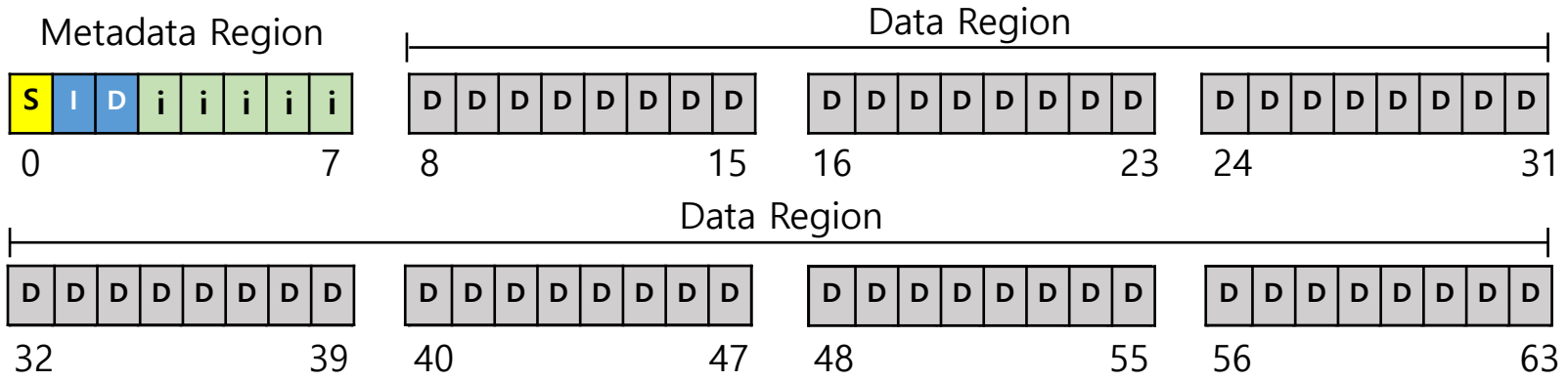
Week 11

Persistent Storage: Basic File System Implementation (Part 2)

Oana Balmau

March 16, 2023

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

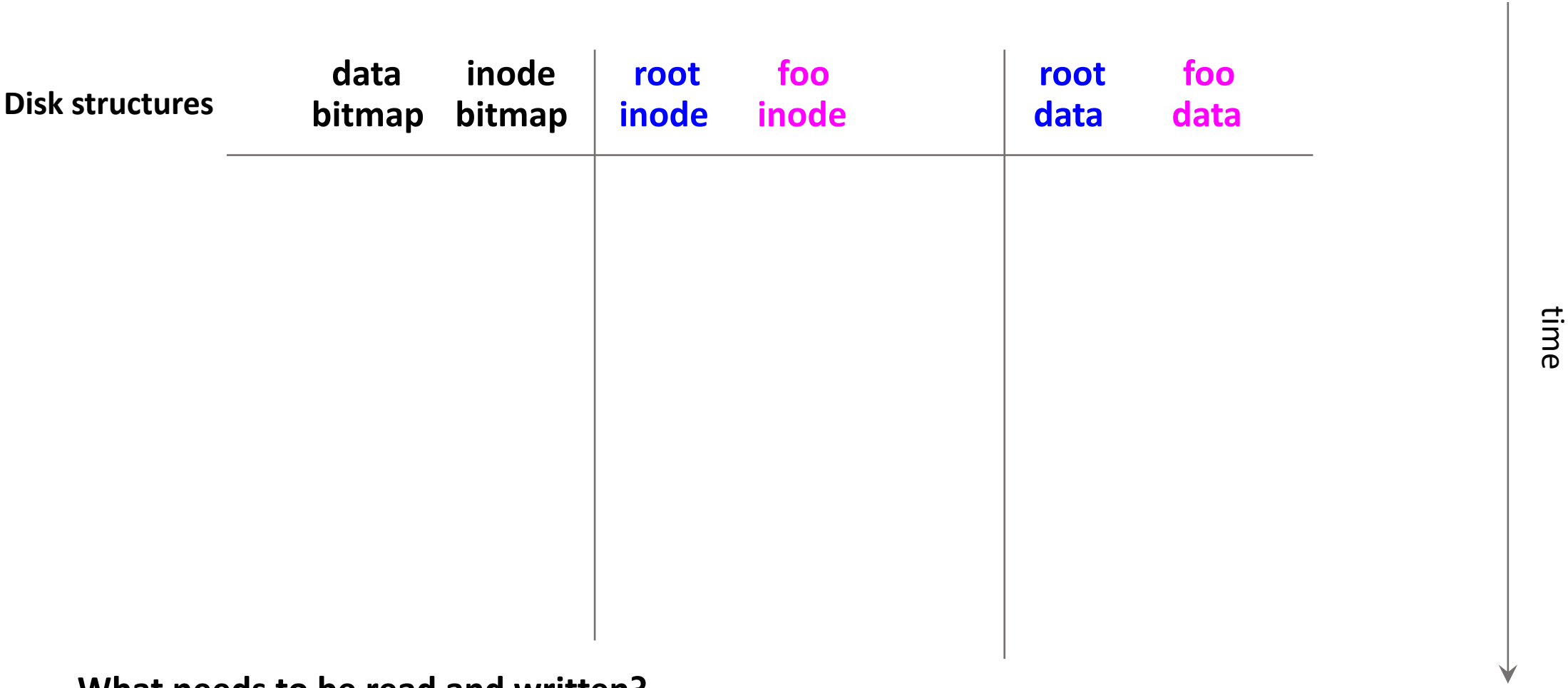
2. **Access methods**
   - How do we open(), read(), write() ?

Remember: File Systems Main Access Methods

- Create
- Open
- Write
- Read
- Close

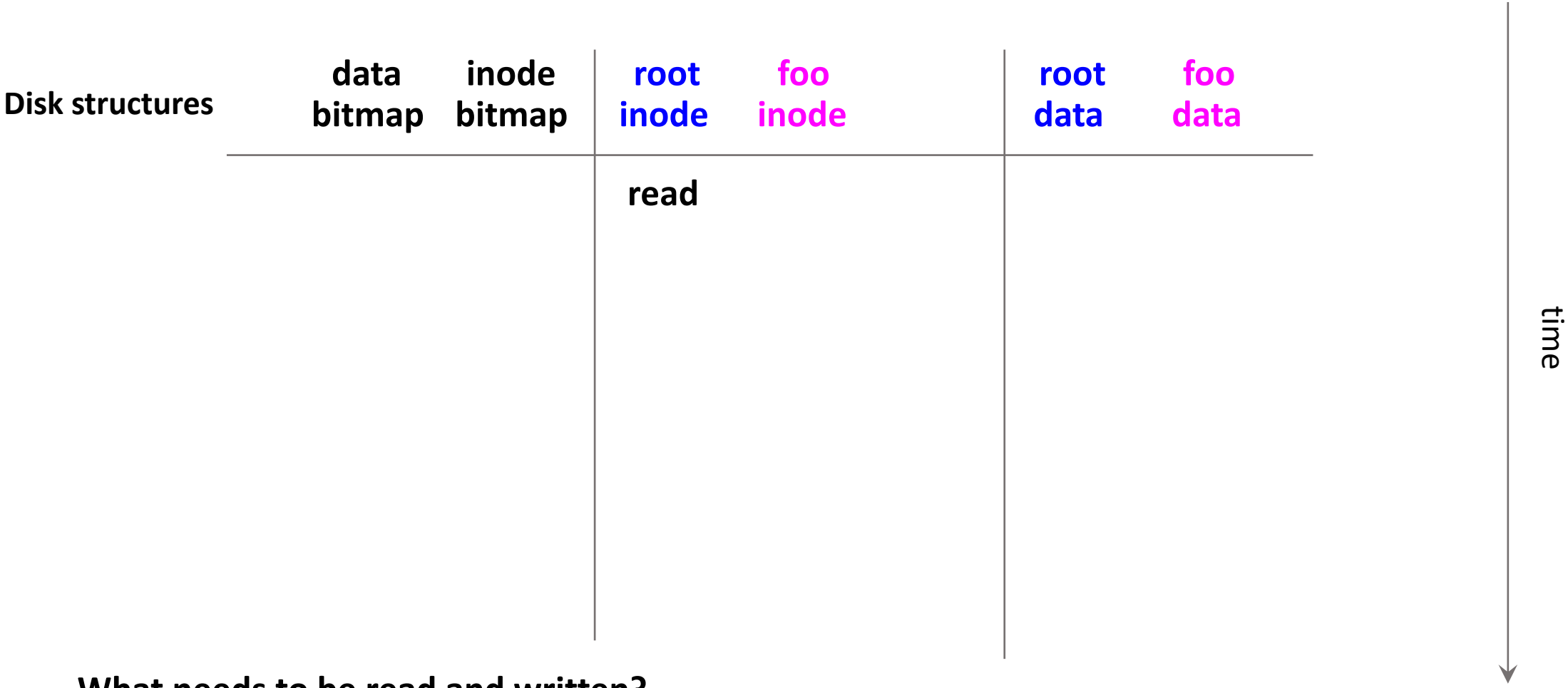
- With some major simplifications
  - No access permission checks, no return value checks, etc.

Create `/foo/bar`



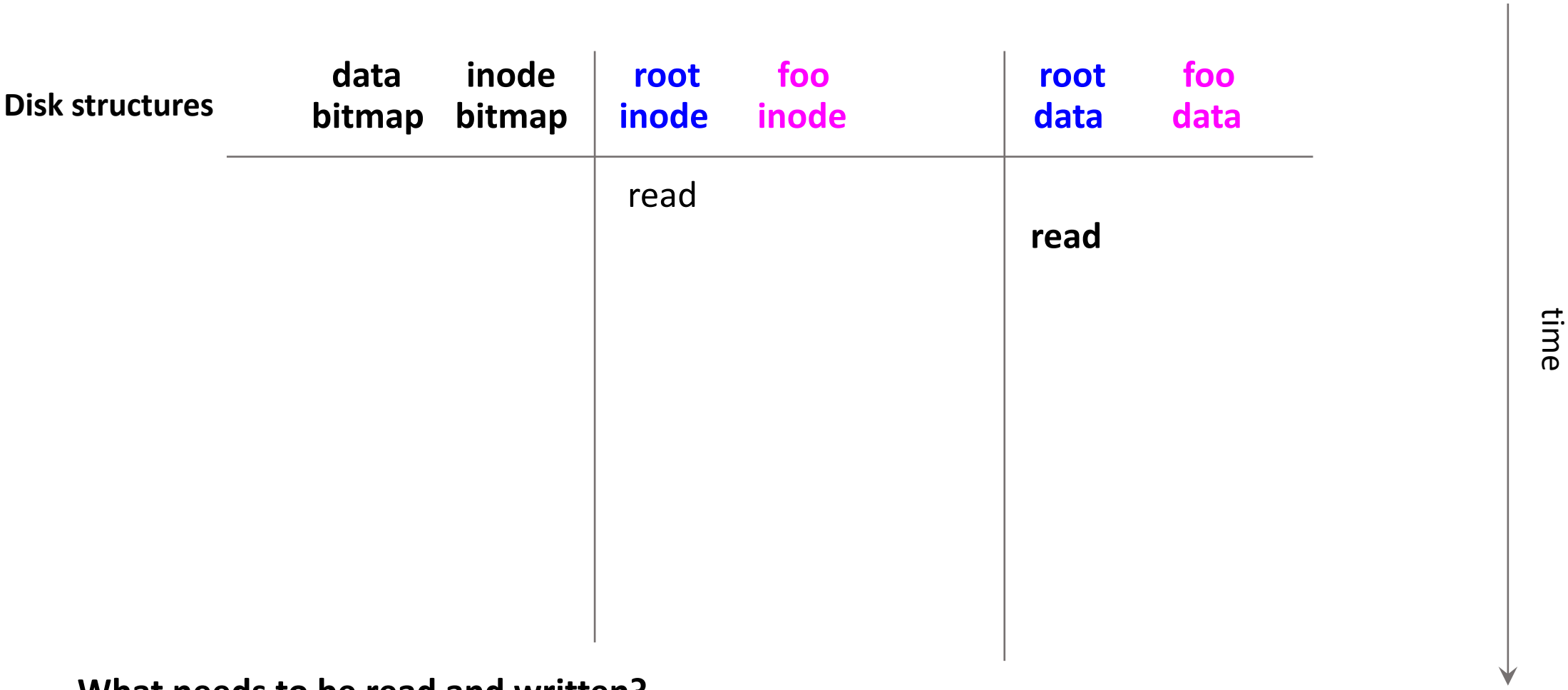
What needs to be read and written?

Create `/foo/bar`



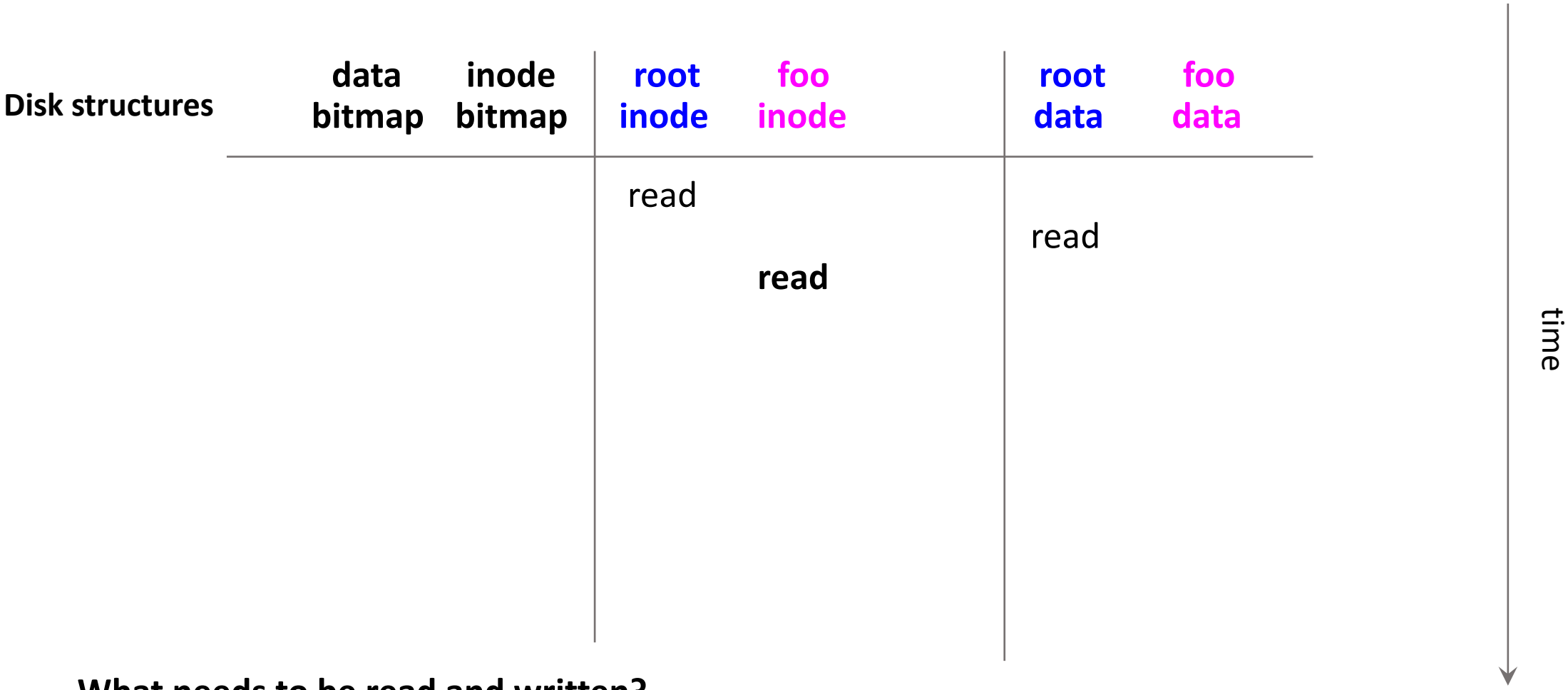
What needs to be read and written?

Create `/foo/bar`



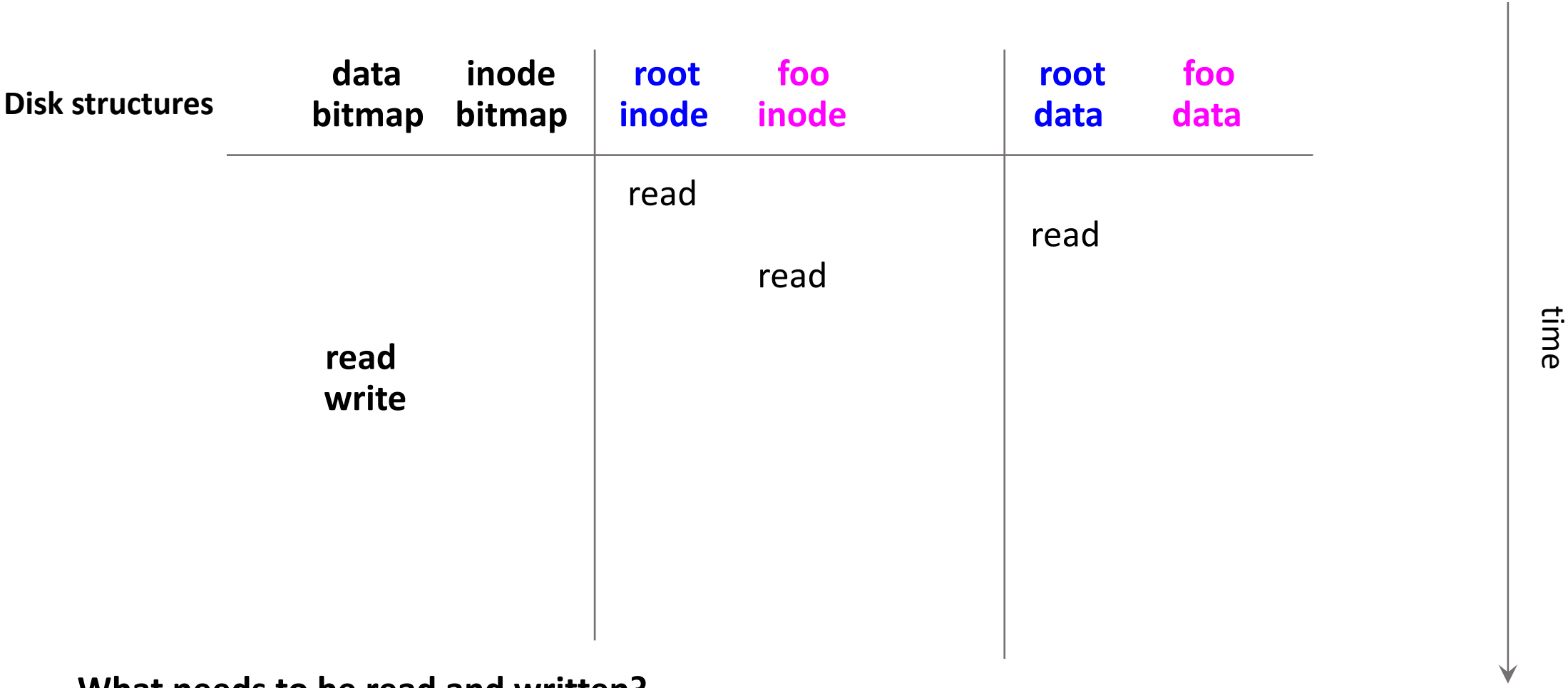
What needs to be read and written?

Create `/foo/bar`



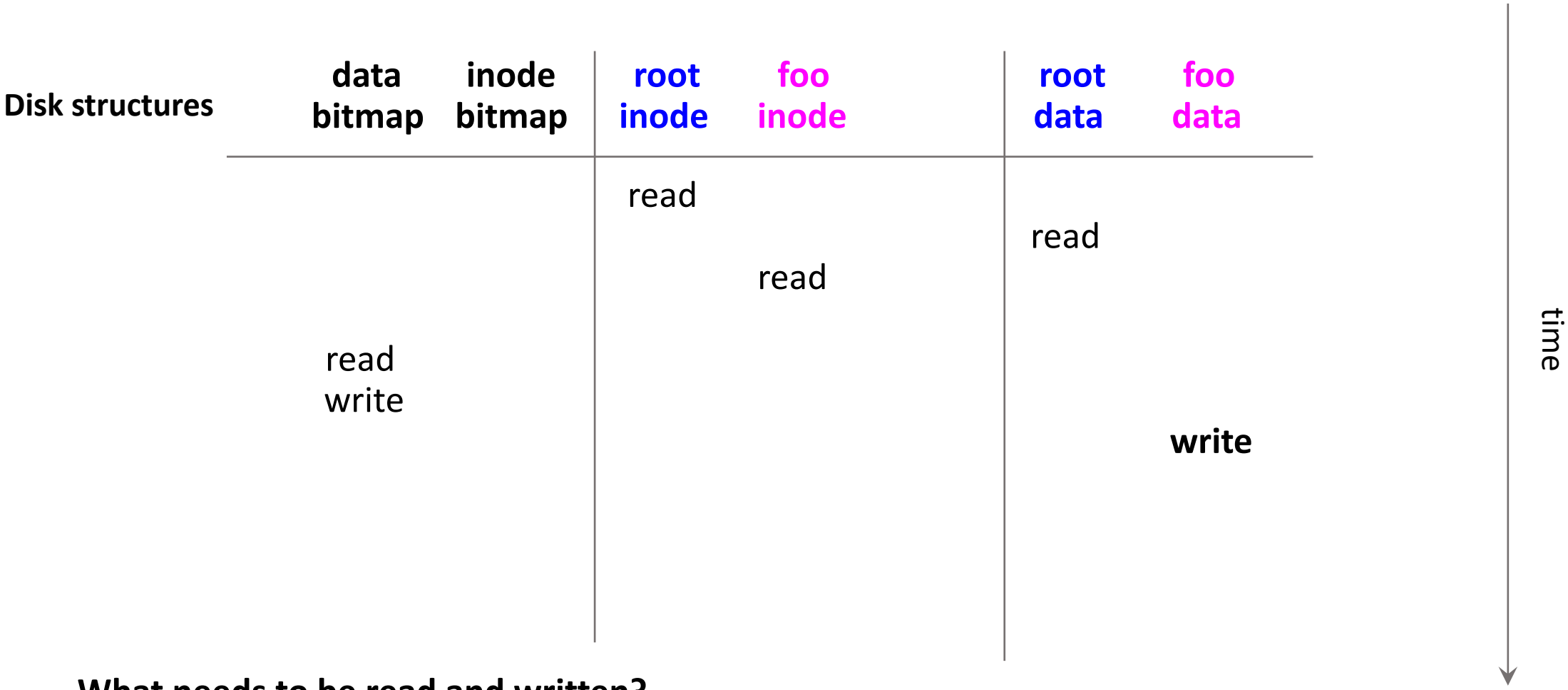
What needs to be read and written?

Create `/foo/bar`



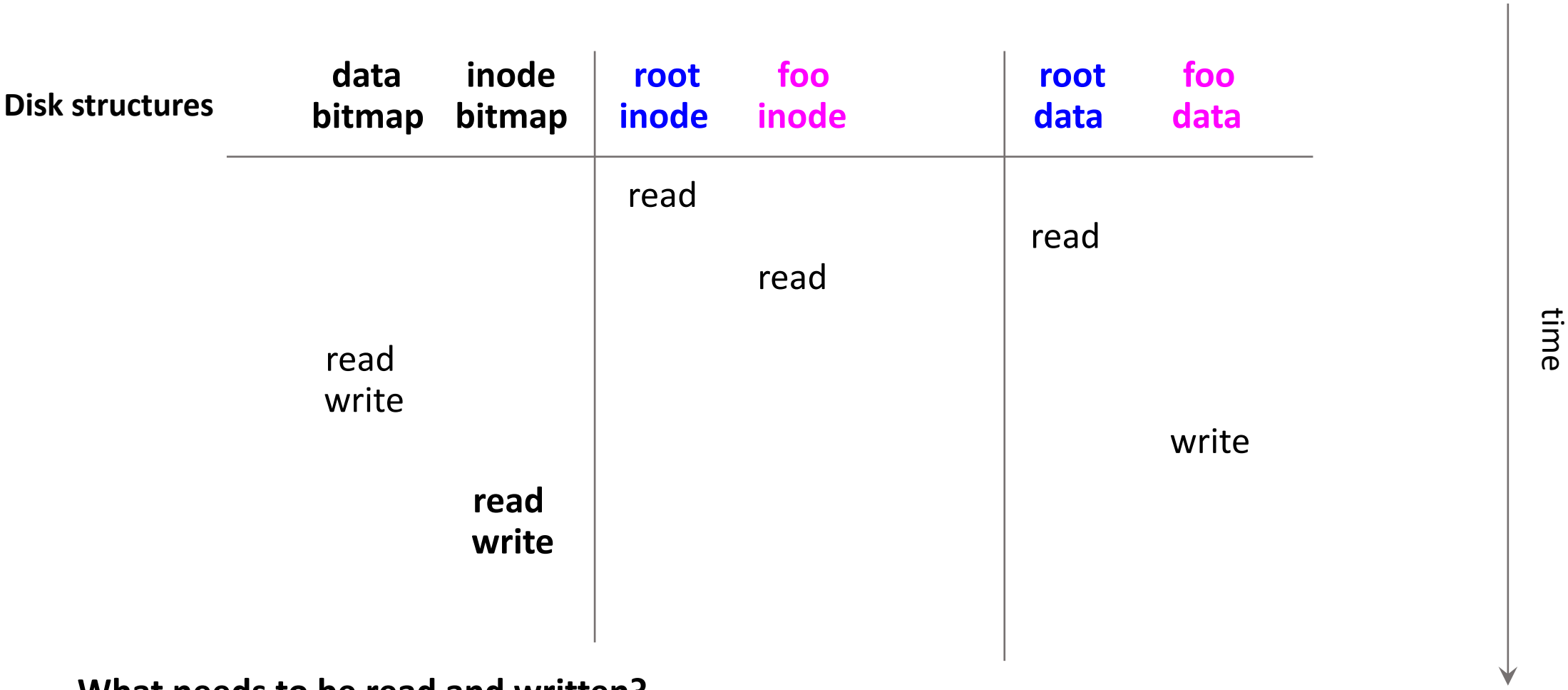
What needs to be read and written?

Create `/foo/bar`



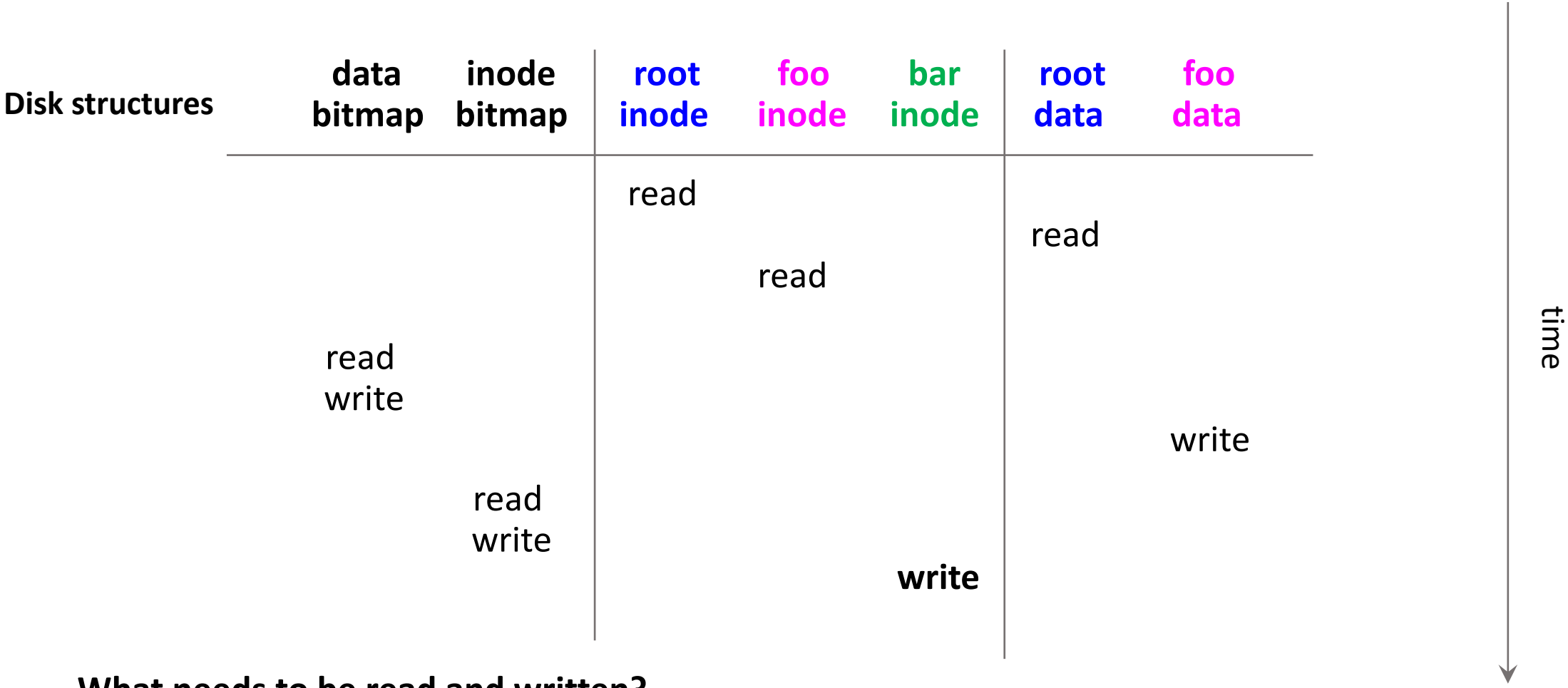
What needs to be read and written?

Create `/foo/bar`



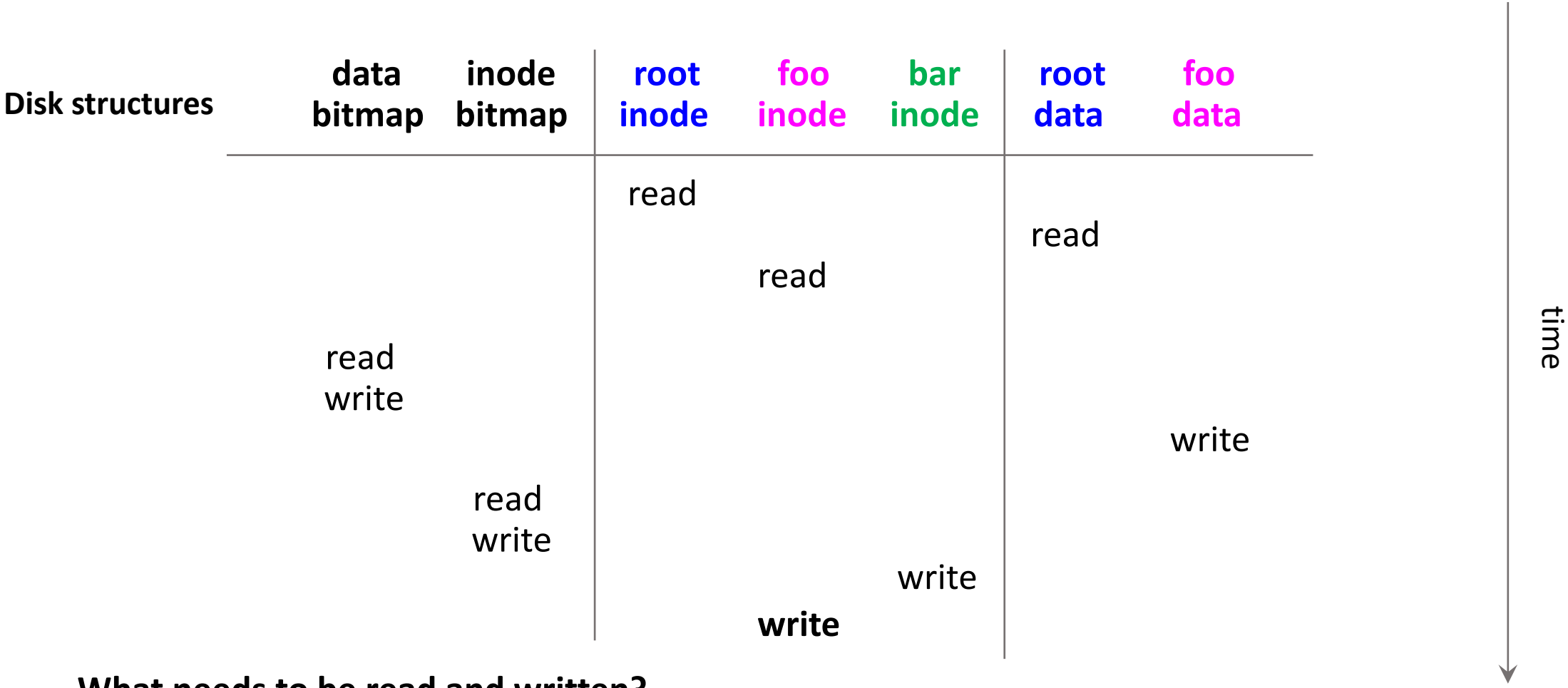
What needs to be read and written?

Create `/foo/bar`



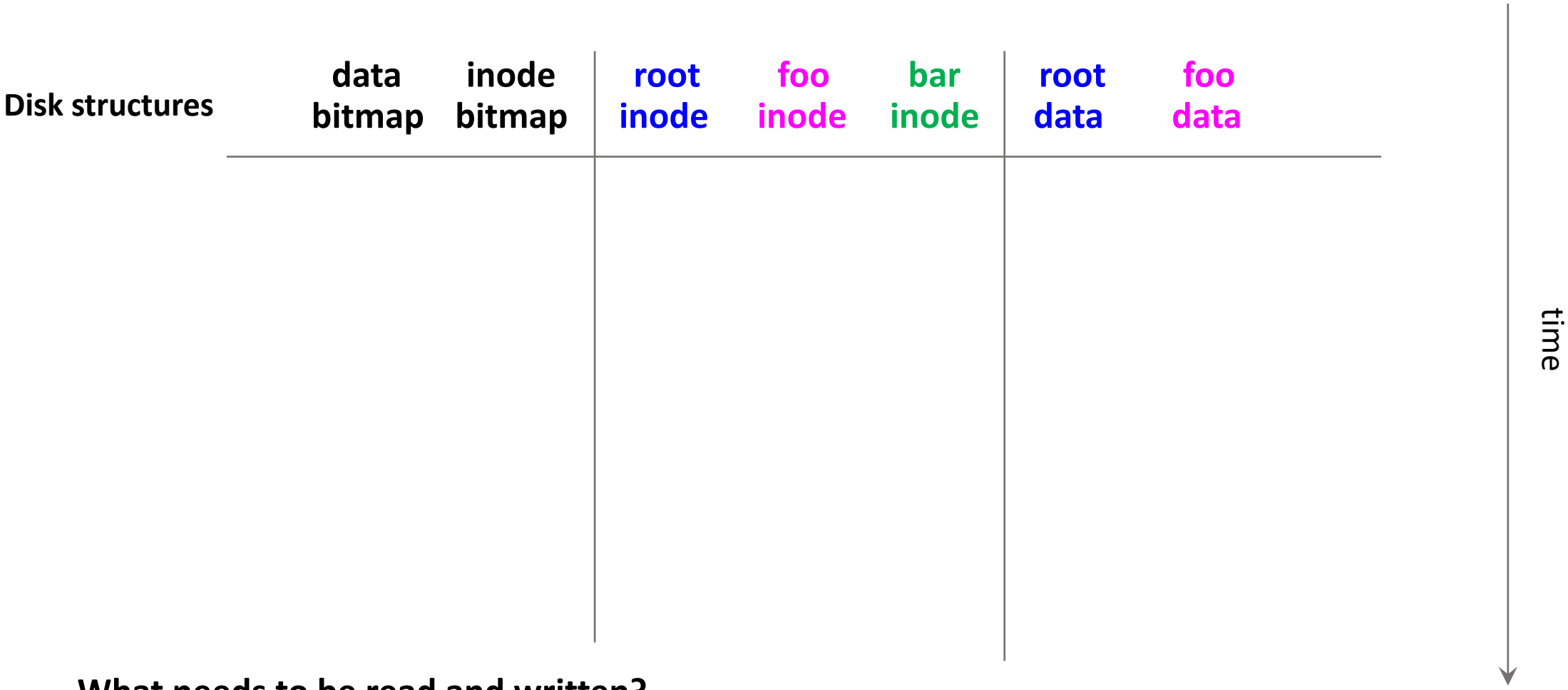
What needs to be read and written?

Create `/foo/bar`



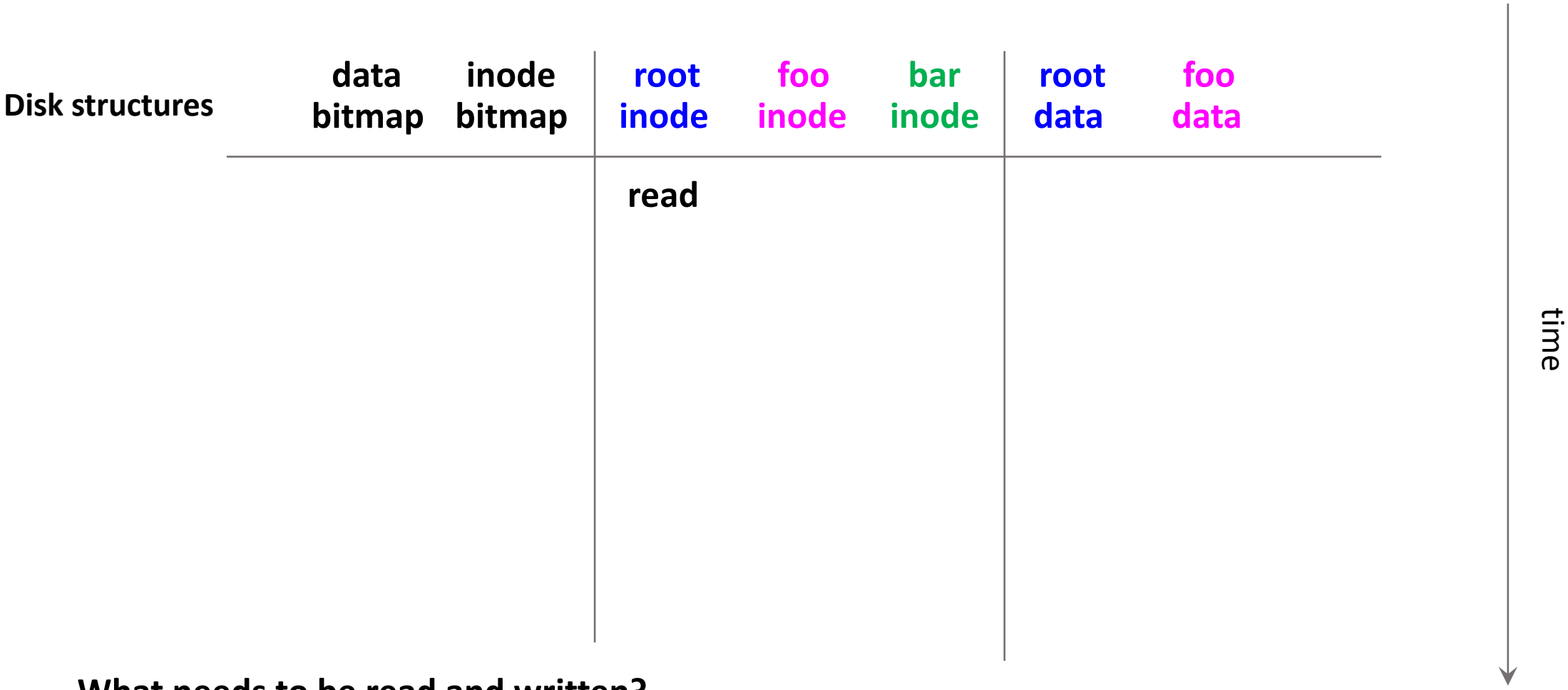
What needs to be read and written?

Open `/foo/bar`



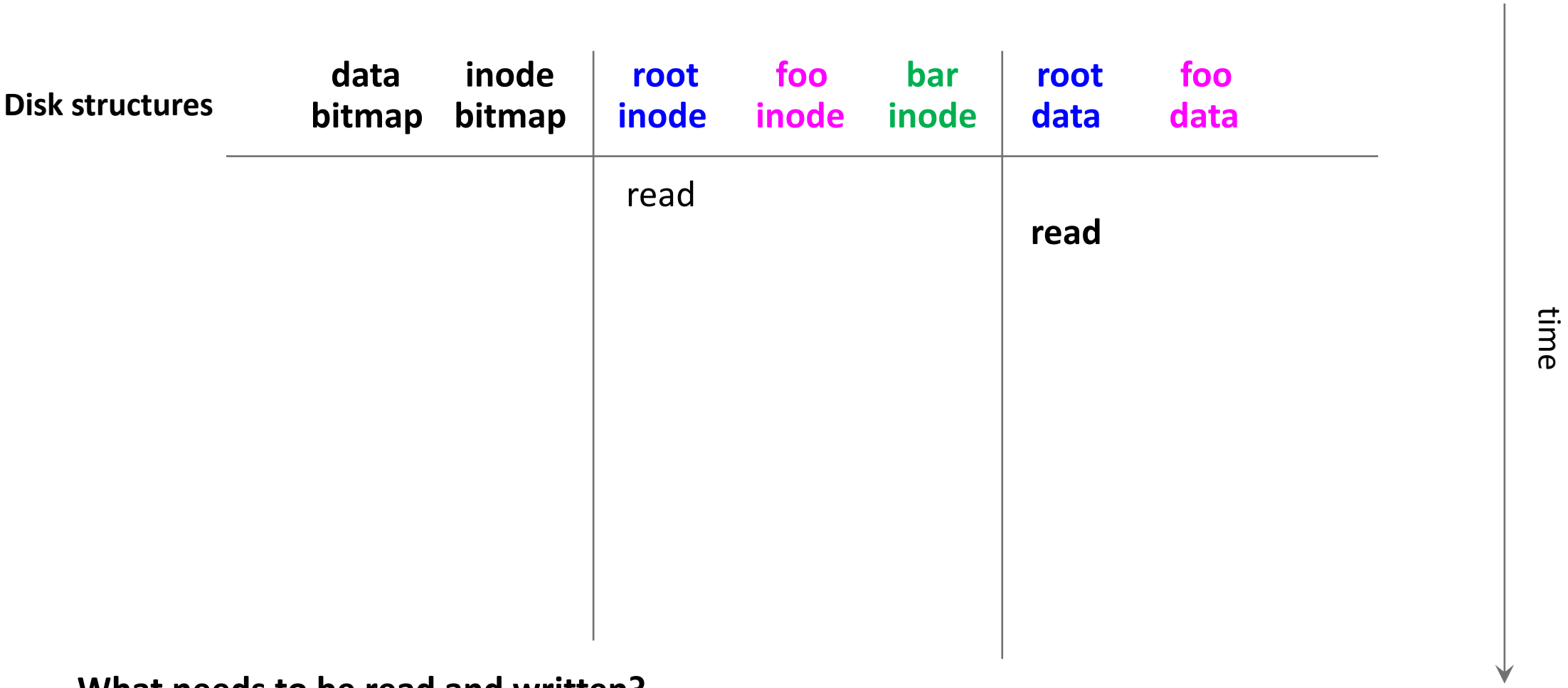
What needs to be read and written?

Open `/foo/bar`



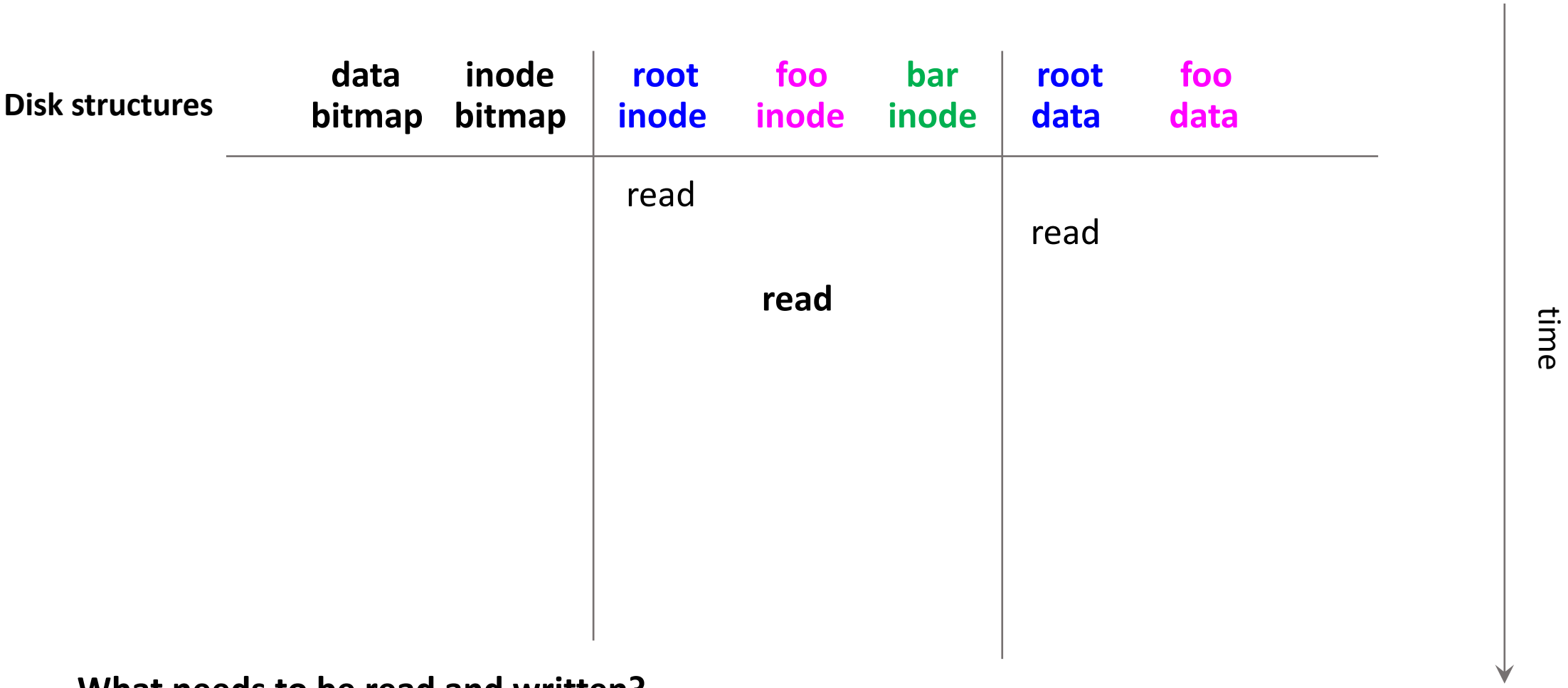
What needs to be read and written?

Open `/foo/bar`



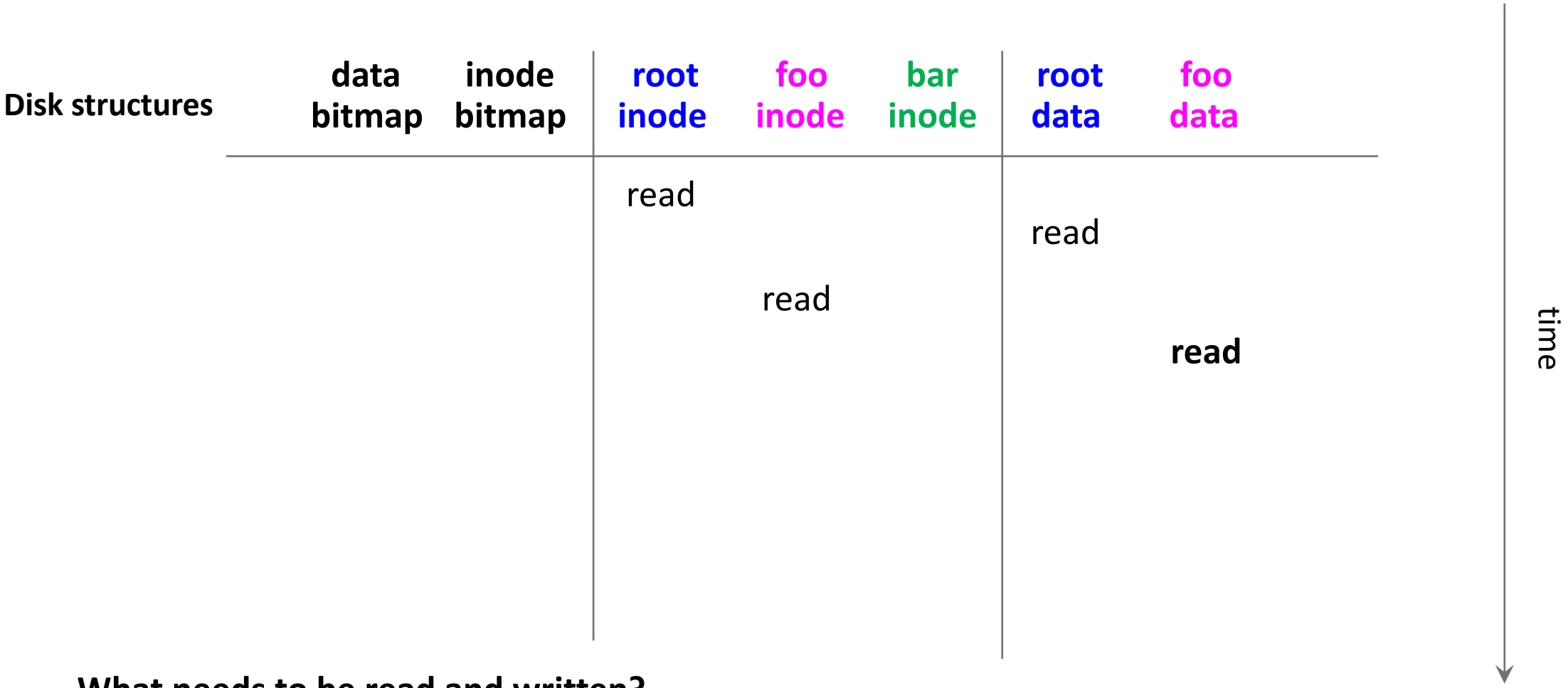
What needs to be read and written?

Open `/foo/bar`



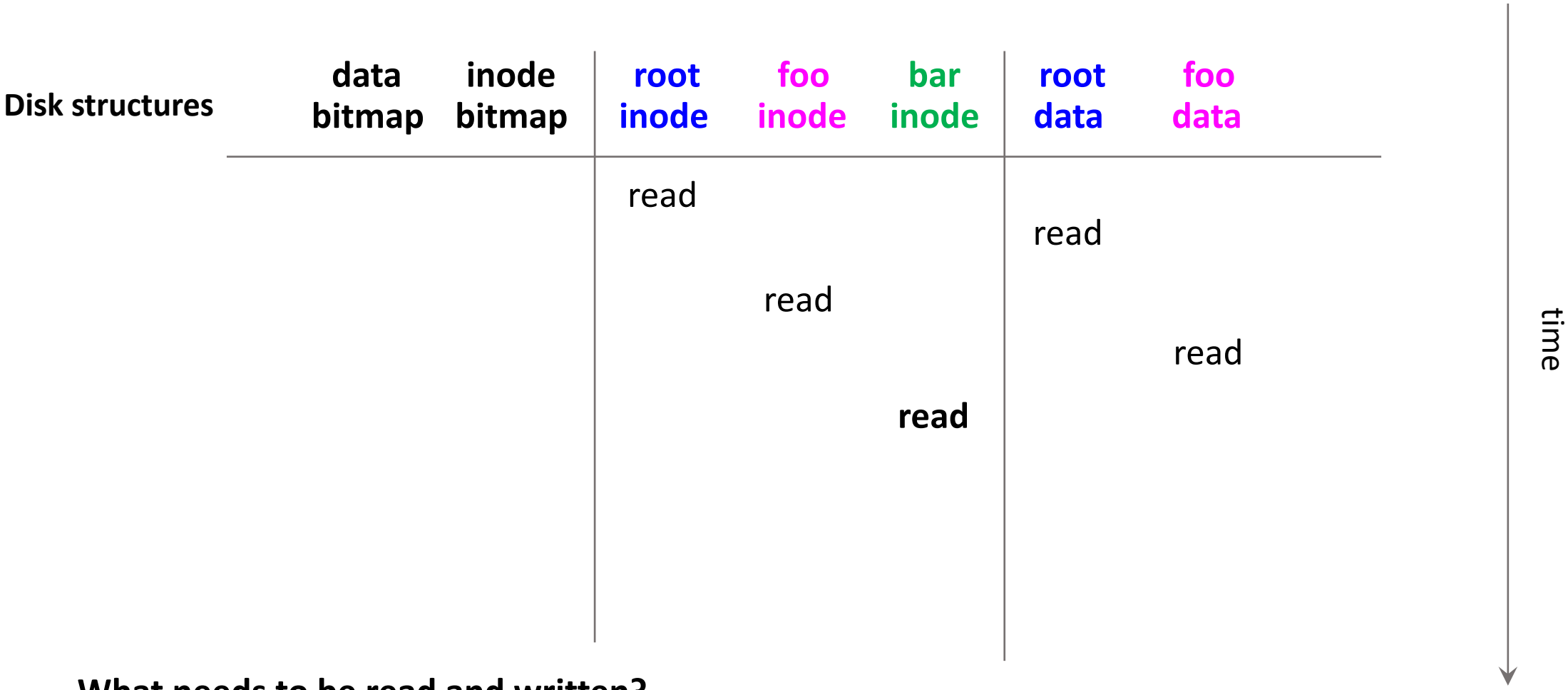
What needs to be read and written?

Open `/foo/bar`



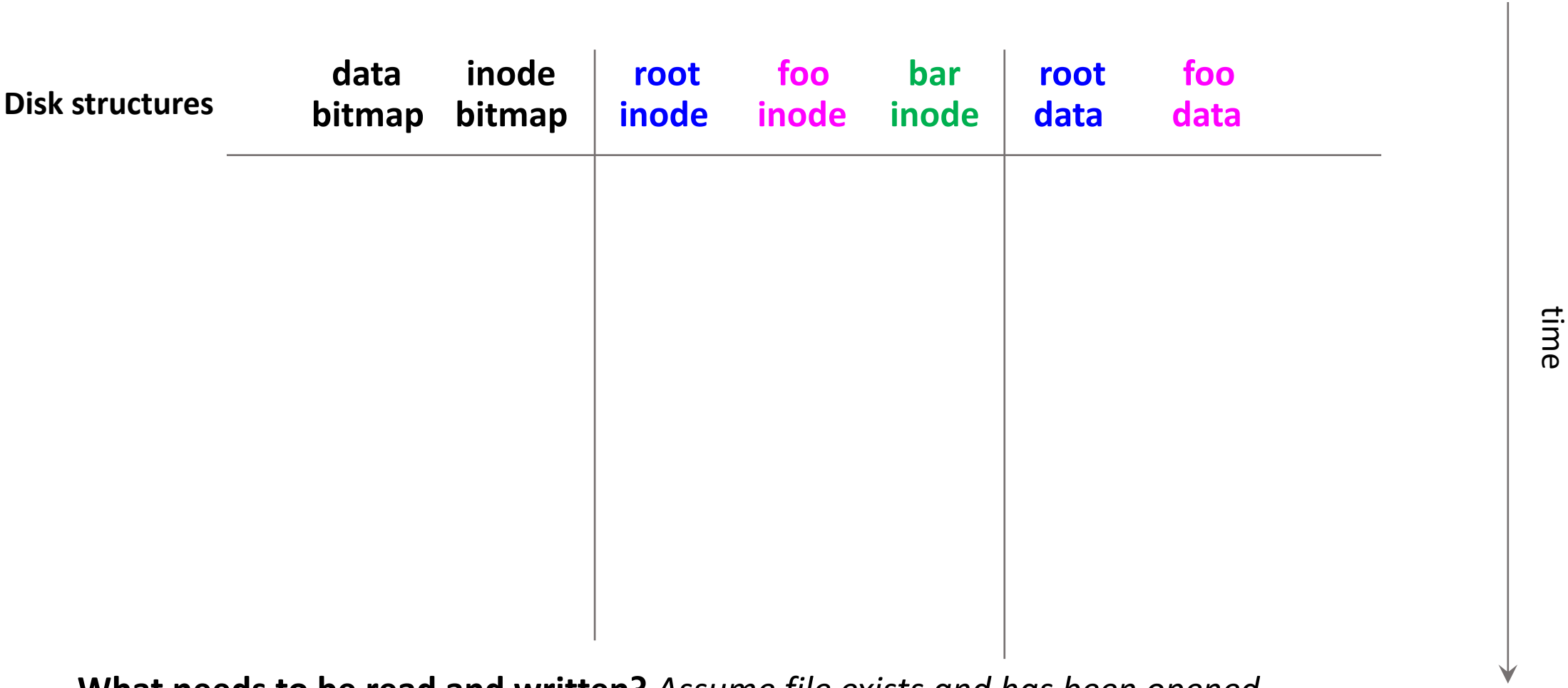
What needs to be read and written?

Open `/foo/bar`



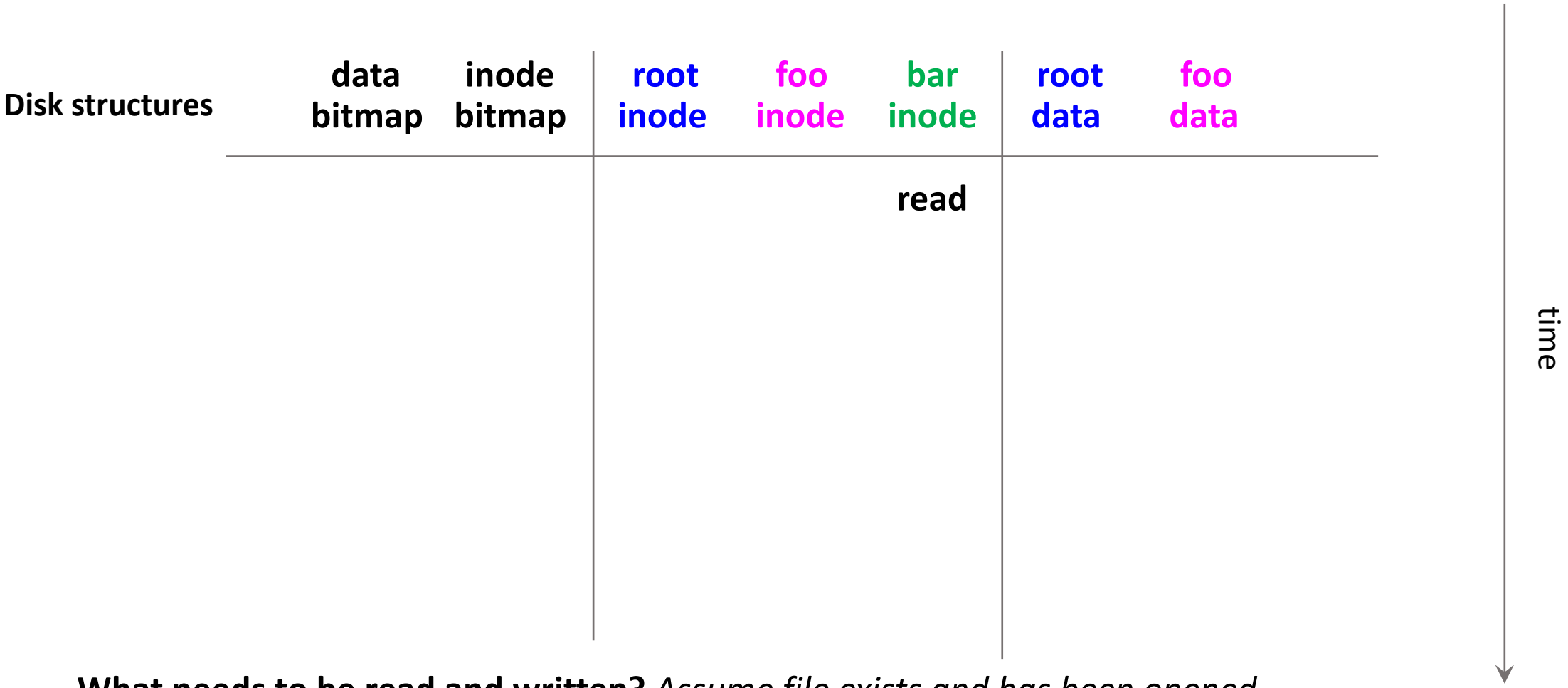
What needs to be read and written?

Write to `/foo/bar`



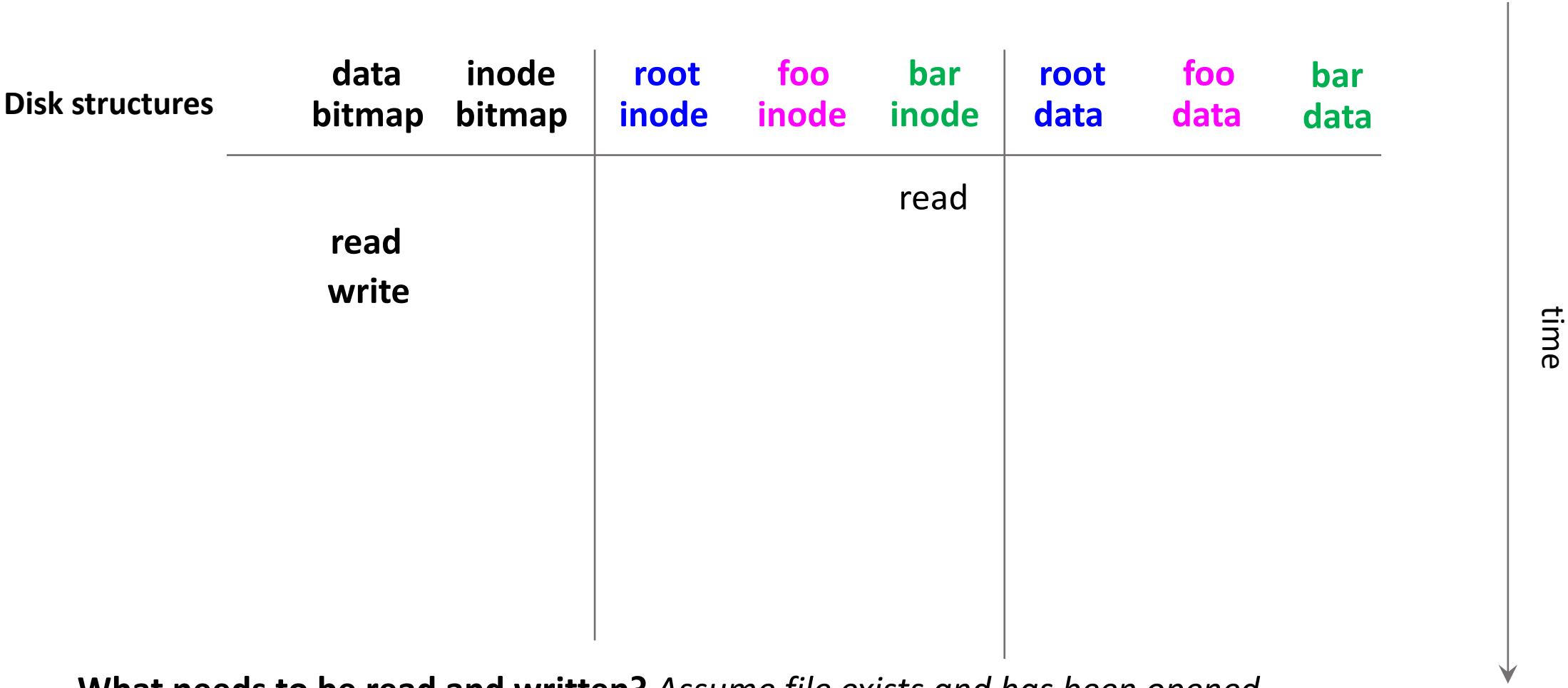
What needs to be read and written? *Assume file exists and has been opened*

Write to `/foo/bar`



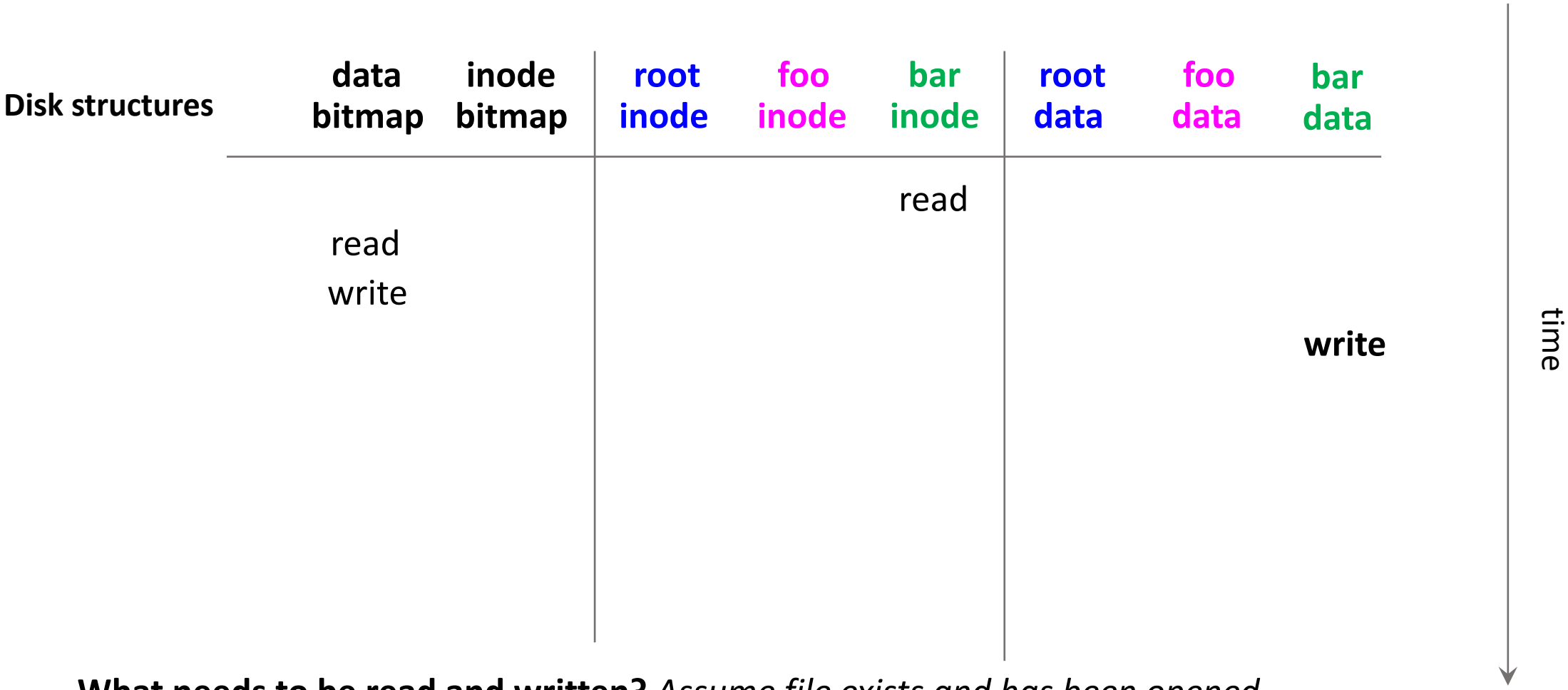
What needs to be read and written? *Assume file exists and has been opened*

Write to `/foo/bar`



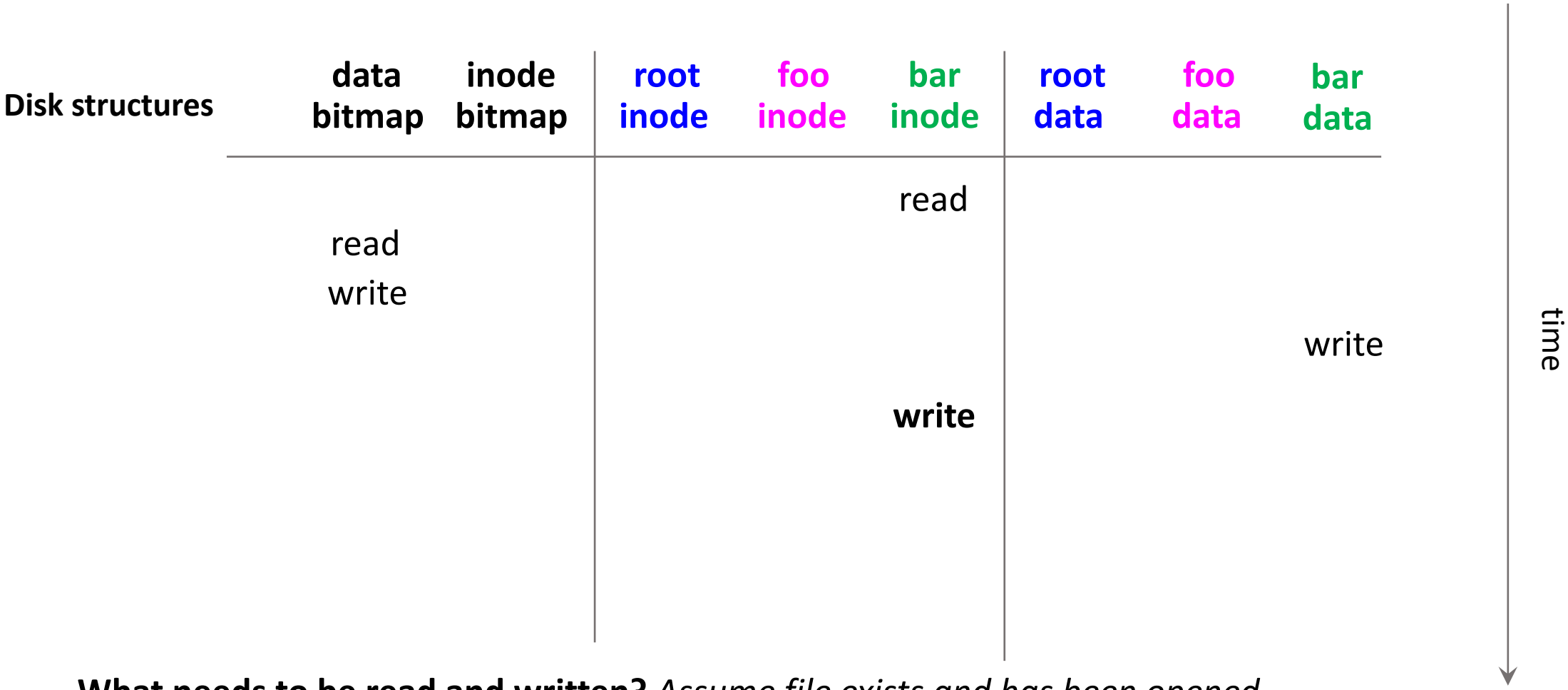
What needs to be read and written? *Assume file exists and has been opened*

Write to `/foo/bar`



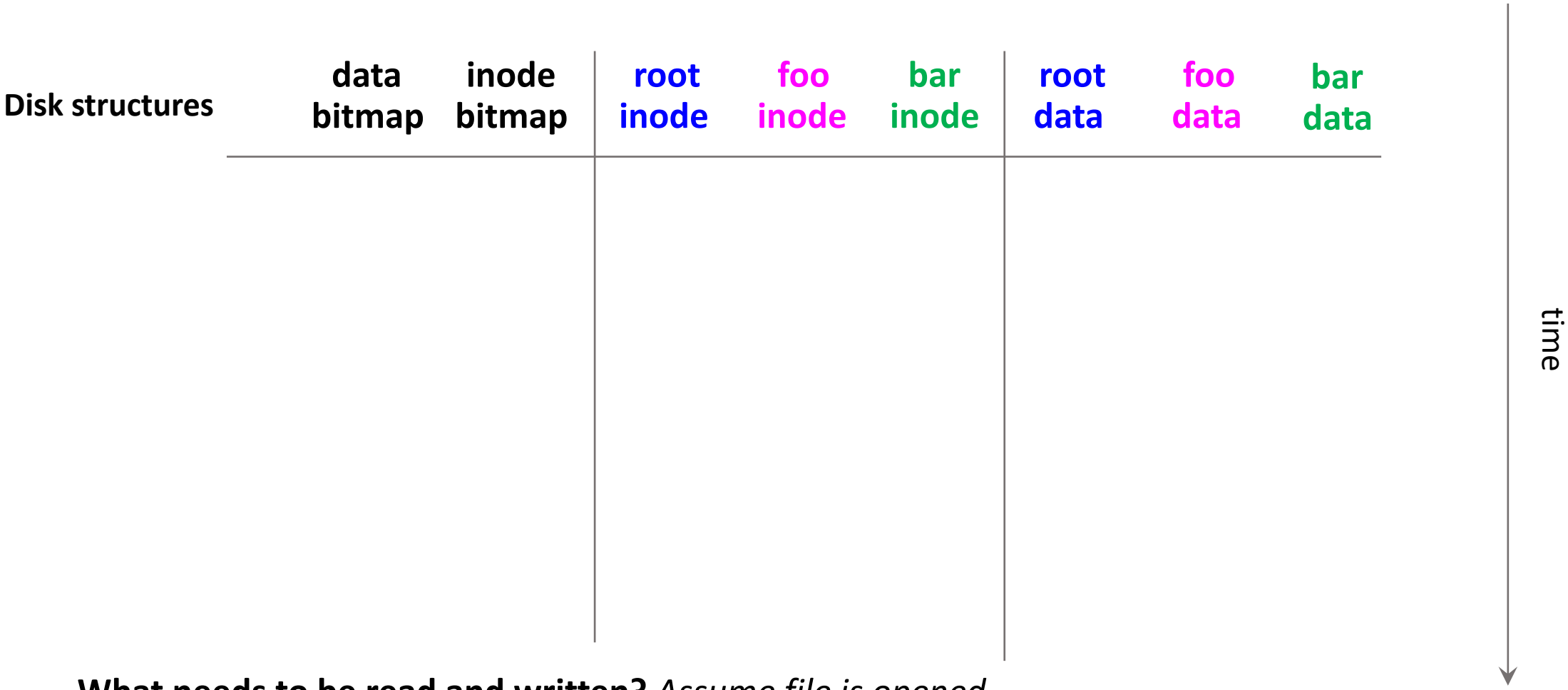
What needs to be read and written? *Assume file exists and has been opened*

Write to `/foo/bar`



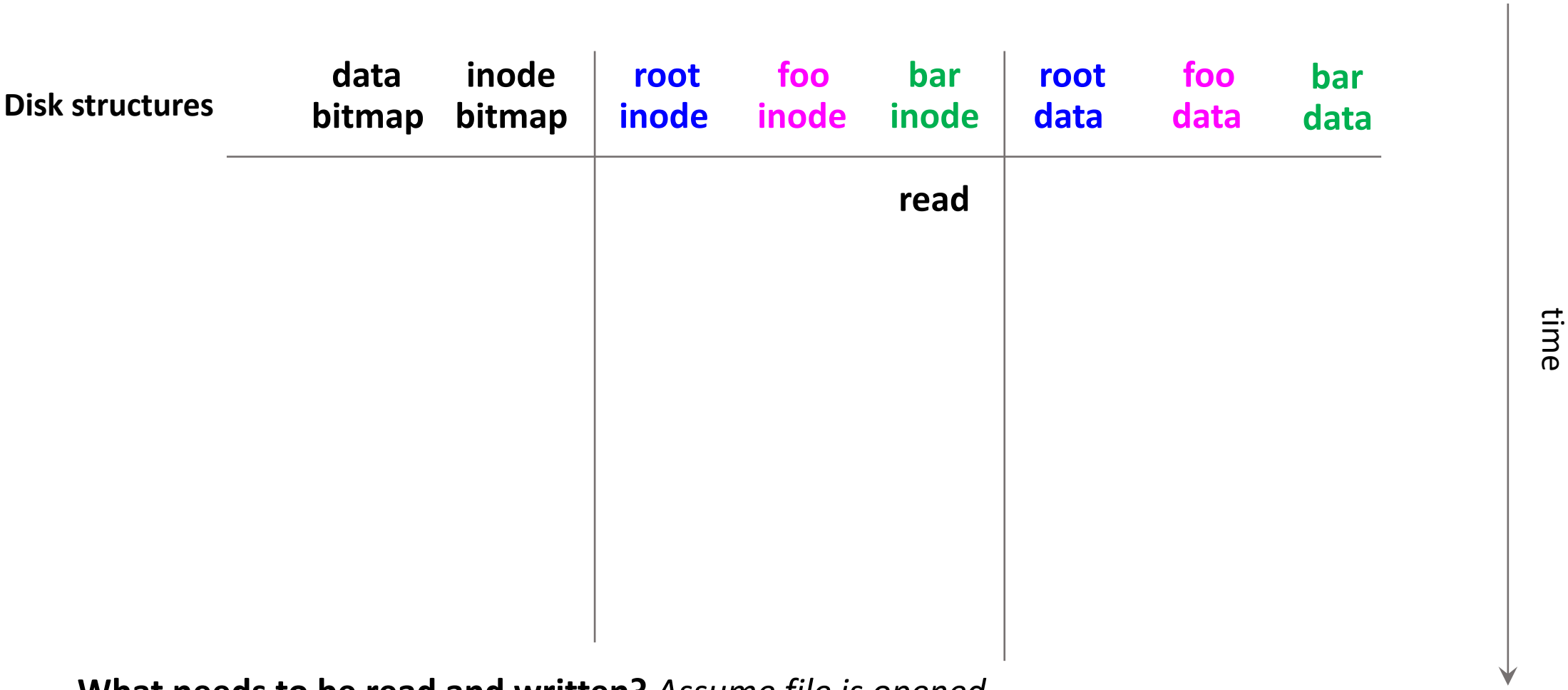
What needs to be read and written? *Assume file exists and has been opened*

Read `/foo/bar`



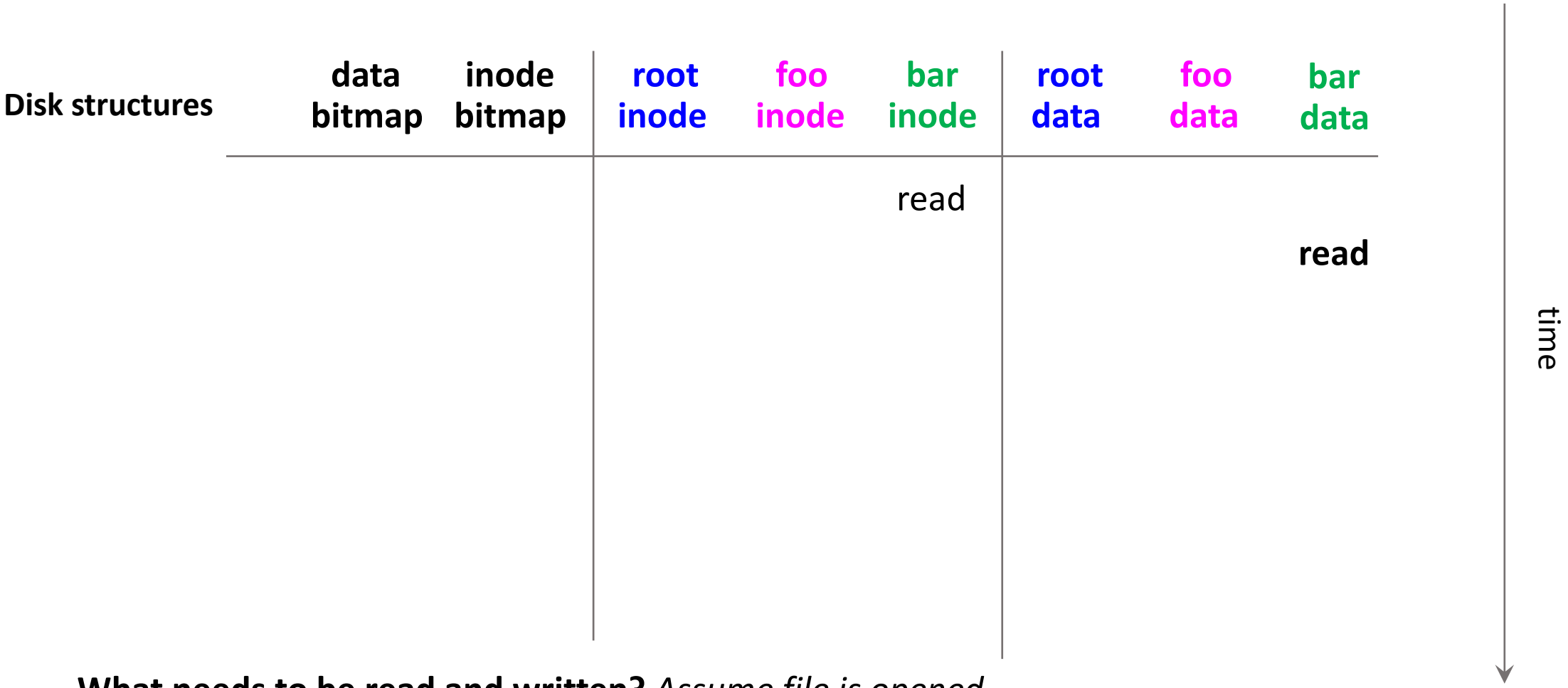
What needs to be read and written? *Assume file is opened*

Read `/foo/bar`



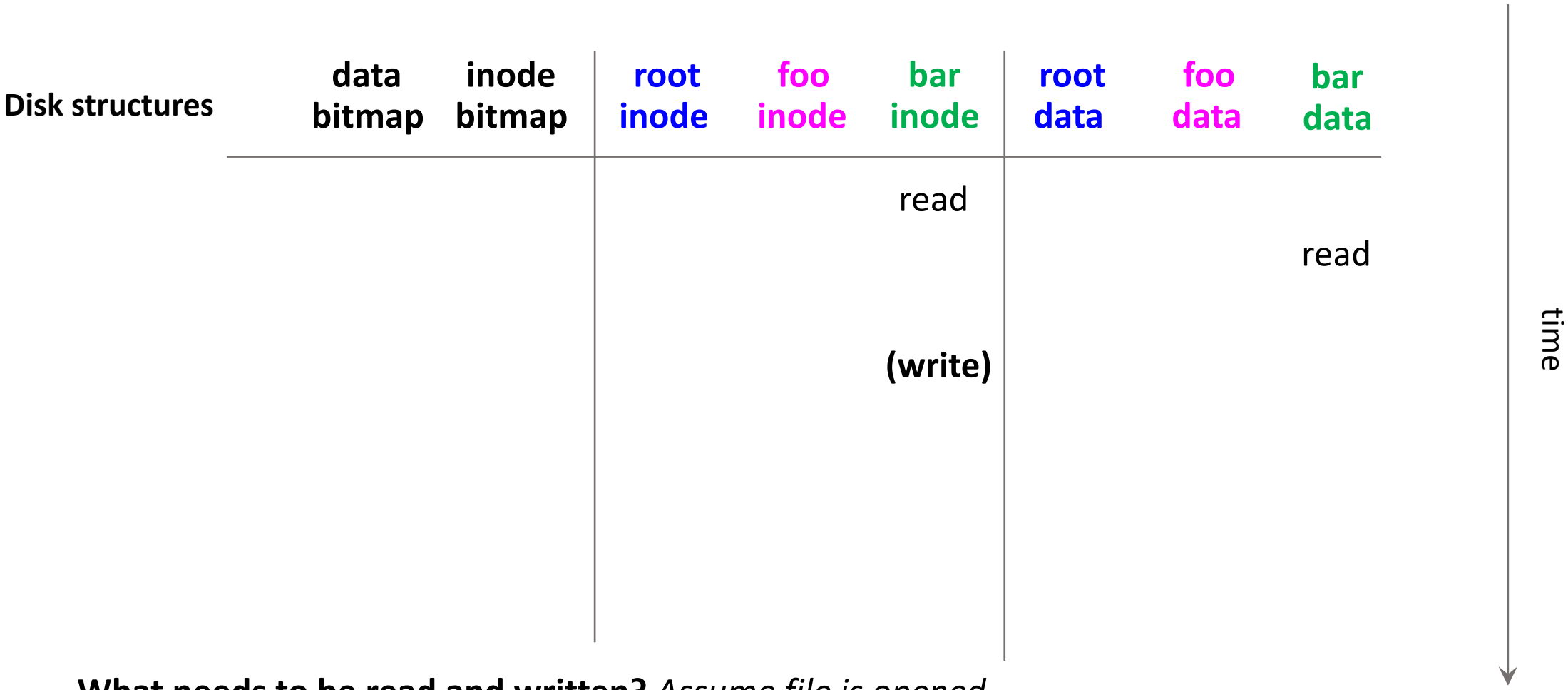
What needs to be read and written? *Assume file is opened*

Read `/foo/bar`



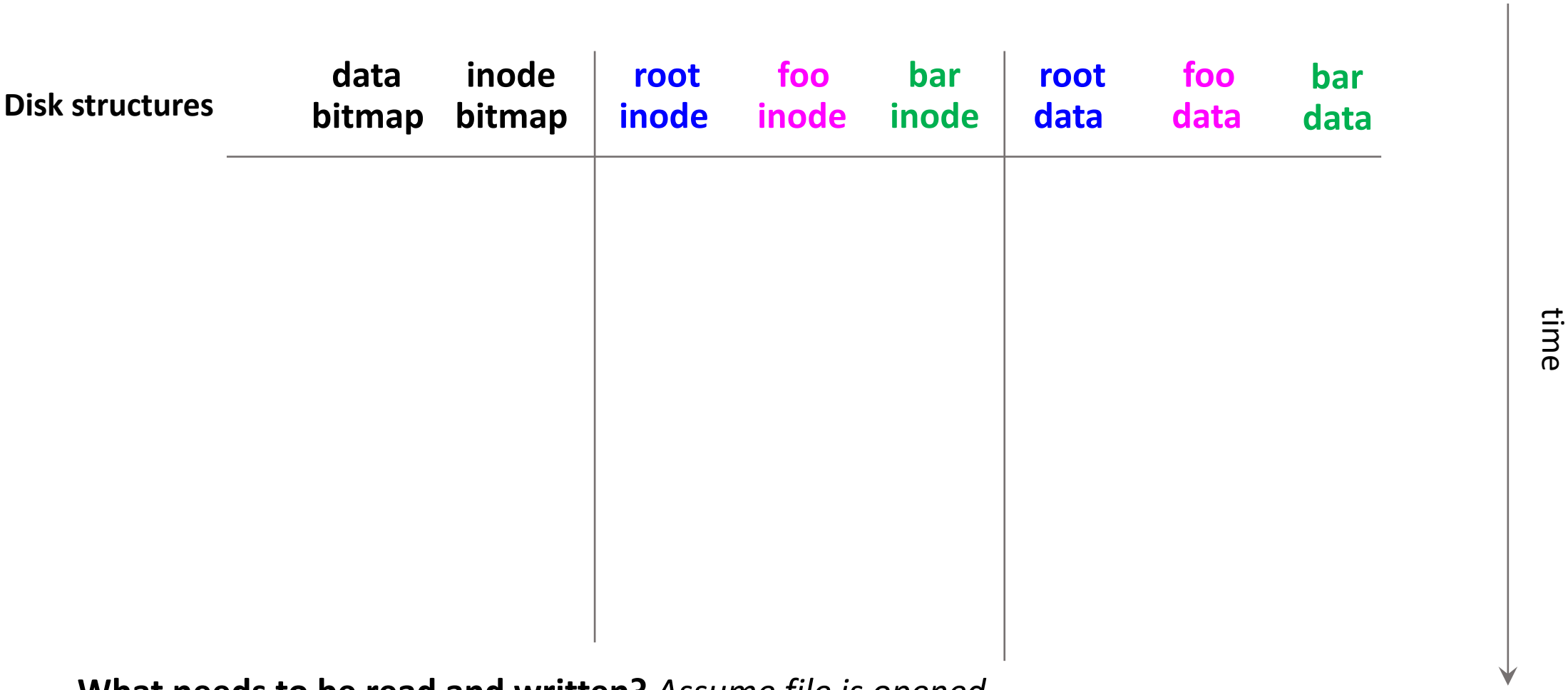
What needs to be read and written? *Assume file is opened*

Read `/foo/bar`



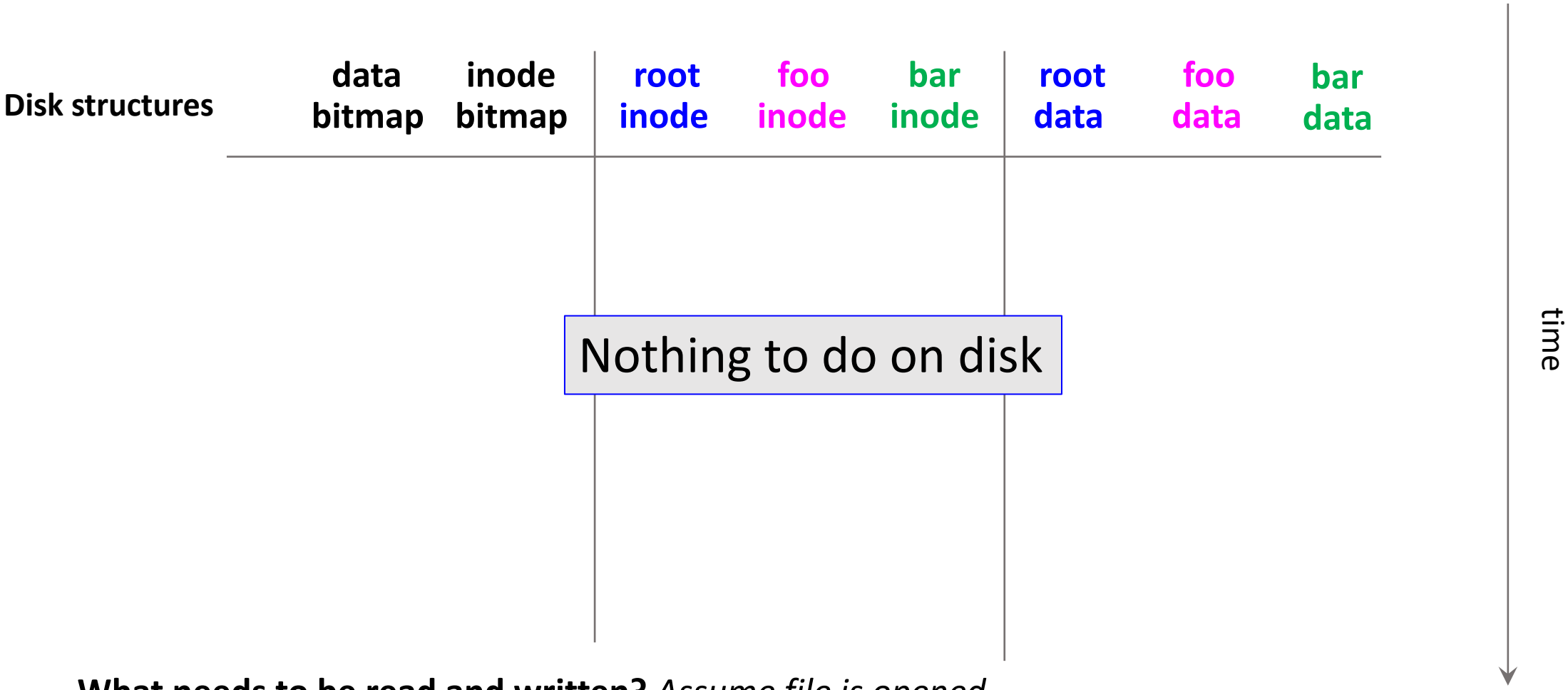
What needs to be read and written? *Assume file is opened*

Close `/foo/bar`



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

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
- index = hash(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

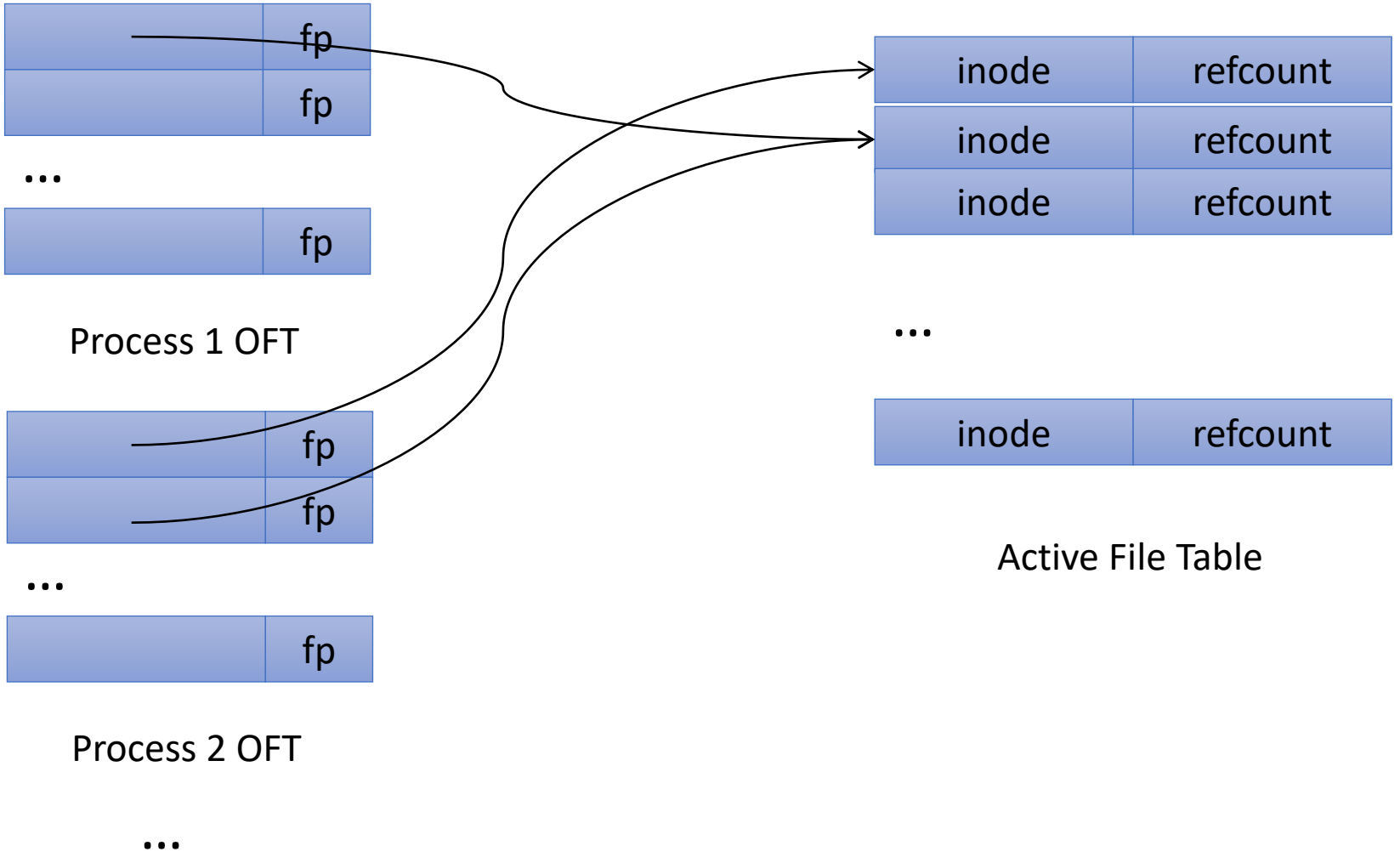
(System-Wide) Active File Table

- 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

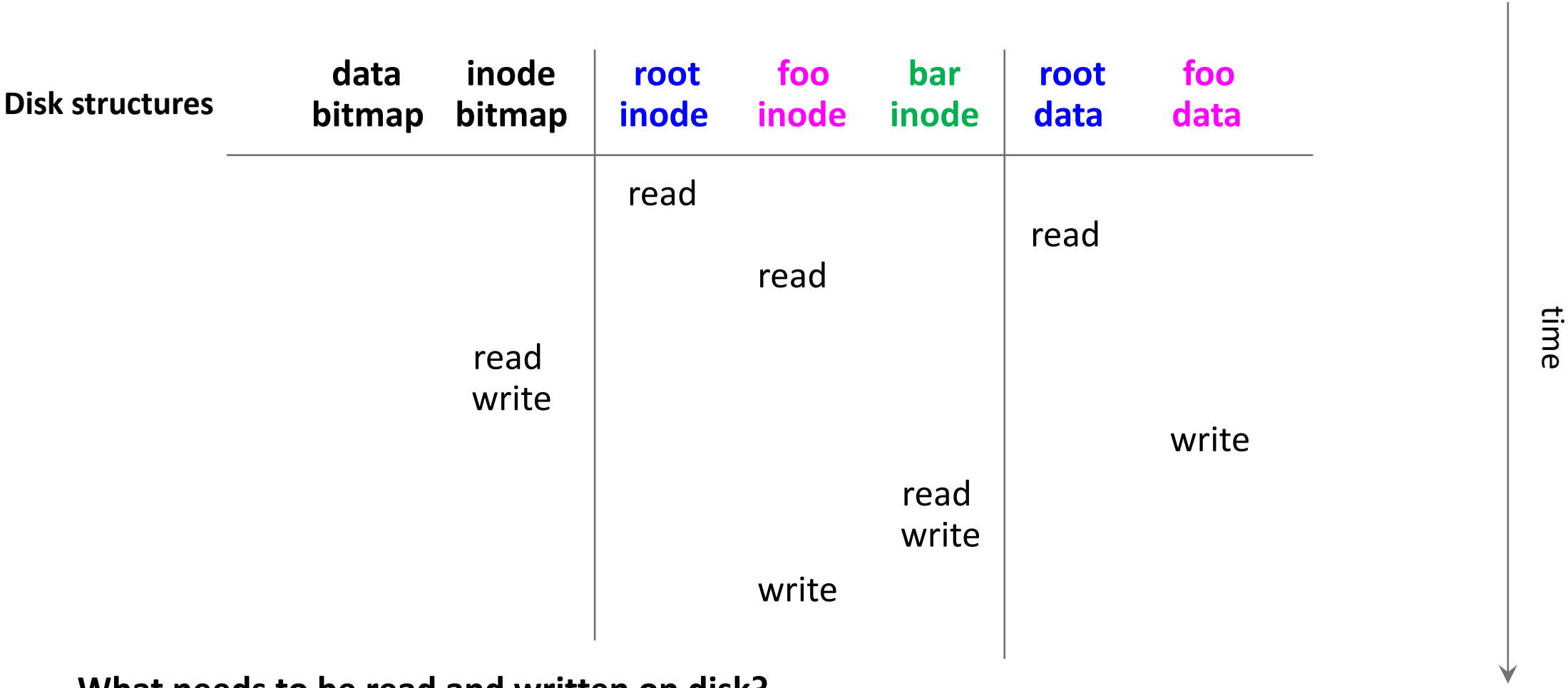
Open File Tables



Putting it All Together

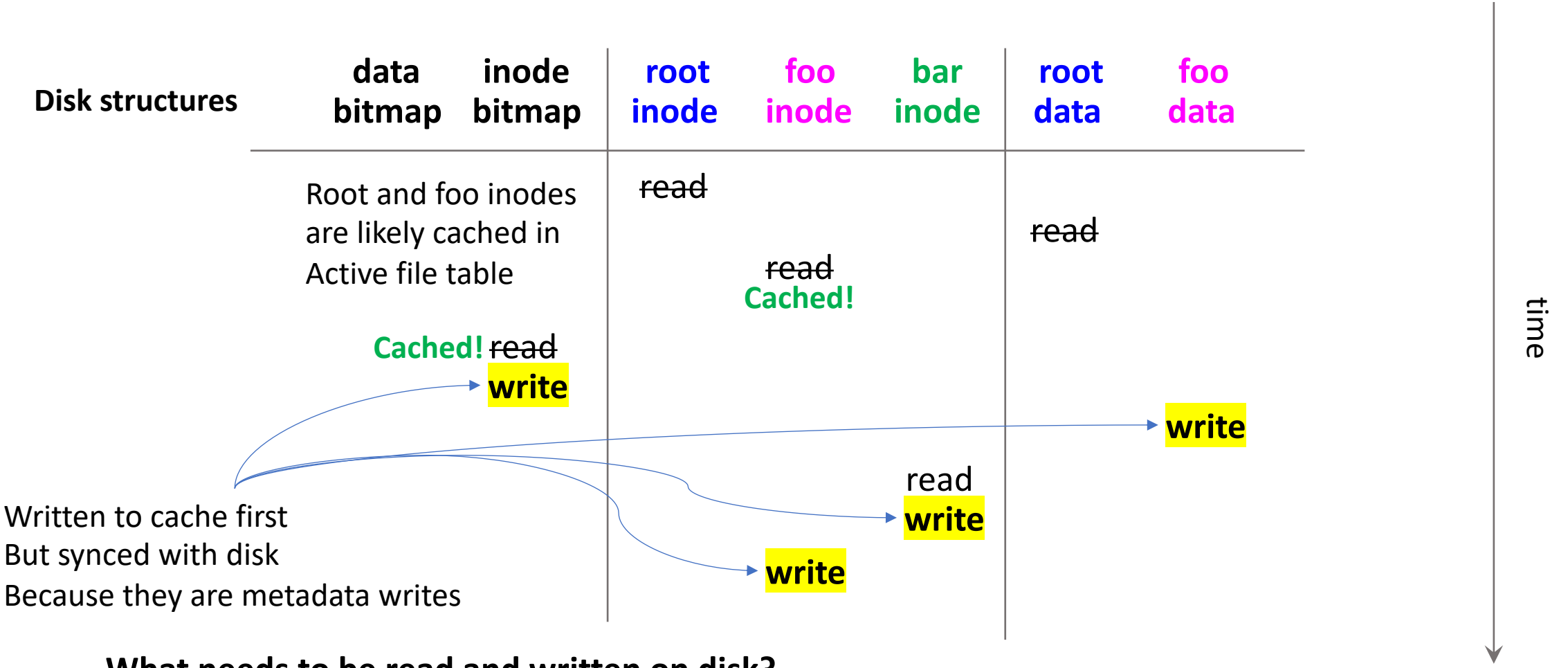
- Create
- Open
- Write
- Read
- Close

Create `/foo/bar` disk structures only



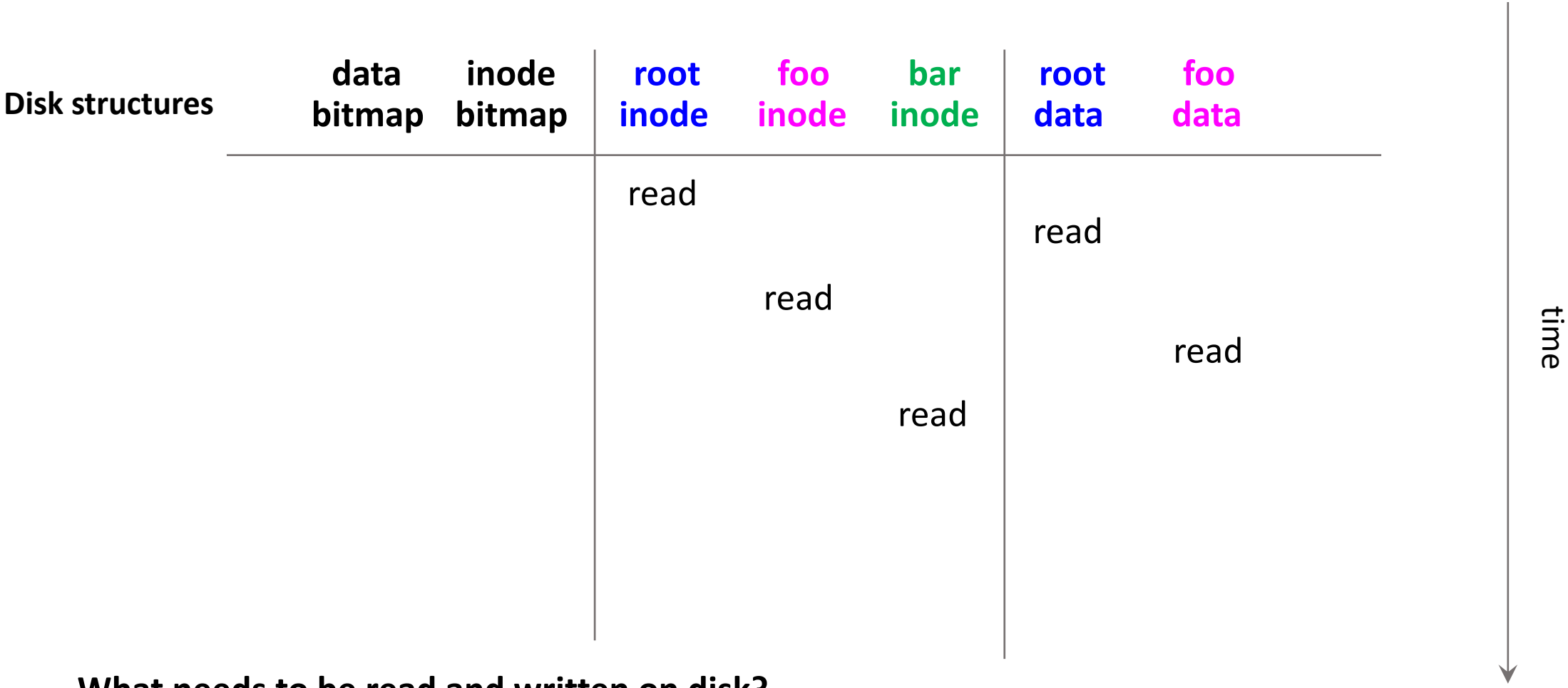
What needs to be read and written on disk?

Create `/foo/bar` disk+memory structures



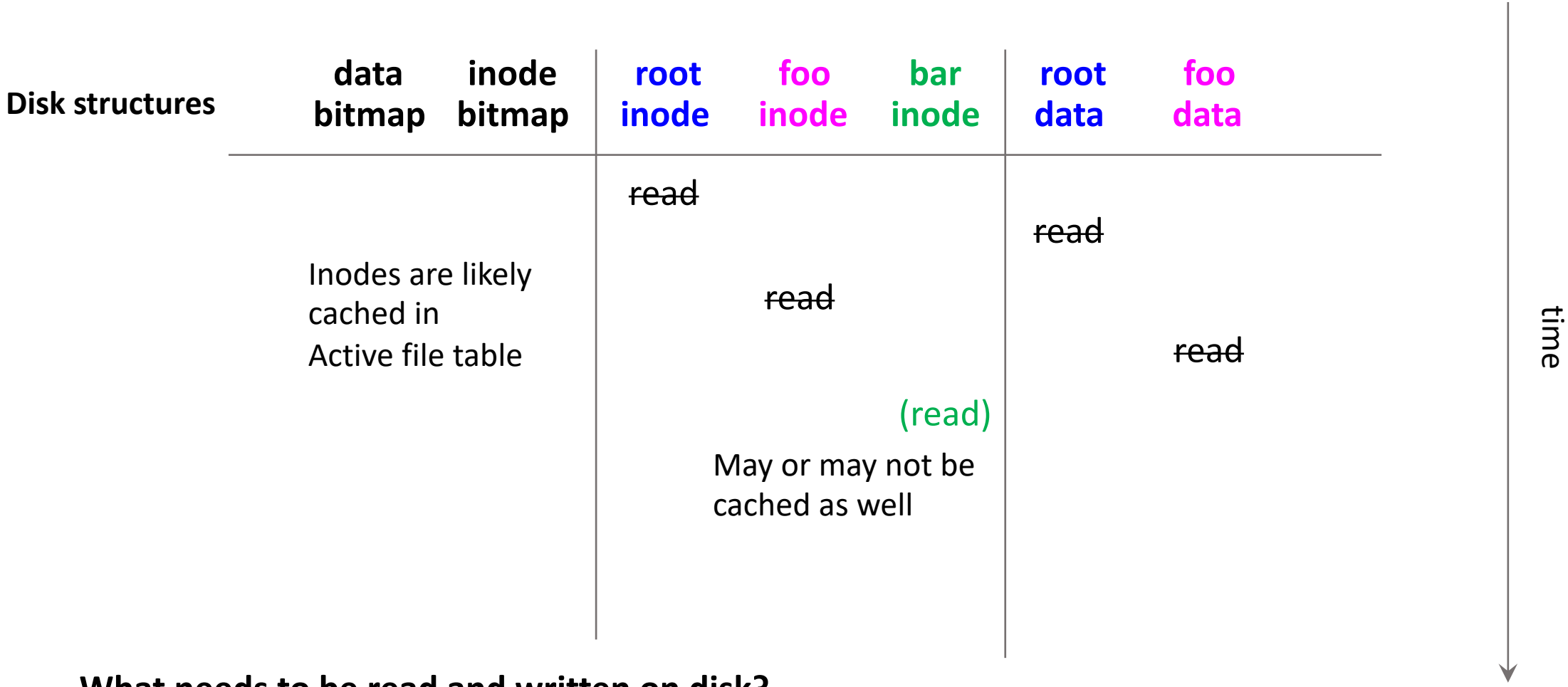
What needs to be read and written on disk?

Open `/foo/bar` disk structures only



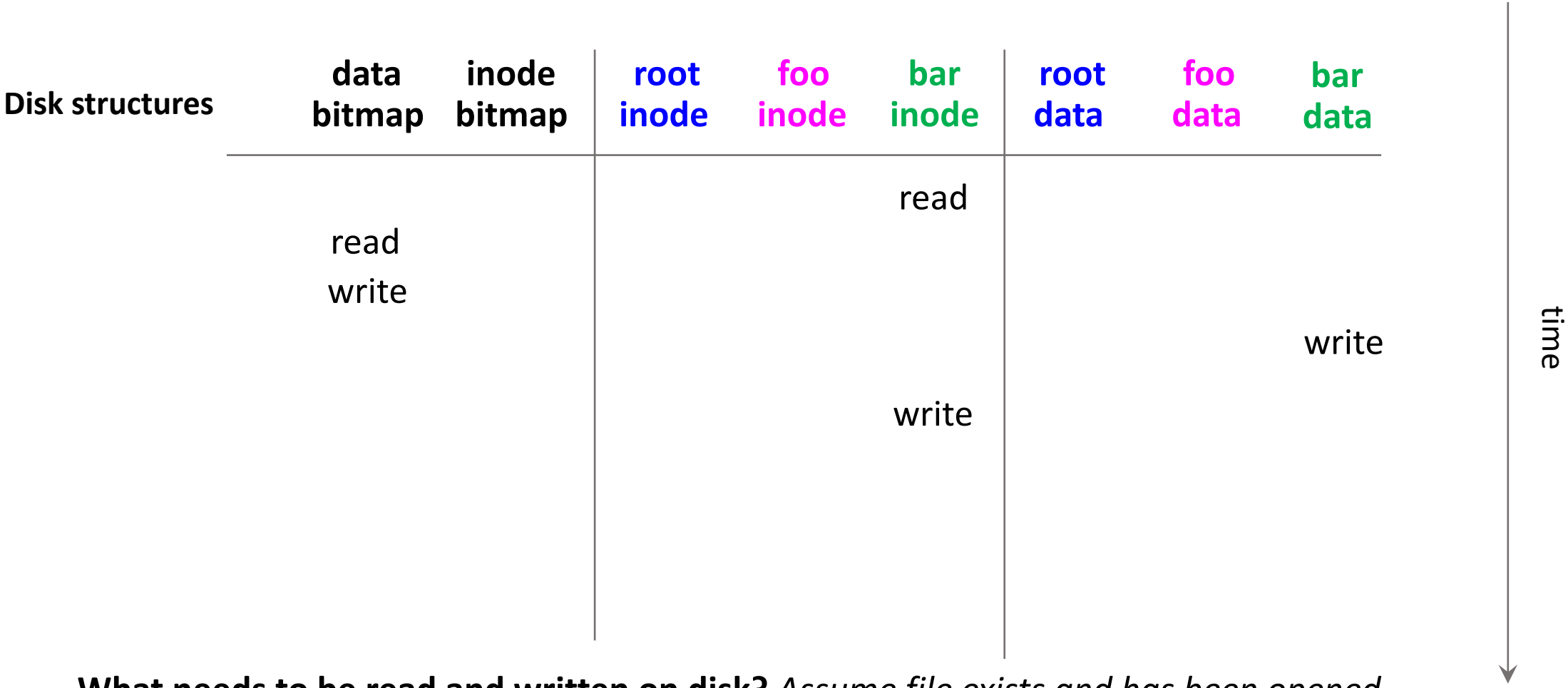
What needs to be read and written on disk?

Open `/foo/bar` disk+memory structures



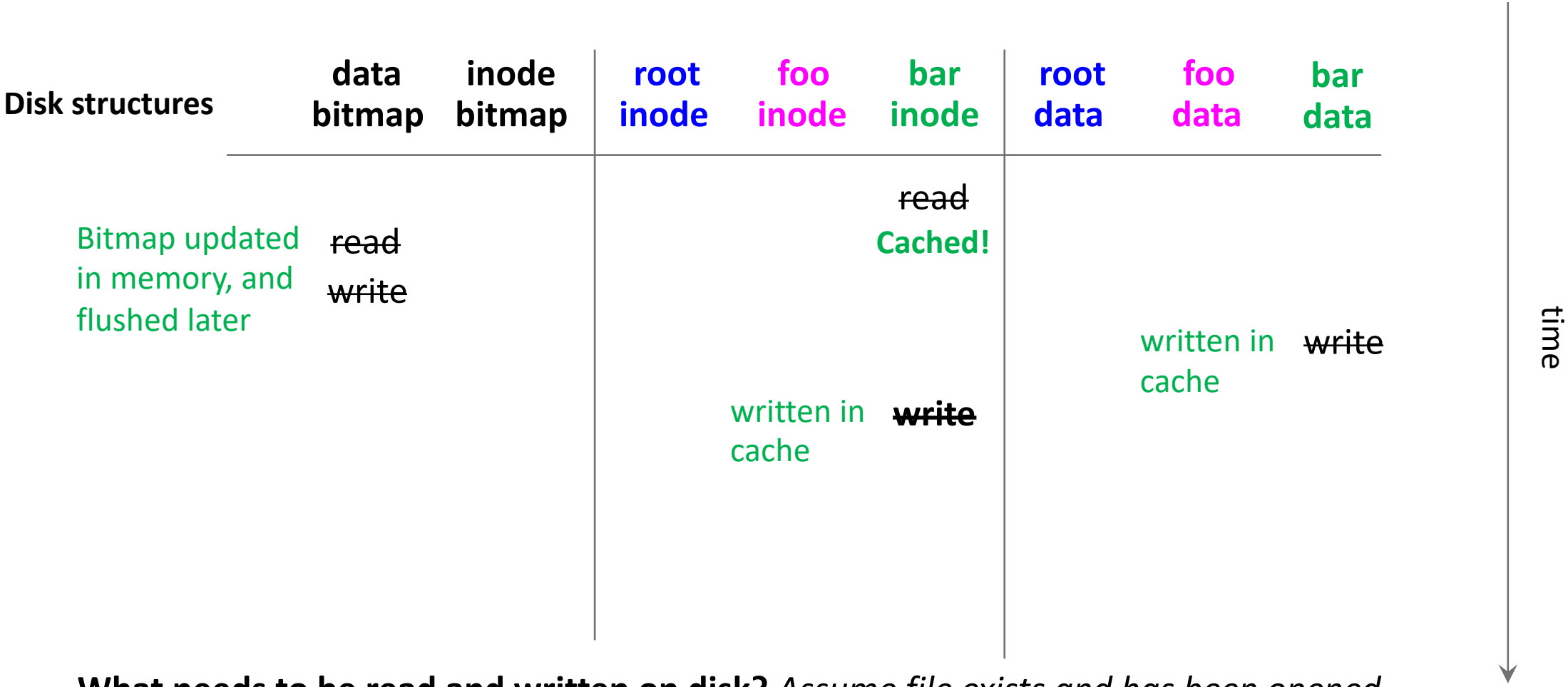
What needs to be read and written on disk?

Write to `/foo/bar` disk structures only



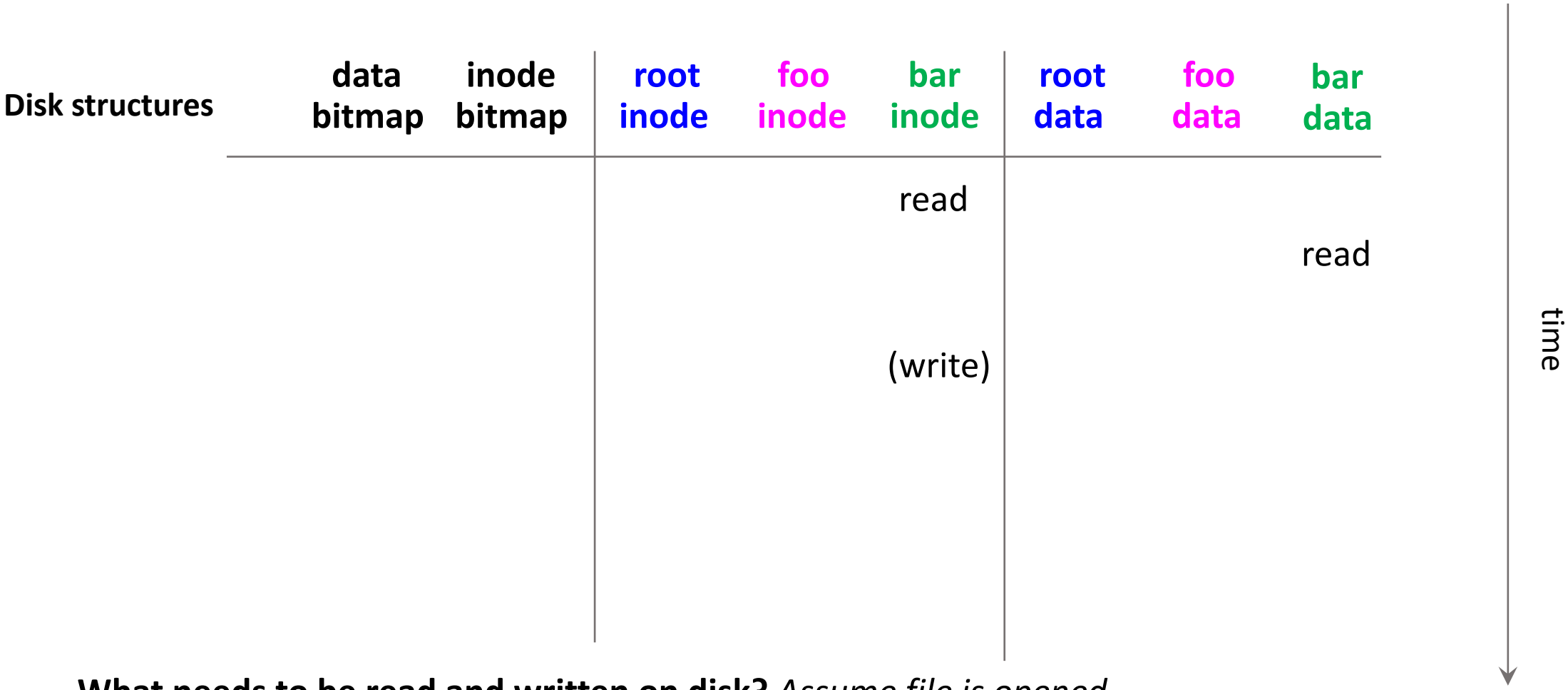
What needs to be read and written on disk? *Assume file exists and has been opened*

Write to `/foo/bar` disk+memory structures



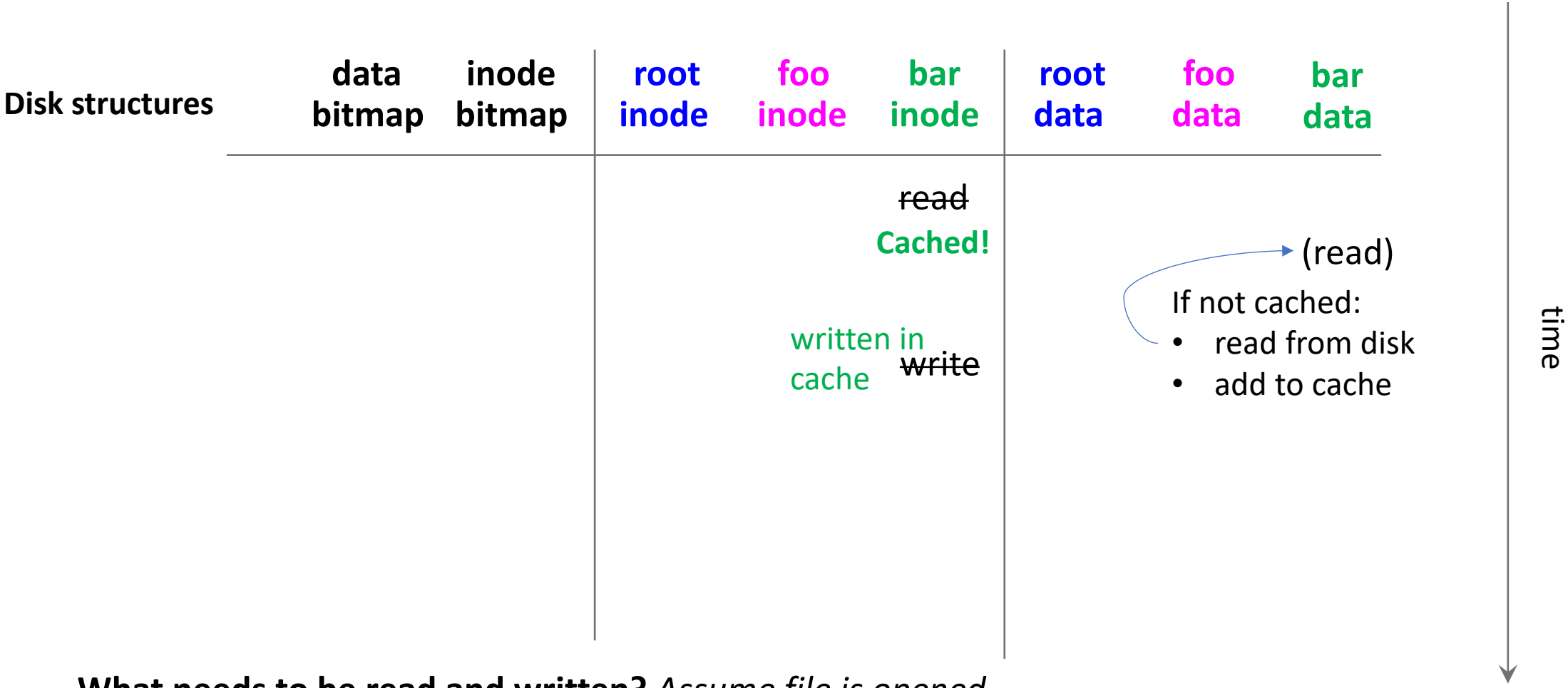
What needs to be read and written on disk? *Assume file exists and has been opened*

Read `/foo/bar` disk structures only



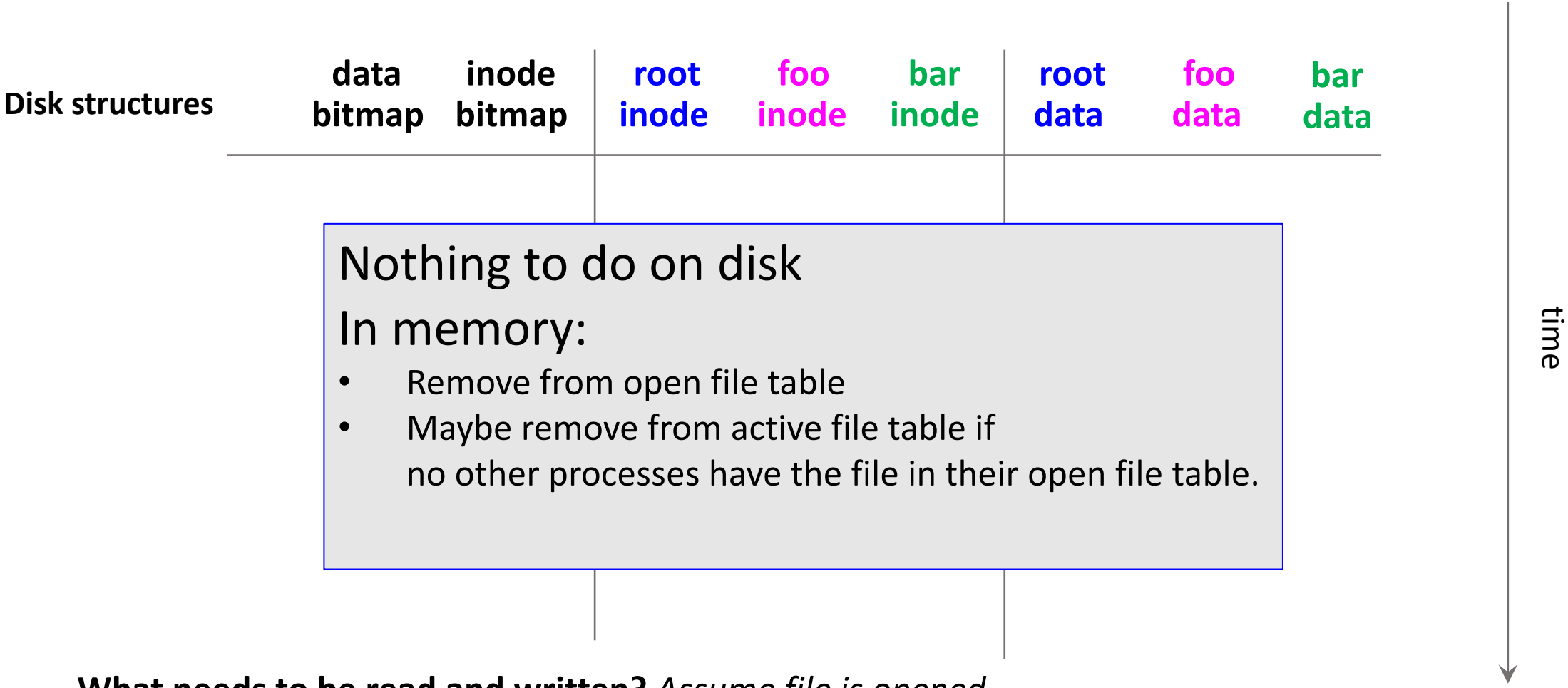
What needs to be read and written on disk? *Assume file is opened*

Read `/foo/bar` disk+memory structures



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

Close `/foo/bar` disk+memory structures



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

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.

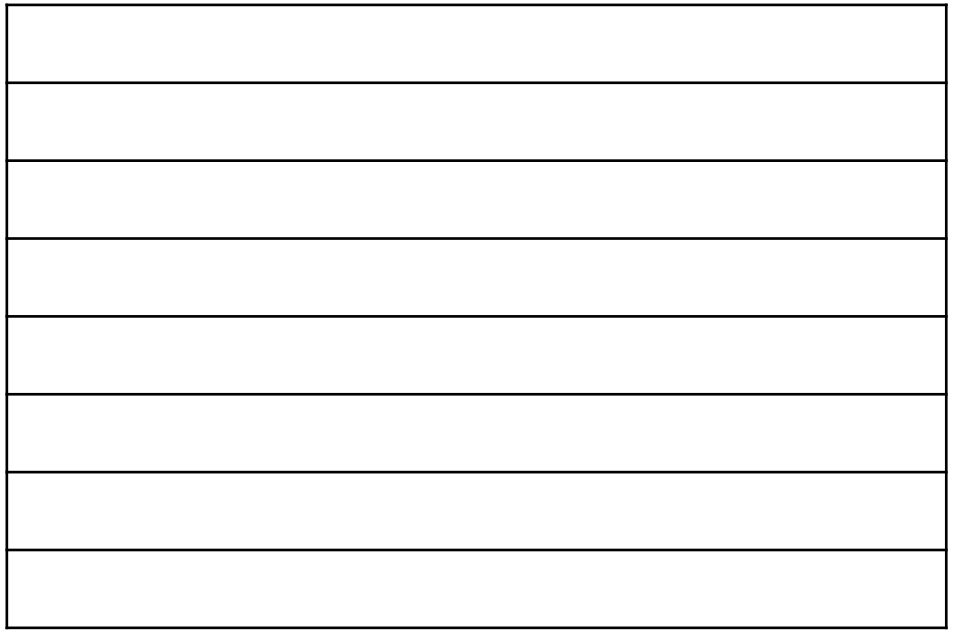
Open file table P1

Open file table P2

Open file table P3

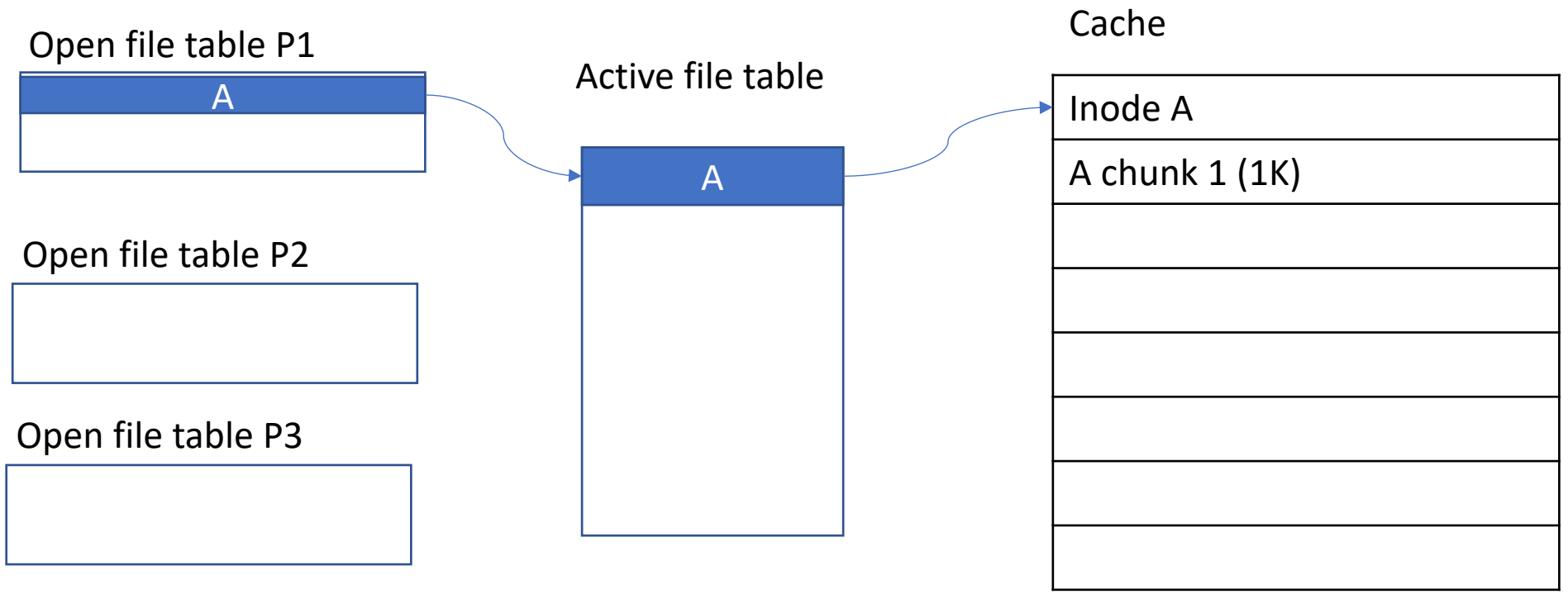
Active file table

Cache



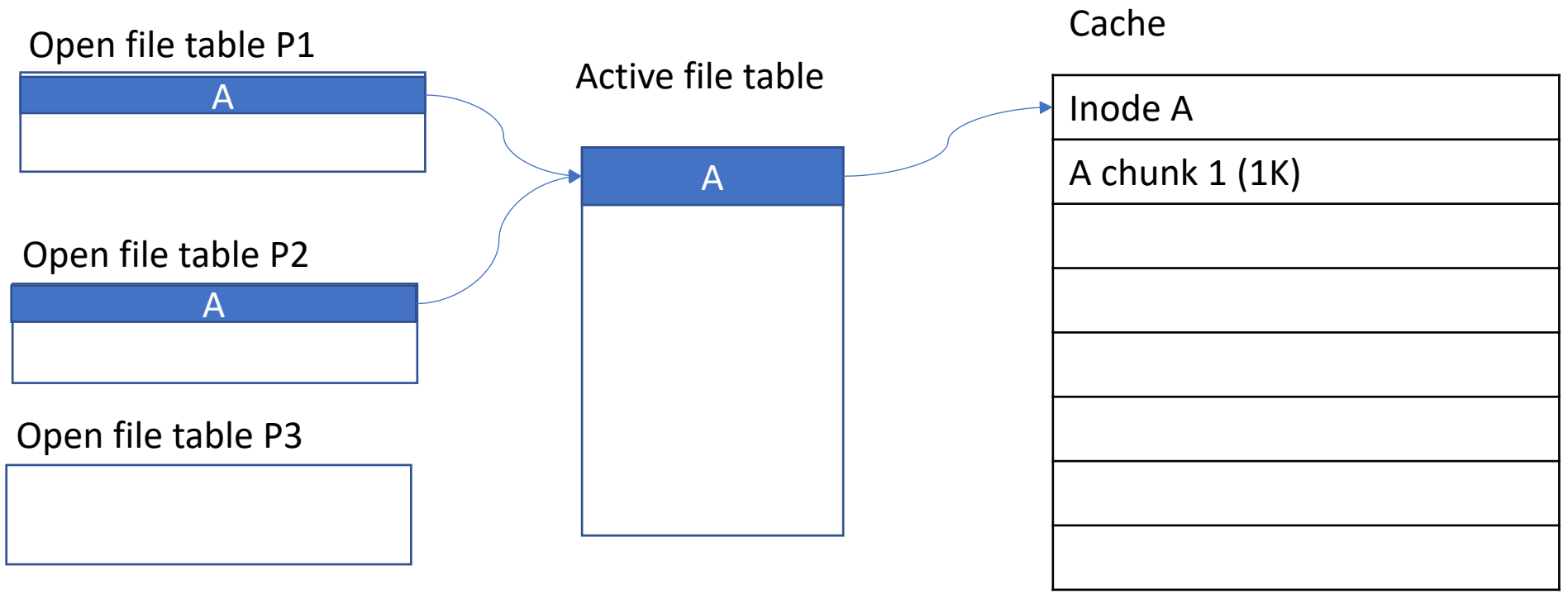
- **Process P1 opens file A and reads 512 bytes from it.**
- Process P2 opens file A and reads 1024 bytes from it.
- Process P3 creates file B opens it and writes 8192 bytes to it.
- Process P1 opens file B and reads 1024 bytes from it.

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



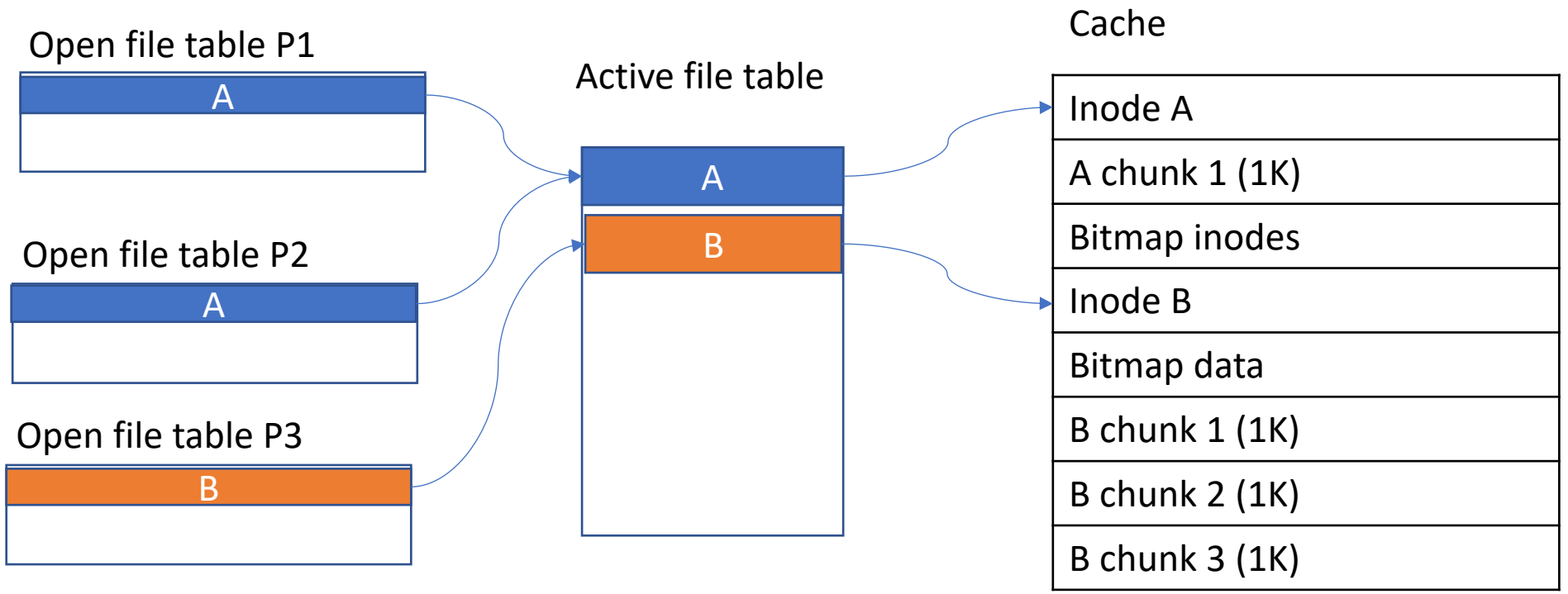
- Process P1 opens file A and reads 512 bytes from it.
- **Process P2 opens file A and reads 1024 bytes from it.**
- Process P3 creates file B opens it and writes 8192 bytes to it.
- Process P1 opens file B and reads 1024 bytes from it.

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



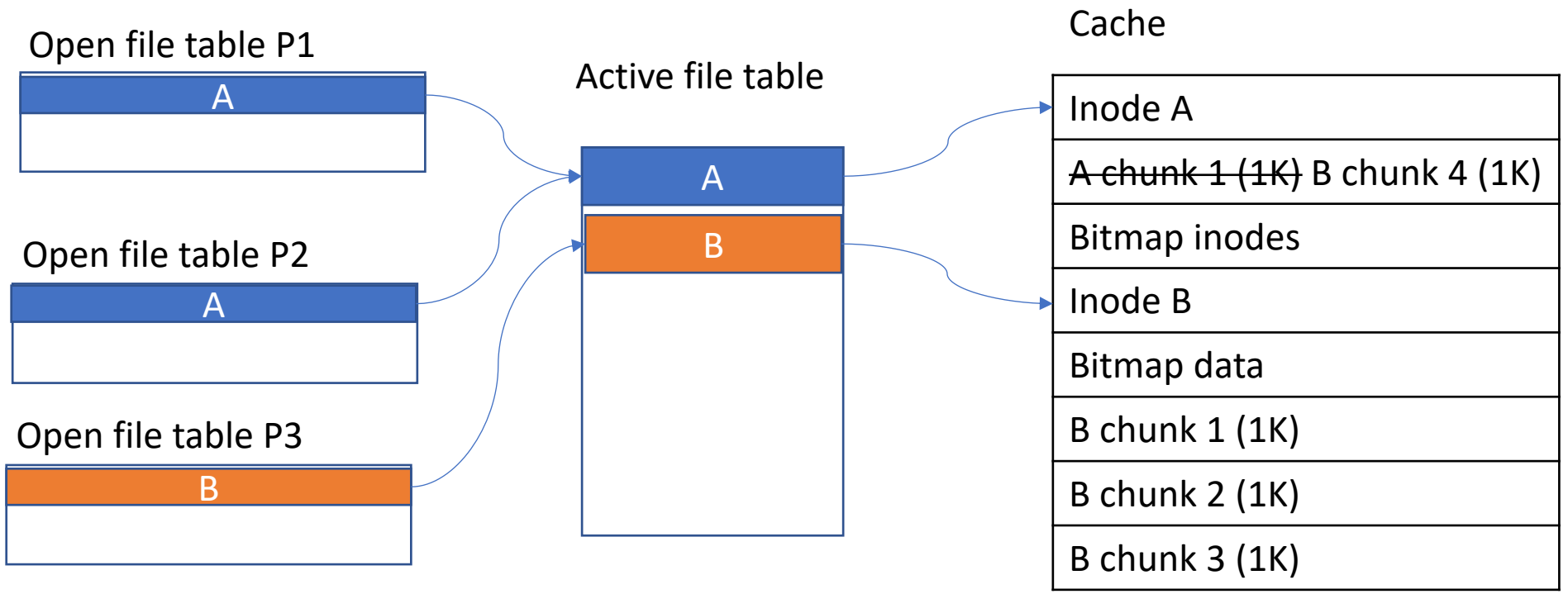
- Process P1 opens file A and reads 512 bytes from it.
- Process P2 opens file A and reads 1024 bytes from it.
- **Process P3 creates file B opens it and writes 8192 bytes to it.**
- Process P1 opens file B and reads 1024 bytes from it.

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



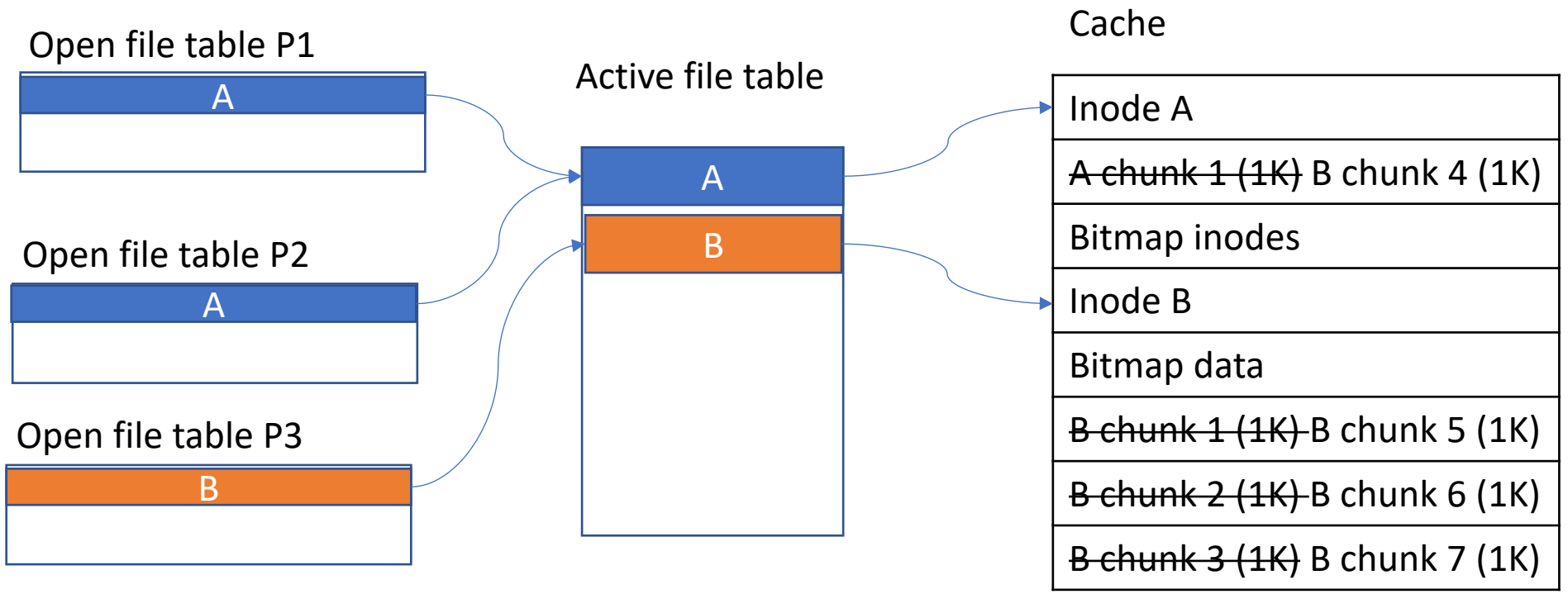
- Process P1 opens file A and reads 512 bytes from it.
- Process P2 opens file A and reads 1024 bytes from it.
- **Process P3 creates file B opens it and writes 8192 bytes to it.**
- Process P1 opens file B and reads 1024 bytes from it.

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



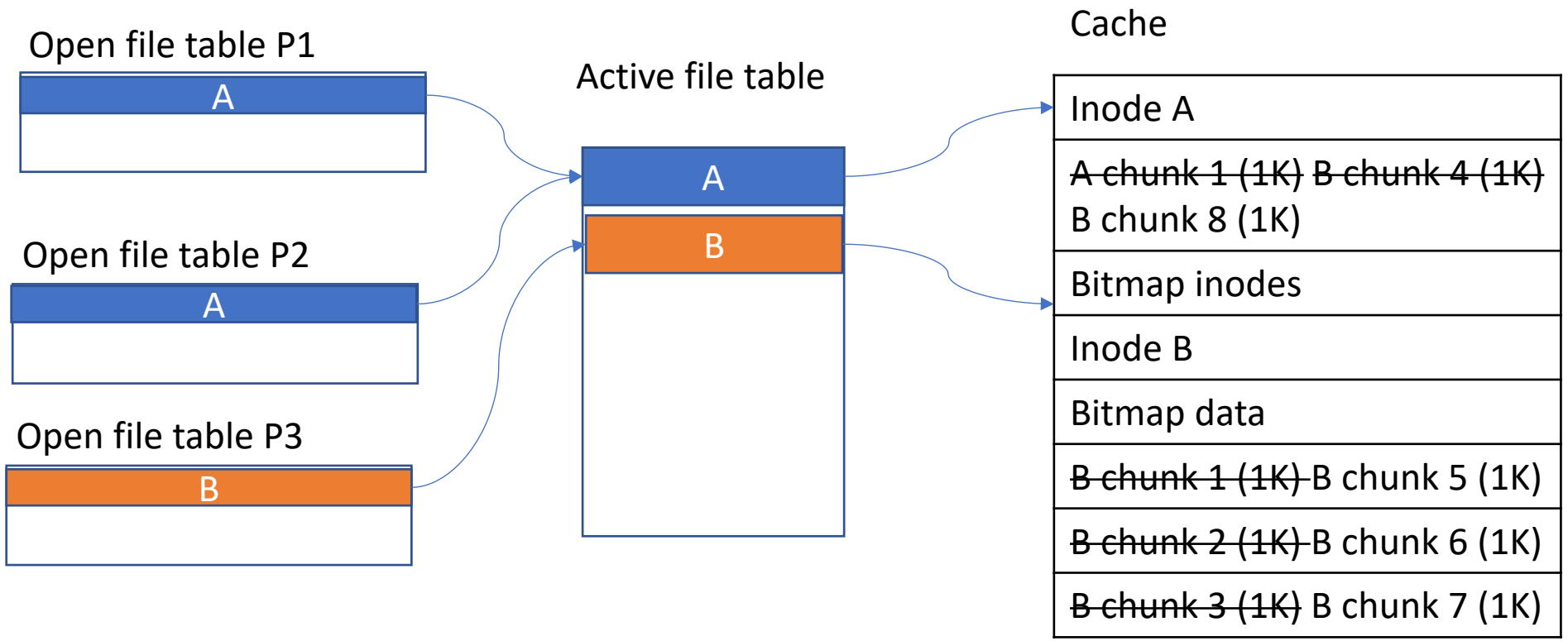
- Process P1 opens file A and reads 512 bytes from it.
- Process P2 opens file A and reads 1024 bytes from it.
- **Process P3 creates file B opens it and writes 8192 bytes to it.**
- Process P1 opens file B and reads 1024 bytes from it.

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



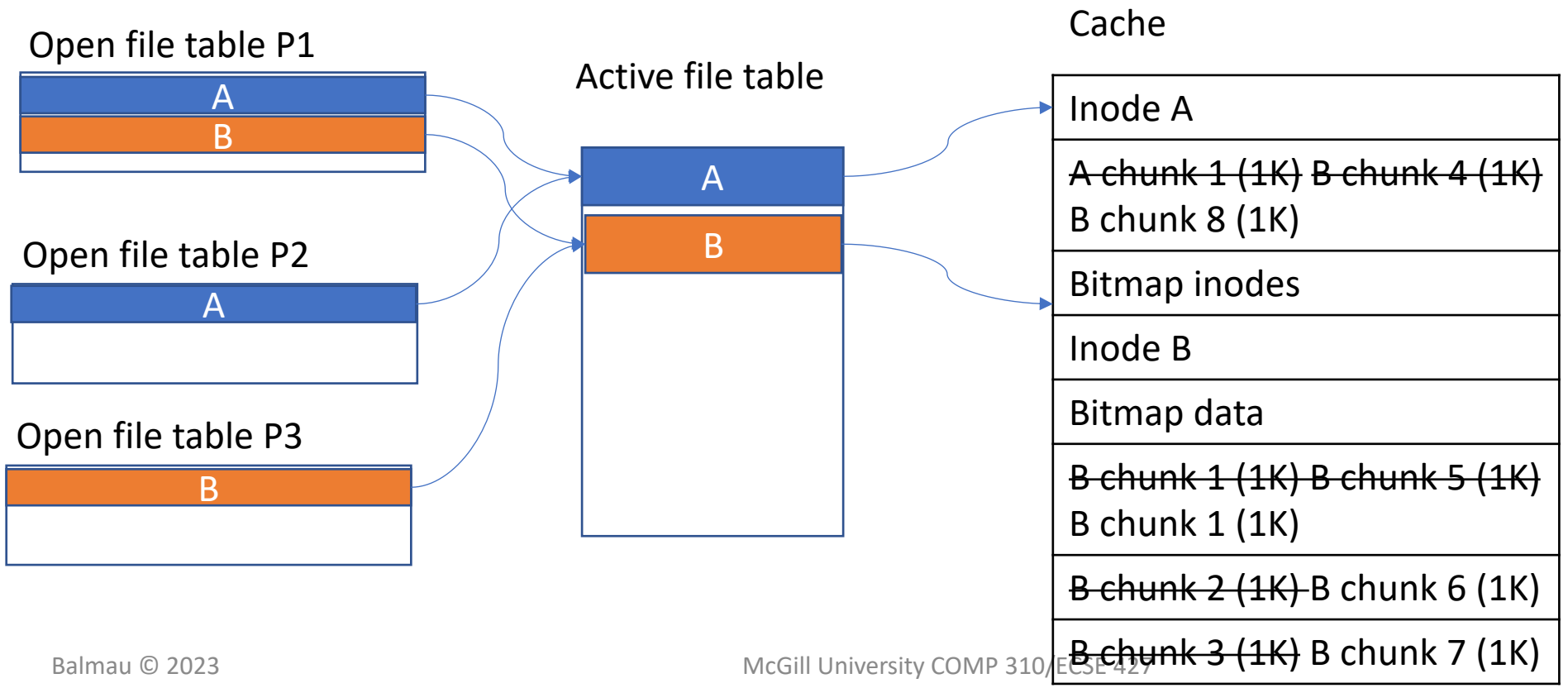
- Process P1 opens file A and reads 512 bytes from it.
- Process P2 opens file A and reads 1024 bytes from it.
- **Process P3 creates file B opens it and writes 8192 bytes to it.**
- Process P1 opens file B and reads 1024 bytes from it.

- 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

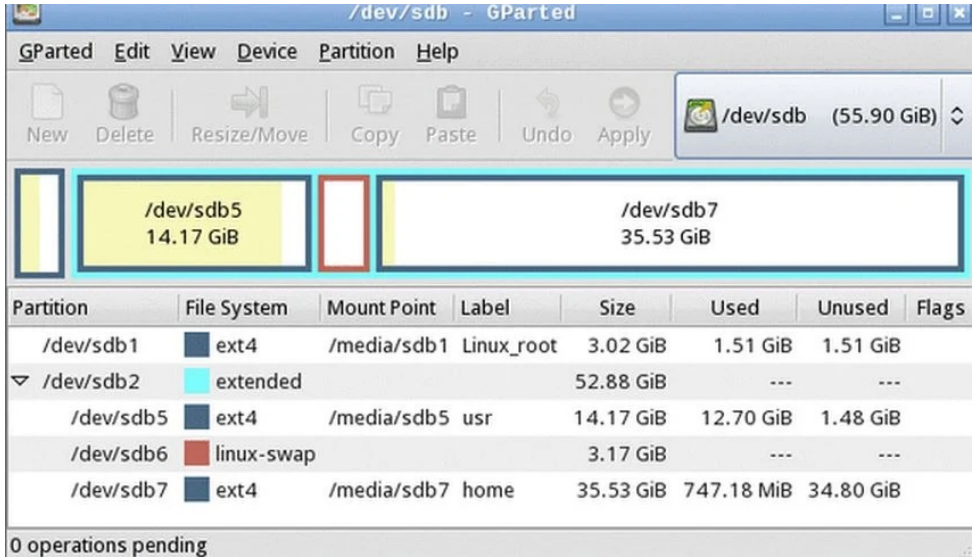
Setting up the FS

- 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



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

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?

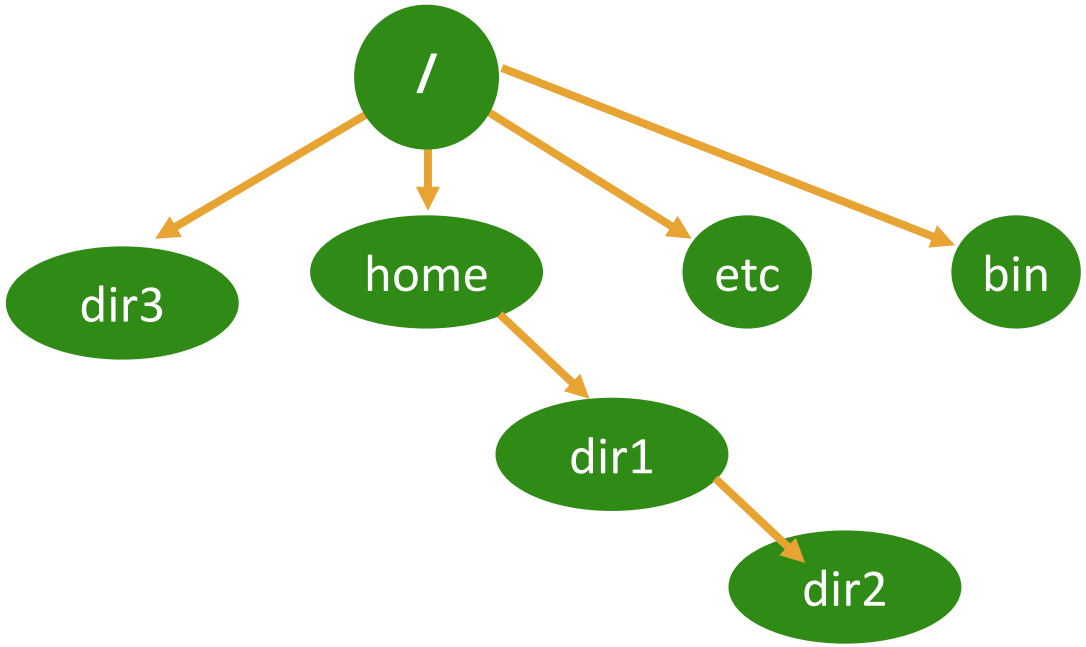
Multiple FS

- 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”!

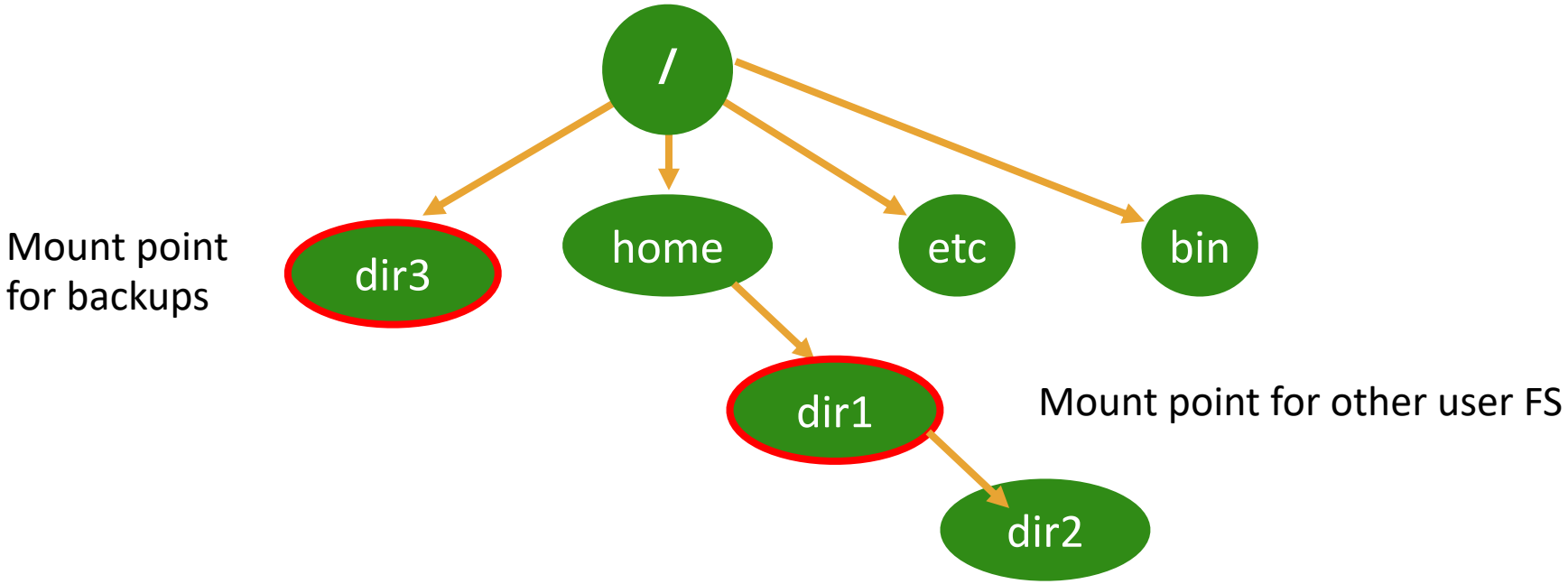
Multiple FS

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

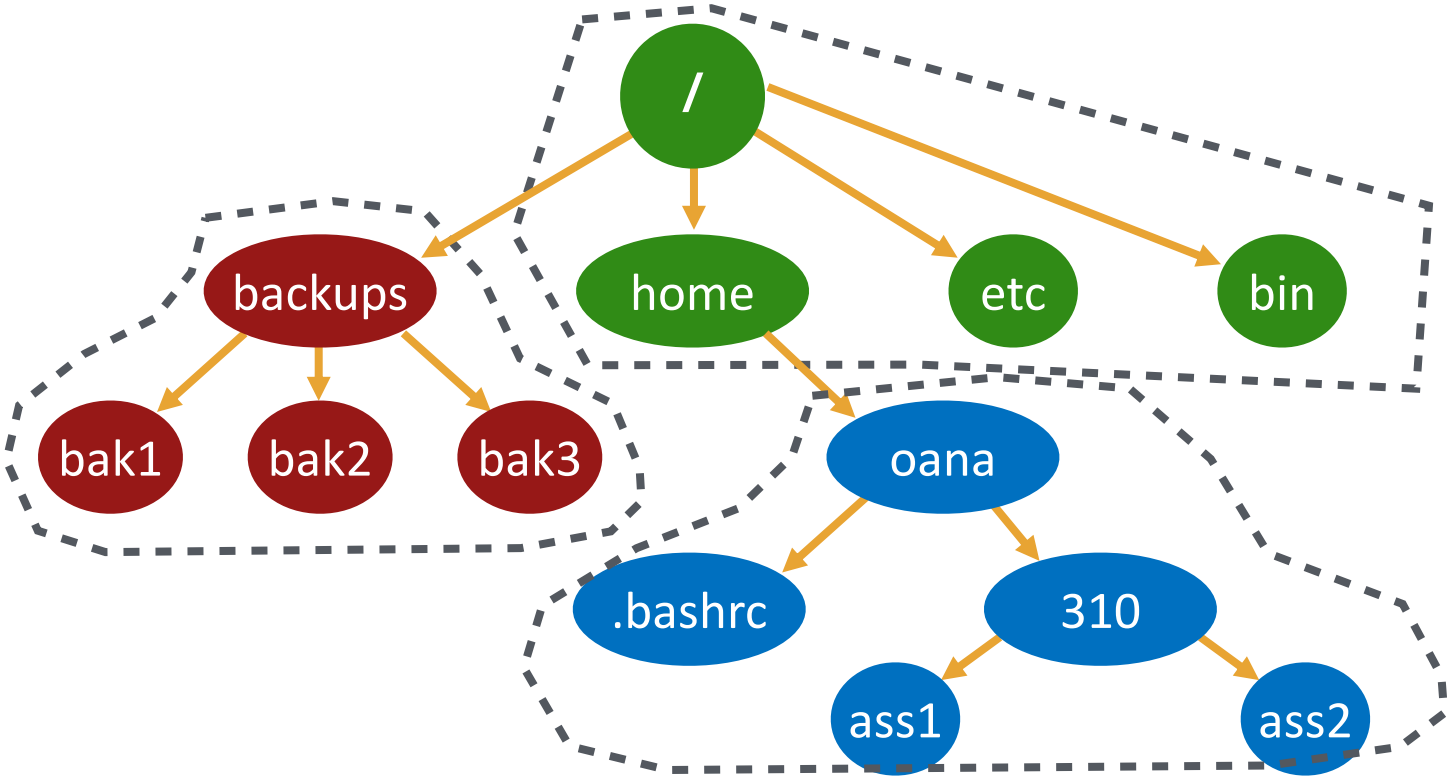
Example with 3 mounted FS



Example with 3 mounted FS

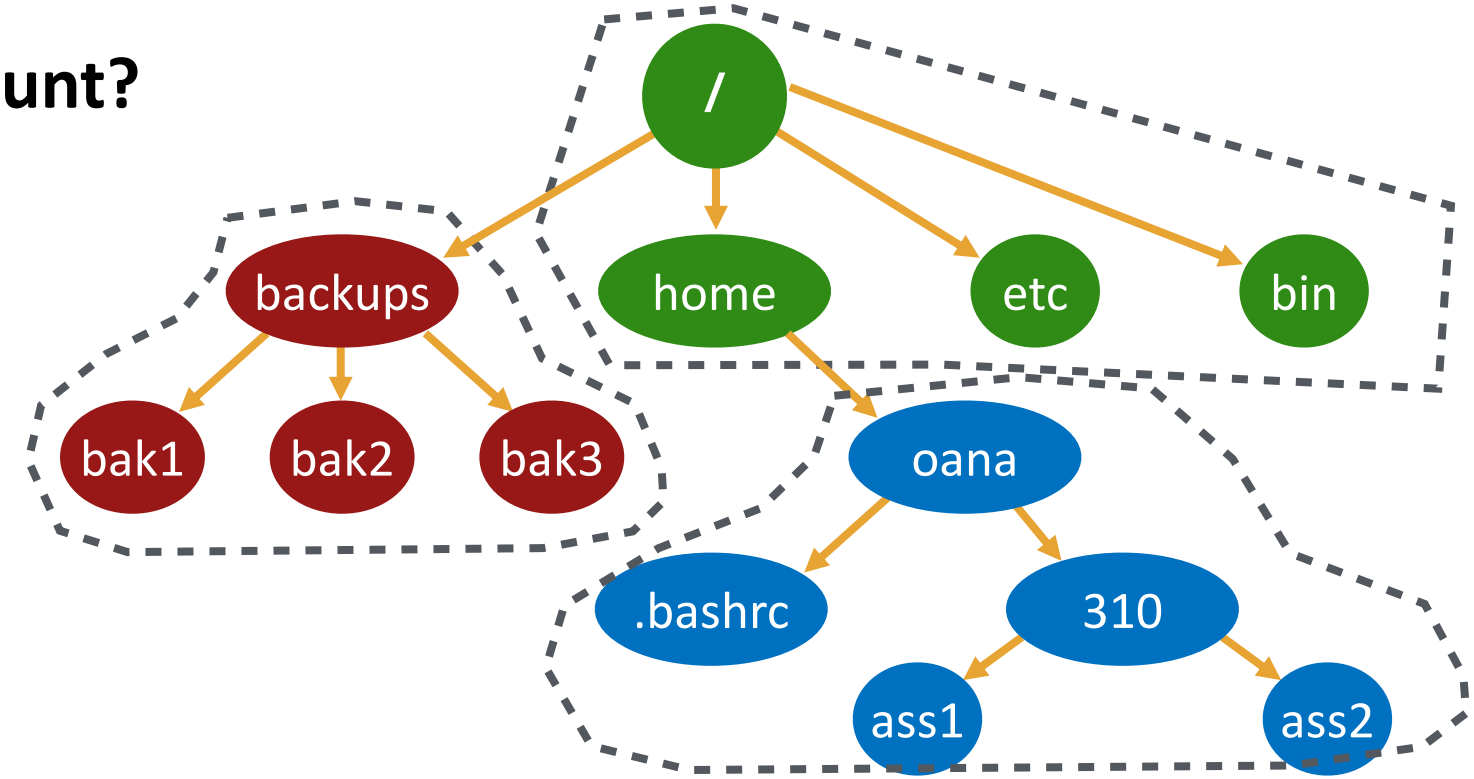


Example with 3 mounted FS



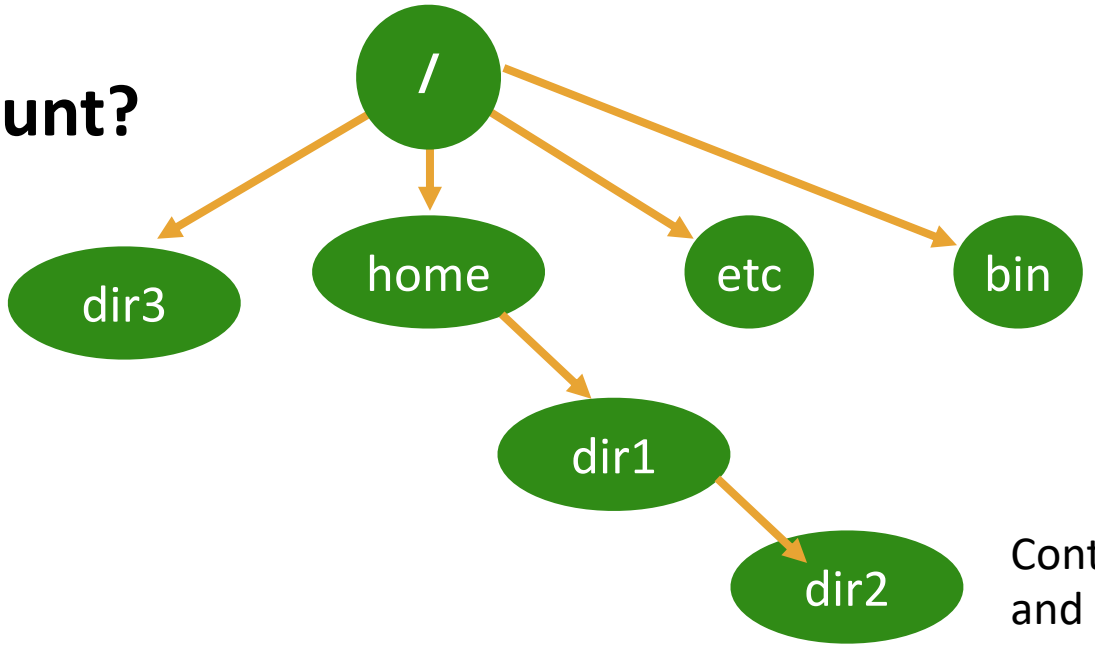
Example with 3 mounted FS

Unmount?



Example with 3 mounted FS

Back to initial
State after unmount?



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

But typically, we mount in empty directories.

Booting

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

Booting

- Very first thing that happens when computer turns on
- BIOS looks for clues on what it needs to start OS
- First place BIOS checks is the **boot block**

Boot Block

- At fixed location on disk (usually sector 0)
  - Contains boot loader
  - Contains partition table
- Read by BIOS on machine boot

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)

File System Check

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

Replication

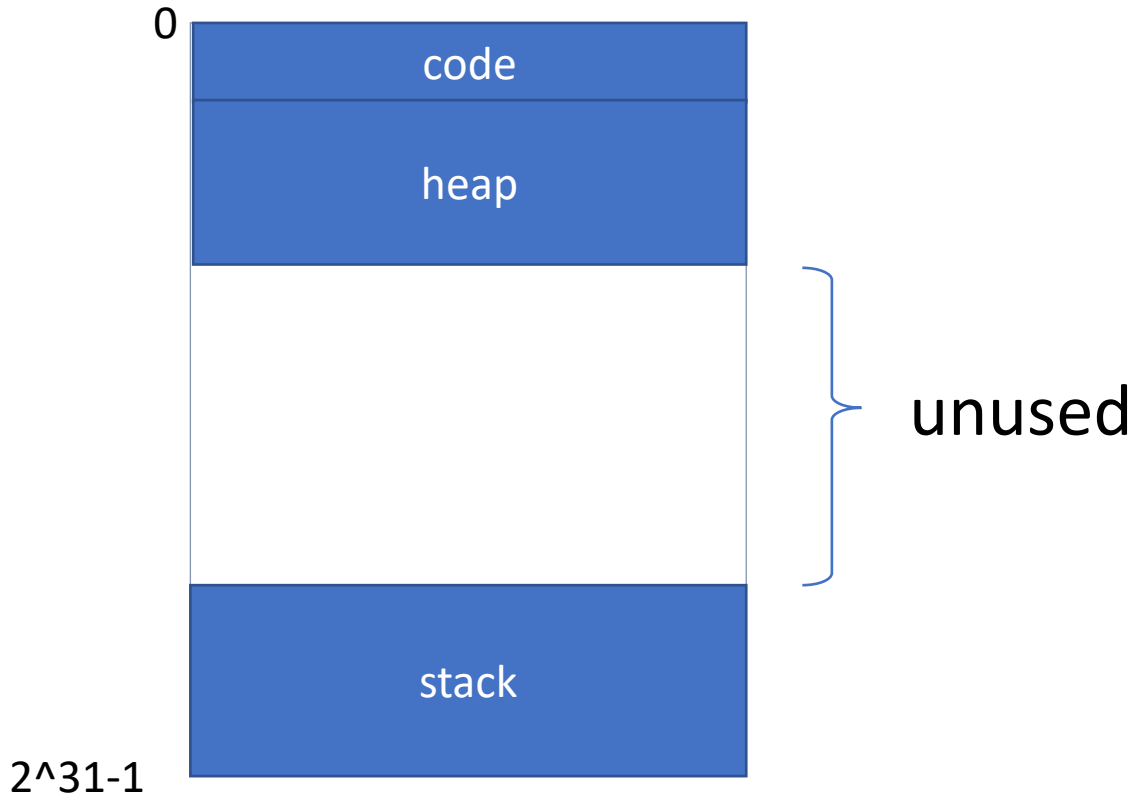
- Some key sectors are replicated
  - Boot blocks
  - Sometimes also inode blocks

Alternative File Access Method: Memory Mapping

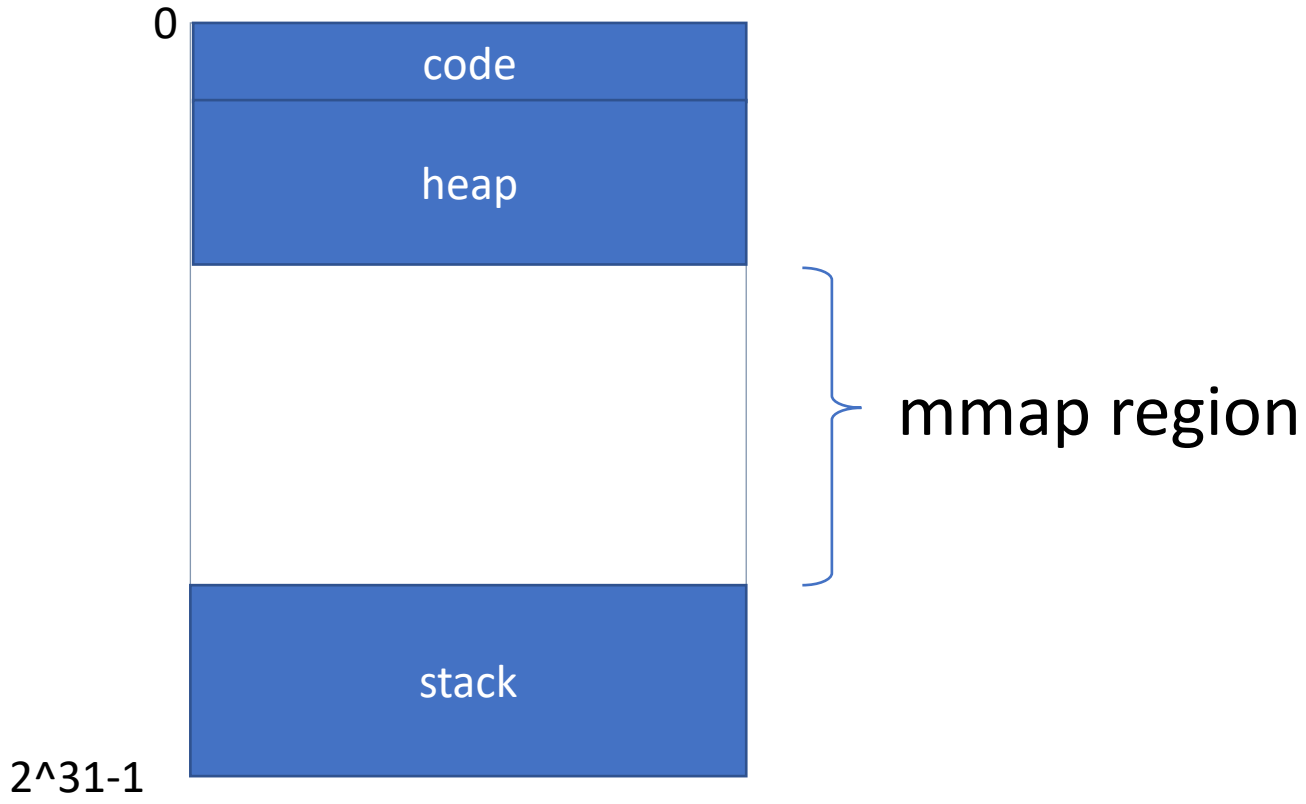
Alternative File Access Method: Memory Mapping

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

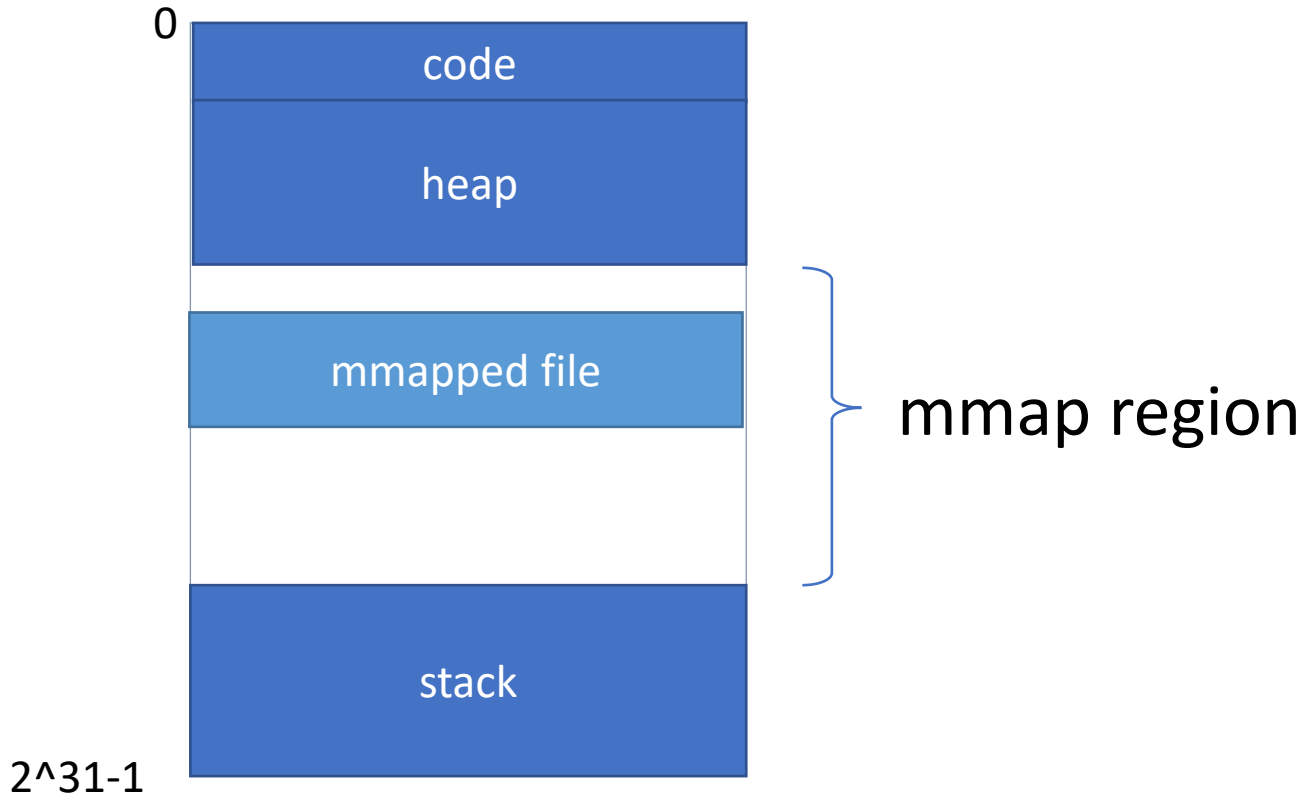
Remember this Picture?
Typical Virtual Address Space



Remember this Picture?
Typical Virtual Address Space



Remember this Picture?
Typical Virtual Address Space



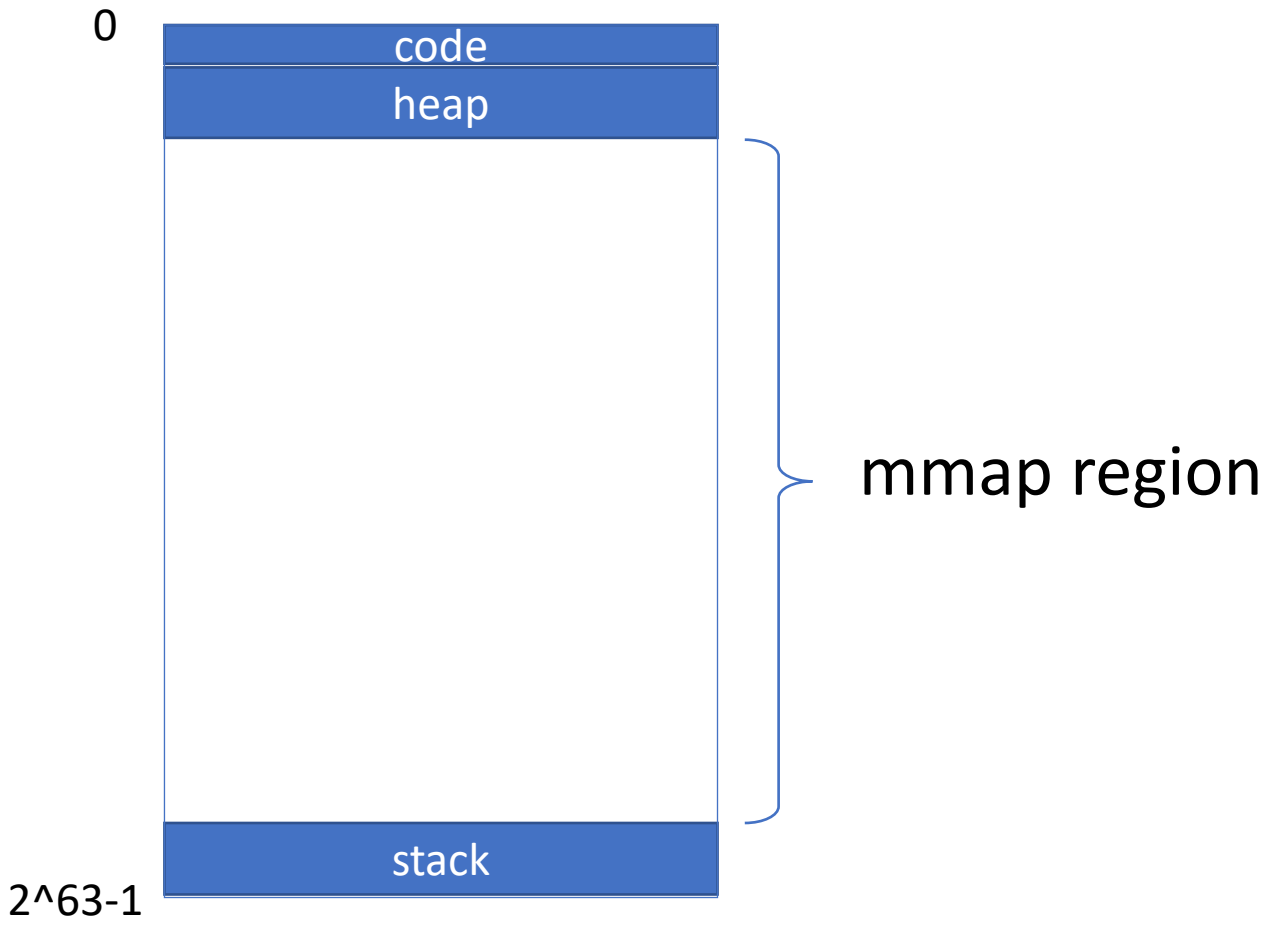
Remember Large Address Spaces?

- 64 bit address space

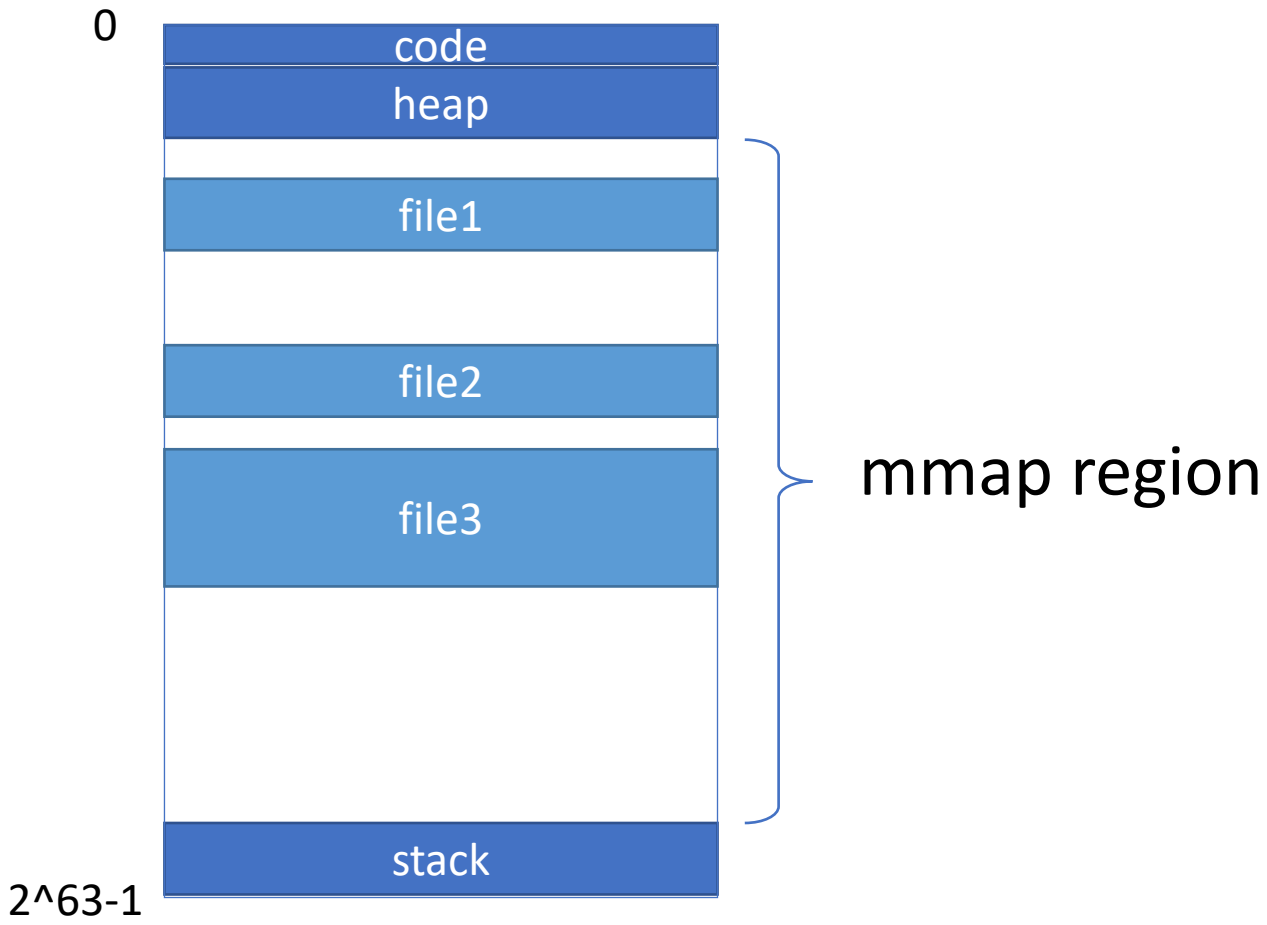
Do you know now why desirable?

- 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

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

Issues with mmap()

- Alignment on page boundary
- Not easy to extend a file
- For small files
  - Read() more efficient than mmap() + page fault

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).