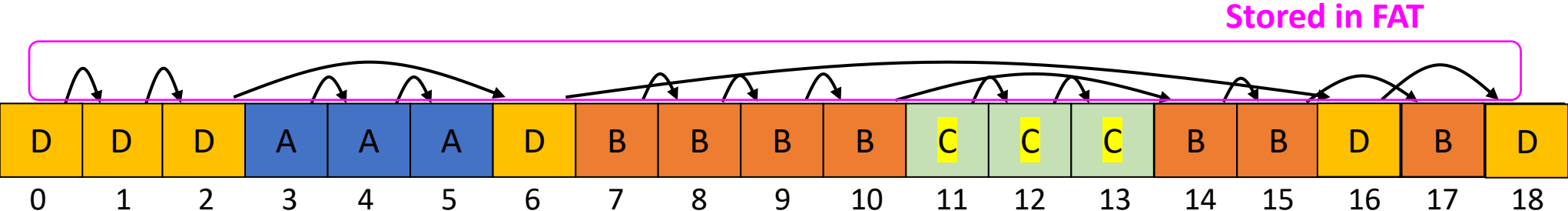
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Block index	Next block
7	8
8	9
9	10
10	14
11	12
12	13

Block index	Next block
13	-1
14	15
15	17
16	18
17	-1
18	-1

File-Allocation Tables (FAT)



Draw the FAT, assum

Block index	Next
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Note 1: **Metadata**

- -1 tells us where a file ends.
- How do we know where files start?
 - **Need to remember position of first block for each file**

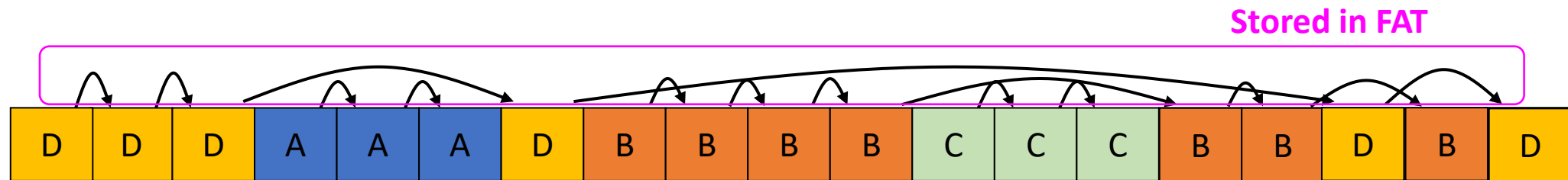
Note 2: **FAT Data Structure**

- This example has a linked list FAT
- FAT implementation can use other data structures too

File-Allocation Tables (FAT)

Strategy: Keep links information for all files in a data structure on disk, called the **file allocation table (FAT)**. *Optimization: the FAT can be cached in memory.*

Metadata: Location of first block of file, plus FAT table itself.



Fragmentation

Same as linked list 😊

Growing files

Same as linked list 😊

Sequential access

Same as linked list

Random access

Significantly better than linked list if FAT cached in memory 😊

Metadata overhead

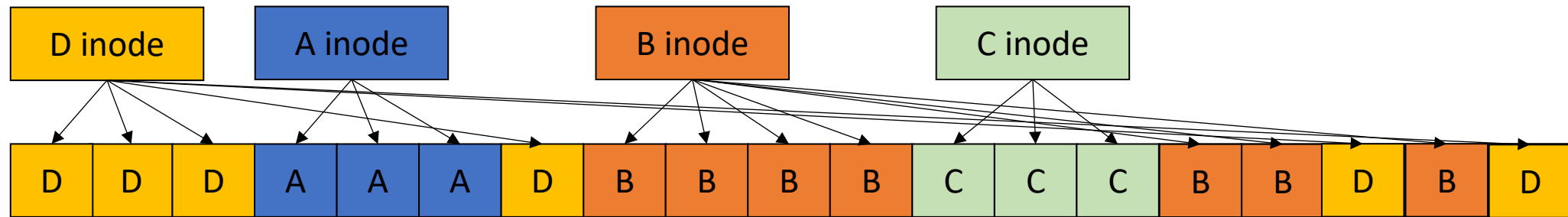
Low for small files. What about large files? 😞

Common practice in SD cards

Indexed Allocation

Strategy: Keep pointers to blocks of file in **an index in the file's inode**. Cap at maximum N pointers.

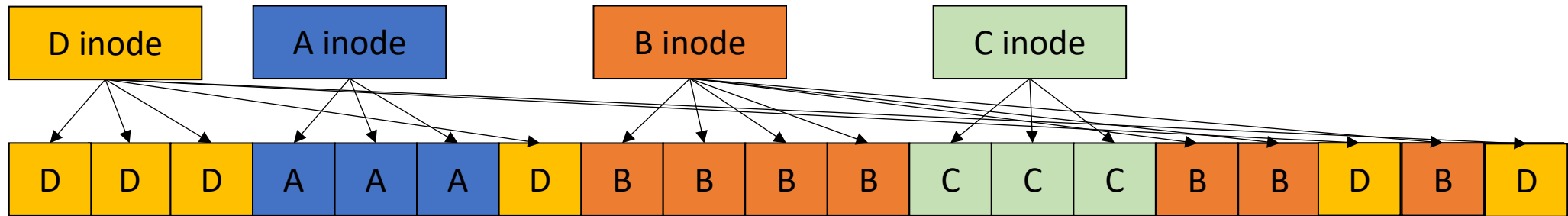
Metadata: Index for each file.



Indexed Allocation

Strategy: Keep pointers to blocks of file in **an index in the file's inode**. Cap at maximum N pointers.

Metadata: Index for each file.



Fragmentation

Growing files

Sequential access

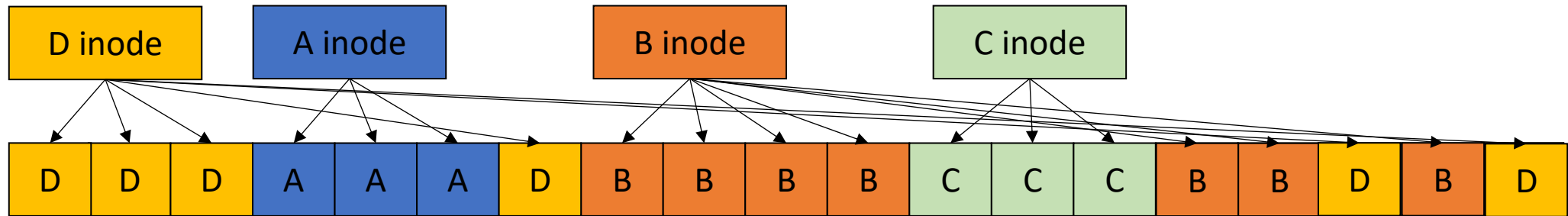
Random access

Metadata overhead

Indexed Allocation

Strategy: Keep pointers to blocks of file in **an index in the file's inode**. Cap at maximum N pointers.

Metadata: Index for each file.



Fragmentation

No external fragmentation, low internal fragmentation 😊

Growing files

Can grow easily 😊

Sequential access

Efficient 😊

Random access

Efficient 😊

Metadata overhead

Low for small files. What if file size > N * size of data block?

Can be combined with
extent allocation

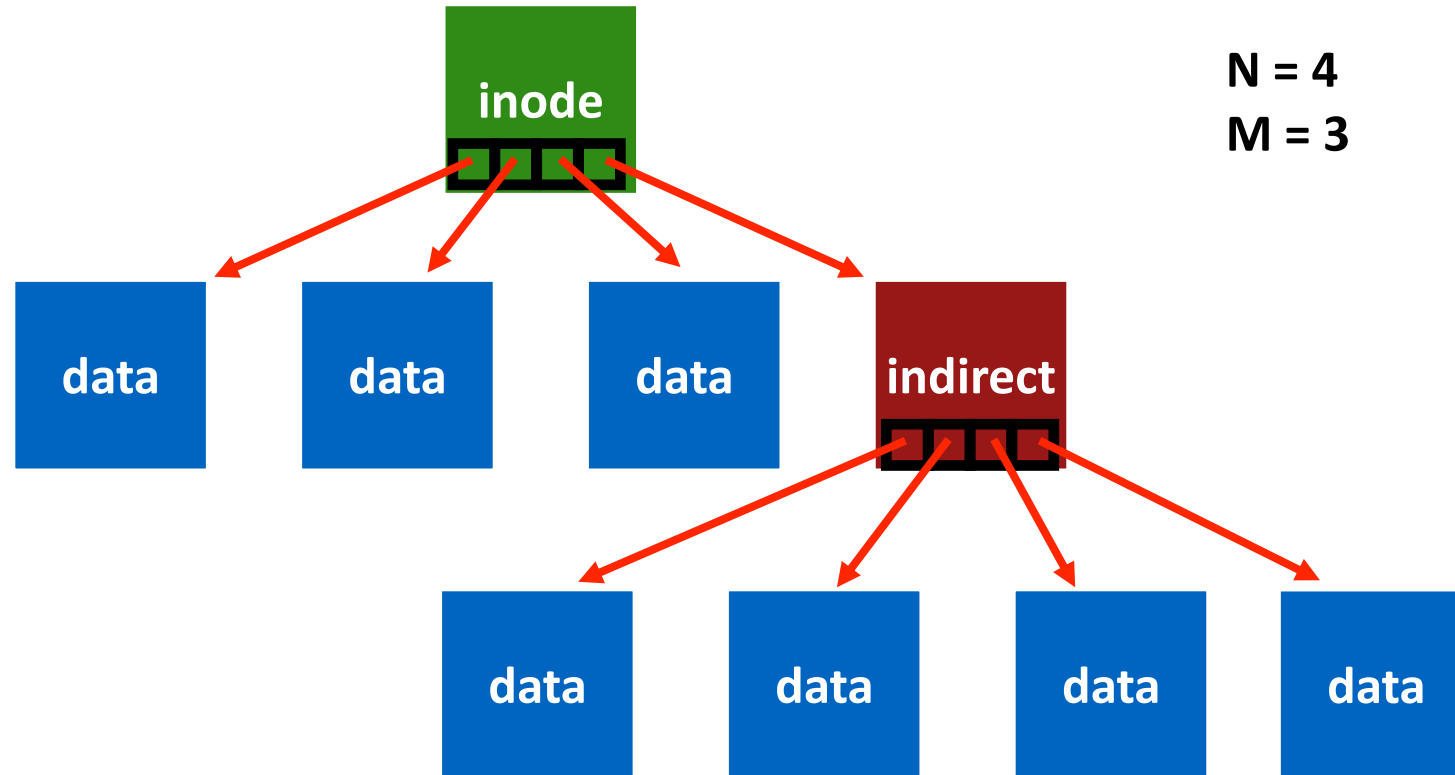
Indexed Allocation with Indirect Blocks

- N pointers in inode block
- First M ($< N$) point to first M **data blocks**
- Blocks M+1 to N point to ***indirect blocks***

Indirect blocks

- Do not contain data
 - But pointers to subsequent data blocks
- Double-indirect blocks also possible

Indexed Allocation with Indirect Blocks



Indexed Allocation with Indirect Blocks

- Same advantages as indexed allocation
- plus
- Possible to extend to very large files
 - Can be used with extents
 - (default FS on linux, ext4, does exactly this)

What About Directories?

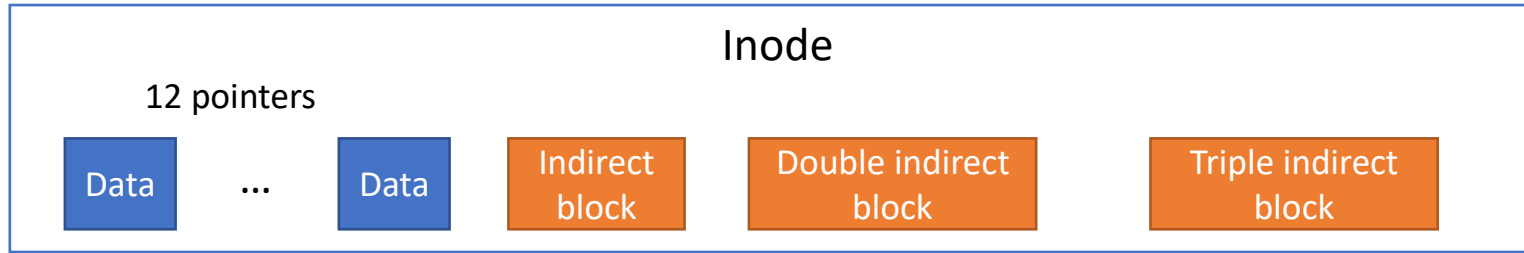
- Directories stored as files

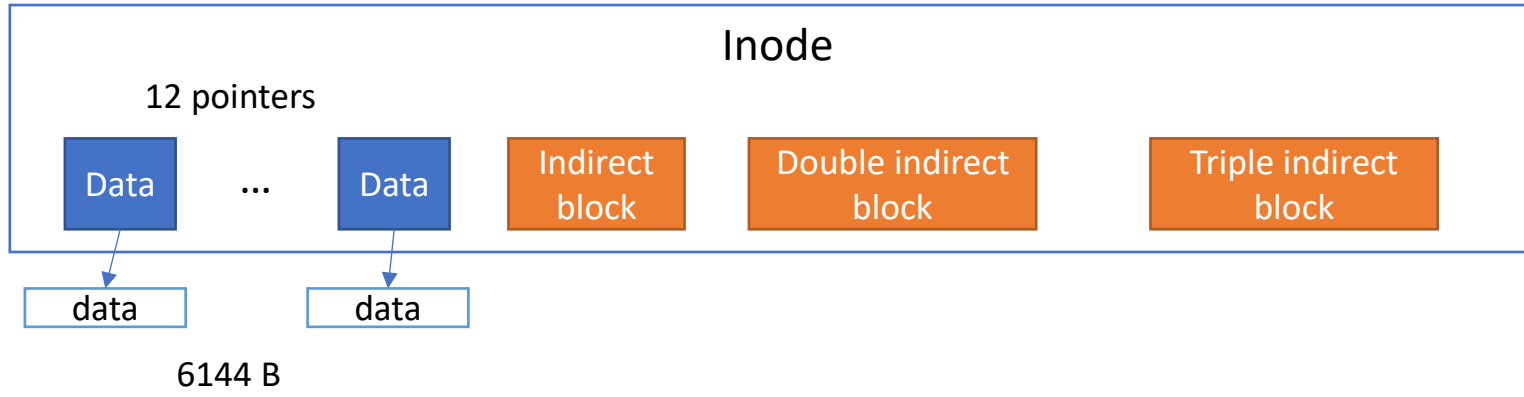
Let's practice!

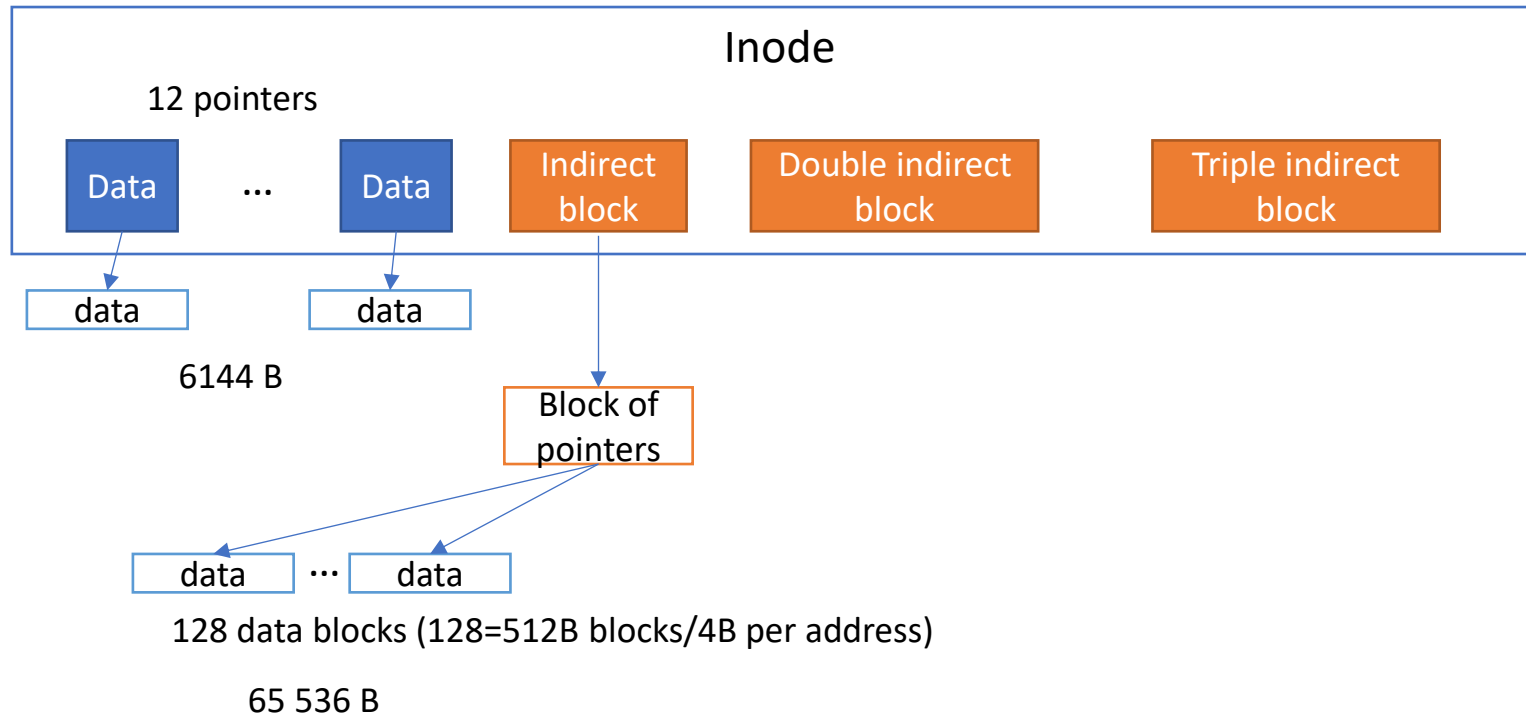
Assume we have a disk with 512-byte disk sectors and 4-byte disk addresses. The file system uses inodes with **12 direct block pointers, 1 indirect block pointer, and 1 double-indirect block pointer, and 1 triple-indirect block pointer**. The block size is identical to the sector size of the disk.

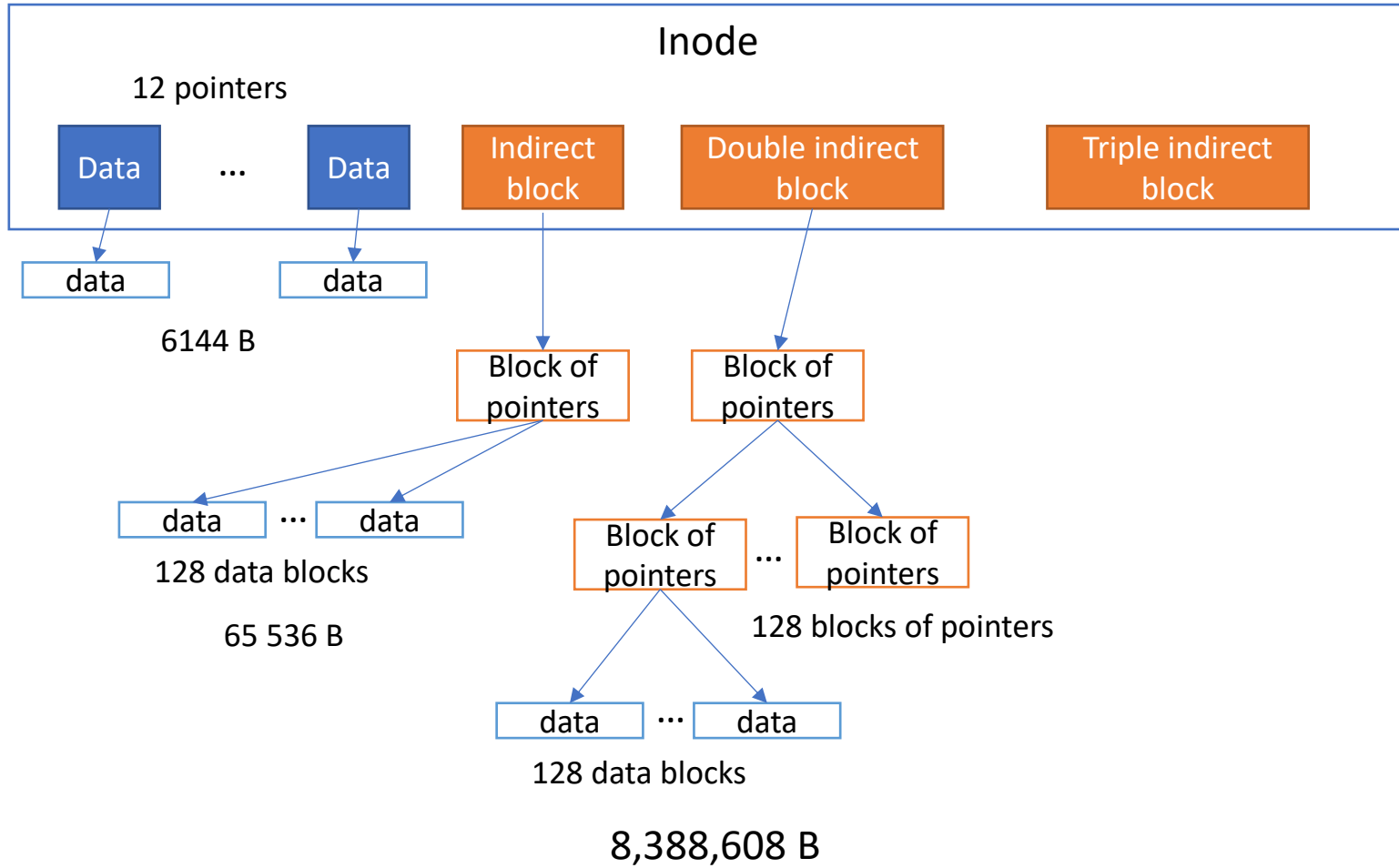
A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses? Explain your answer by **drawing a diagram illustrating which disk accesses occur**.

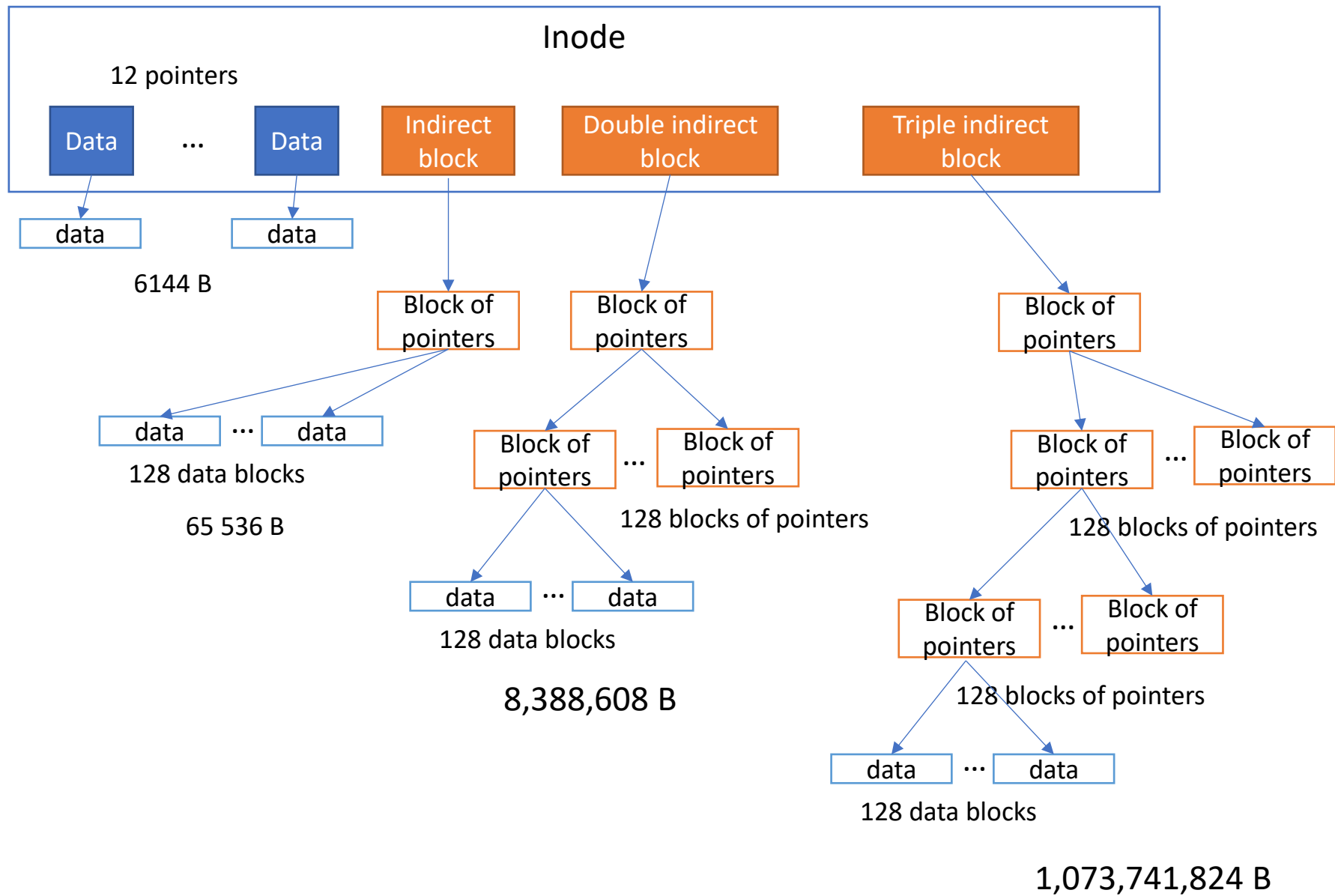
Assume that prior to the open, the inode of the file and the blocks belonging to the file are *not* in the cache, but that there is space in the cache to store all of them. You may also assume that there is no file system activity going on, other than the accesses to this particular file.

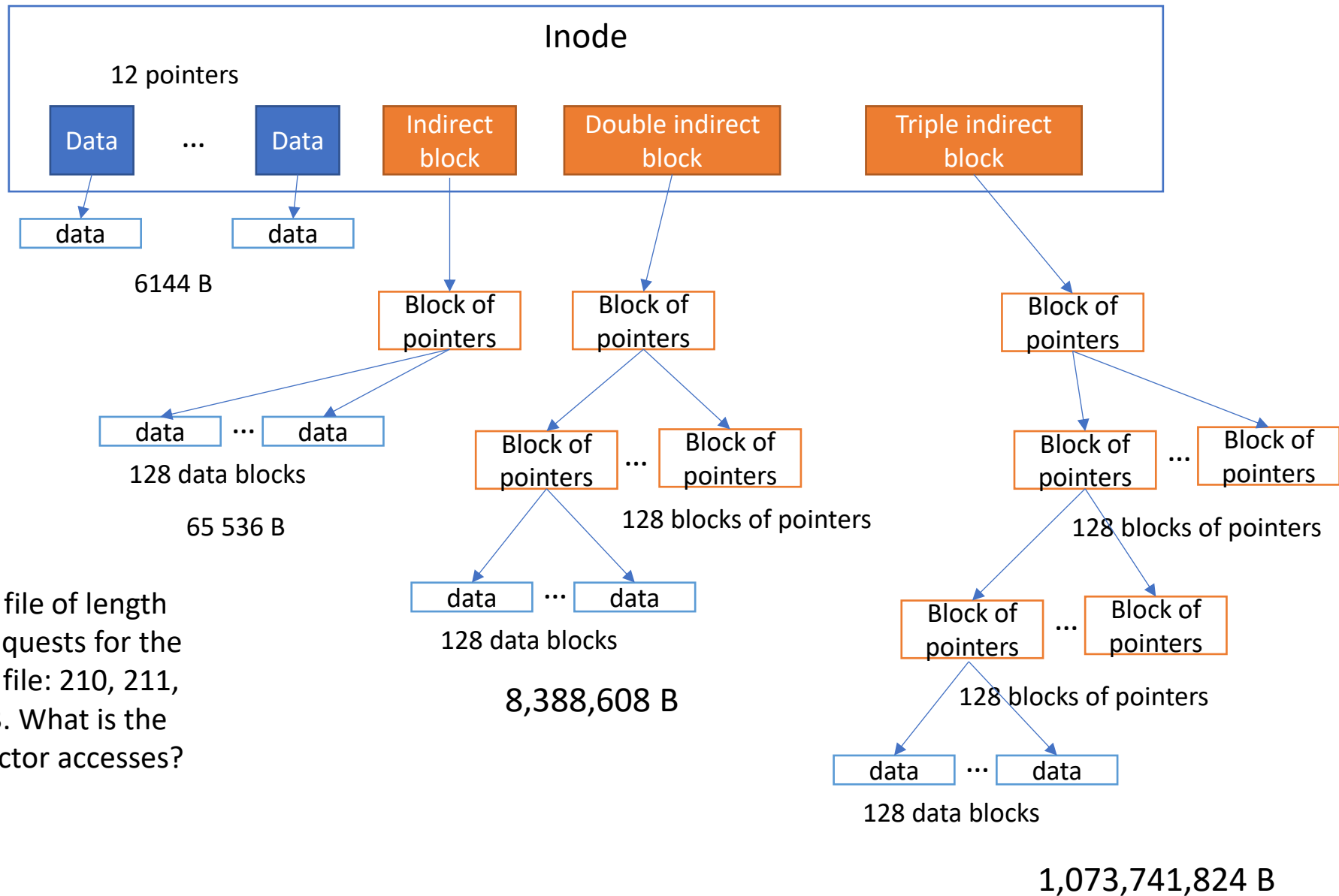




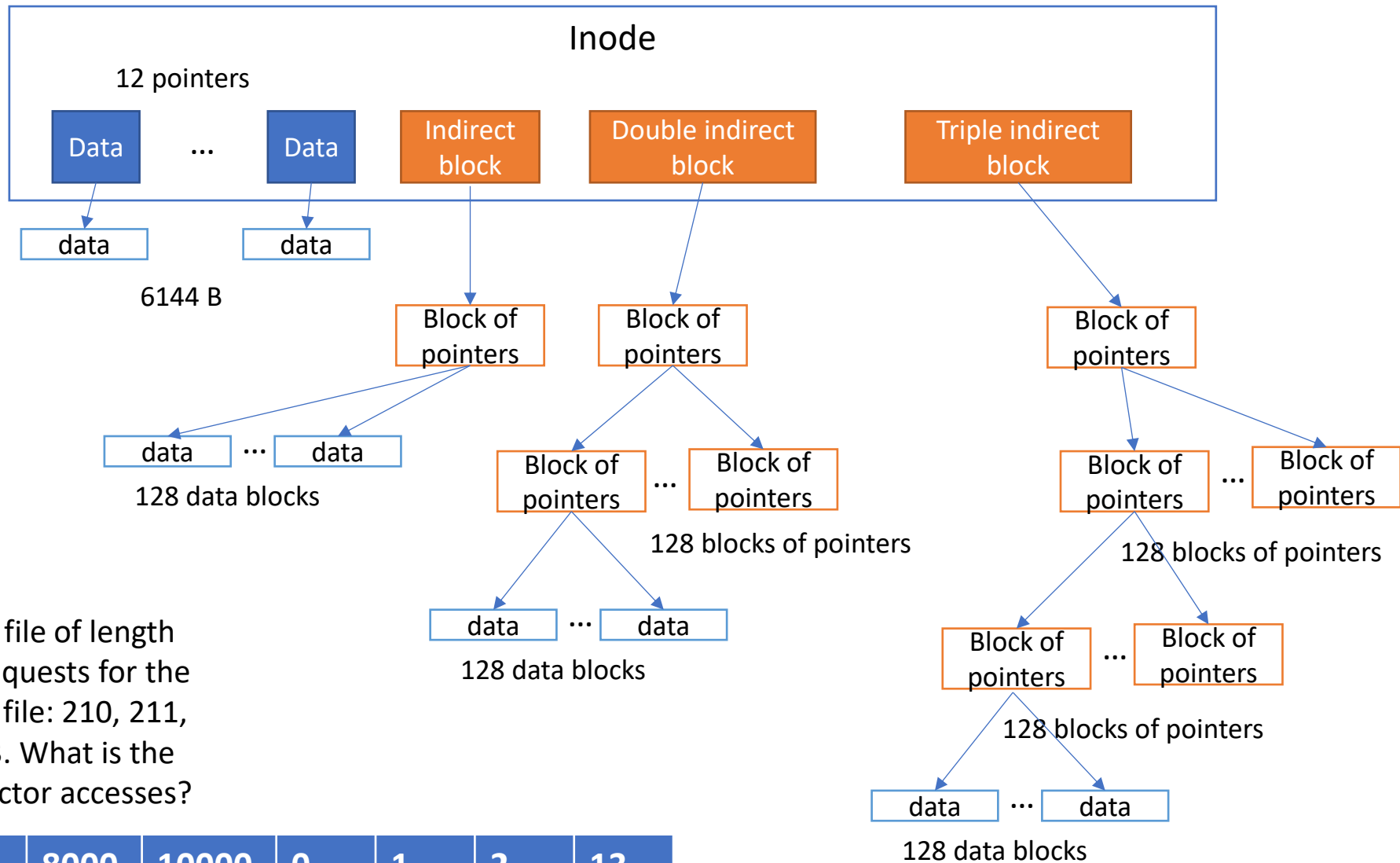






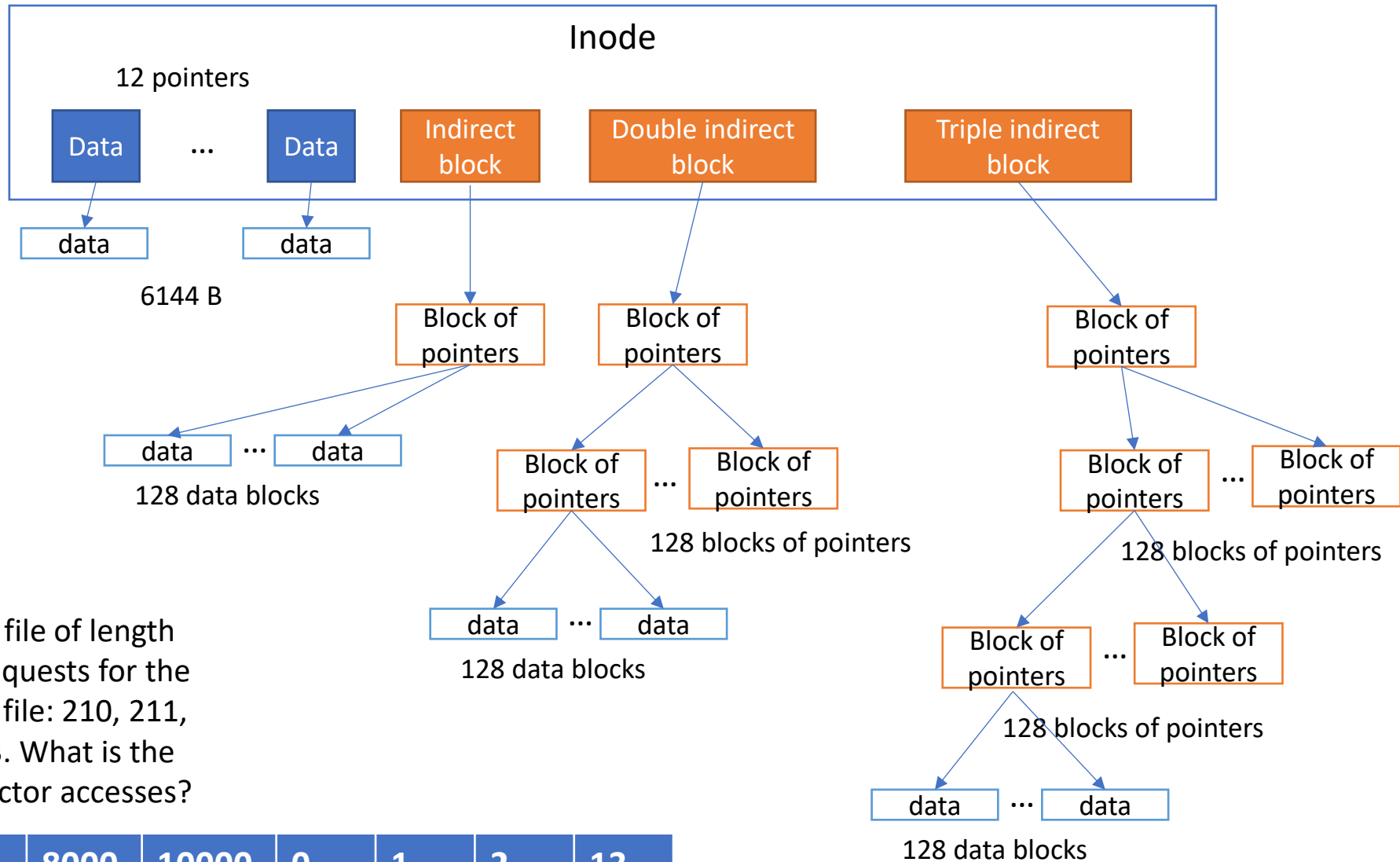


A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses?



A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses?

inode	210	211	8000	10000	0	1	2	13



A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses?

inode	210	211	8000	10000	0	1	2	13
1	3	1	2	2	1	1	1	2

14 disk accesses

inode	210	211	8000	10000	0	1	2	13	
1	3	1	2	2	1	1	1	2	14 disk accesses

Explanations:

- Inode: 1 access because the problem statement assumption is that nothing is cached.
- 210: lives in the double indirect block. Nothing is cached apart from the inode, so we have 3 accesses: 2 for the pointer blocks and 1 for the data block
- 211: lives in the double indirect block, neighboring 210. So, the 2 pointer blocks (and the inode) are cached from the previous accesses. We only need 1 disk access for the data block.
- 8000: lives in the double indirect block. The first pointer block (and inode) are cached from reading 210, and 211. But, the second pointer block and data block are not cached, so we have 2 disk accesses
- 10000: same explanation as for 8000 (note that 8000 and 10000 are more than 128 blocks apart, so the second pointer block is not cached from when we read 8000, as it was the case for reading 211 after 210).
- 0, 1, 2: 1 access because they live in the direct blocks.
- 13: lives in the indirect block. We thus have 2 accesses: one for the pointer block and one for the data block.

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 40, 41, 45.

<https://pages.cs.wisc.edu/~remzi/OSTEP/>

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).

Week 11

Persistent Storage: Basic File System Implementation (Part 2)

Max Kopinsky
20 March, 2025

Remember: Disk Data Structures for File System



Remember: File System Implementation

Key aspects of the system:

1. Data structures

- On disk
- In memory

← disk structures are enough to implement access methods

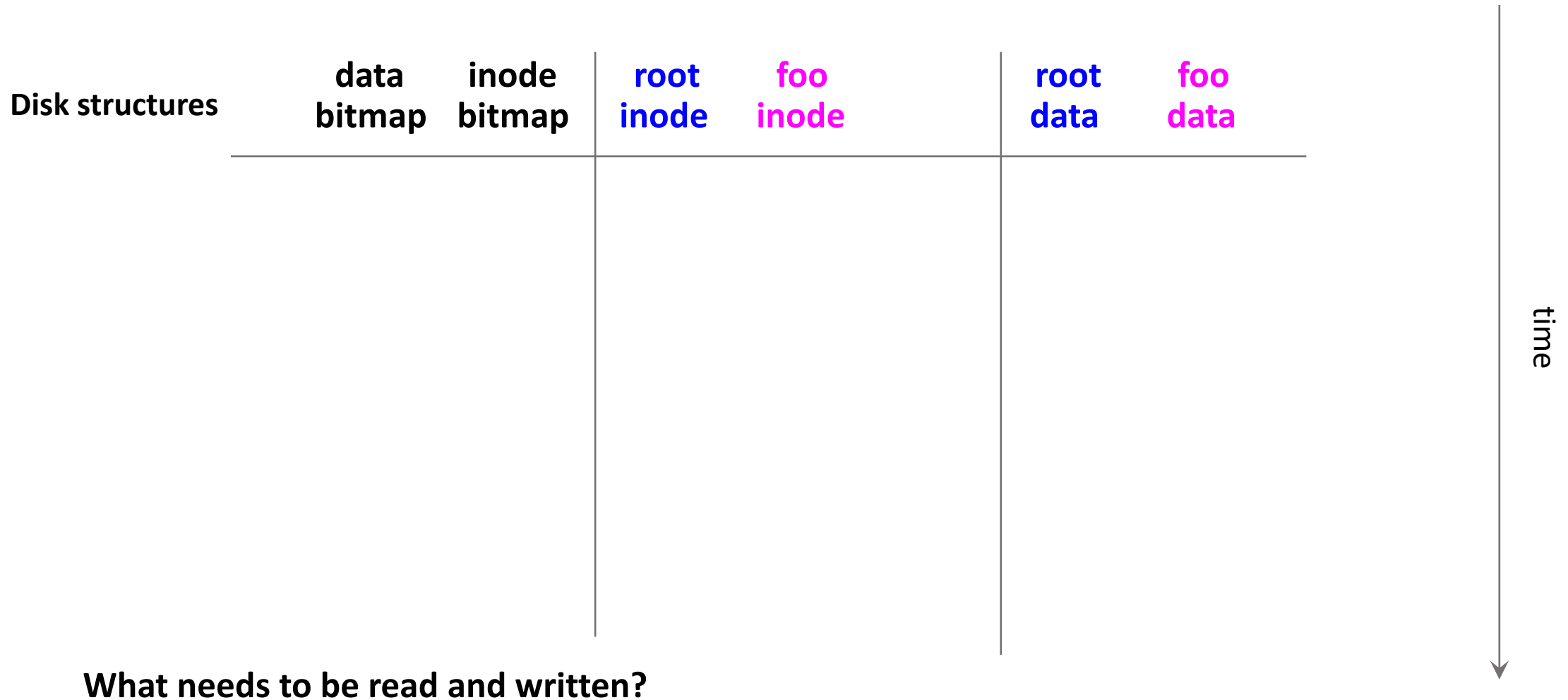
2. Access methods

- How do we open(), read(), write() ?

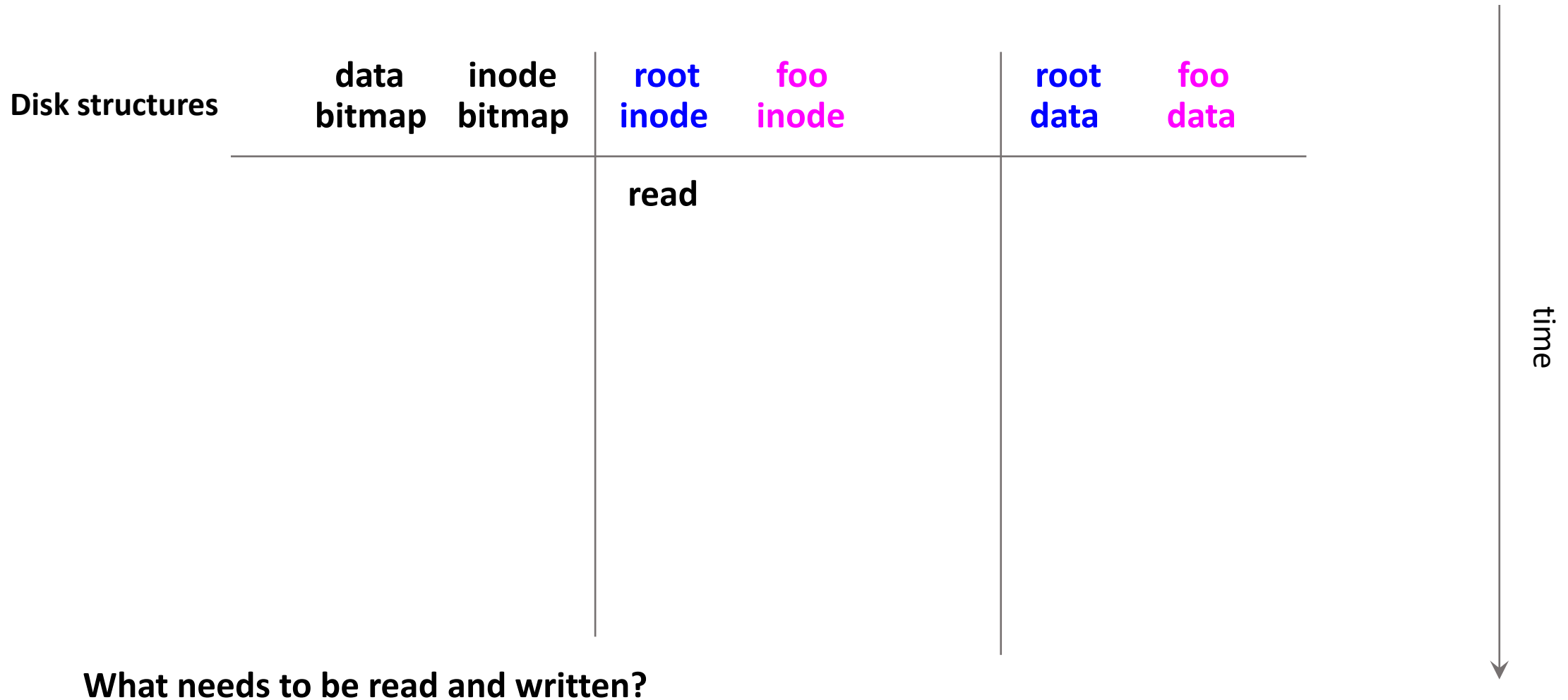
Remember: File Systems Main Access Methods

- Create
 - Open
 - Write
 - Read
 - Close
-
- With some major simplifications
 - No access permission checks, no return value checks, etc.

Create /foo/bar



Create /foo/bar



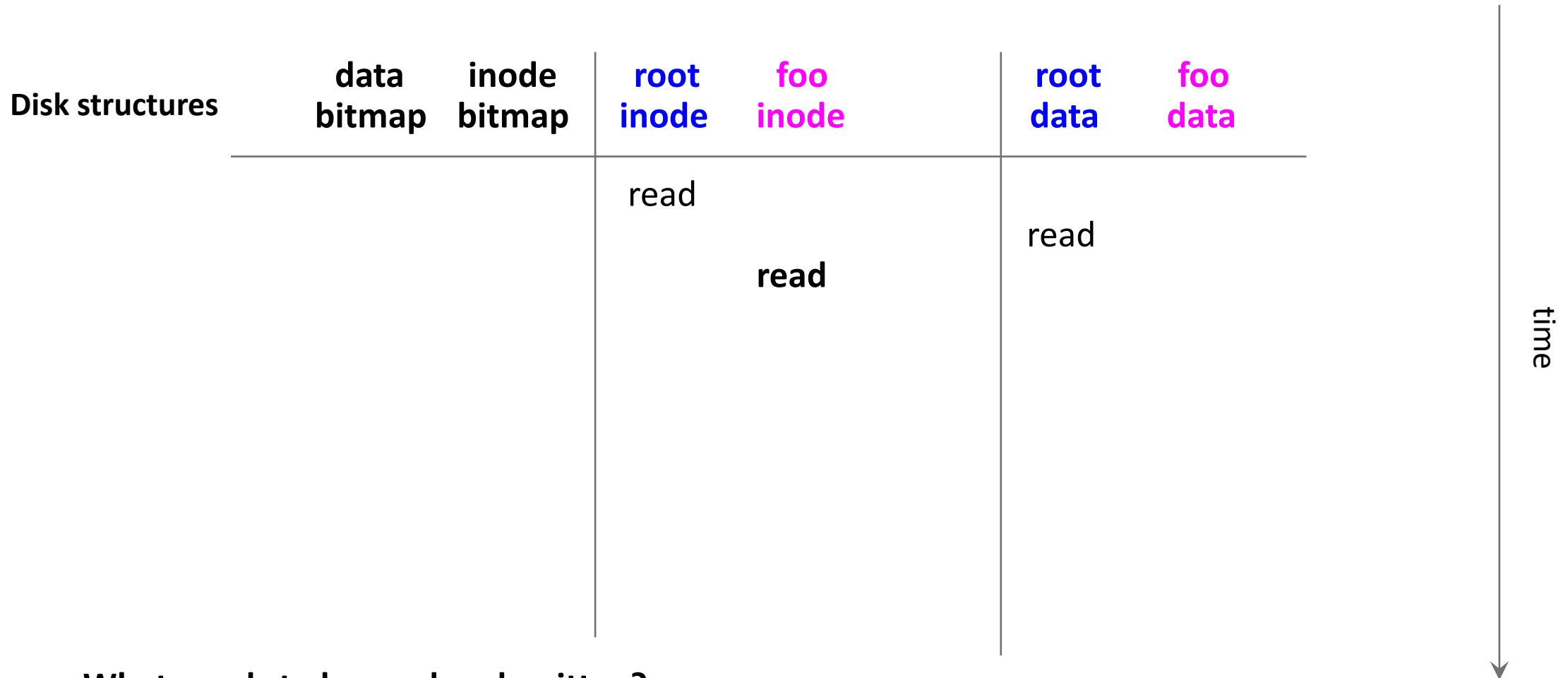
Create /foo/bar

Disk structures	data bitmap	inode bitmap	root inode	foo inode	root data	foo data
			read		read	

time

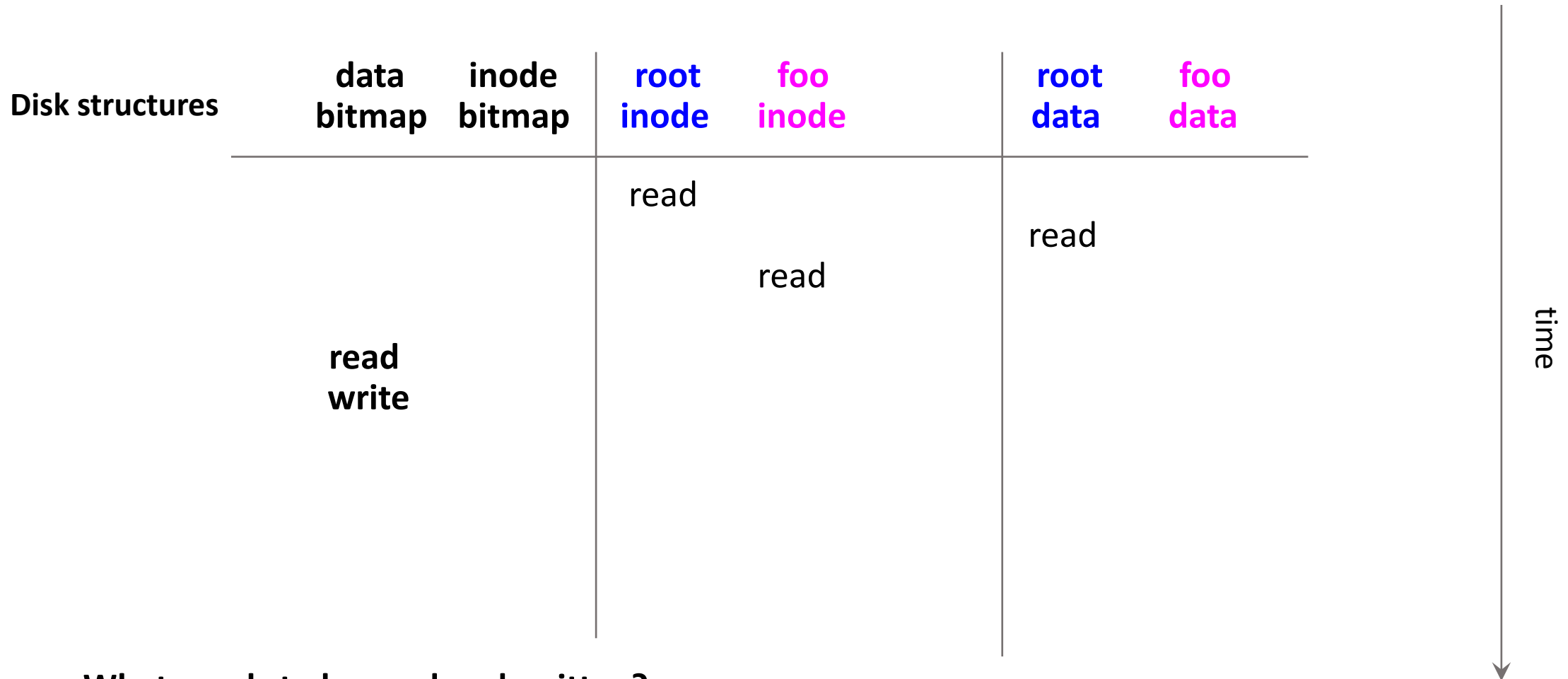
What needs to be read and written?

Create /foo/bar



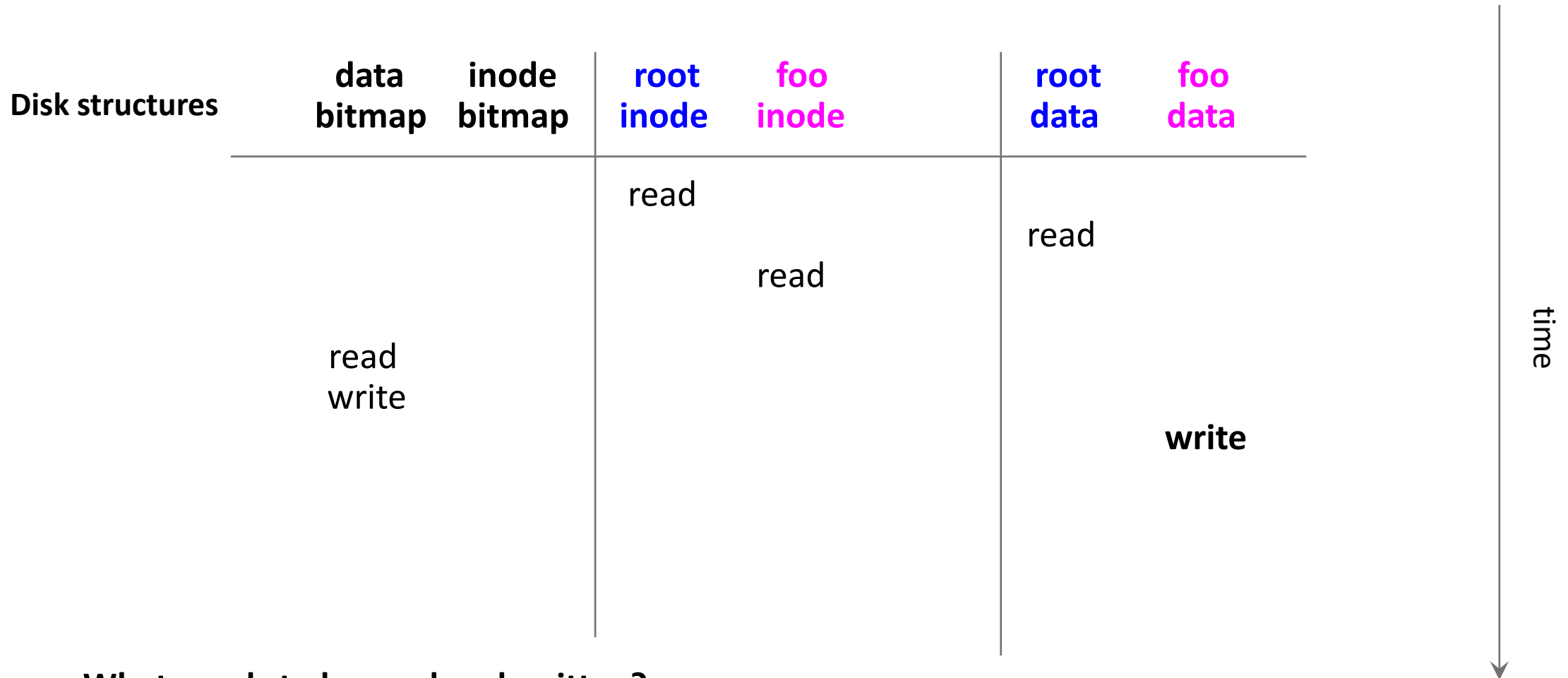
What needs to be read and written?

Create /foo/bar



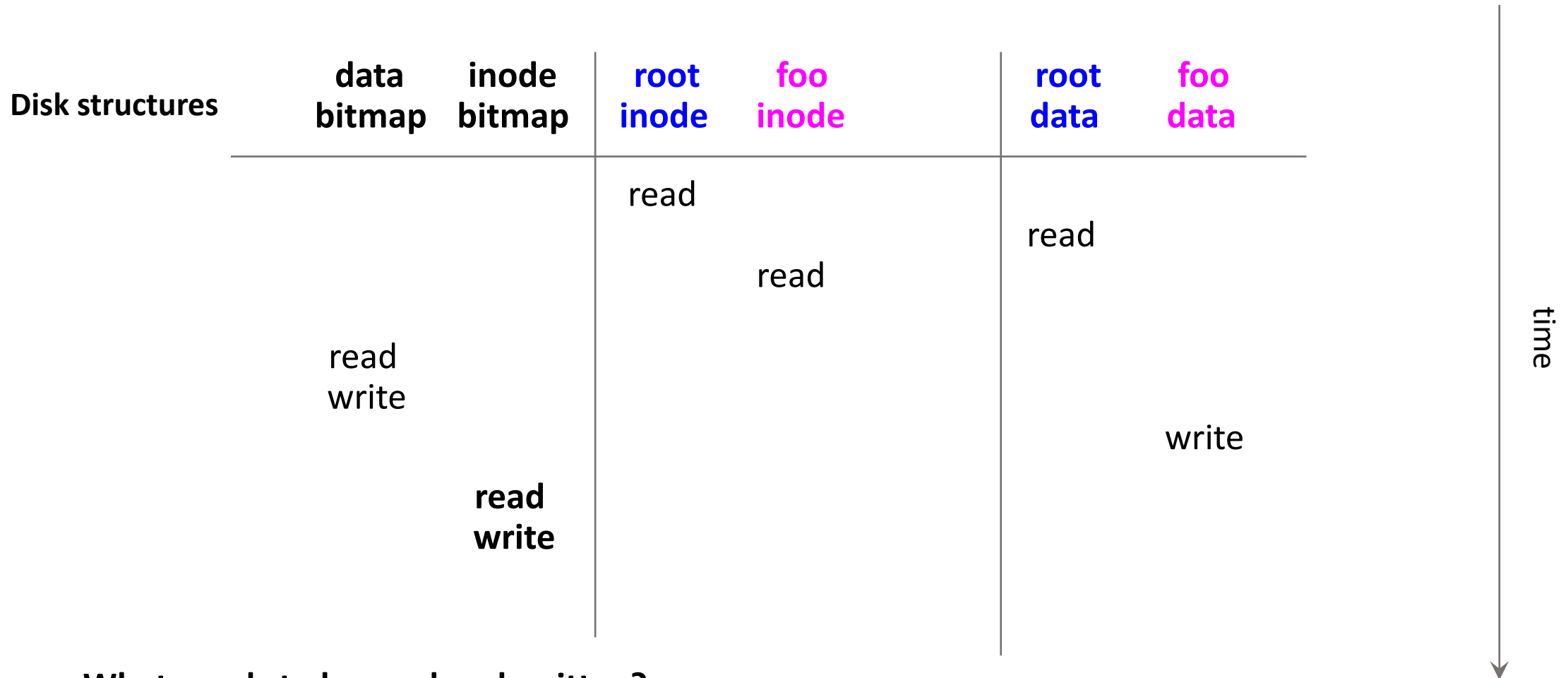
What needs to be read and written?

Create /foo/bar



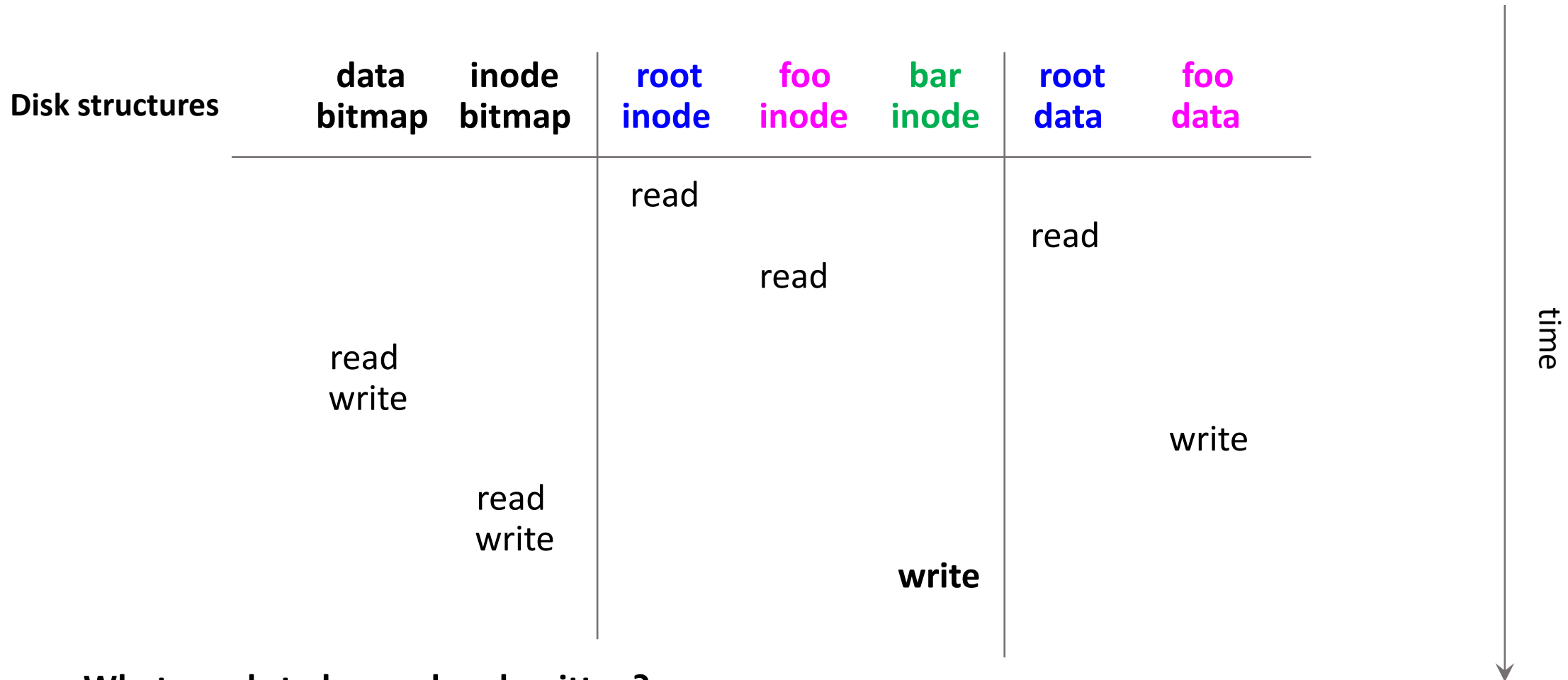
What needs to be read and written?

Create /foo/bar



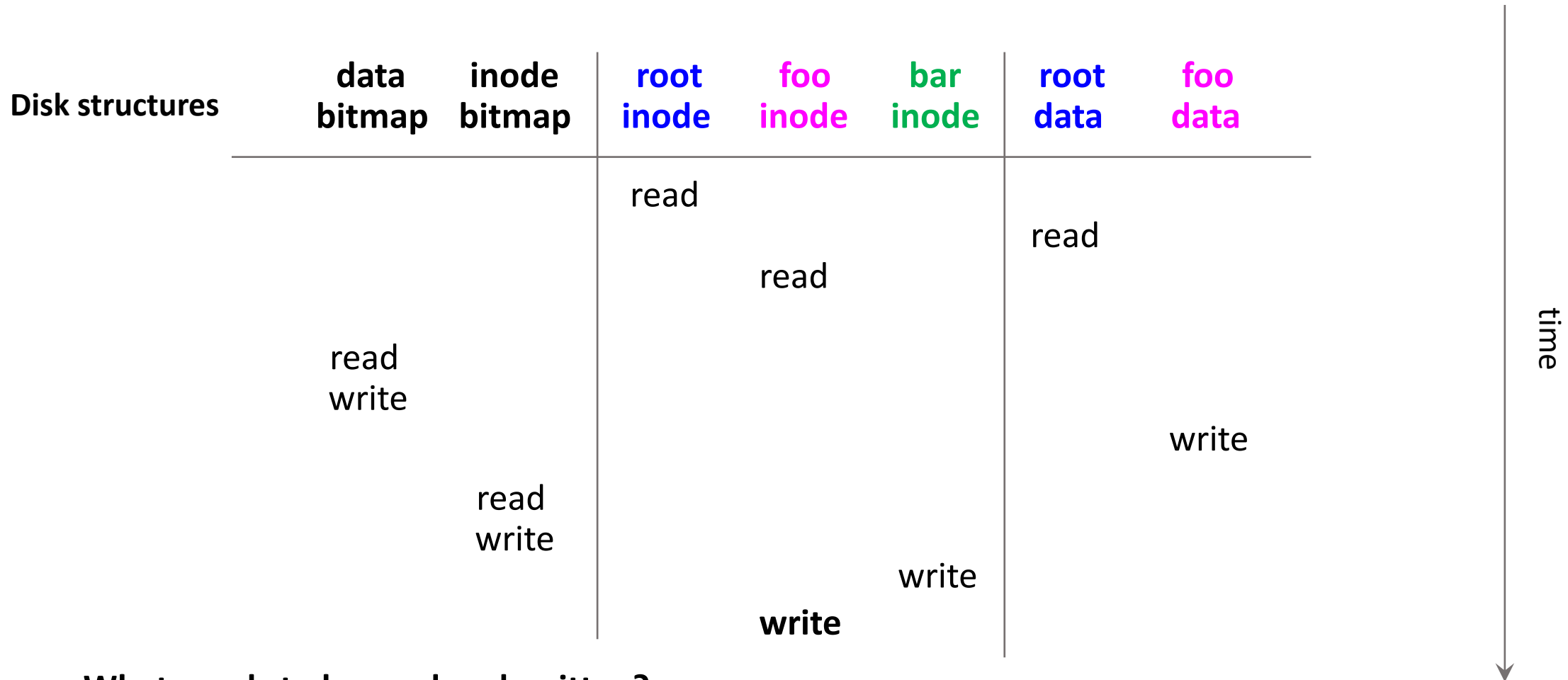
What needs to be read and written?

Create /foo/bar



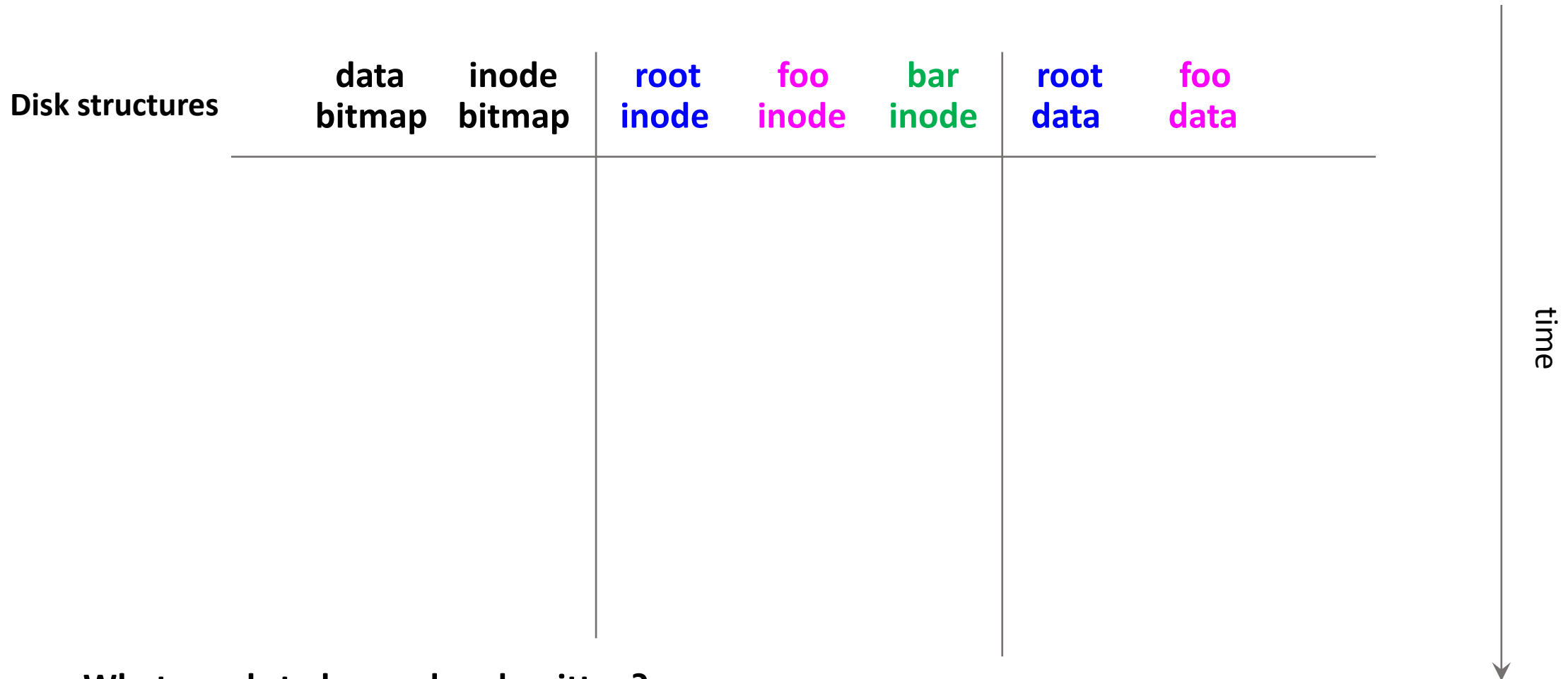
What needs to be read and written?

Create /foo/bar

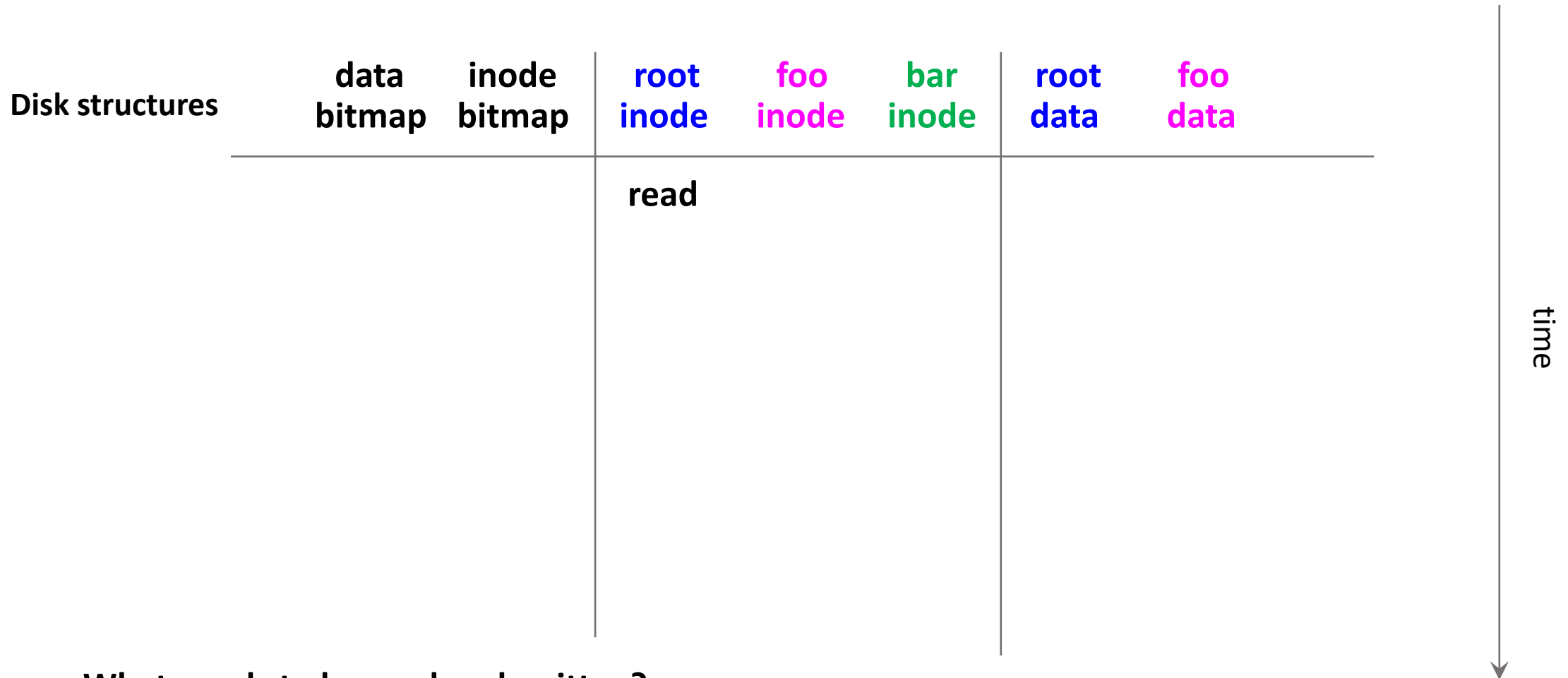


What needs to be read and written?

Open /foo/bar

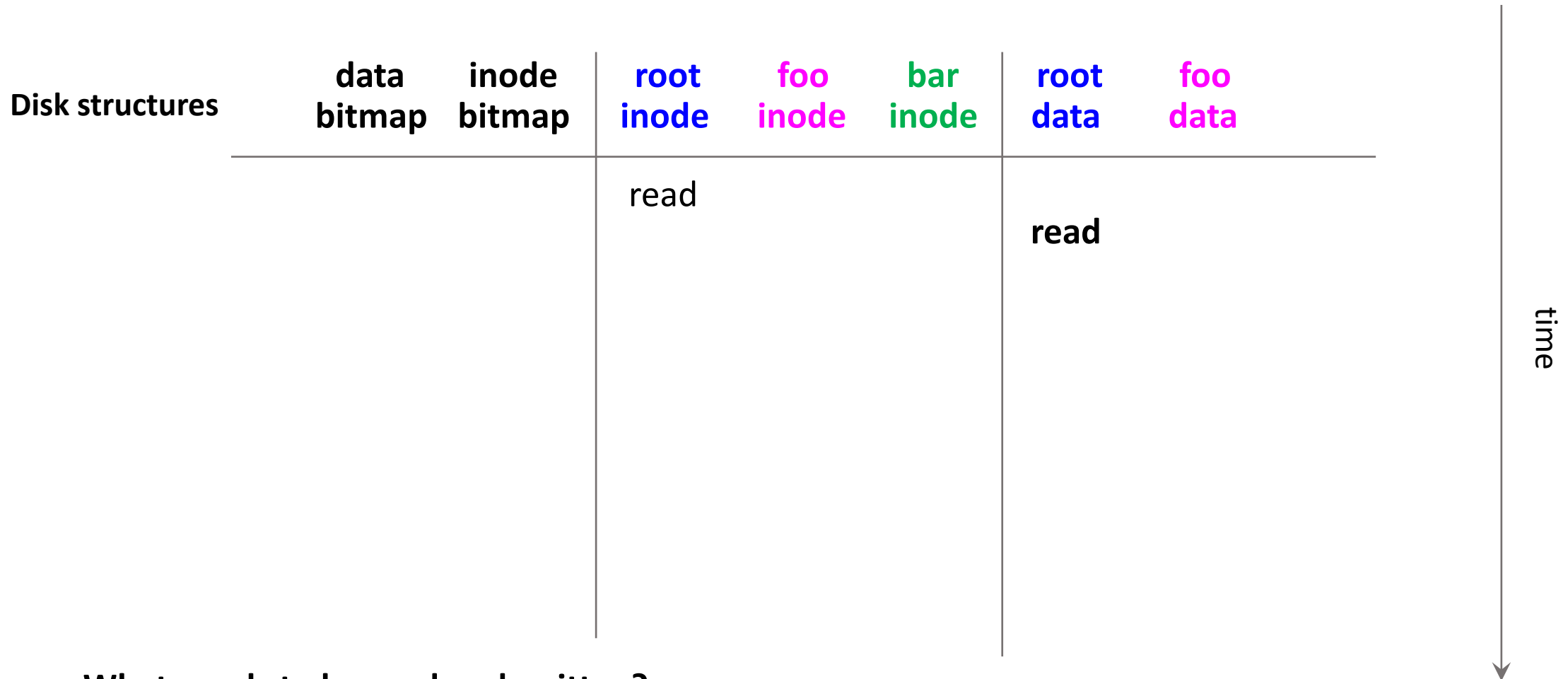


Open /foo/bar



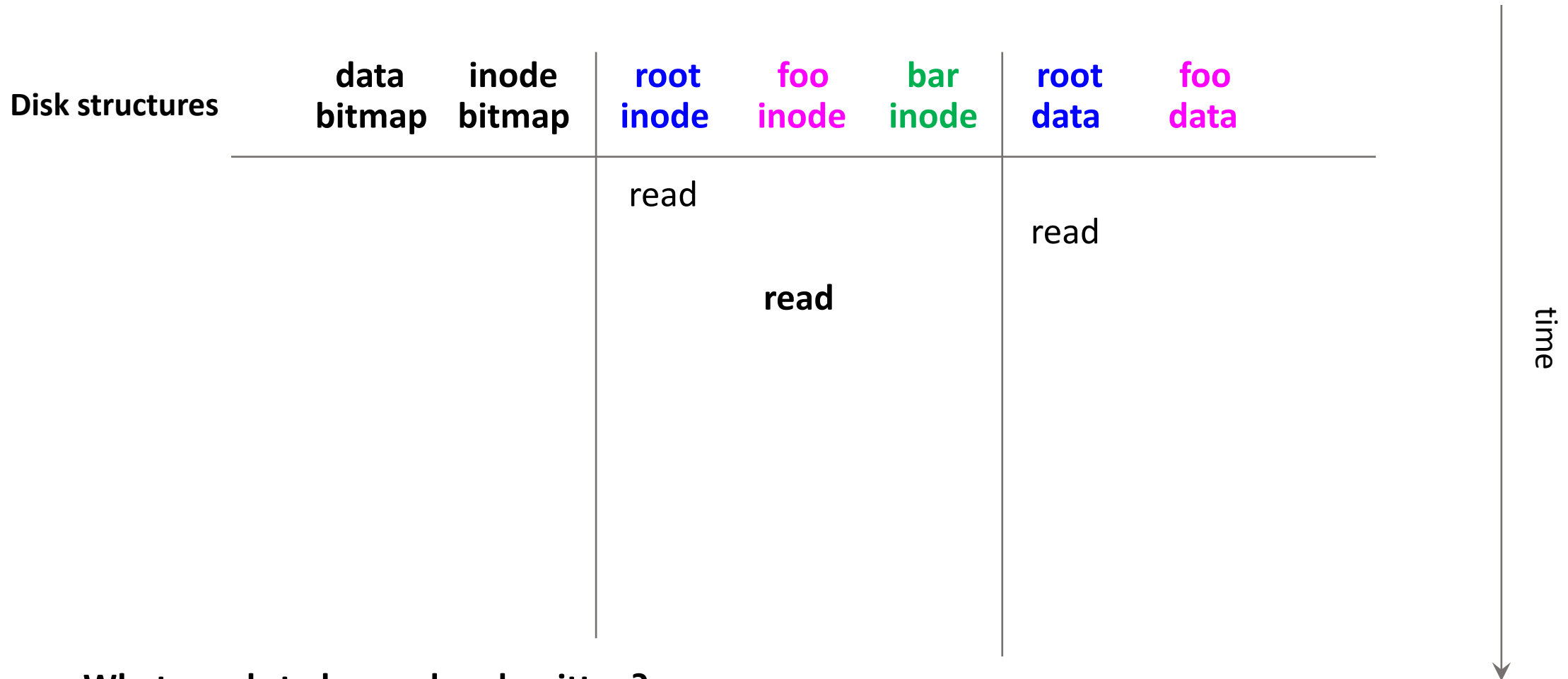
What needs to be read and written?

Open /foo/bar



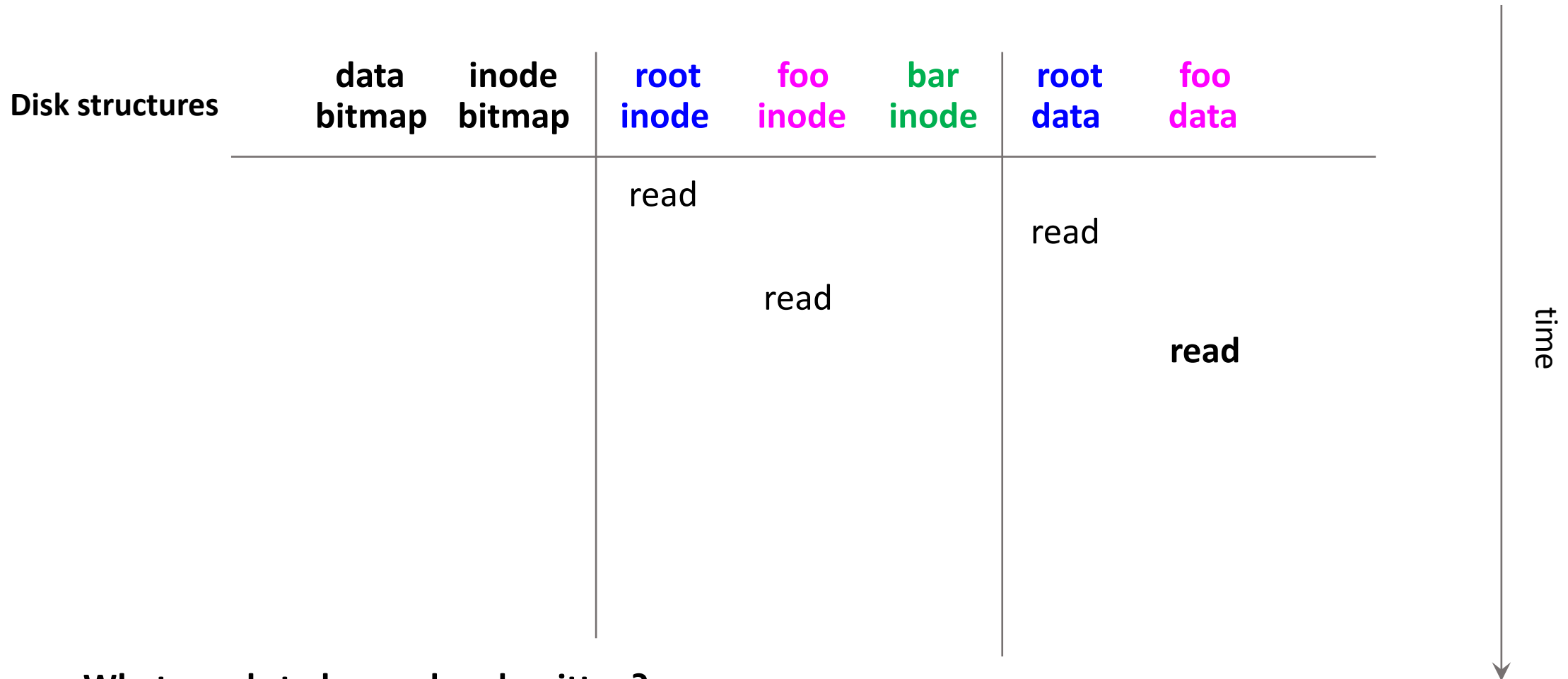
What needs to be read and written?

Open /foo/bar



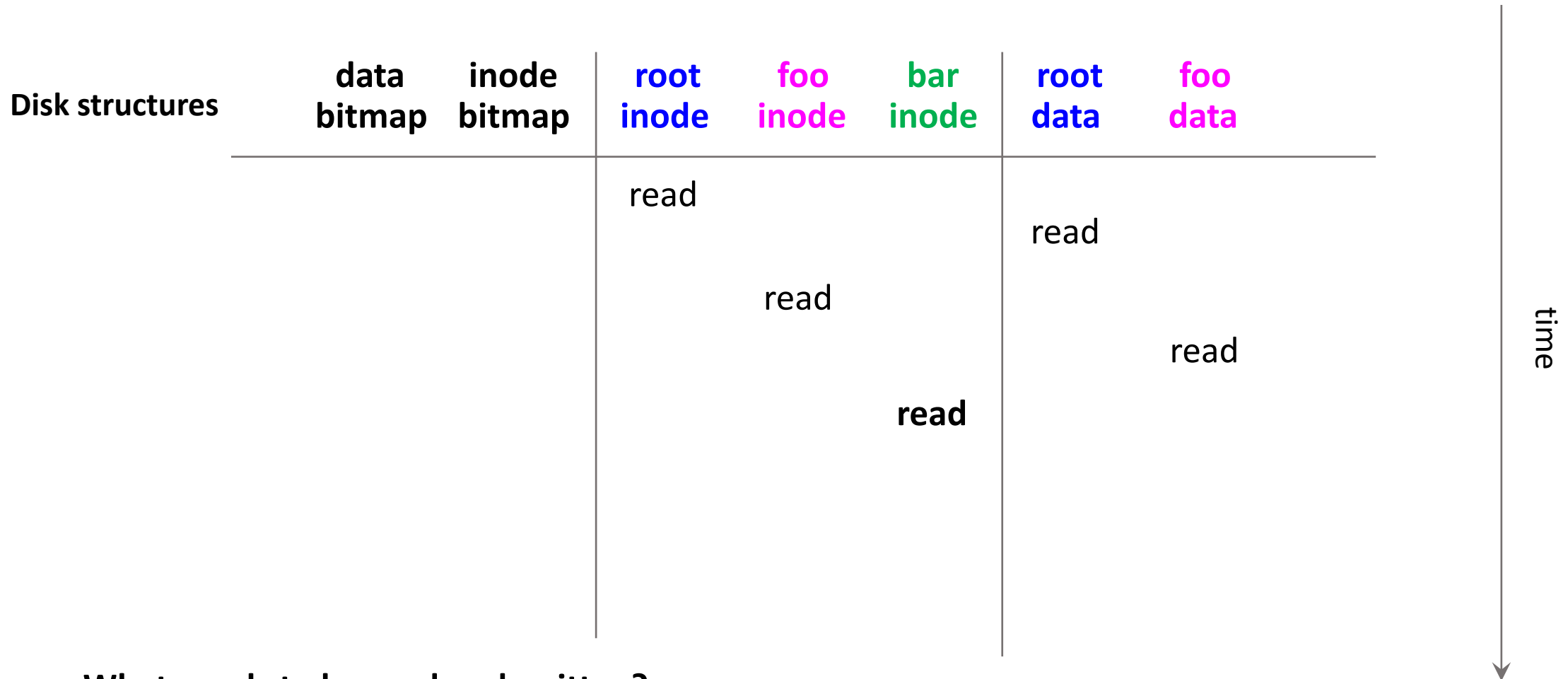
What needs to be read and written?

Open /foo/bar



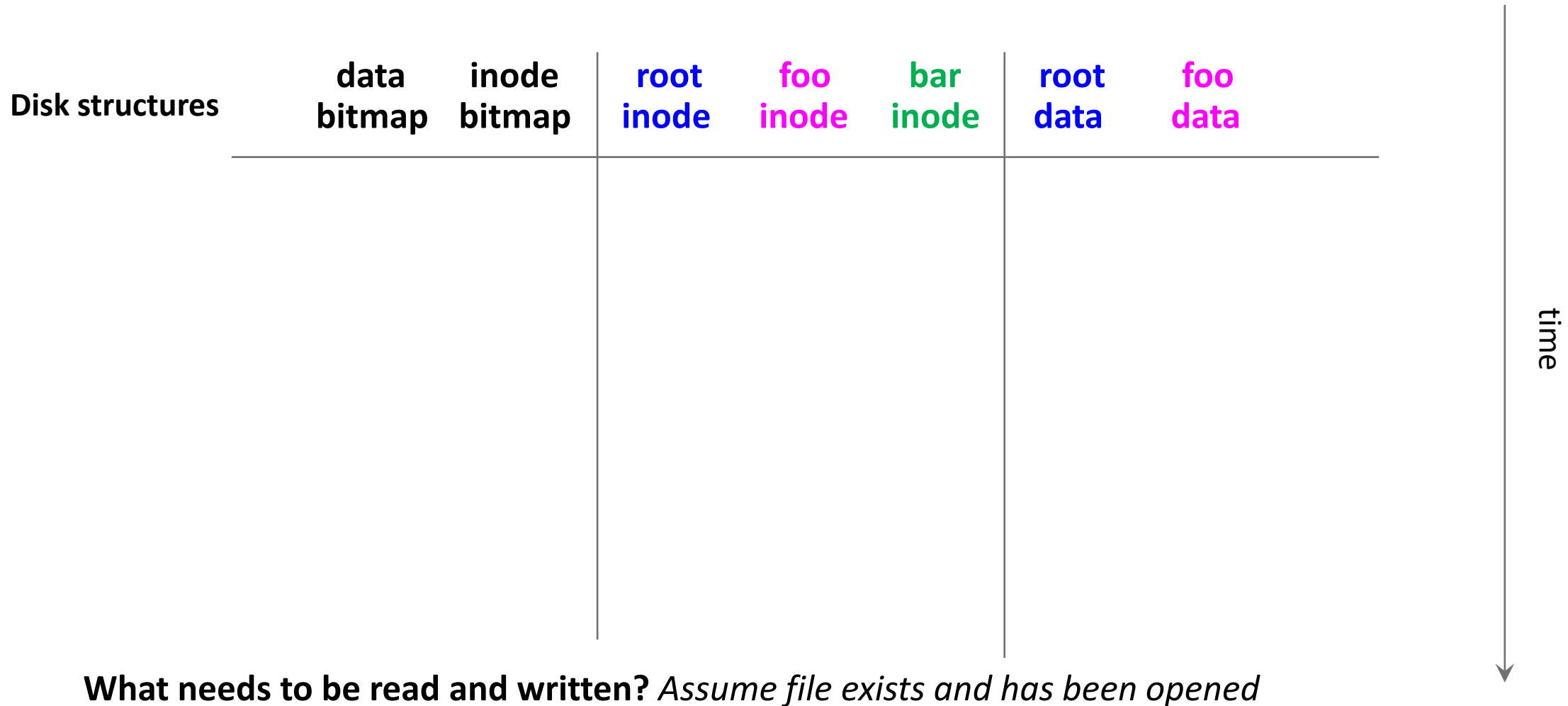
What needs to be read and written?

Open /foo/bar

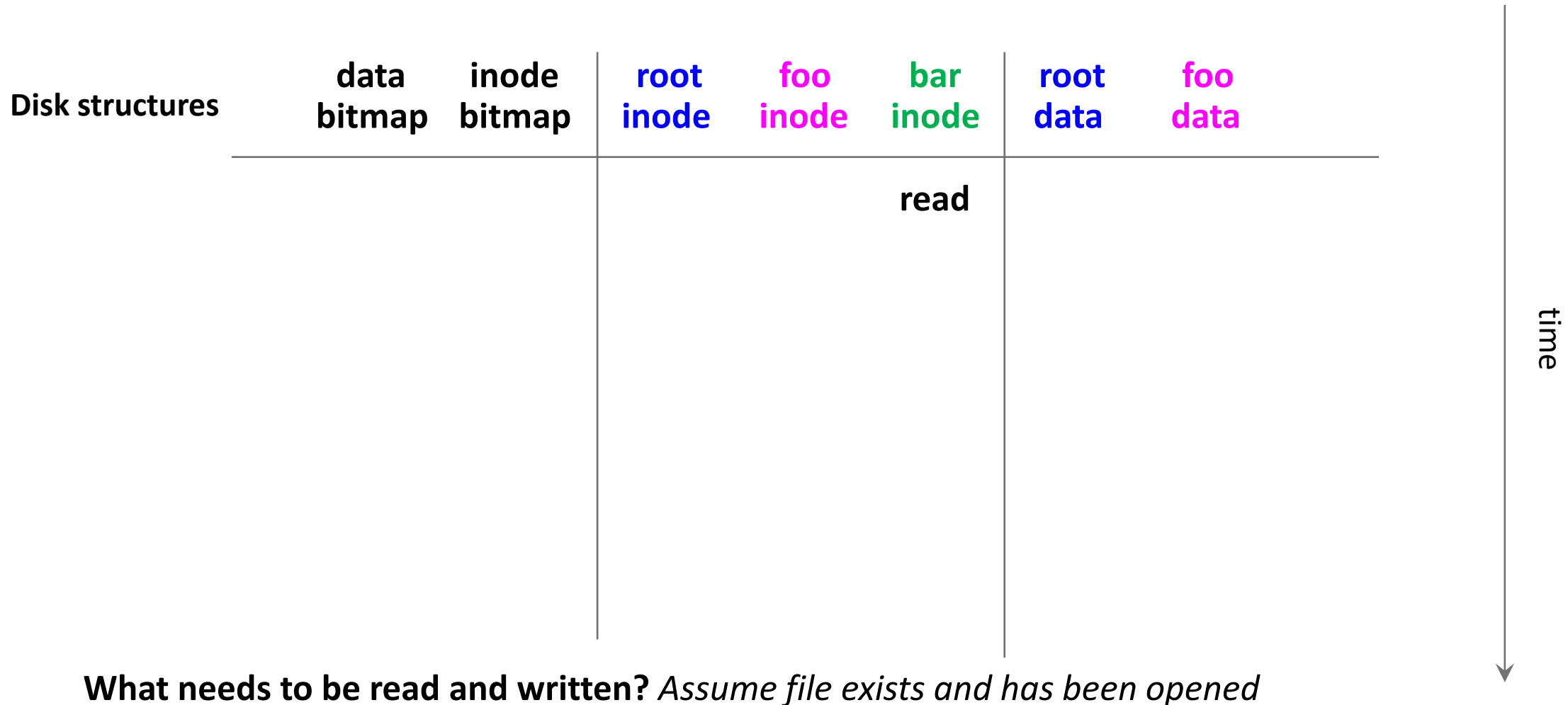


What needs to be read and written?

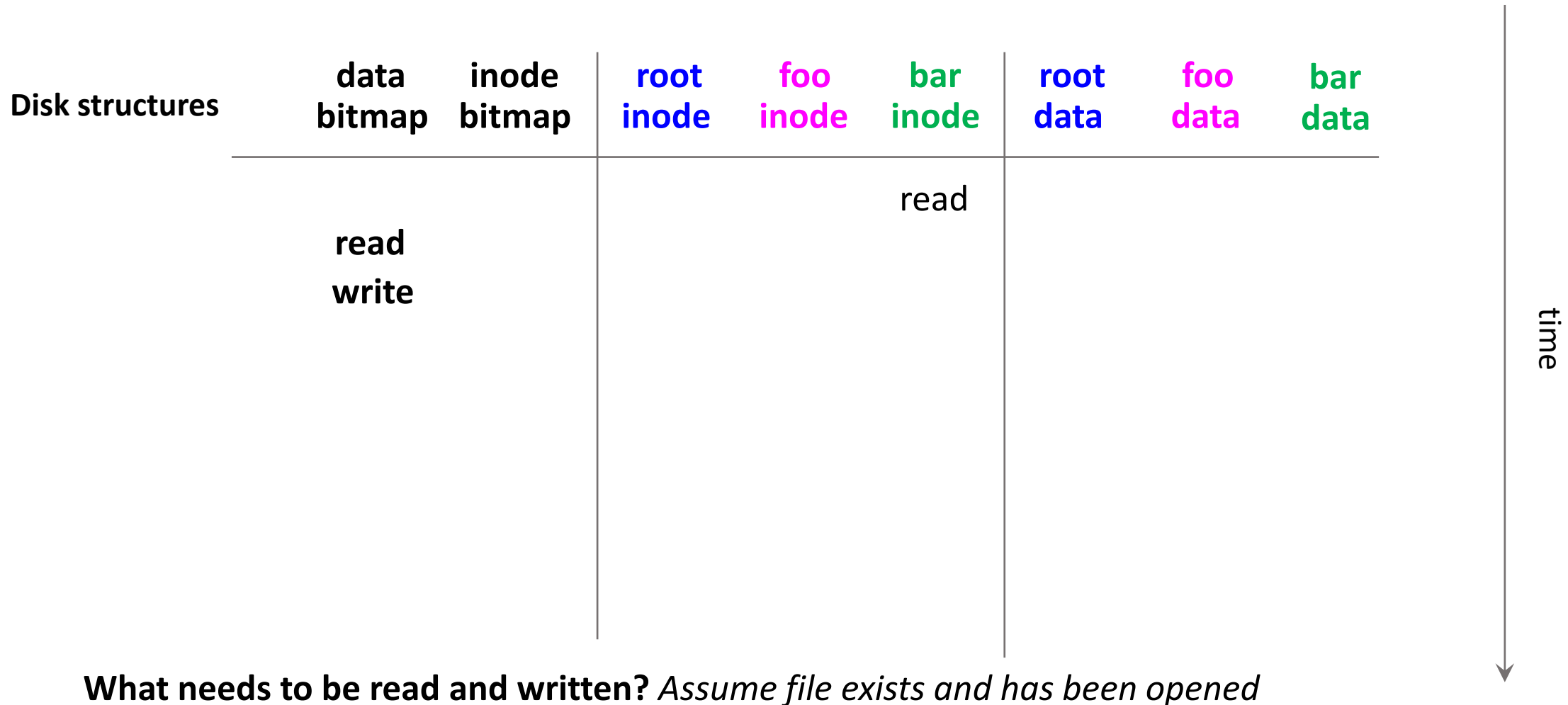
Write to /foo/bar



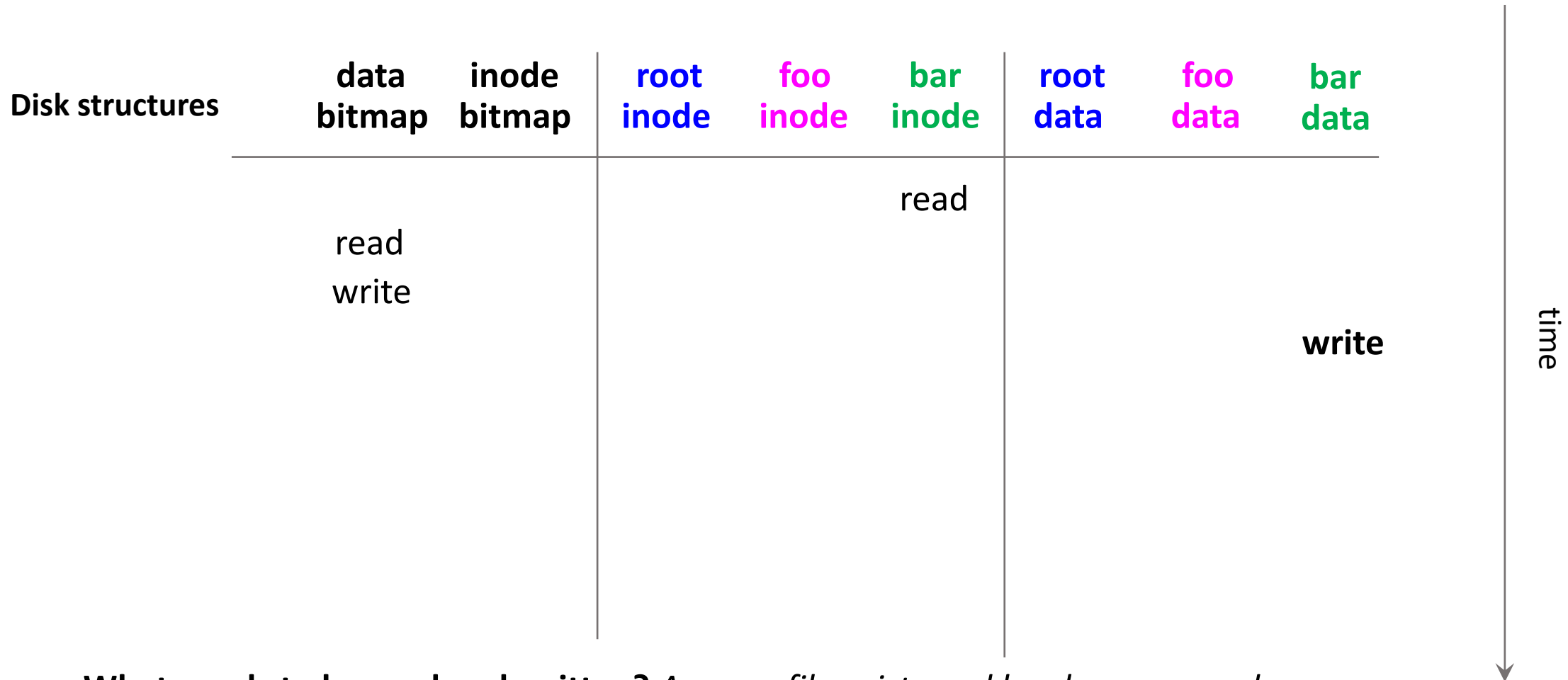
Write to /foo/bar



Write to /foo/bar

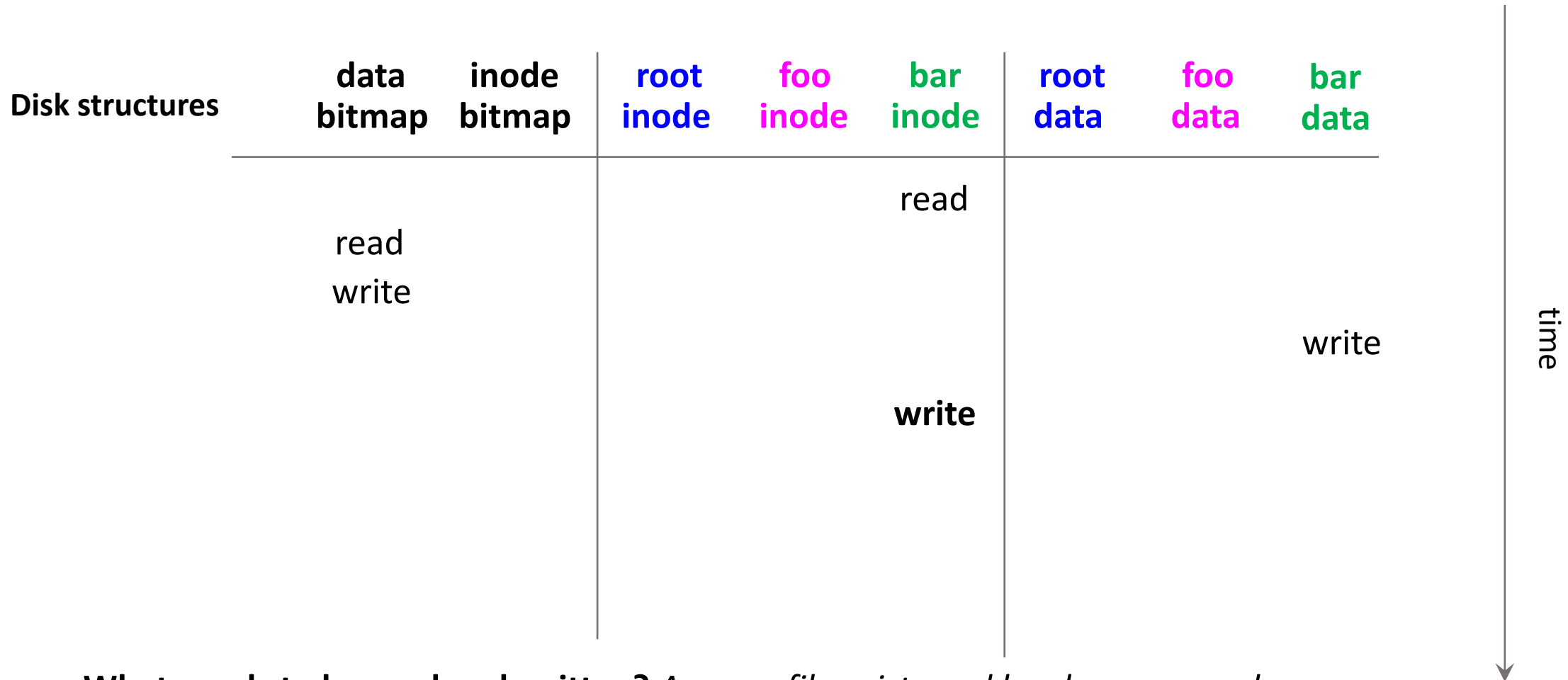


Write to /foo/bar



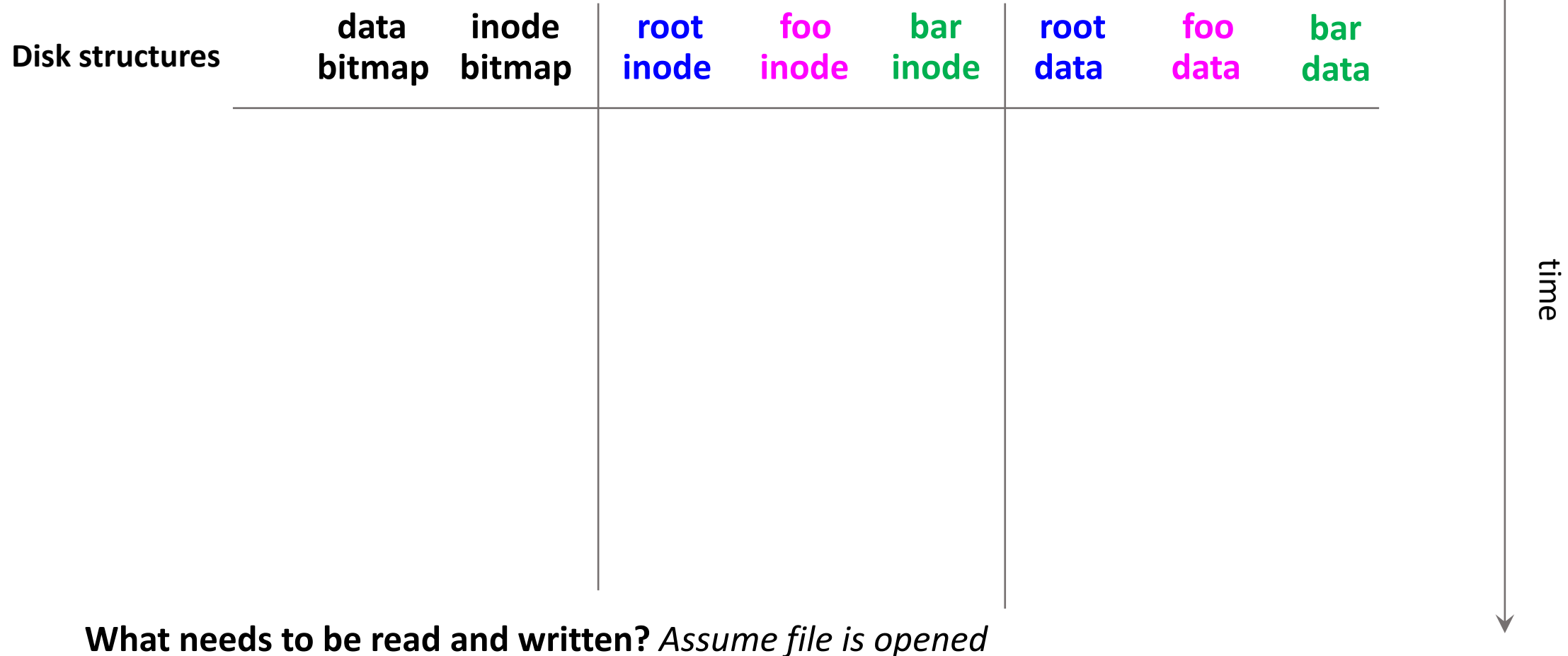
What needs to be read and written? *Assume file exists and has been opened*

Write to /foo/bar

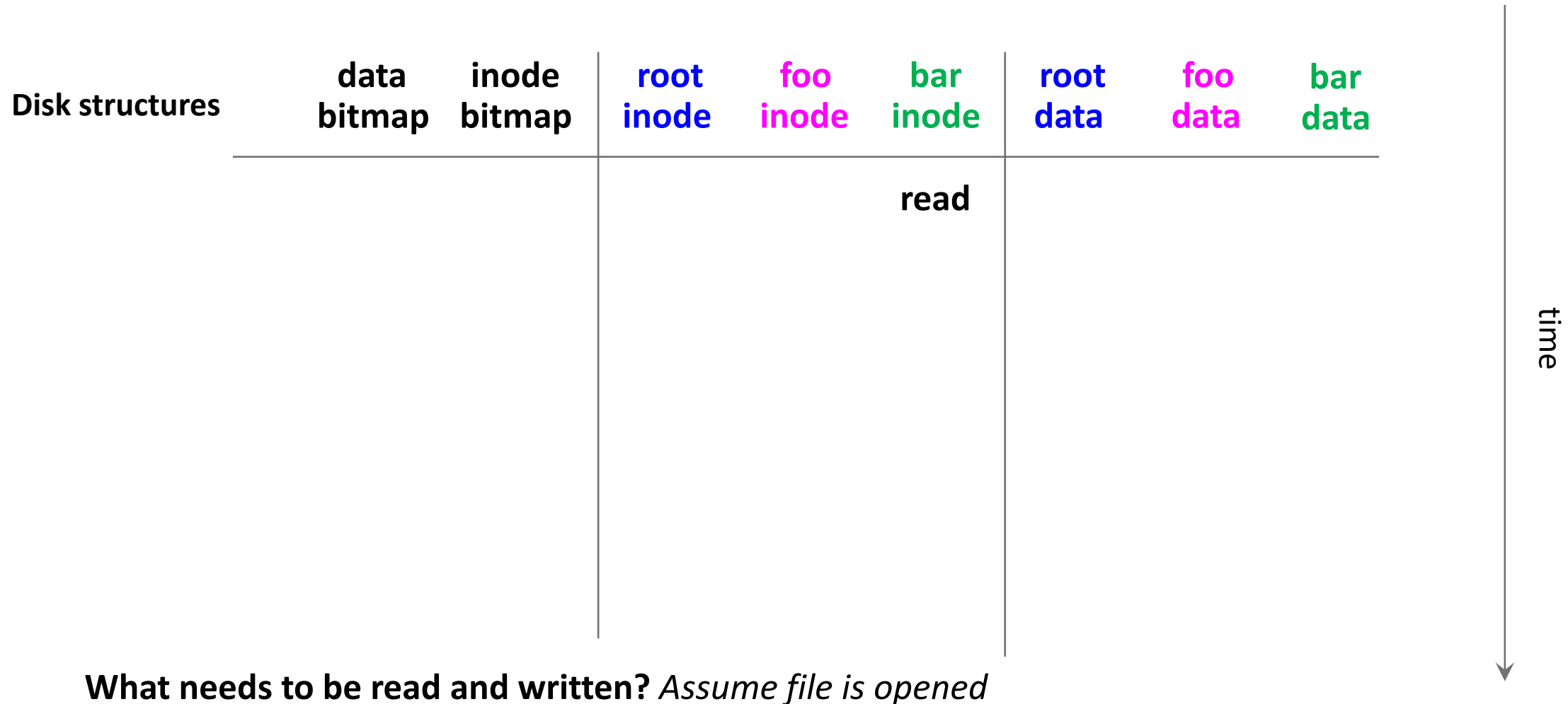


What needs to be read and written? *Assume file exists and has been opened*

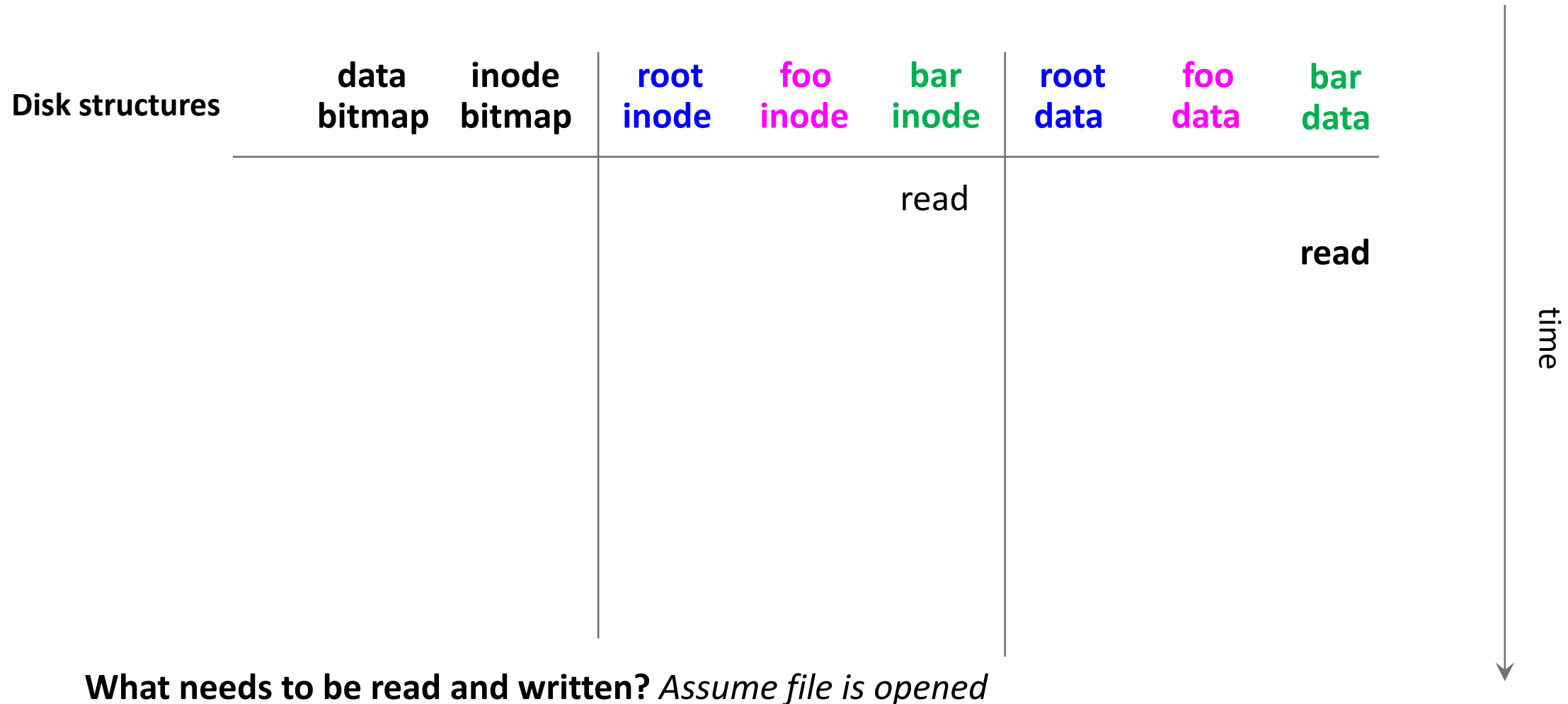
Read / foo / bar



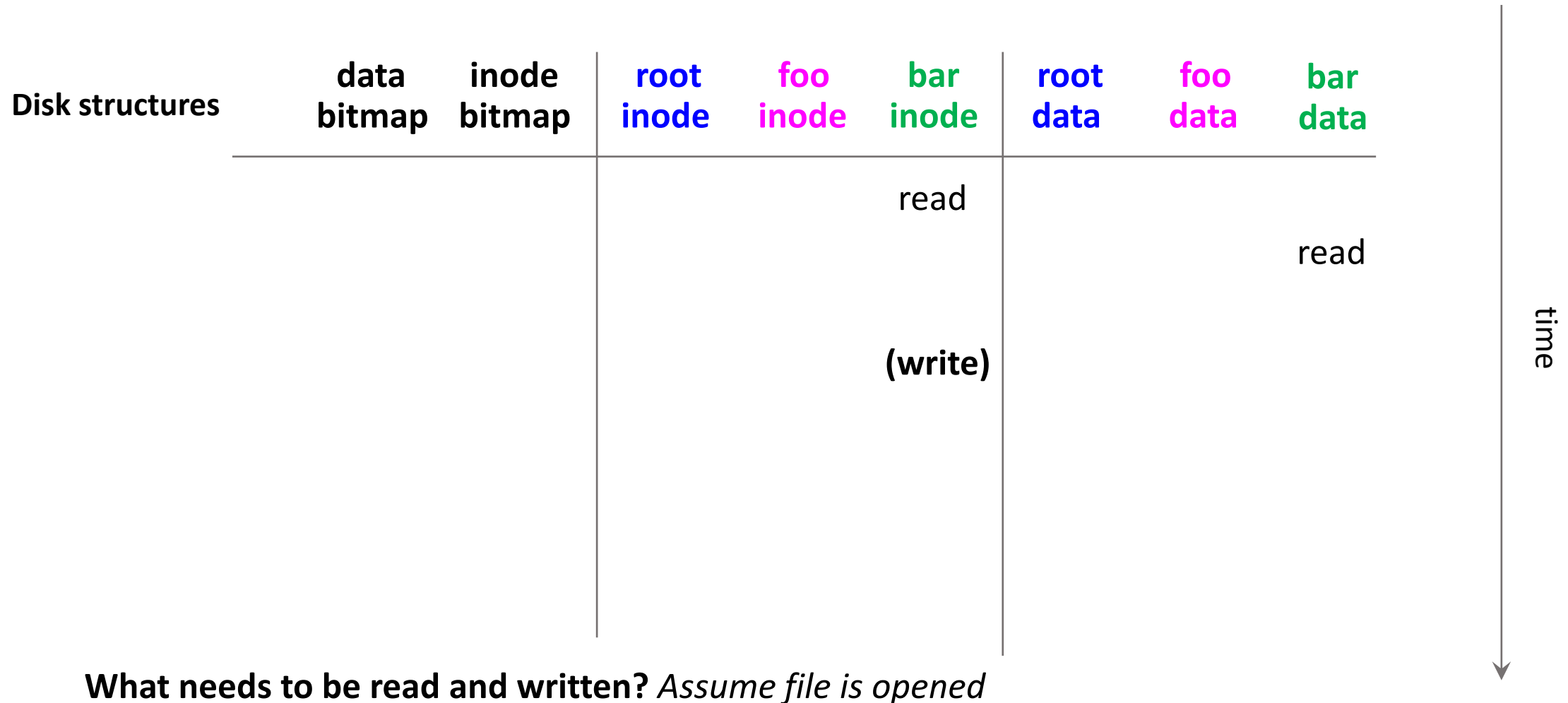
Read / foo / bar



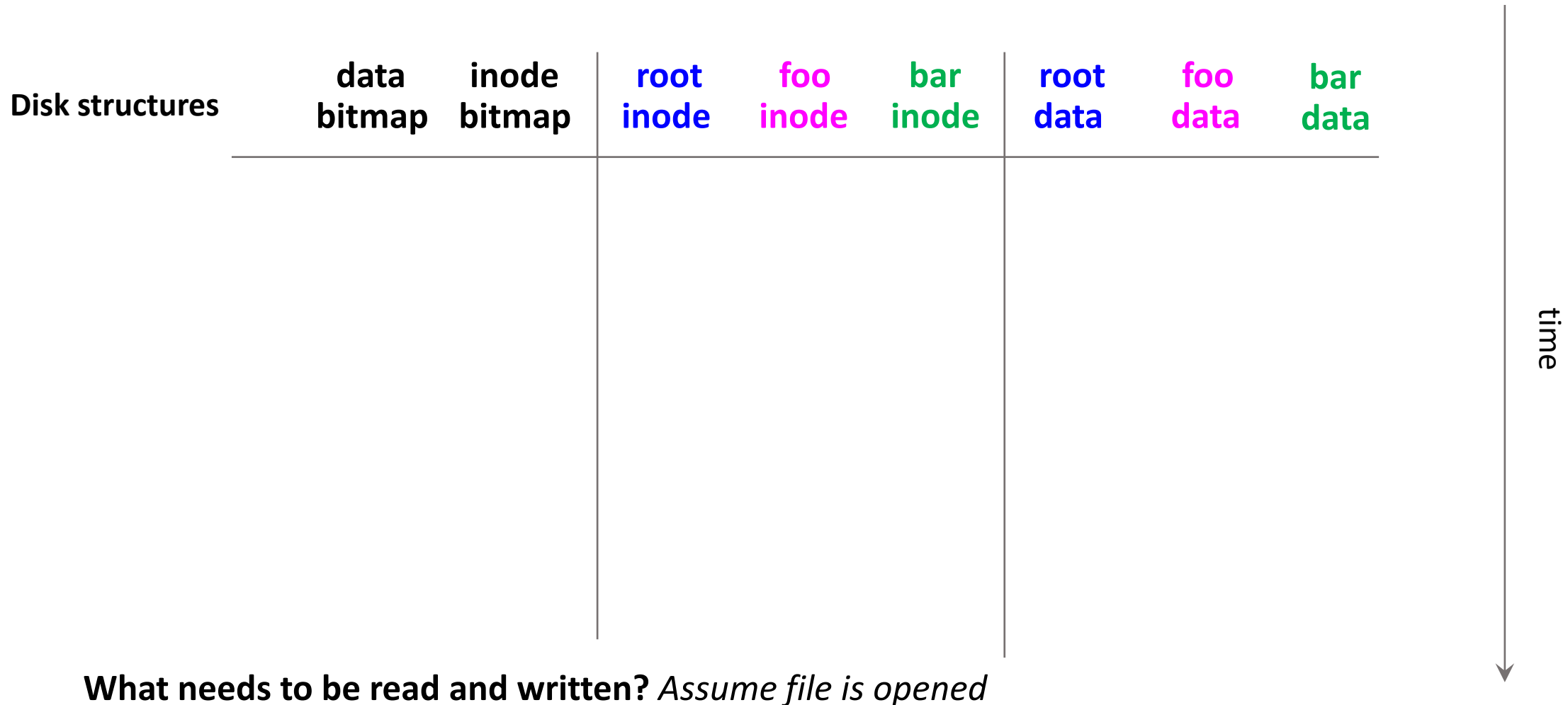
Read / foo / bar



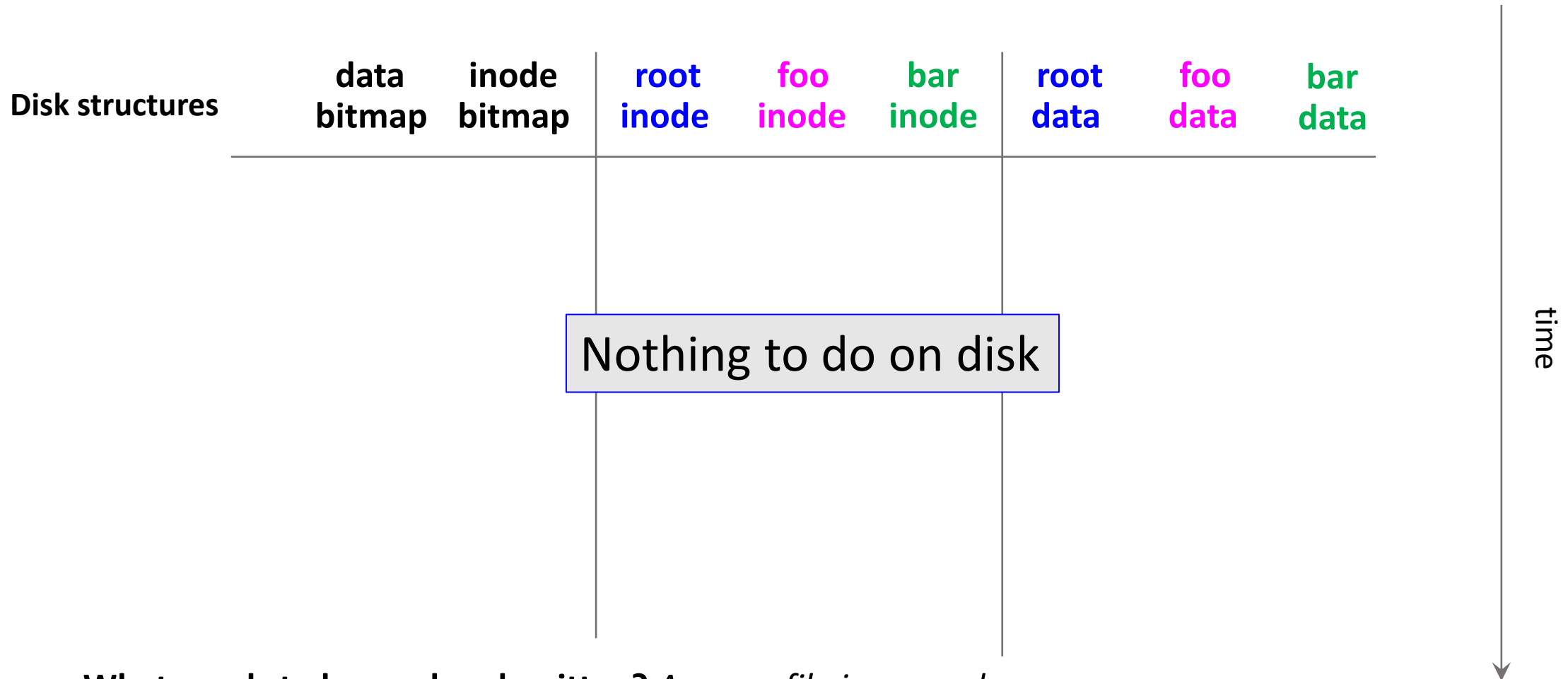
Read / foo / bar



Close / foo / bar



Close / foo / bar



What needs to be read and written? *Assume file is opened*

Efficiency?

- Head moves between
 - Directories
 - Inodes
 - Data

How can we avoid this excessive I/O?
We will see next week.

- Basic access methods need many I/O calls.
 - Particularly creating files

Remember: File System Implementation

Key aspects of the system:

1. Data structures

- On disk
- In memory

← In memory data structures are used to make I/O more efficient

2. Access methods

- How do we open(), read(), write() ?

In-Memory Data Structures

- Cache
- Cache directory
- Queue of pending disk requests
- Queue of pending user requests
- Active file table
- Open file tables

Cache

- Fixed contiguous area of kernel memory
- Size = max number of cache blocks x block size
- A large chunk of memory of the machine

Cache

- In general, write-behind is used
- For user data ok
- For metadata
 - Written to disk more aggressively
 - Affects integrity of file system

Cache Directory

- Usually a hash table
- $\text{index} = \text{hash}(\text{disk address})$
- With an overflow list in case of collision
- Usually has a “dirty” bit

Cache Replacement

- Keep LRU list
 - Unlike memory management, here easy to do
 - Accesses are far fewer (file vs memory access)
- If no more free entries in the cache
 - Replace “clean” block according to LRU
 - Replace “dirty” block according to LRU

Cache Flush

- Find “dirty” entries in cache
- Write them back to disk
 - Periodically (30 seconds)
 - When disk is idle

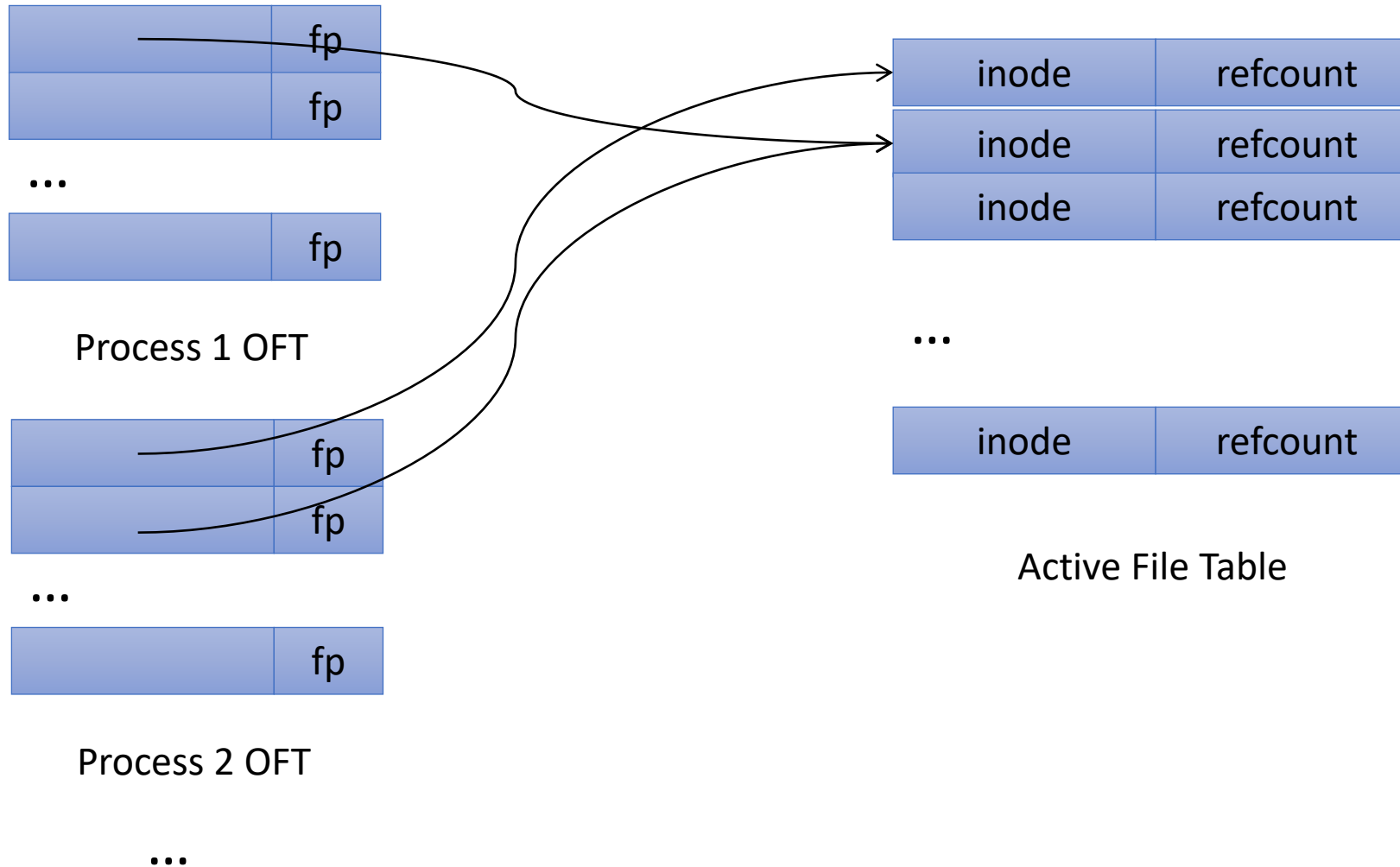
(System-Wide) Active File Table

- One array for the entire system
- One entry per *open file*
- Each entry contains
 - File inode
 - Additional information
 - Reference count of number of file opens

(Per-Process) Open File Tables

- One array per process
- One entry per *file open* of that process
- Indexed by file descriptor *fd*
- Each entry contains
 - Pointer to file inode in active file table
 - File pointer *fp*
 - Additional information

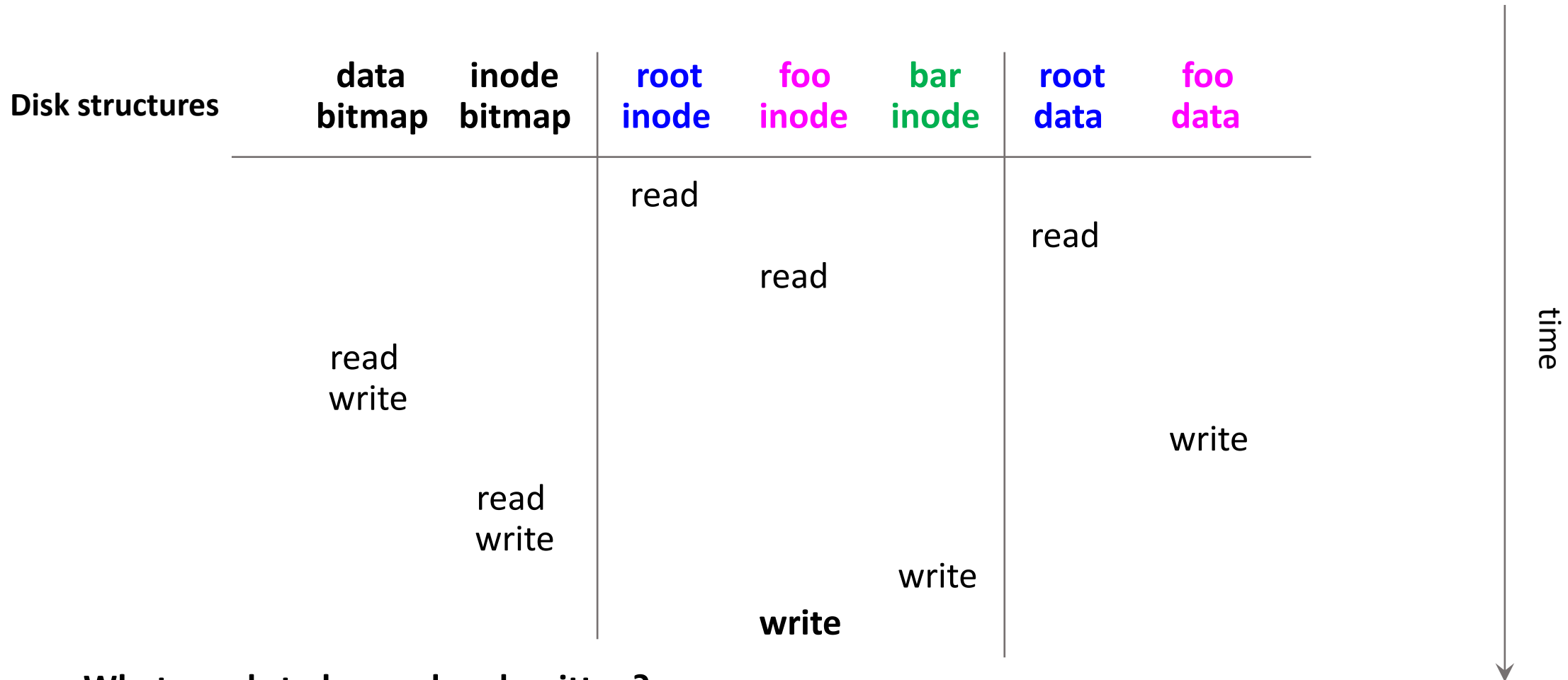
Open File Tables



Putting it All Together

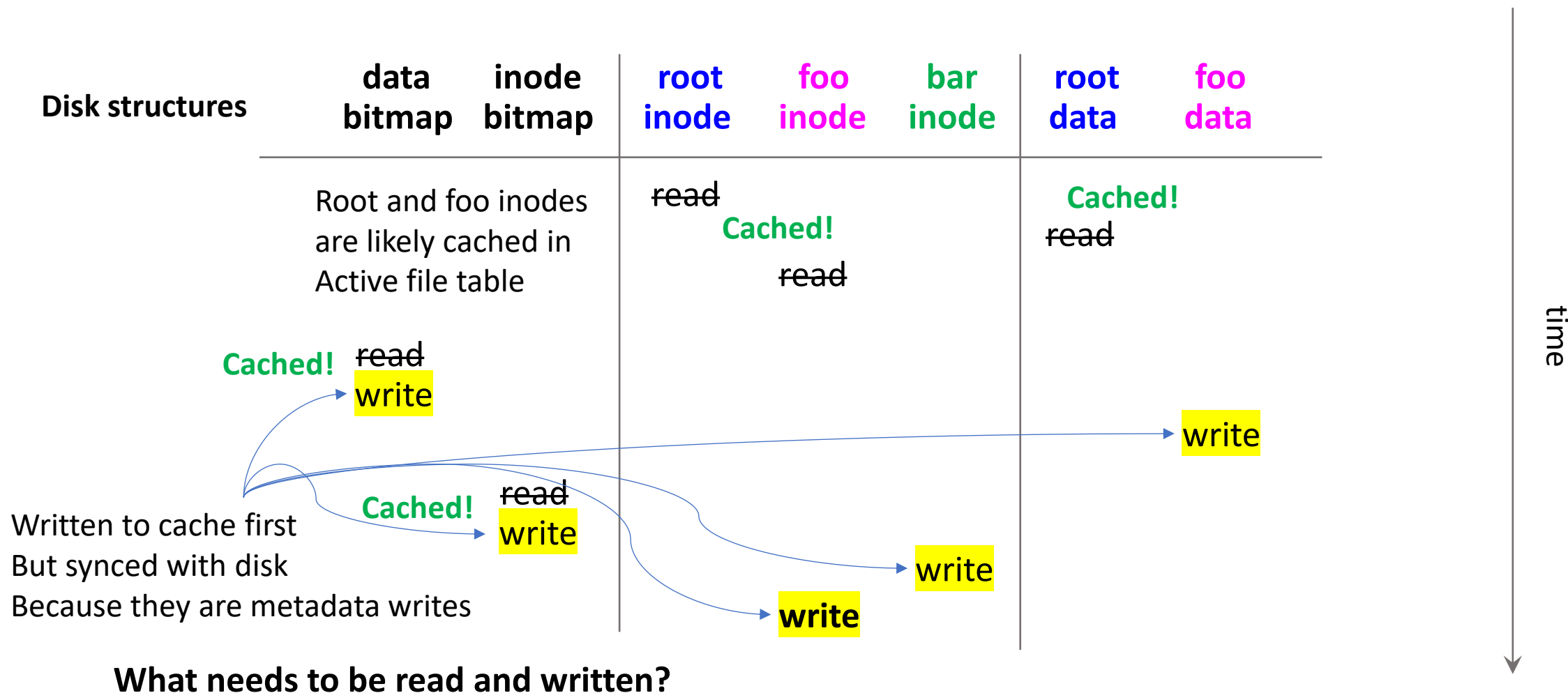
- Create
- Open
- Write
- Read
- Close

Create /foo/bar disk structures only

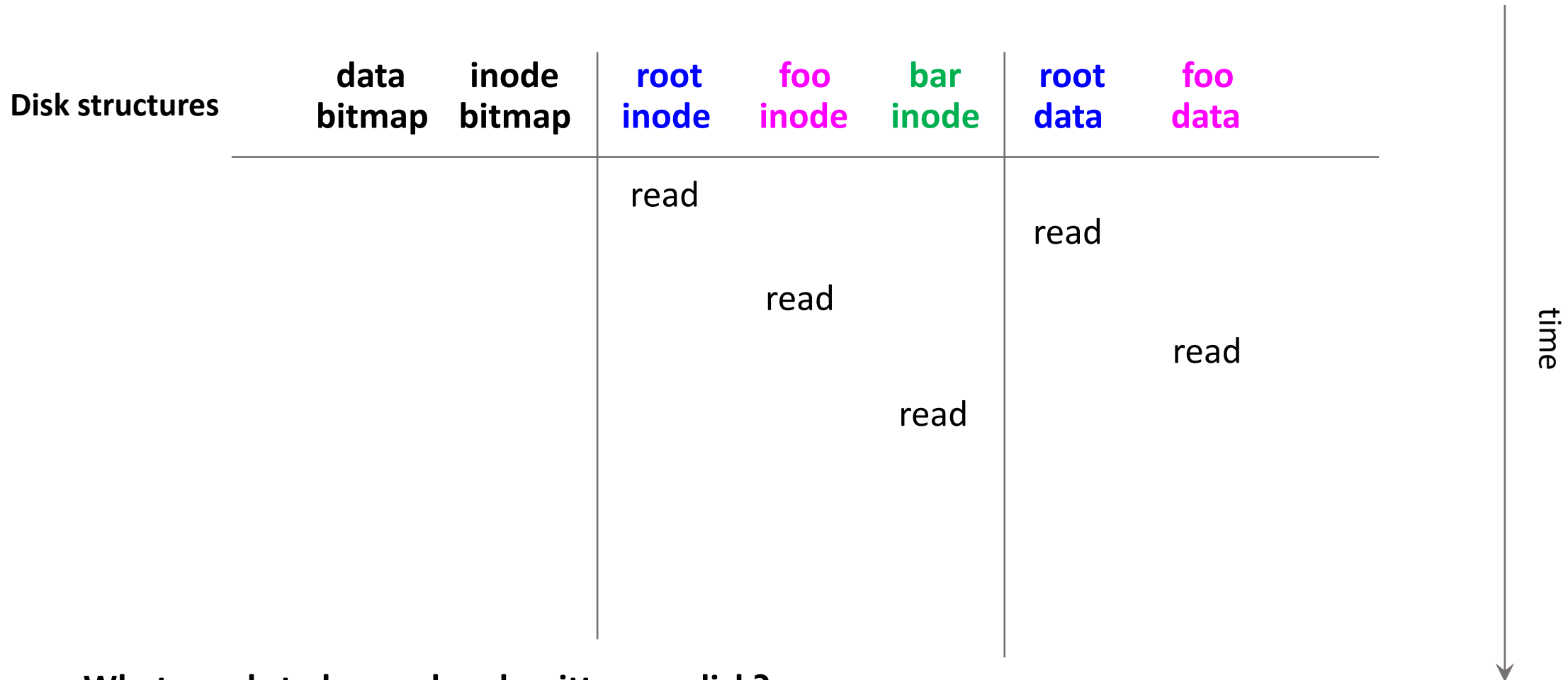


What needs to be read and written?

Create /foo/bar disk+memory structures

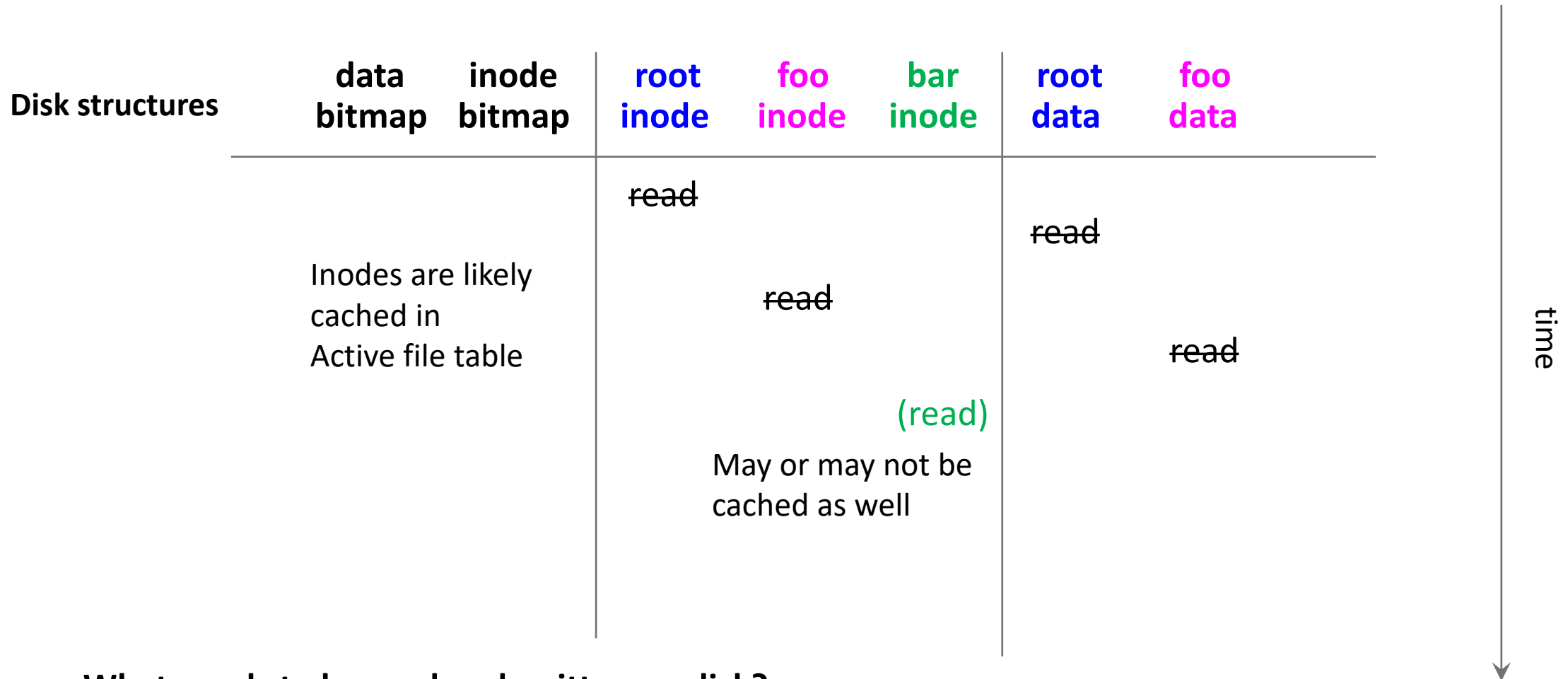


Open /foo/bar disk structures only



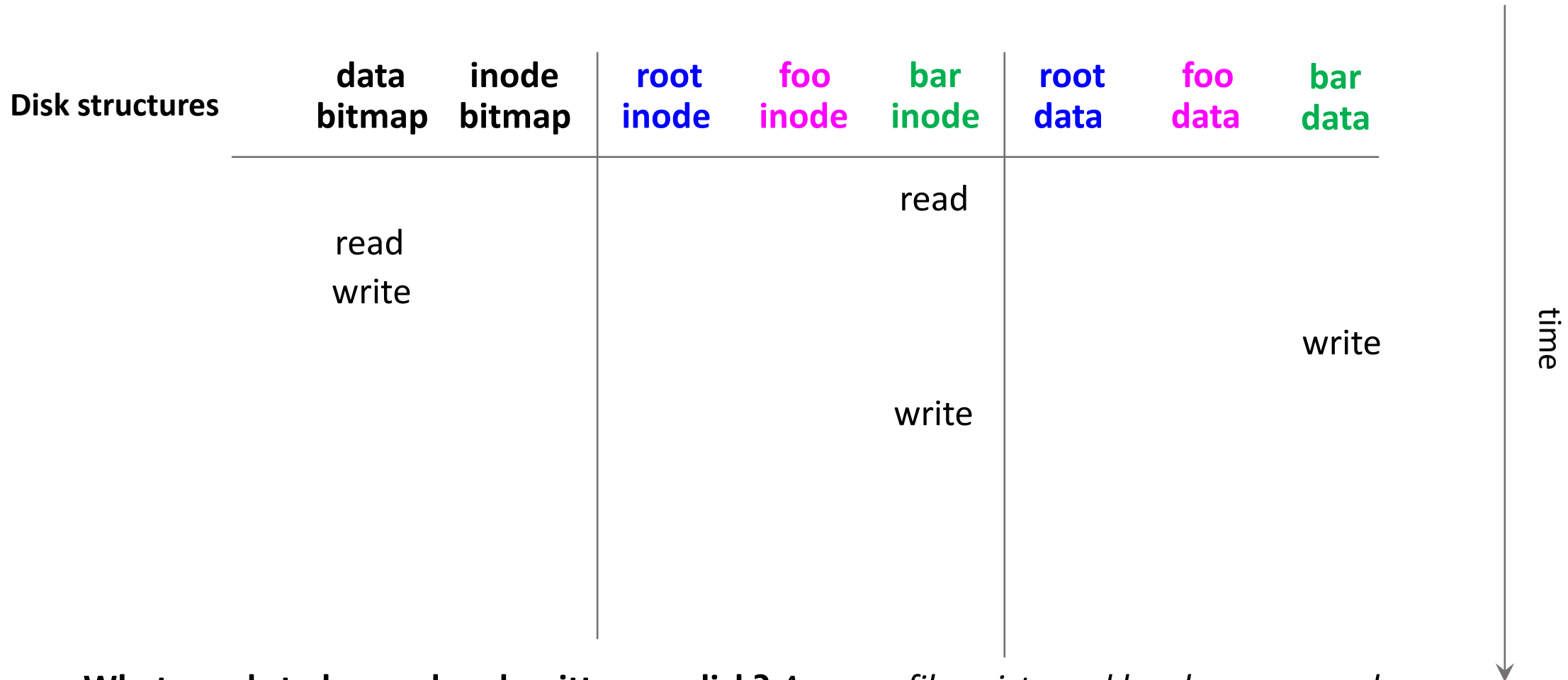
What needs to be read and written on disk?

Open /foo/bar disk+memory structures



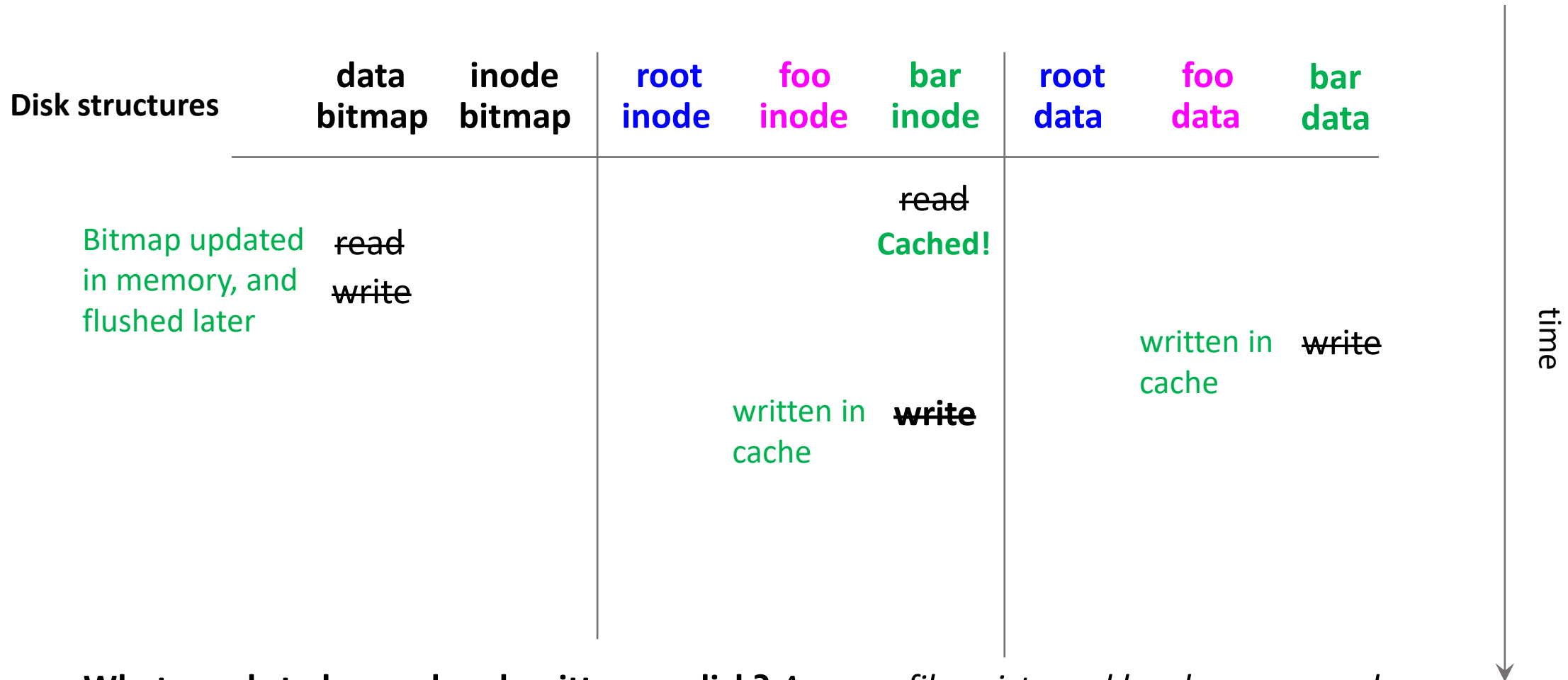
What needs to be read and written on disk?

Write to `/foo/bar` disk structures only



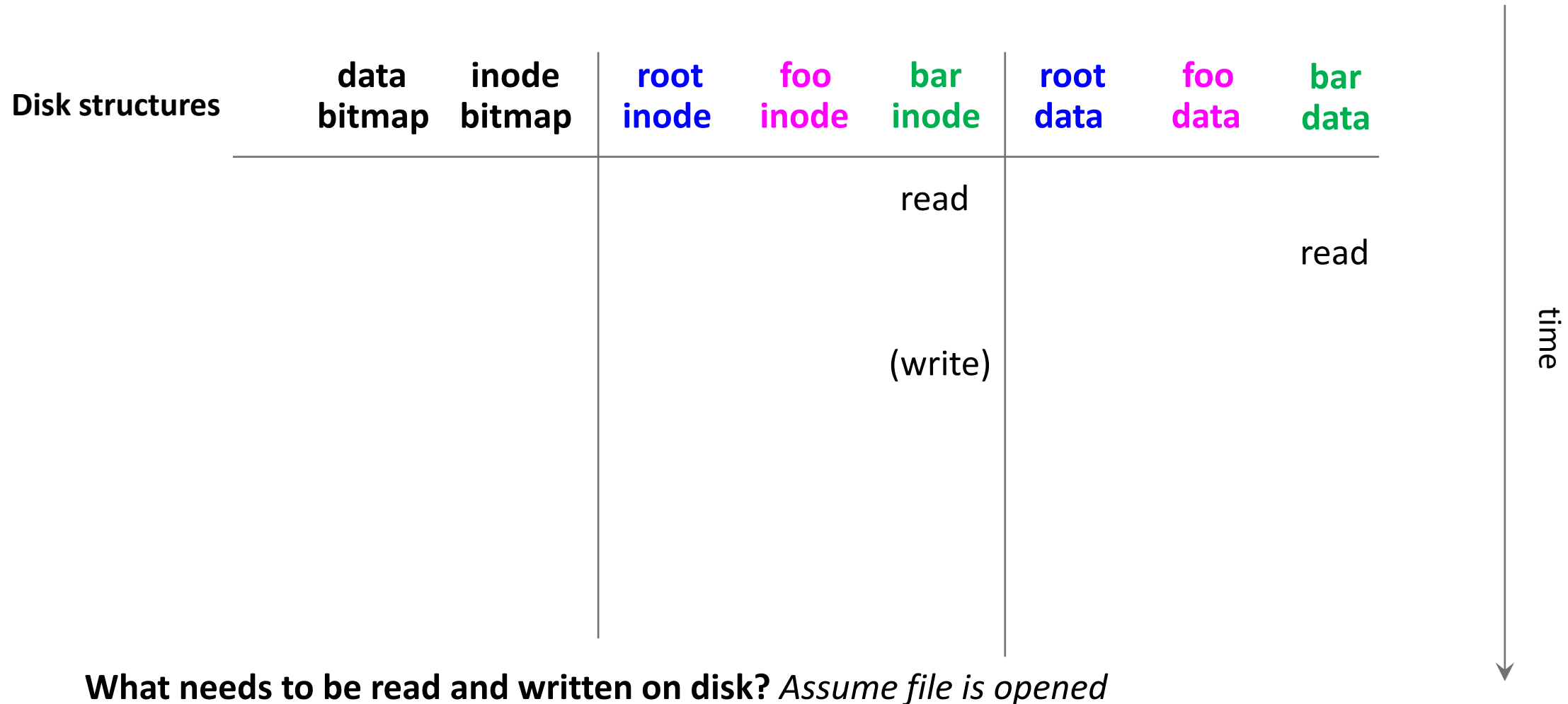
What needs to be read and written on disk? *Assume file exists and has been opened*

Write to /foo/bar disk+memory structures

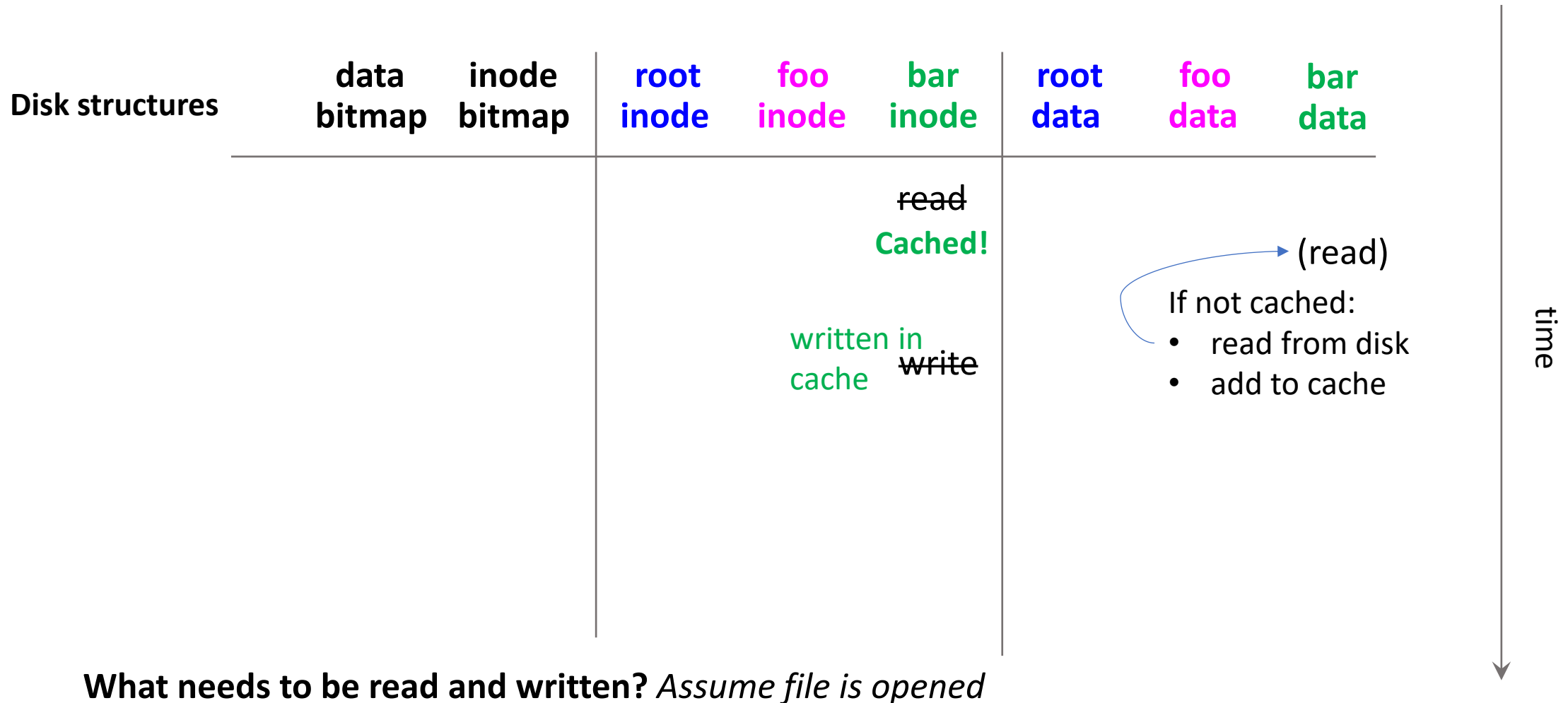


What needs to be read and written on disk? *Assume file exists and has been opened*

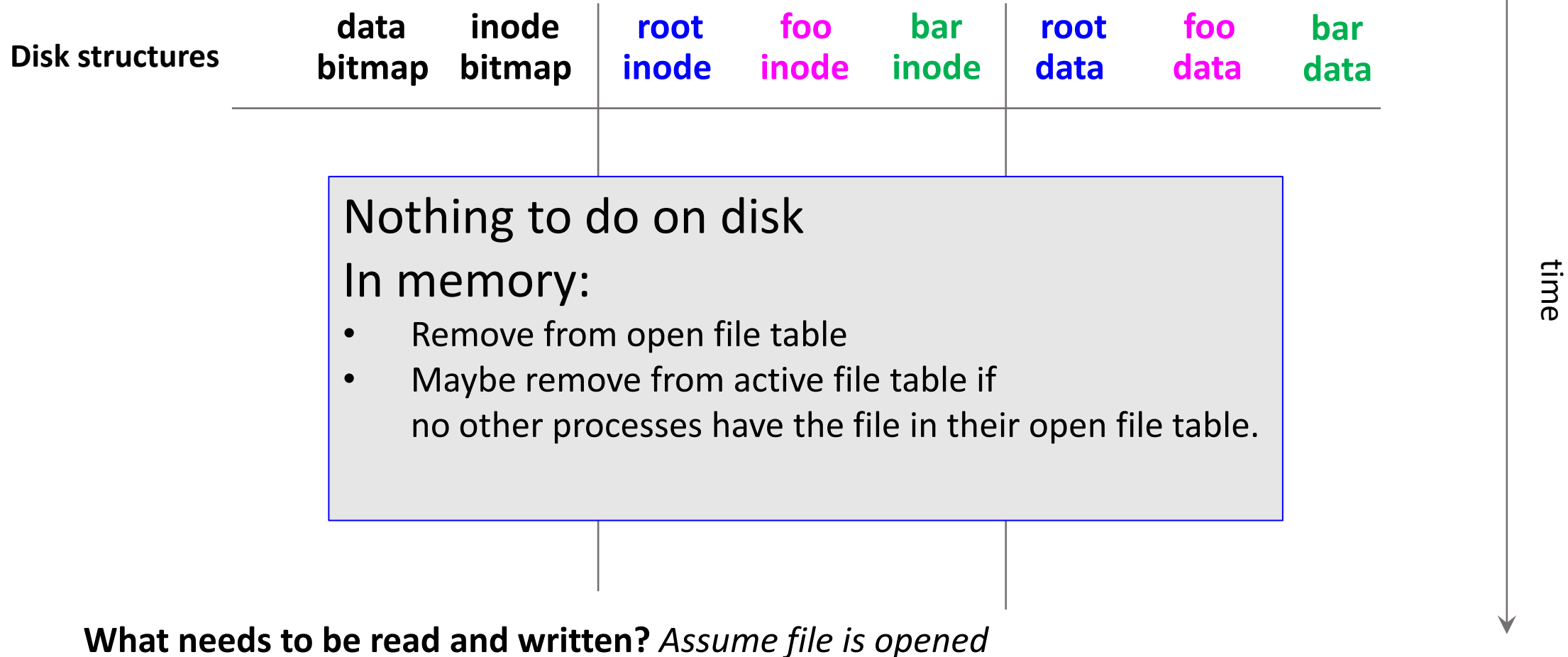
Read/foo/bar disk structures only



Read/ foo/ bar disk+memory structures



Close/fo o/bar disk+memory structures



Let's practice In-Memory Data Structures

- Process P1 opens file A and reads 512 bytes from it.
- Process P2 opens file A and reads 1024 bytes from it.
- Process P3 creates file B opens it and writes 8192 bytes to it.
- Process P1 opens file B and reads 1024 bytes from it.

At the end of this sequence of operations, describe the contents of the file system's in-memory data structures, including the active file table, open file tables, and cache. You can assume that, prior to this sequence of operations, the file A existed and had length 4096 bytes, and all in-memory data structures were empty. The size of the entries in the cache is 1024 bytes and the cache has 8 entries. Nothing else is happening in the OS during this sequence of operations.

Cache replacement policy: first kick out data chunks (LRU), followed by bitmaps (LRU if more than one bitmap), followed by inodes (LRU, if more than one inode).

- Process P1 opens file A and reads 512 bytes from it.
- Process P2 opens file A and reads 1024 bytes from it.
- Process P3 creates file B opens it and writes 8192 bytes to it.
- Process P1 opens file B and reads 1024 bytes from it.

- File A has length 4096 bytes
- The size of the entries in the cache is 1024 bytes and the cache has 8 entries.

Open file table P1

--

Open file table P2

--

Open file table P3

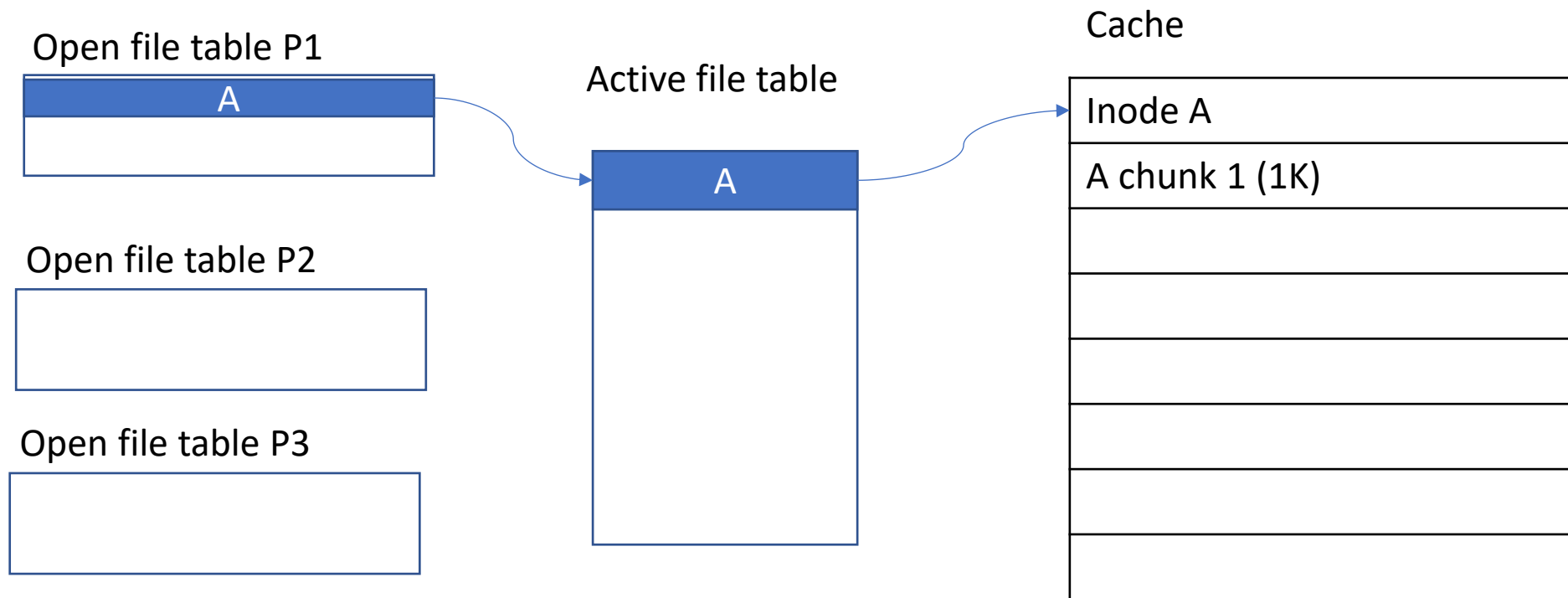
--

Active file table

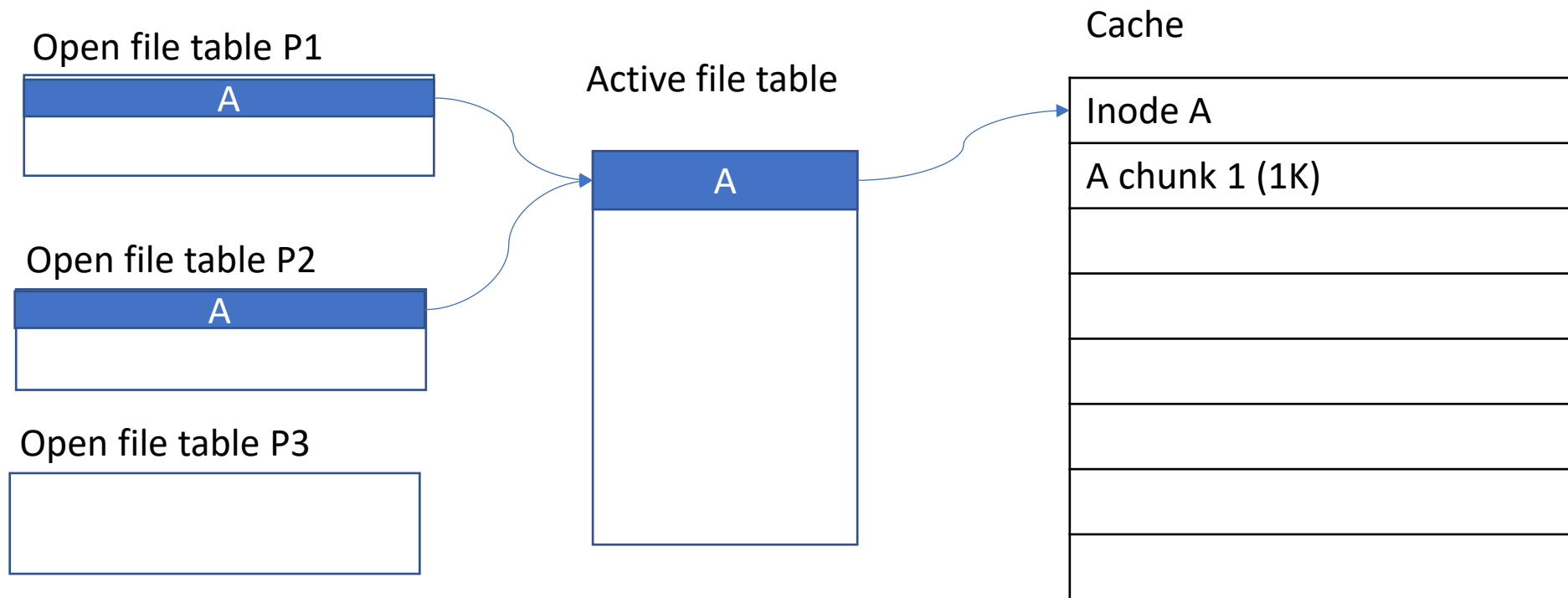
--

Cache

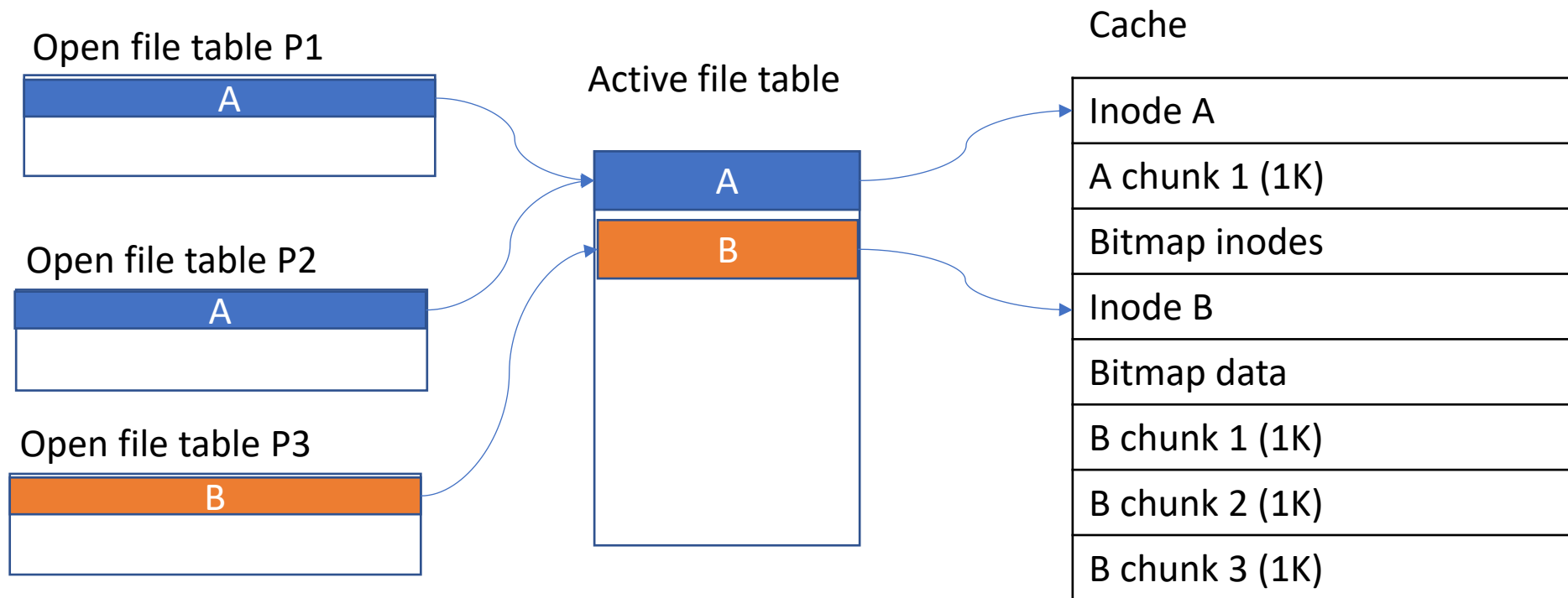
- **Process P1 opens file A and reads 512 bytes from it.**
 - Process P2 opens file A and reads 1024 bytes from it.
 - Process P3 creates file B opens it and writes 8192 bytes to it.
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



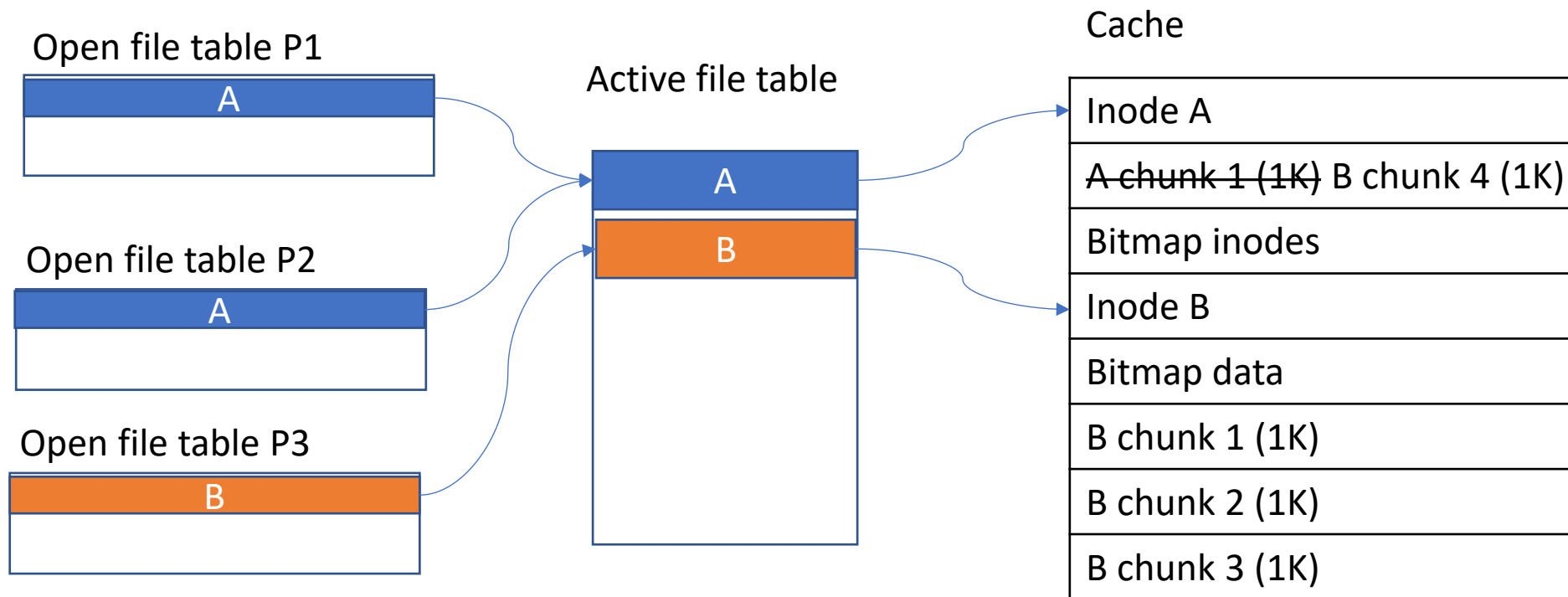
- Process P1 opens file A and reads 512 bytes from it.
 - **Process P2 opens file A and reads 1024 bytes from it.**
 - Process P3 creates file B opens it and writes 8192 bytes to it.
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



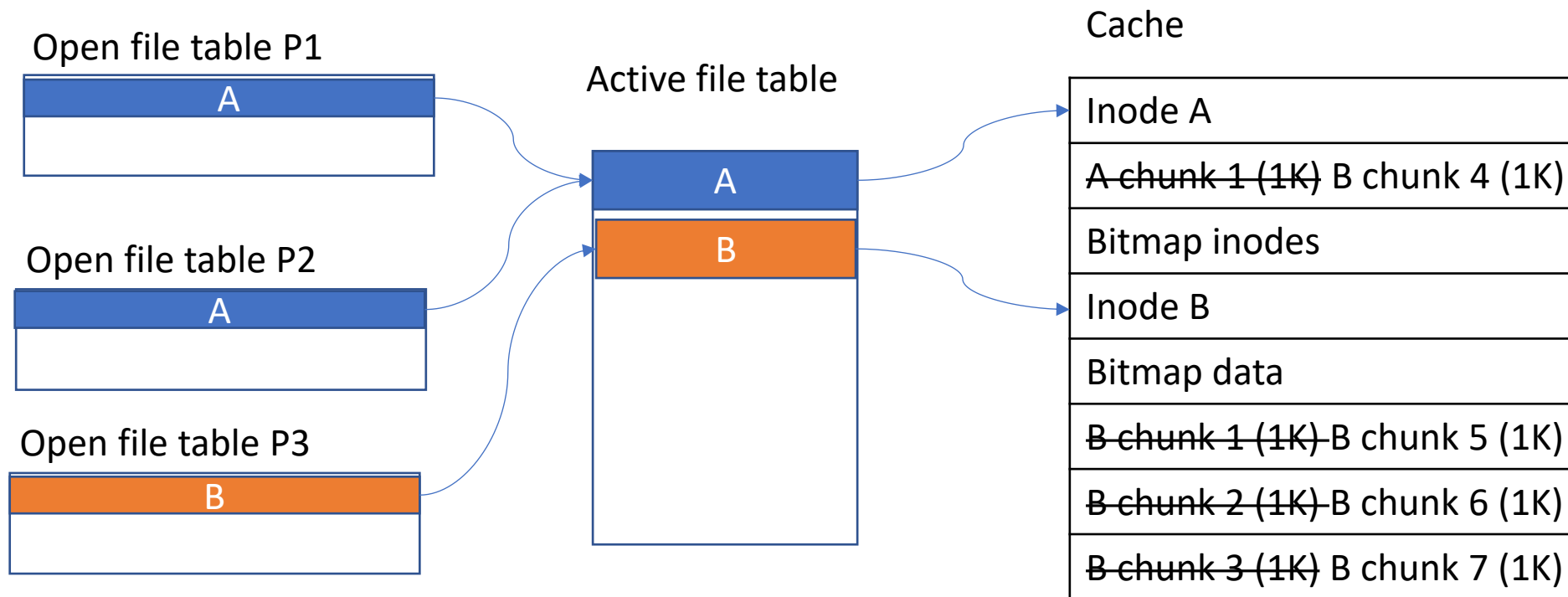
- Process P1 opens file A and reads 512 bytes from it.
 - Process P2 opens file A and reads 1024 bytes from it.
 - **Process P3 creates file B opens it and writes 8192 bytes to it.**
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



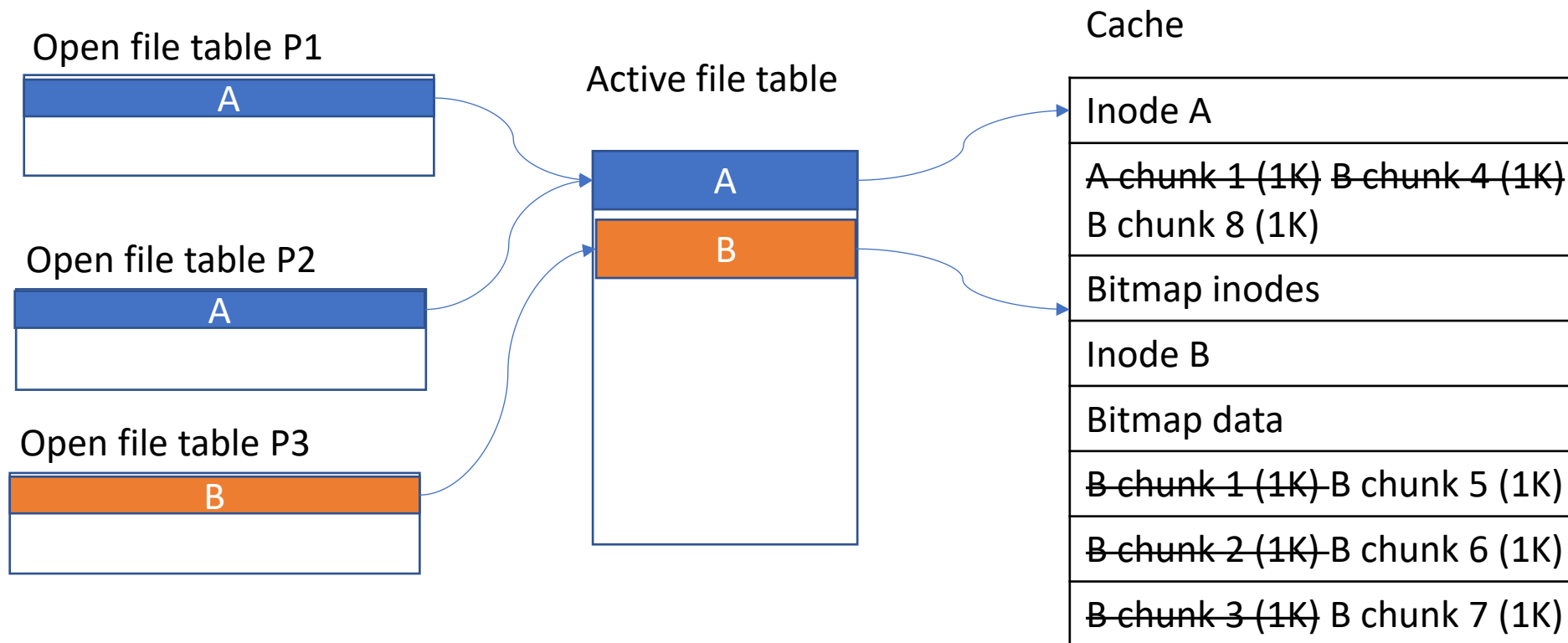
- Process P1 opens file A and reads 512 bytes from it.
 - Process P2 opens file A and reads 1024 bytes from it.
 - **Process P3 creates file B opens it and writes 8192 bytes to it.**
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



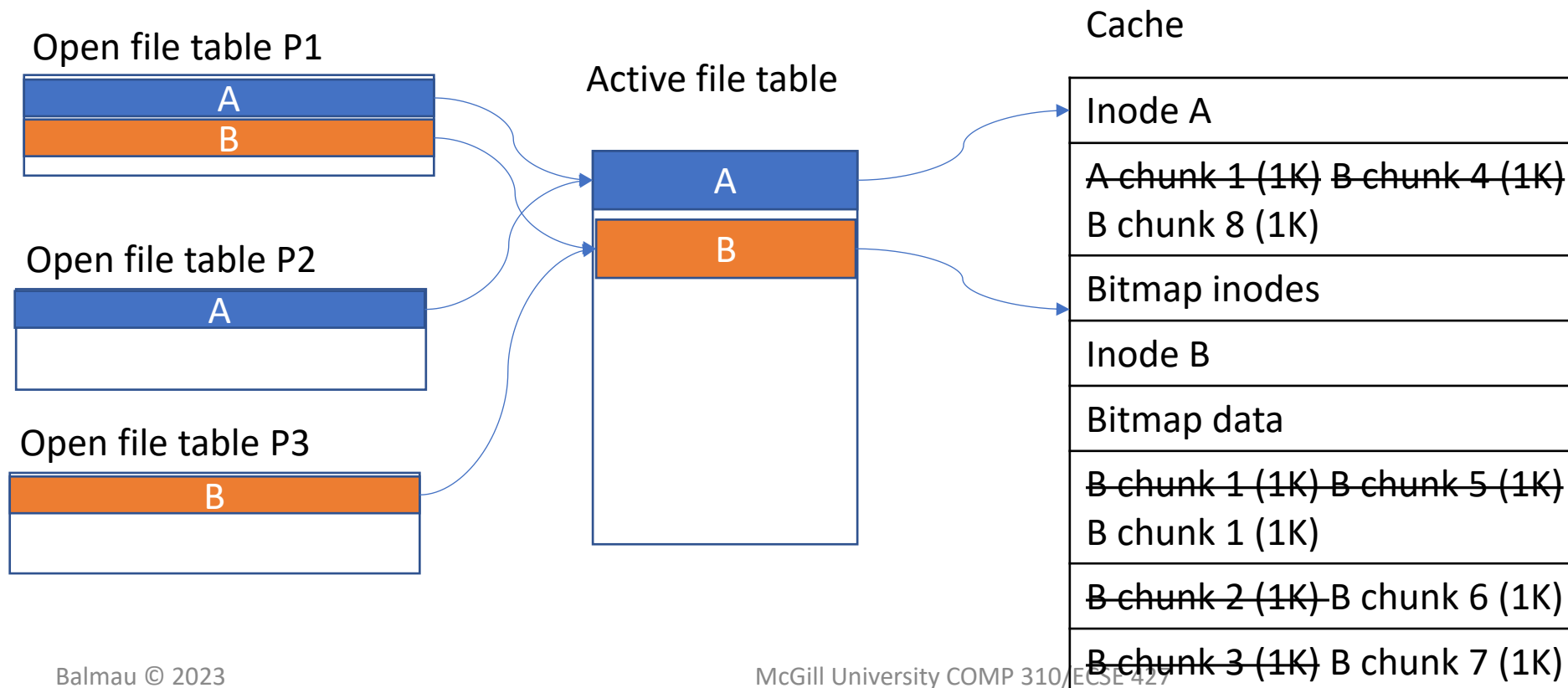
- Process P1 opens file A and reads 512 bytes from it.
 - Process P2 opens file A and reads 1024 bytes from it.
 - **Process P3 creates file B opens it and writes 8192 bytes to it.**
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



- Process P1 opens file A and reads 512 bytes from it.
 - Process P2 opens file A and reads 1024 bytes from it.
 - **Process P3 creates file B opens it and writes 8192 bytes to it.**
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



- Process P1 opens file A and reads 512 bytes from it.
 - Process P2 opens file A and reads 1024 bytes from it.
 - **Process P3 creates file B opens it and writes 8192 bytes to it.**
 - Process P1 opens file B and reads 1024 bytes from it.
- File A has length 4096 bytes
 - The size of the entries in the cache is 1024 bytes and the cache has 8 entries.



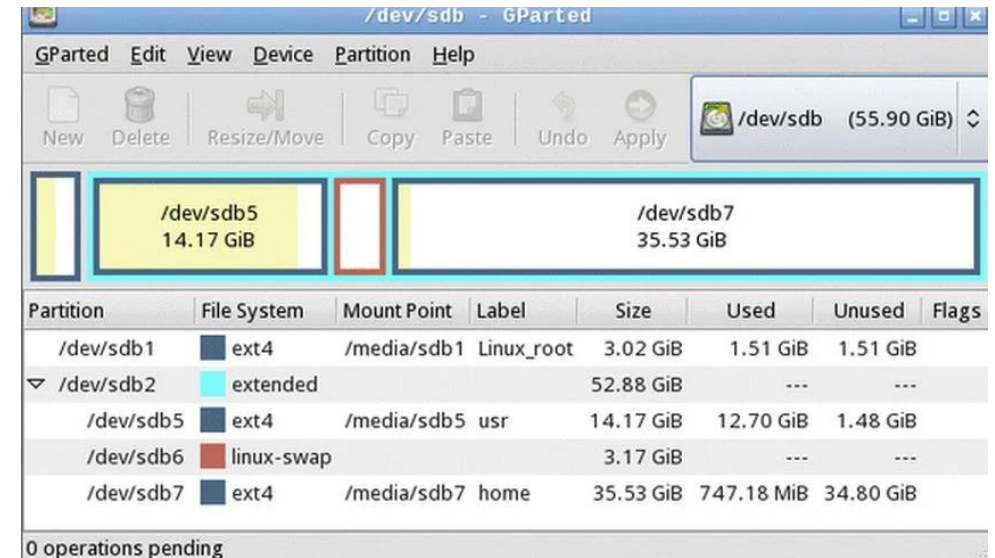
Setting up the FS

Setting up the FS

- By default, OS sees all storage devices as
 - chunks of unallocated space
 - which are unusable
- Cannot start writing files to a blank drive!
- Need to set up the FS first

Disk Partitioning (or Slicing)

- FS needs a “container” on the storage device
- Container is called a **partition**
- Partitioning allows different FS to be installed on same OS



The screenshot shows the GParted application window titled "/dev/sdb - GParted". The interface includes a menu bar (GParted, Edit, View, Device, Partition, Help) and a toolbar with icons for New, Delete, Resize/Move, Copy, Paste, Undo, and Apply. Below the toolbar, a visual representation of the disk shows several partitions. Two partitions are highlighted with blue borders: /dev/sdb5 (14.17 GiB) and /dev/sdb7 (35.53 GiB). Below this visual, a table lists the details of the partitions.

Partition	File System	Mount Point	Label	Size	Used	Unused	Flags
/dev/sdb1	ext4	/media/sdb1	Linux_root	3.02 GiB	1.51 GiB	1.51 GiB	
▼ /dev/sdb2	extended			52.88 GiB	---	---	
/dev/sdb5	ext4	/media/sdb5	usr	14.17 GiB	12.70 GiB	1.48 GiB	
/dev/sdb6	linux-swap			3.17 GiB	---	---	
/dev/sdb7	ext4	/media/sdb7	home	35.53 GiB	747.18 MiB	34.80 GiB	

0 operations pending

Disk Partitioning (or Slicing)

- Each partition appears to OS as a logical disk.
- Disk stores partition info in **partition table**:
 - Partition locations
 - Partition sizes
- OS reads partition table before any other part of the disk.

Mounting a File System (FS)

- FS lives inside a partition
- But OS cannot read/write files yet
- FS needs to be **mounted** for OS to access its files
 - (subtle problem: where is the OS itself?)

Mounting a File System (FS)

- Mounting attaches FS to a directory
- Directory is called **mount point**

Multiple FS

- Users may want to have many FS at the same time
 - Main disk
 - Backup disk
 - USB drive
 - etc
- How can OS support this?

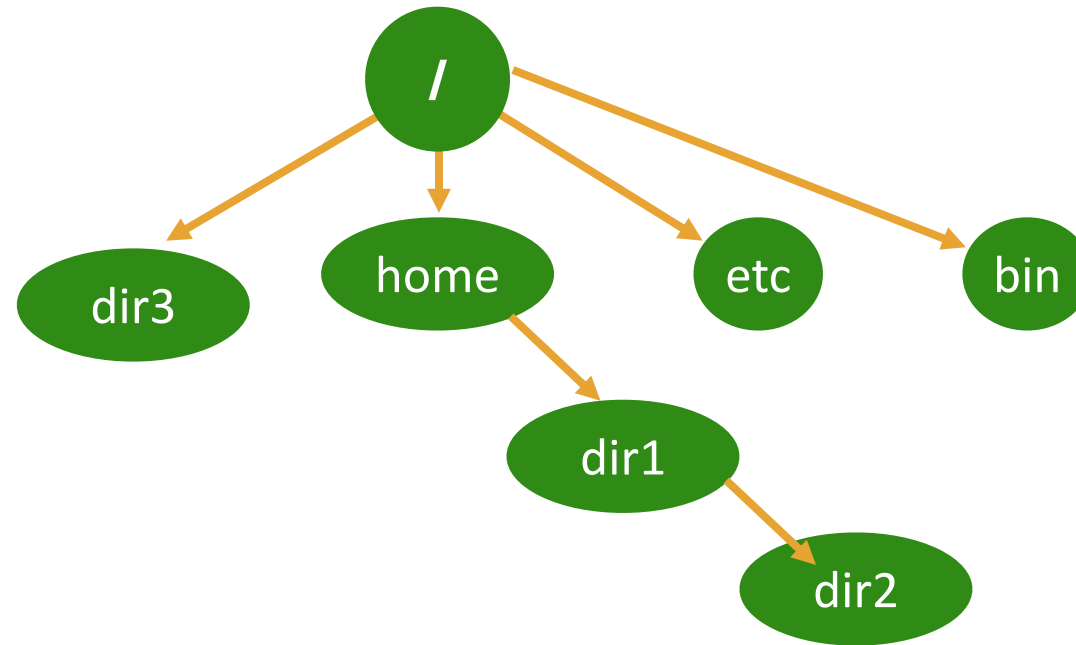
Multiple FS

- Users may want to have many FS at the same time
 - Main disk
 - Backup disk
 - USB drive
 - etc
- How can OS support this?
- **Idea:** Stitch all the file systems together into a “super file system”!

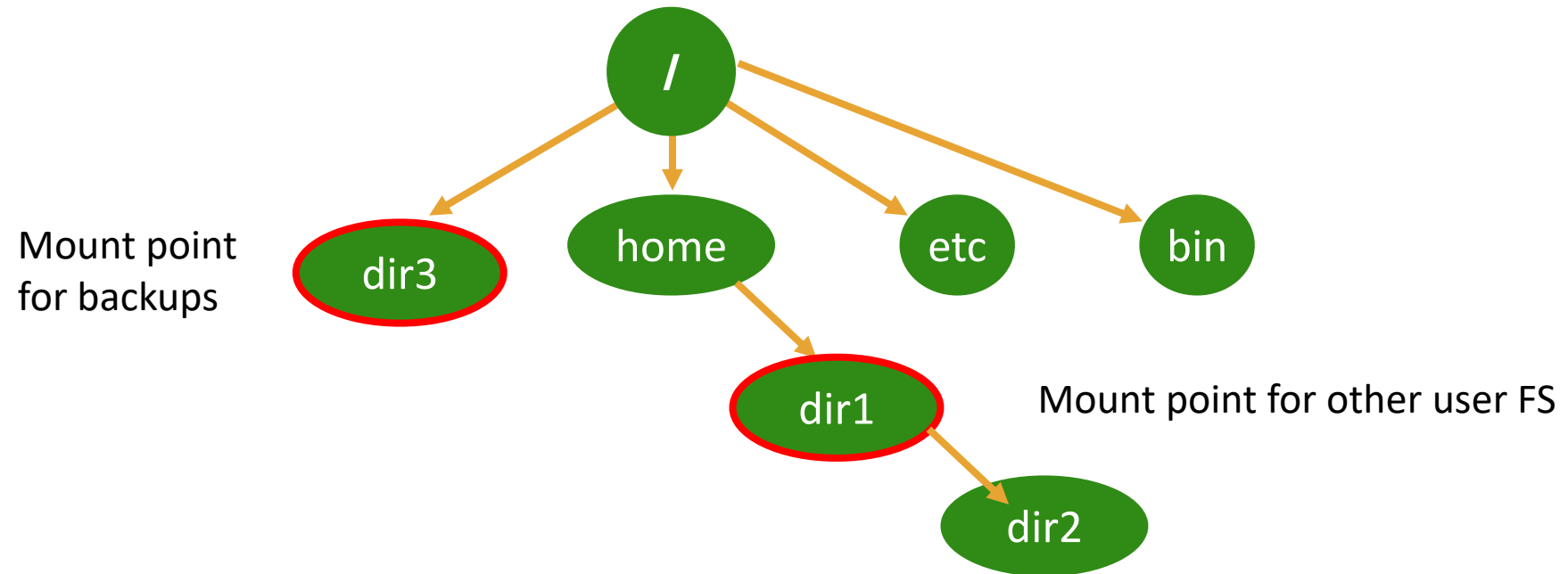
Multiple FS

- `root (/)` file system is always mounted.
- OS keeps track of mounted FS in Mounted FS Table

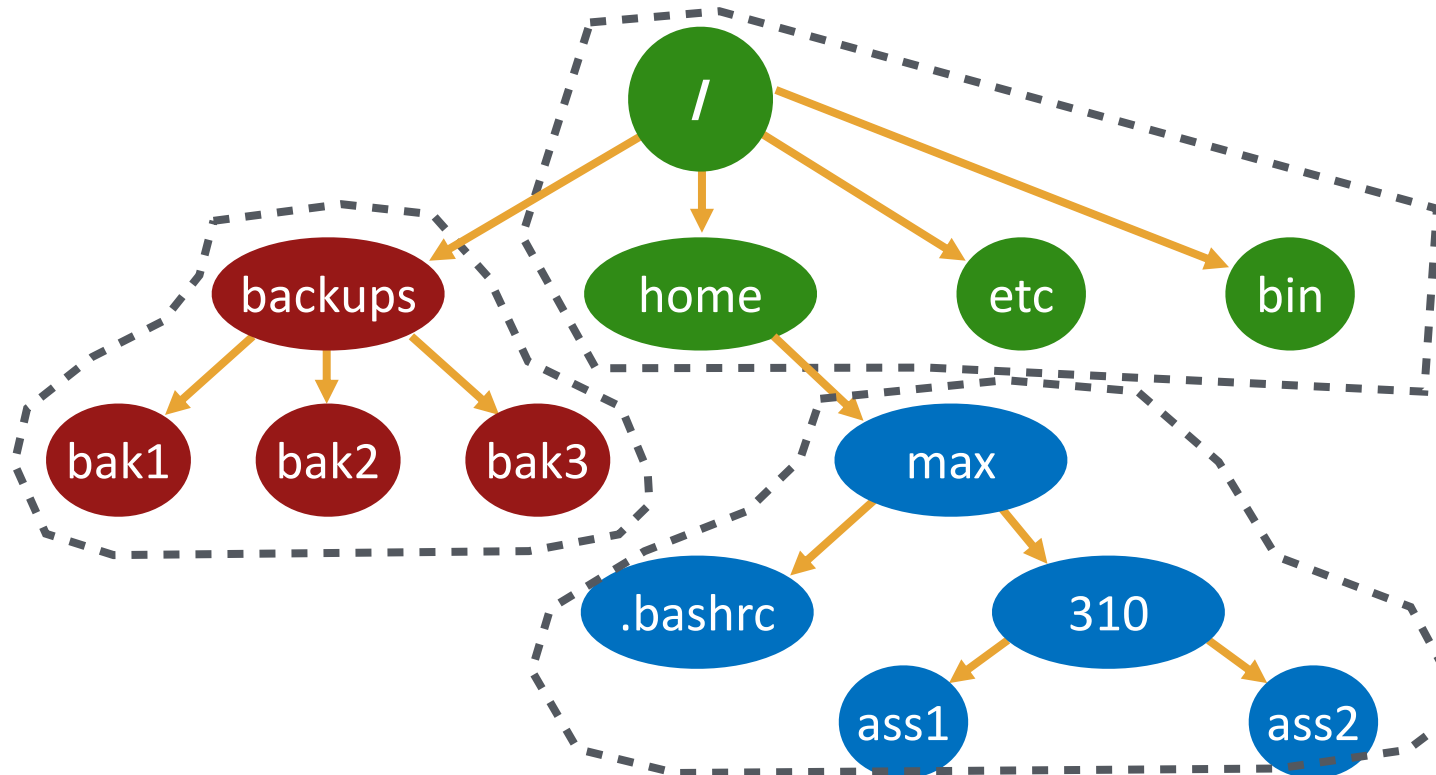
Example with 3 mounted FS



Example with 3 mounted FS

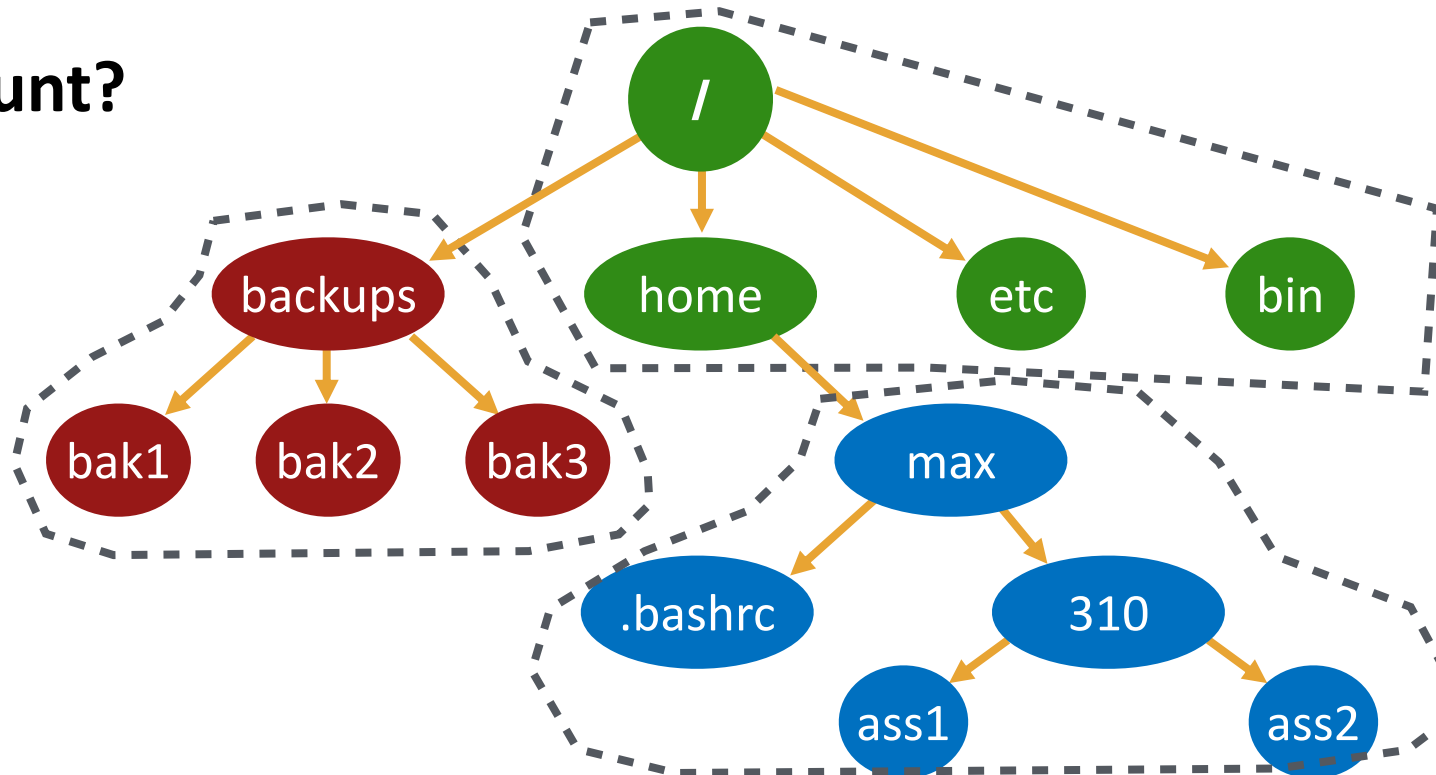


Example with 3 mounted FS



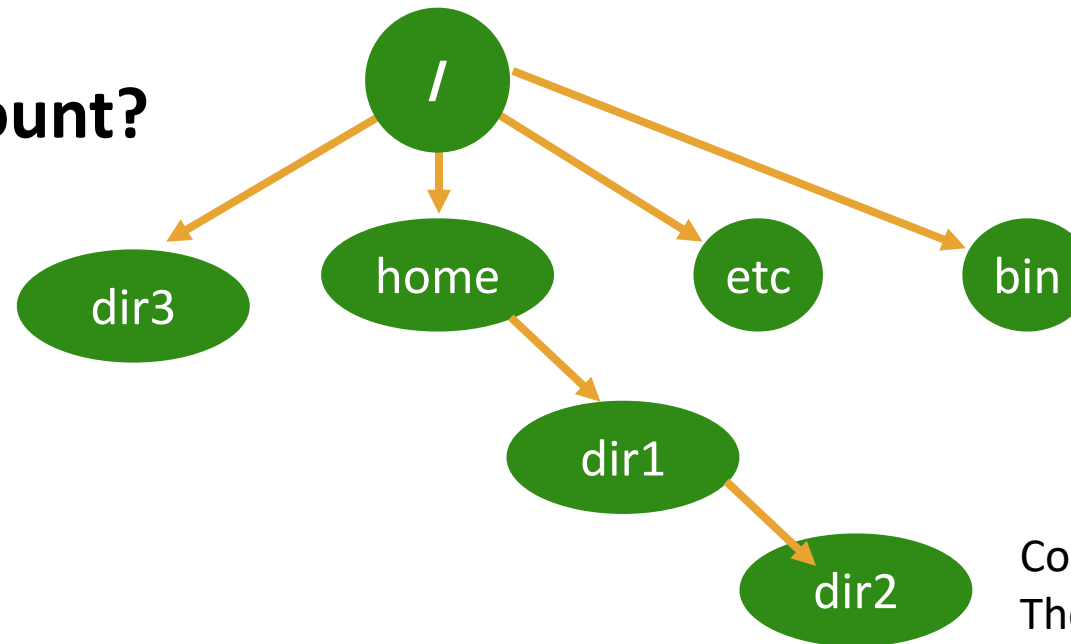
Example with 3 mounted FS

Unmount?



Example with 3 mounted FS

Back to initial
State after unmount?



Contents of dir1 and dir2 are **not lost**.
They were **just hidden** by the mount

But typically, we mount in empty directories.

Booting

- We said `root (/)` file system was always mounted.
- How is this done?

Booting

- Process that happens to turn a computer on
- Motherboard firmware (“BIOS”) in charge
 - looks for clues on what it needs to start OS
 - BIOS searches for a “boot block” that can start the system
 - Has to set up devices to search them, but no drivers from OS yet. Not easy!
- Disk contains Master Boot Record (MBR), suitable boot block
 - When installing an OS on a new computer, boot block might come from CD or USB

Boot Block

- At fixed location on disk (usually sector 0)
 - Contains boot loader
 - Contains partition table
- Read by BIOS before OS is even selected (e.g. dual booting)
- BIOS gives control to boot loader
- Boot loader loads OS into RAM, then gives it control

File System Startup

- Normally, nothing would be necessary
- Sometimes things are not normal
 - Disk sector goes bad
 - File system software has bugs
 - ...
- Common to “check” the file system (fsck)
 - “Early” step in OS
 - There are thousands of things the OS needs to do to start. They are all “early.”
 - This is lower priority than setting up the MMU, but higher priority than loading non-disk drivers. This check takes a **long** time.

File System Check

- No sectors are allocated twice
- No sectors are allocated and on free list
- Reconstruct free list

Replication

- Some key sectors are replicated
 - Boot blocks
 - Sometimes also inode blocks

Brief Break

- File system implementation
- Handling multiple file systems
- Brief description of booting

- One more topic: memory mapped files

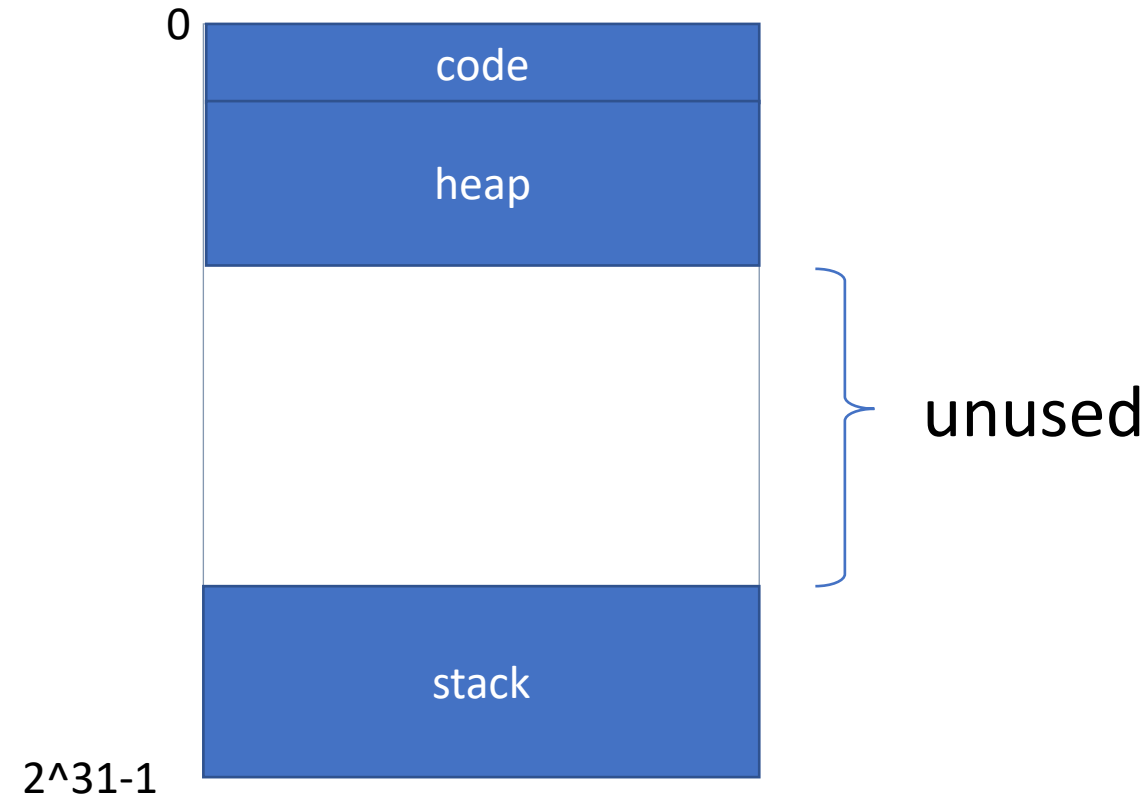
Alternative File Access Method: Memory Mapping

Alternative File Access Method: Memory Mapping

- `mmap()`
 - Map the contents of a file in memory
- `munmap()`
 - Remove the mapping

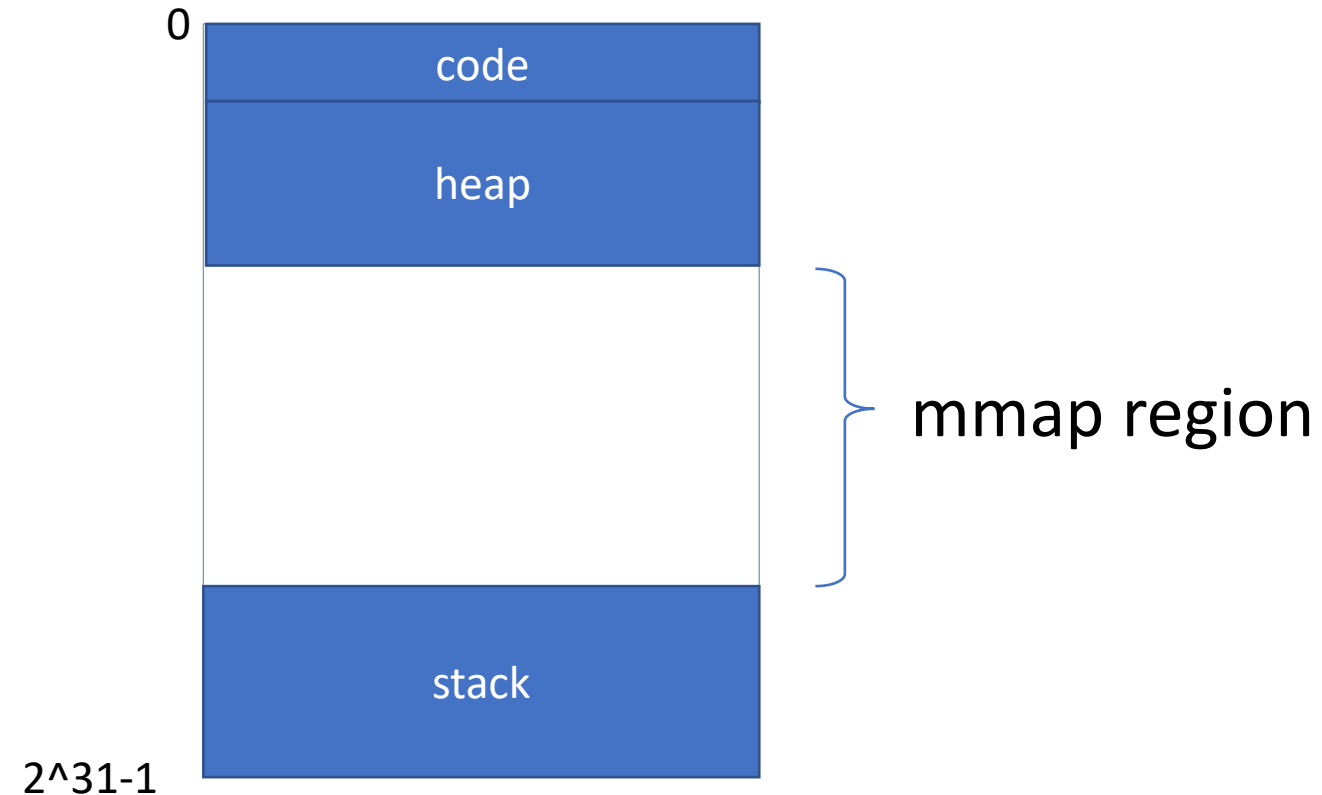
Remember this Picture?

Typical Virtual Address Space



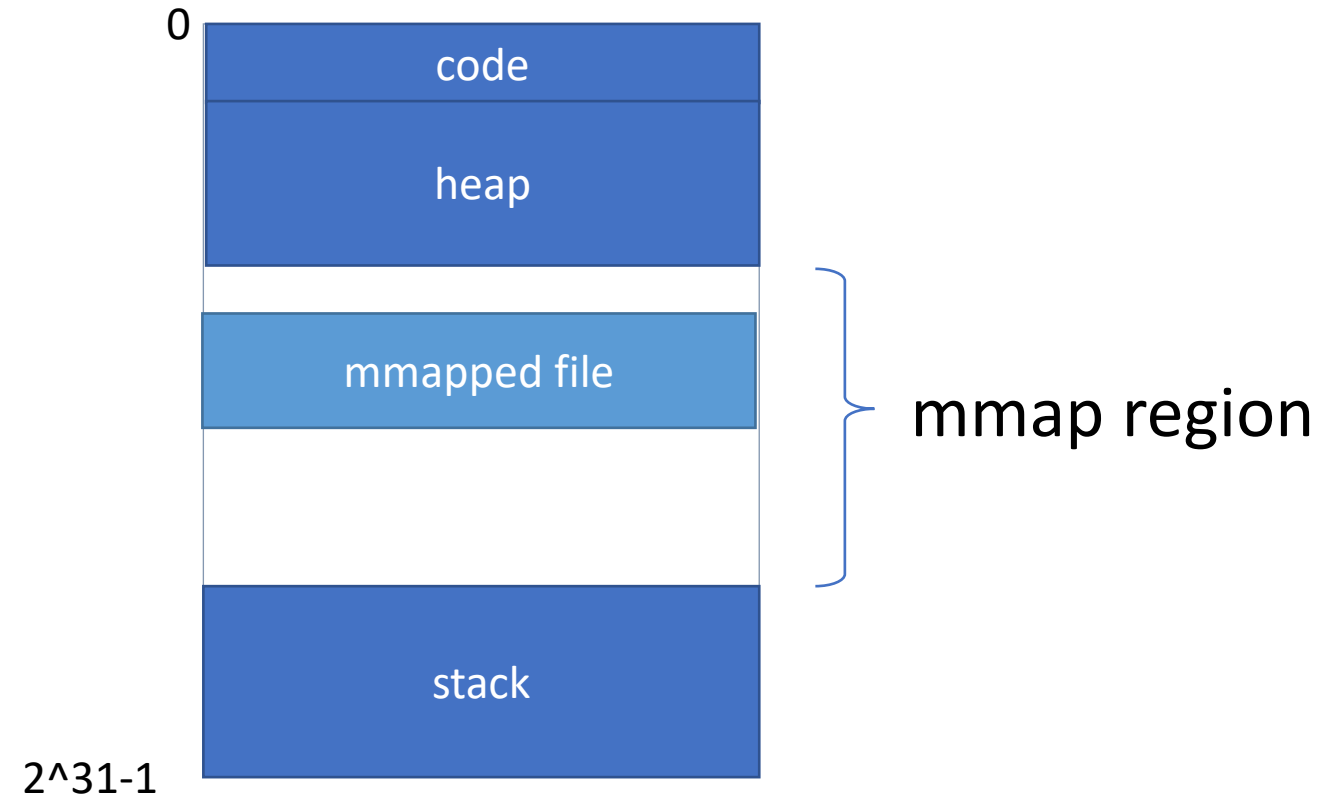
Remember this Picture?

Typical Virtual Address Space



Remember this Picture?

Typical Virtual Address Space



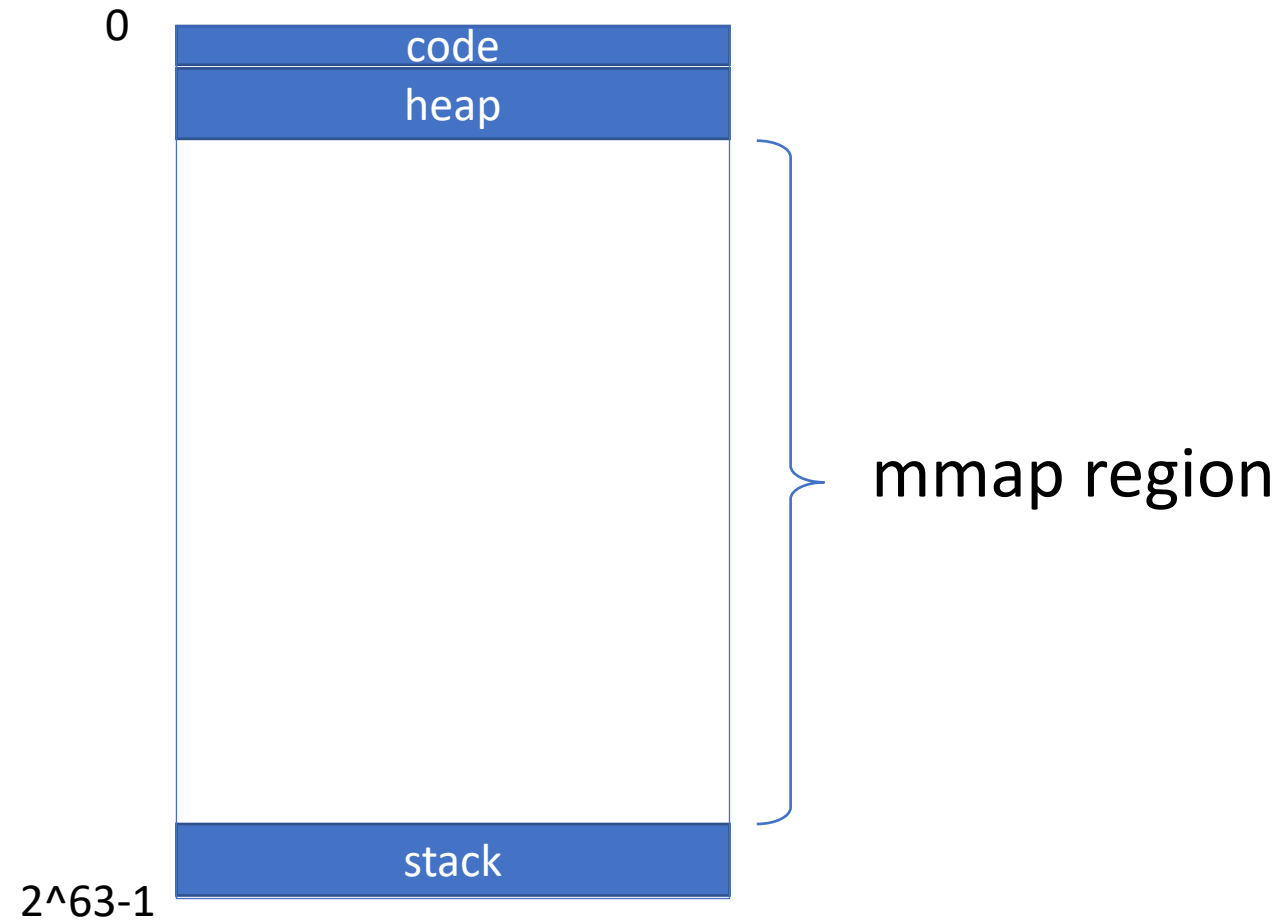
Remember Large Address Spaces?

- 64 bit address space

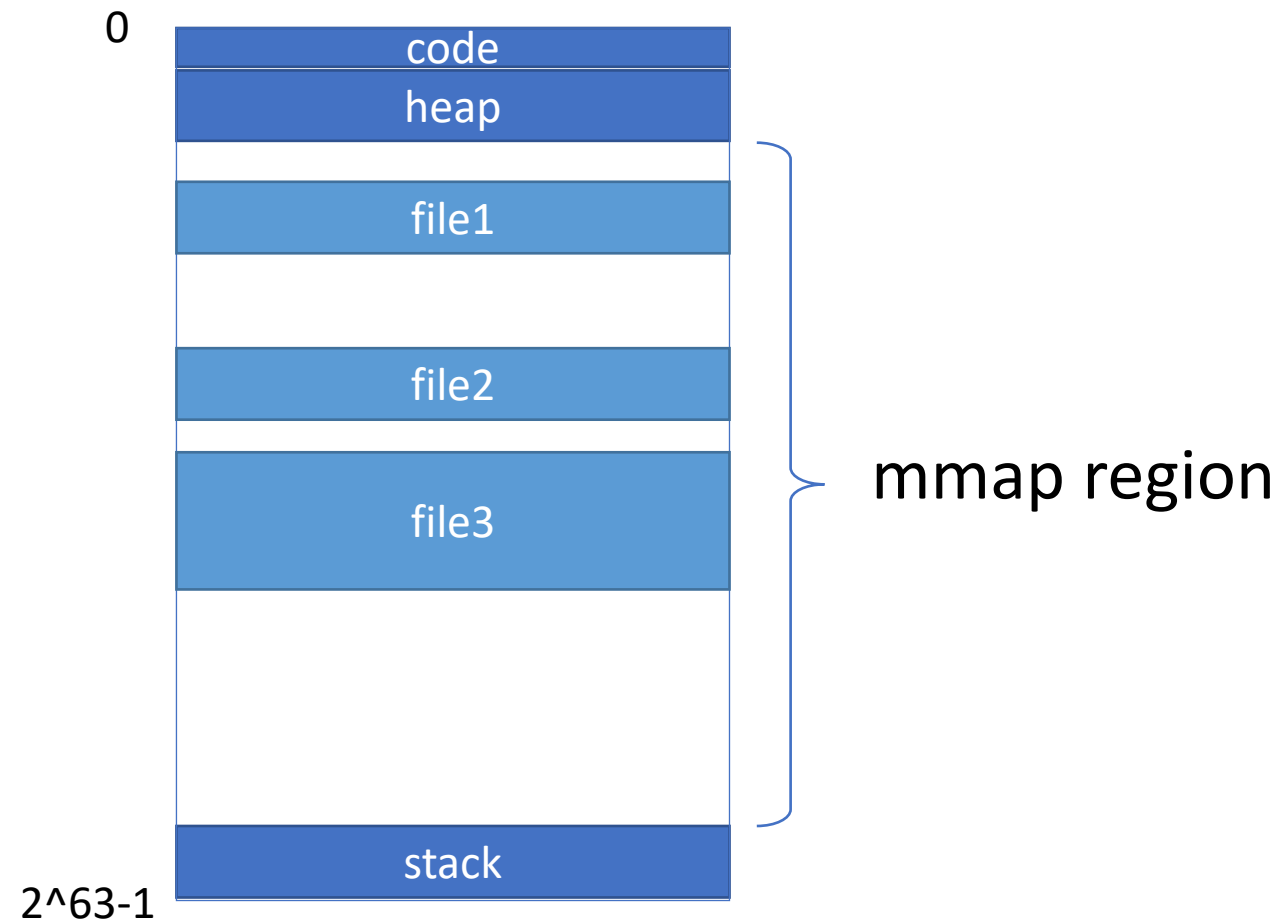
Do you know now why desirable?

- 32 bits → 4 Gbytes
- A few big files mmap()-ed
- You are out of virtual address space!

64-bit Address Space: Huge mmap() Region



Example with 3 (Large) Files Mapped



Access to mmap()-ed Files

- Access to mmap()-ed memory region
- Causes page fault
- Causes page/block of file to be brought in
 - “Backing store” for that address is the file, instead of the VMEM backing store

mmap() Implementation

- On mmap()
 - Allocate page table entries
 - Set valid bit to “invalid”

mmap() Implementation

- On mmap()
 - Allocate page table entries
 - Set valid bit to “invalid”
- On access,
 - Page fault
 - File = backing store for mapped region of memory
 - Just like in demand paging
 - Except paged from mapped file

mmap() Implementation

- On mmap()
 - Allocate page table entries
 - Set valid bit to “invalid”
- On access,
 - Page fault
 - File = backing store for mapped region of memory
 - Just like in demand paging
 - Except paged from mapped file
- After page fault handling
 - Set valid bit to true

How to get data to disk for mmap?

- Through normal page replacement
- Or through an explicit call *msync()*

What is mmap() good for?

- Random access to large file

Random Access with mmap()

- `addr = mmap()`
- Use memory addresses in `[addr, addr+len-1]`

Random Access with Read() Interface

- Open
- Read entire file into memory buffer
- Then use memory address in buffer

Advantage with mmap()

- Only accessed portions brought in memory
- Huge advantage
 - **For large files**
 - **Sparsely accessed**

Random Access with Seek()

- Open
- Seek
- Read into Buffer
- Seek
- Read into Buffer

Advantage with mmap()

- Much easier programming model
 - Follow pointer in memory
 - As opposed to (Seek, Read) every time
- Easier if reuse
 - VM system keeps page for you
 - Otherwise, have to do your own replacement

mmap() Advantages for Random Access

- Easy to write
- Only bring in memory what you read
- Easy reuse
- Mapping the same file twice causes page sharing
 - Could be helpful in some cases

Issues with mmap()

- Alignment on page boundary
- Not easy to extend a file
- For small files
 - Read() more efficient than mmap() + page fault
- Mapping the same file twice causes page sharing
 - Can be extremely confusing

Summary – Key Concepts

- File system “mental model”
 - Data structures : on disk, in memory
 - File data allocation methods
 - Contiguous, extent-based, linked, FAT, indexed, indirect blocks
 - File access methods
 - Create, open, write, read, close
- Setting up the file system
 - Partitioning, mounting, boot
- Memory-mapped files

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 40, 41, 45.

<https://pages.cs.wisc.edu/~remzi/OSTEP/>

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).