File System Implementation

ICS332

Operating Systems

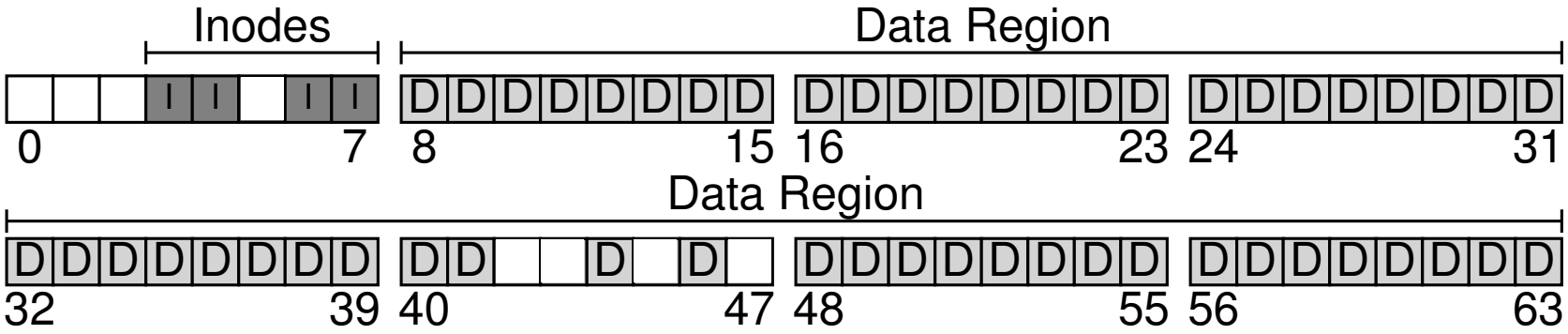
Henri Casanova (henric@hawaii.edu)

It's a Data Structure

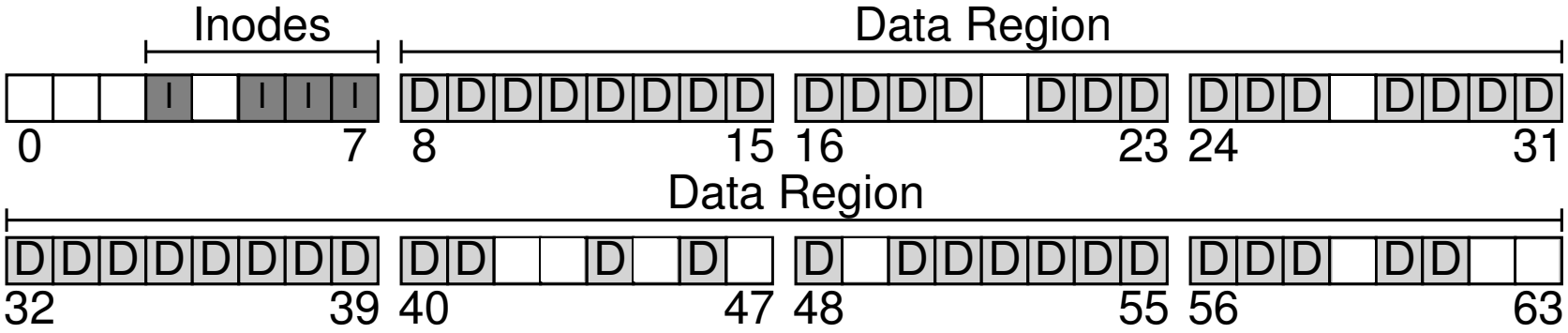
- A File System basically implements an (on-disk) data structure to store the directory structure, the files, and their content
  - Part of that on-disk data structure is sometimes brought into RAM temporarily
- A File System implements simple operations on this data structures to manage files:
  - open, read, write, delete, move, etc.
- The data structure and operation implementations should be efficient, should not waste too much space, and should be able to survive data corruptions

Files as Blocks

- The content of a file is stored as a set of blocks
  - Stored on HDD blocks or SDD pages
- For each file we need an inode data structure to track where the file blocks are and store all kind of useful information about the file (e.g., permission, data of creation)
  - "**inode**" is UNIX/Linux terminology, which OSTEP uses throughout
  - Let's call it "inode" here too, just like OSTEP,
- We now have two "regions" on disk
  - The inode region (one block contains more than one inode)
  - The data region

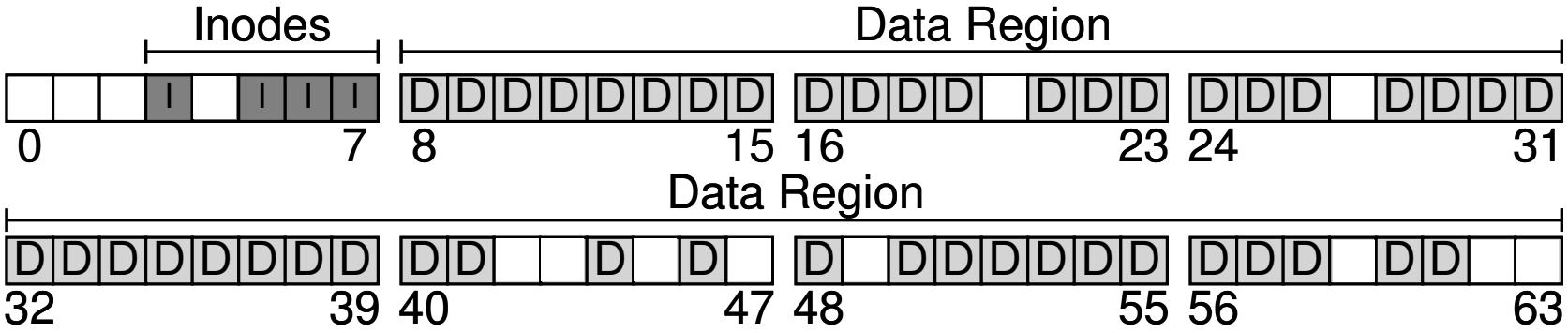


Non-Contiguous Allocations



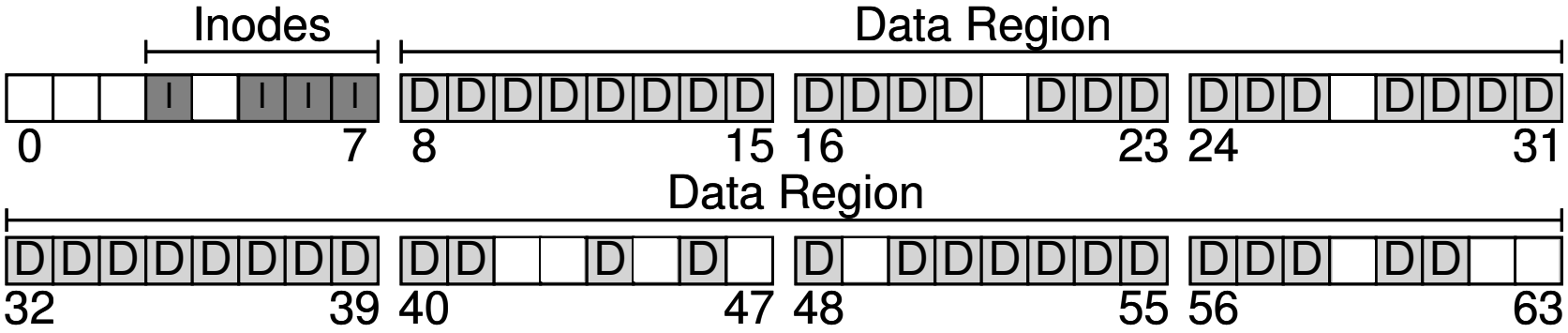
- We already know from the MainMemory module that **contiguous allocation is not a good idea**
  - Fragmentation makes it difficult for find a large enough contiguous set of holes to store a new file
  - What if we want to append to the end of a file and there are no free block after its last block?
- So we do non-contiguous allocation
  - A file's blocks are not necessarily next to each other on the disk
- The inode structure needs to keep track of where each block is
  - It can't just be "index of first block and index of last block"!

Keep Track of Free Blocks?



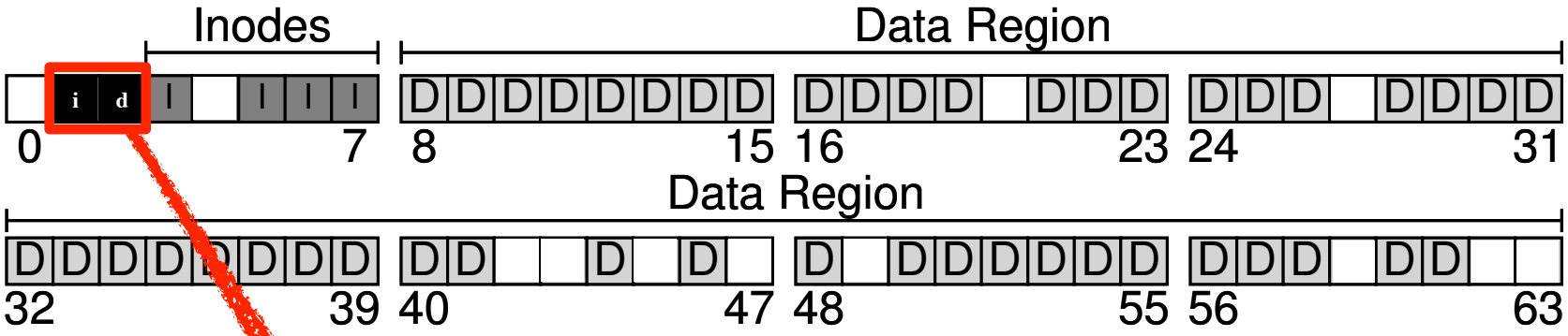
- There are free blocks on the disk
  - Shown as empty squares in the figure above
- **The file system needs to keep track of where free blocks are**
- One option: an on-disk linked list of free blocks
  - Each block stores a few bytes that encode the index of the next free block
  - The file system just needs to index of the “first” free block
  - This could be optimized by keeping the number of consecutive free blocks
    - So that we have a linked list of groups of contiguous free blocks
  - The fact that a linked list is O(n) for traversal is ok since we really never need to traverse it
- An other option: a bitmap

Bitmap of Free Blocks



- Main idea: we have an array of bits, one per block index on disk
  - 0 means “free”, 1 means “not free”
- This is called a bitmap
- A bitmap doesn’t take much space
  - Say our disk is 2TiB with 4KiB blocks
  - That’s a total of 2^41 / 2^12 = 2^29 blocks
  - So the bitmap needs 2^29 bits
  - That’s 2^26 bytes, or 64MiB
  - This is only 0.003% of the total disk space “wasted” to store the bitmap
- A file system can keep two bitmaps
  - A bitmap of free inode blocks
  - A bitmap of free data blocks

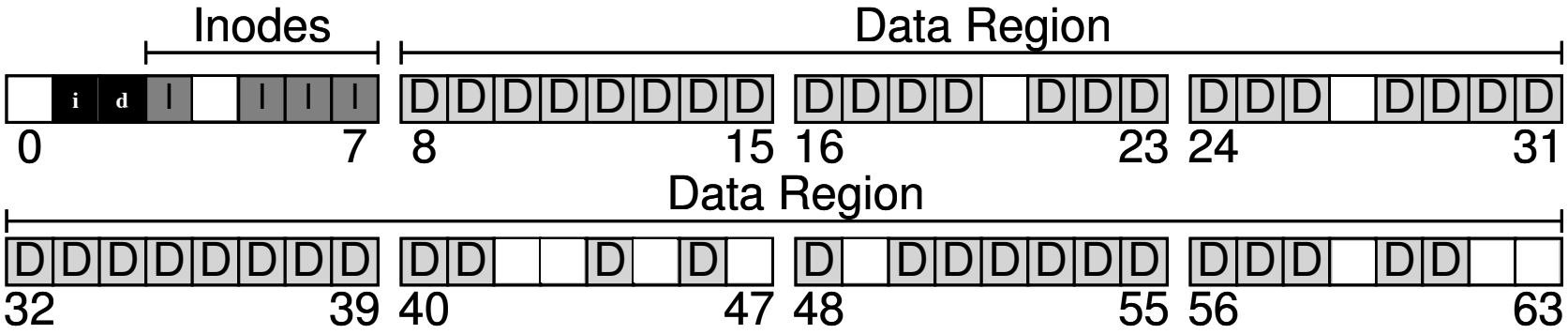
Bitmap of Free Blocks



- Main idea: we have an array of bits, one per block index on disk
  - 0 means "free", 1 means "not free"
- This is called a bitmap
- A bitmap doesn't take much space
  - Say our disk is 2T with 4KiB blocks
  - That's a total of 2^
  - So the bitmap nee
  - That's 2^26 bytes,
  - This is only 0.003% bitmap
- A file system can keep two bitmaps
  - A bitmap of free inode blocks
  - A bitmap of free data blocks

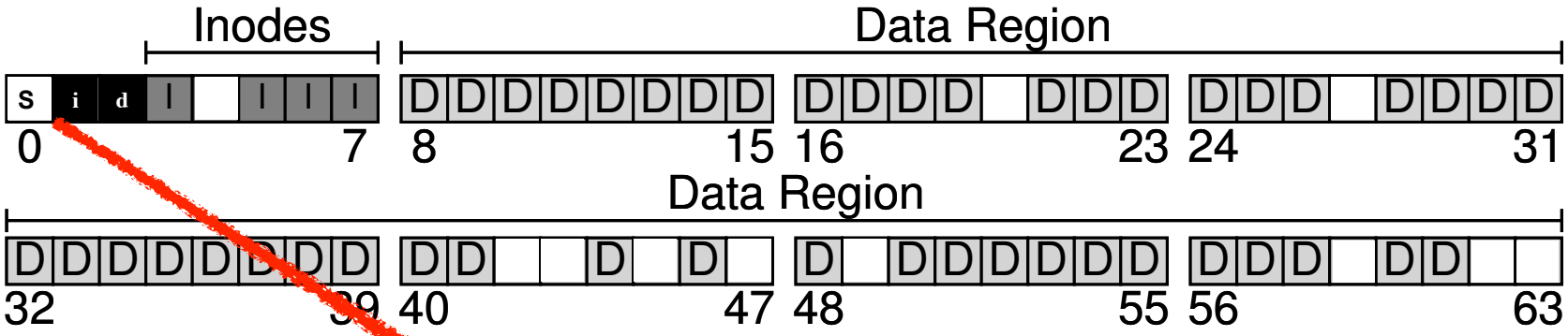
Shown in OSTEP like this, but each bitmap can span fewer or more than one disk block

Superblock



- There needs to be information on disk about the file system as a whole
  - Which type of file system
  - The block size
  - The total number of blocks
  - Where the bitmaps are
  - Where the data region begins
  - Where the inodes region begins
- OSTEP calls this the superblock, A UNIX terminology
  - In NTFS is called the Master File Table, in FAT it's called the boot sector, etc.

Superblock



- There needs to be information on disk about the file system as a whole
  - Which type of file system
  - The block size
  - The total number of blocks
  - Where the bitmaps are
  - Where the data region begins
  - Where the inodes region begins
- OSTEP calls this the superblock, A UNIX terminology
  - In NTFS is called the Master File Table, in FAT it's called the boot sector, etc.

Recap So Far

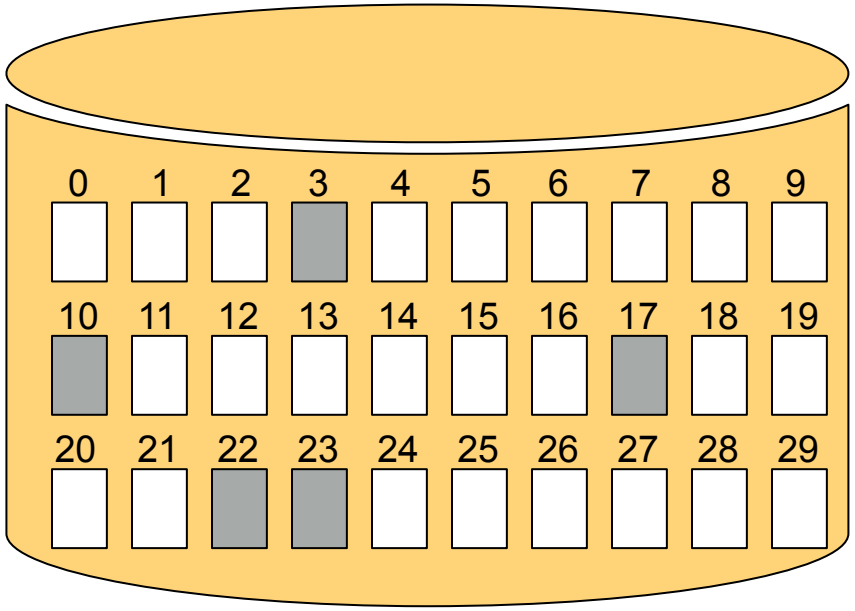
- The File System on disk is, essentially:
  - A bunch of data blocks
  - For each file, blocks that contain a data structure that makes it possible to find information about the file and to locate all its data blocks
  - Blocks that contain data structures that make it possible to keep track of free blocks
  - Blocks that contain a master data structure that makes it possible to find all the other data structures
- Next up: what data structure should we use to keep track of a file's blocks?
  - The "inode"

The inode Data Structure

- The term “inode” comes from “index node”
- This is the “low-level” name of a file
  - That we saw printed on the terminal with `ls -i`
- As we said before, the inode contains all possible information about each file, or metadata
- The key information an inode needs to encode is a way to find all of the file’s blocks
- And all inodes should have the same size otherwise the file system implementation becomes much more complicated
- Let’s look at a few options for the inode data structure…

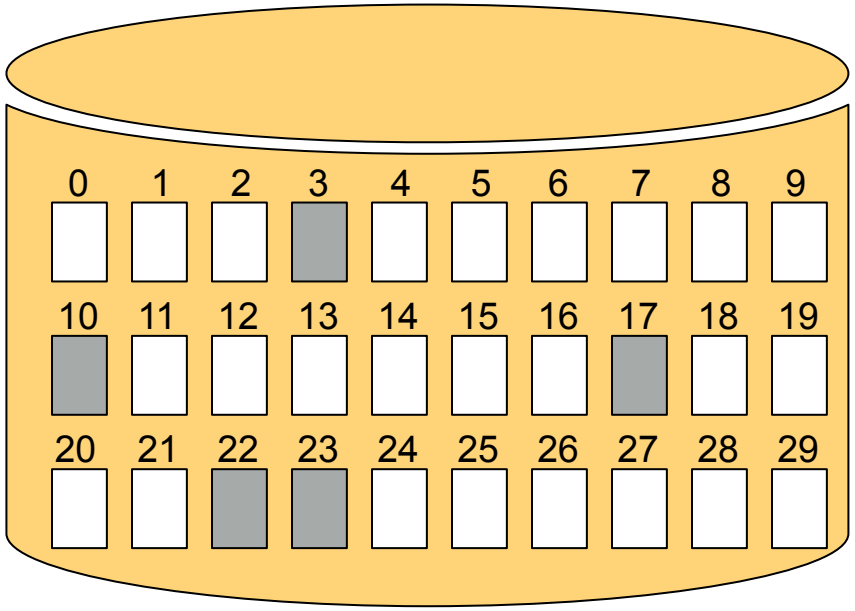
An Array of Direct Pointers?

- Consider a file that consists of these 5 blocks



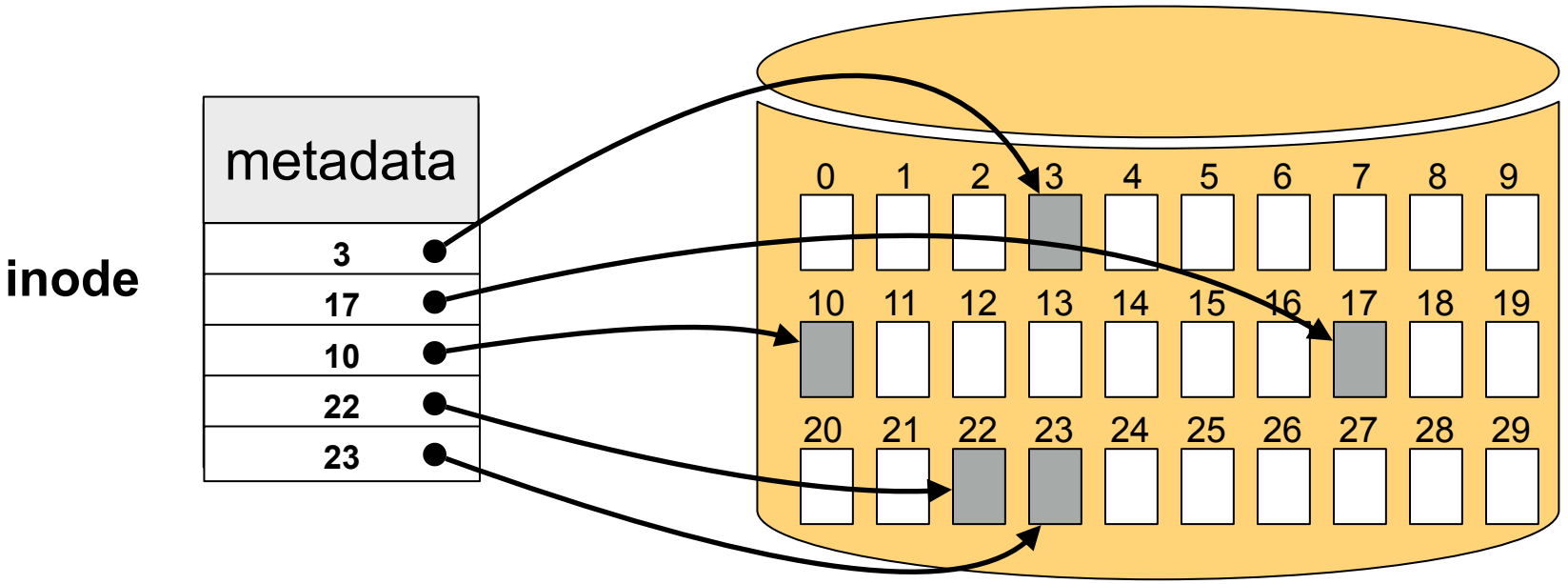
An Array of Direct Pointers?

- **Simple option:** The inode stores an array of direct pointers
  - Basically, the list of all blocks that belong to the file



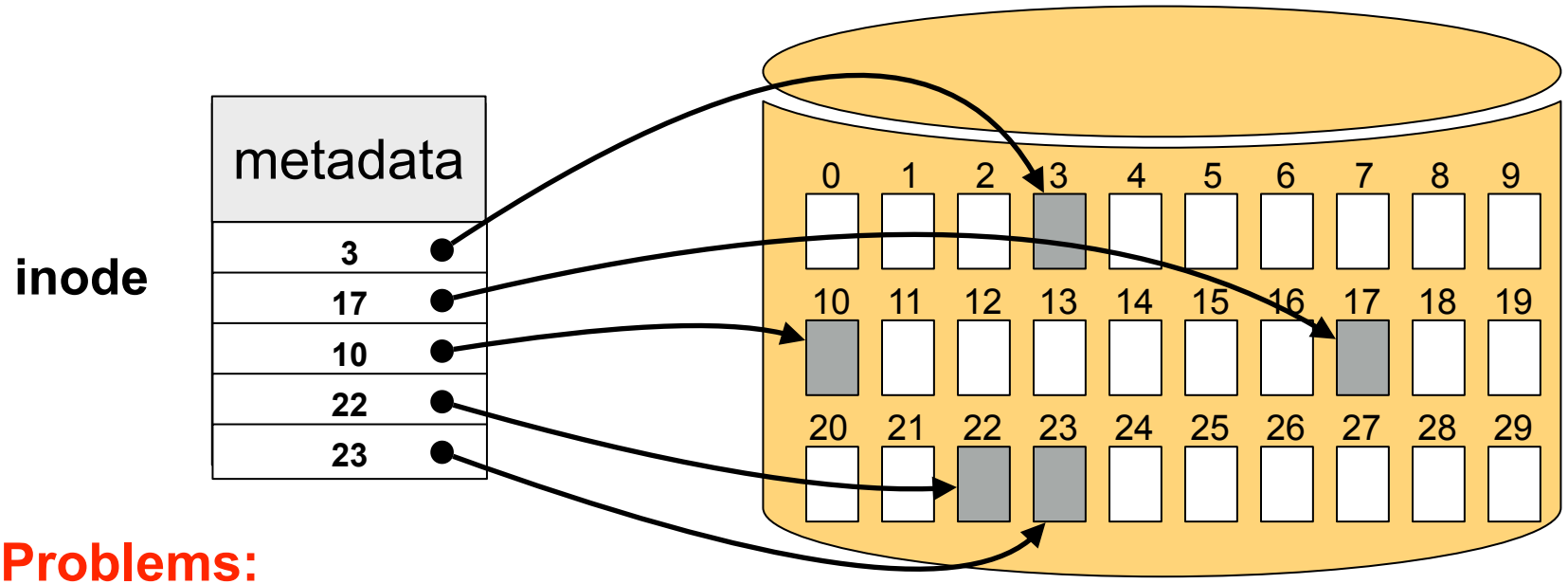
An Array of Direct Pointers?

- **Simple option:** The inode stores an array of direct pointers
  - Basically, the list of all blocks that belong to the file



An Array of Direct Pointers?

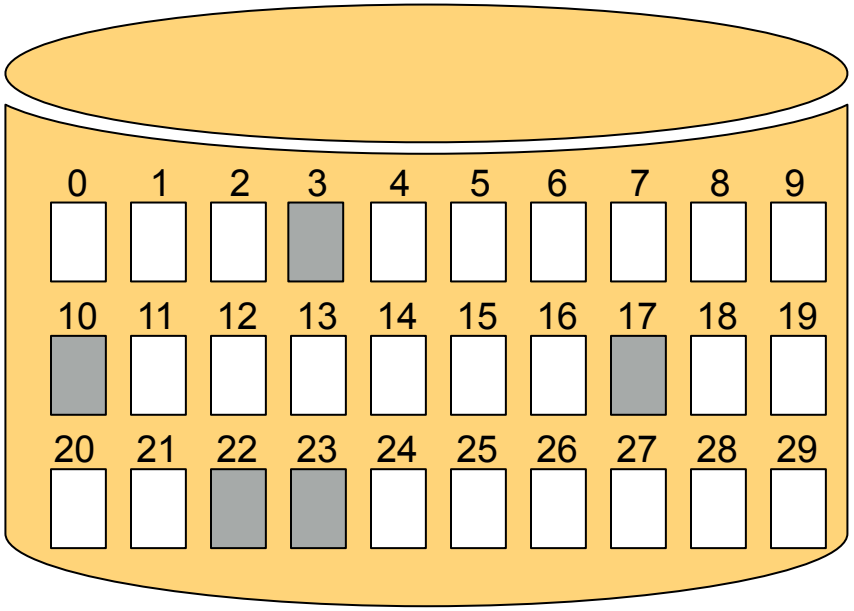
- **Simple option:** The inode stores an array of direct pointers
  - Basically, the list of all blocks that belong to the file



- **Problems:**
  - If the inode has n pointers and you have a 1-block file, you're wasting n-1 pointers: so n should be small
  - If the inode has n pointers and you want to store a file that has more than n blocks, you cannot: so n should be large
  - Picking a good n is not easy :)

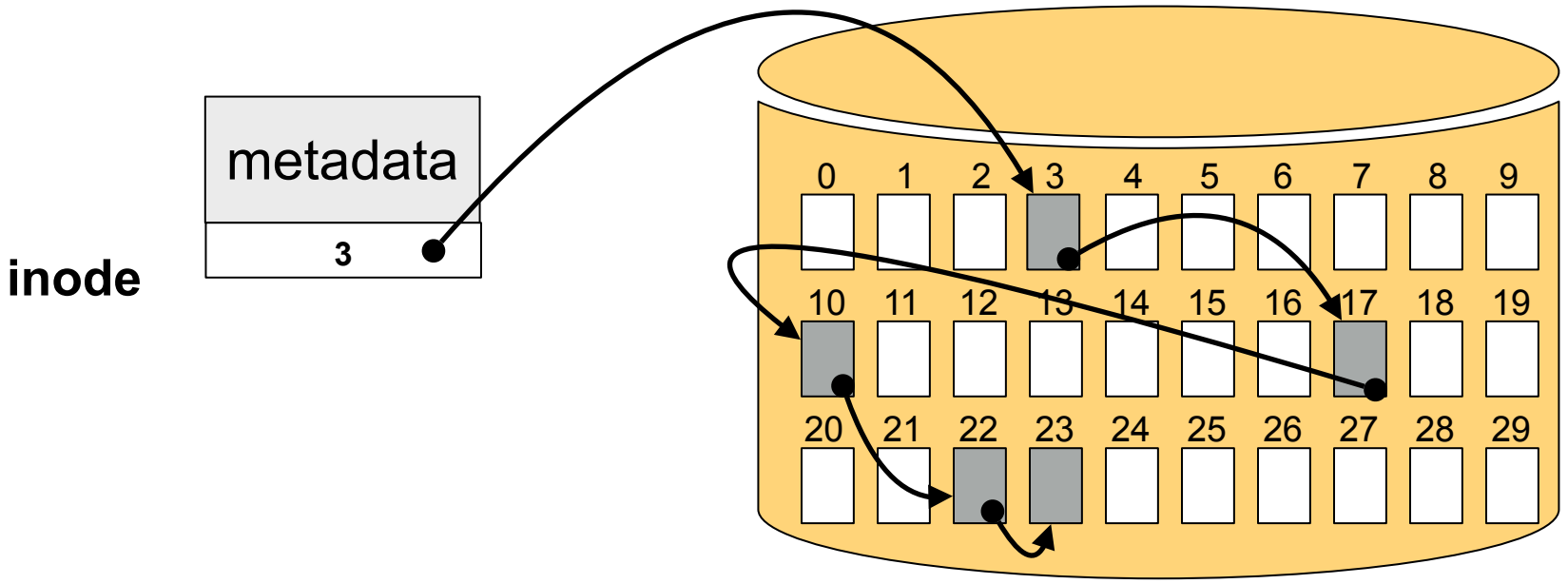
An On-Disk Linked List?

- **One solution:** just use an on-disk linked list
  - A few bytes in each block are used to store the index of the next block



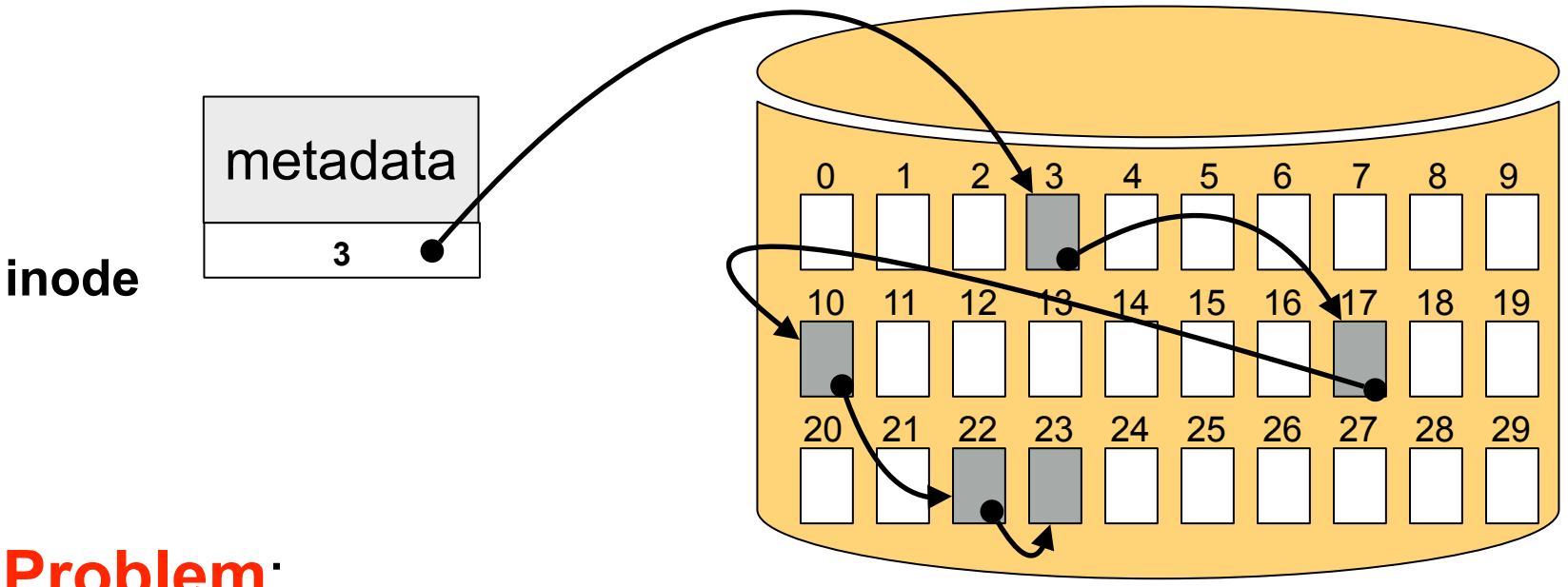
An On-Disk Linked List?

- **One solution:** just use an on-disk linked list
  - A few bytes in each block are used to store the index of the next block



An On-Disk Linked List?

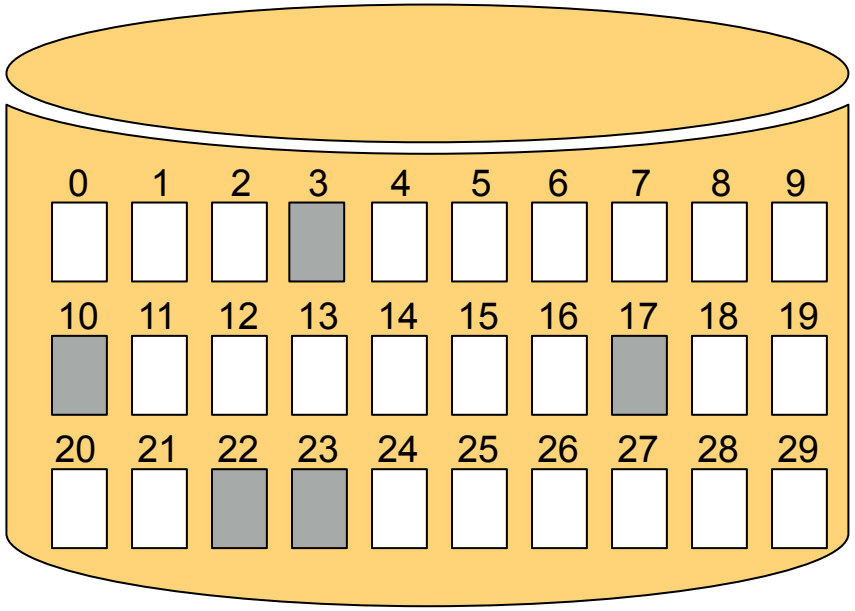
- **One solution:** just use an on-disk linked list
  - A few bytes in each block are used to store the index of the next block



- **Problem**:
  - Random access requires traversing the linked list, which is too slow (disk accesses are slooooow!)
  - If one block gets corrupted, then we lose all blocks after it

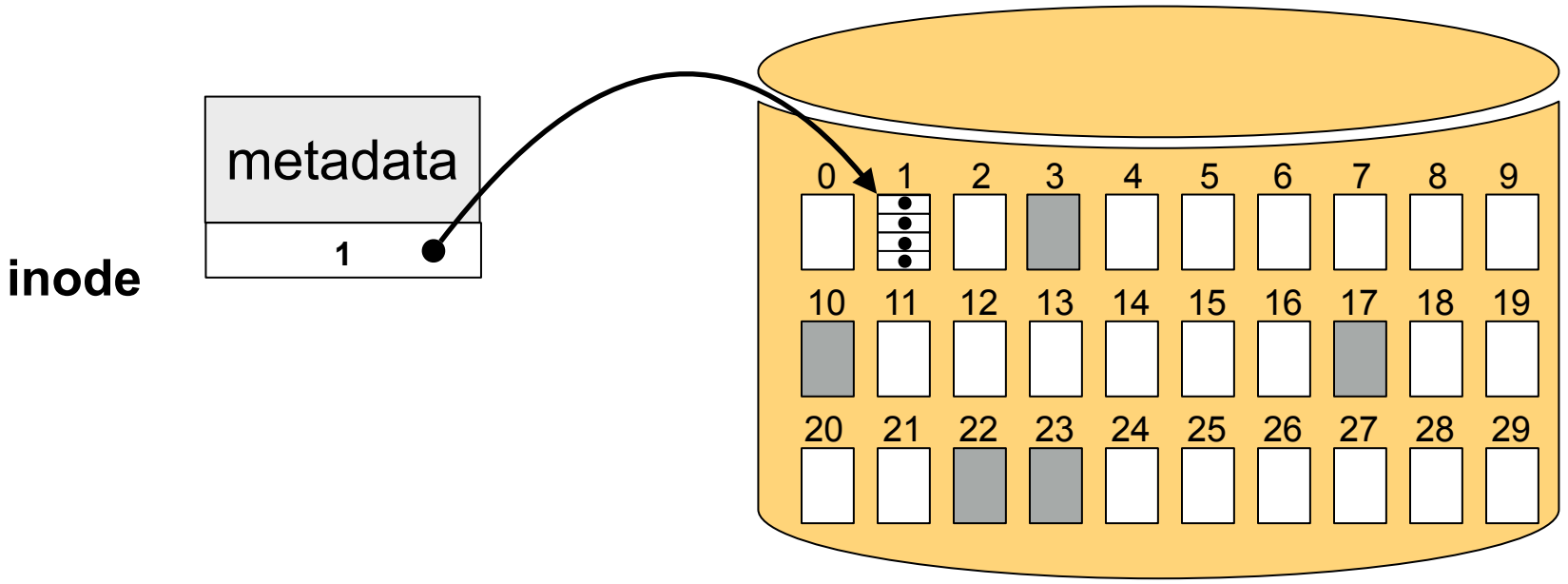
A Hierarchical Index?

- **Another solution:** Same idea as hierarchical page tables
  - The inode points to an index, which points to indices, etc.



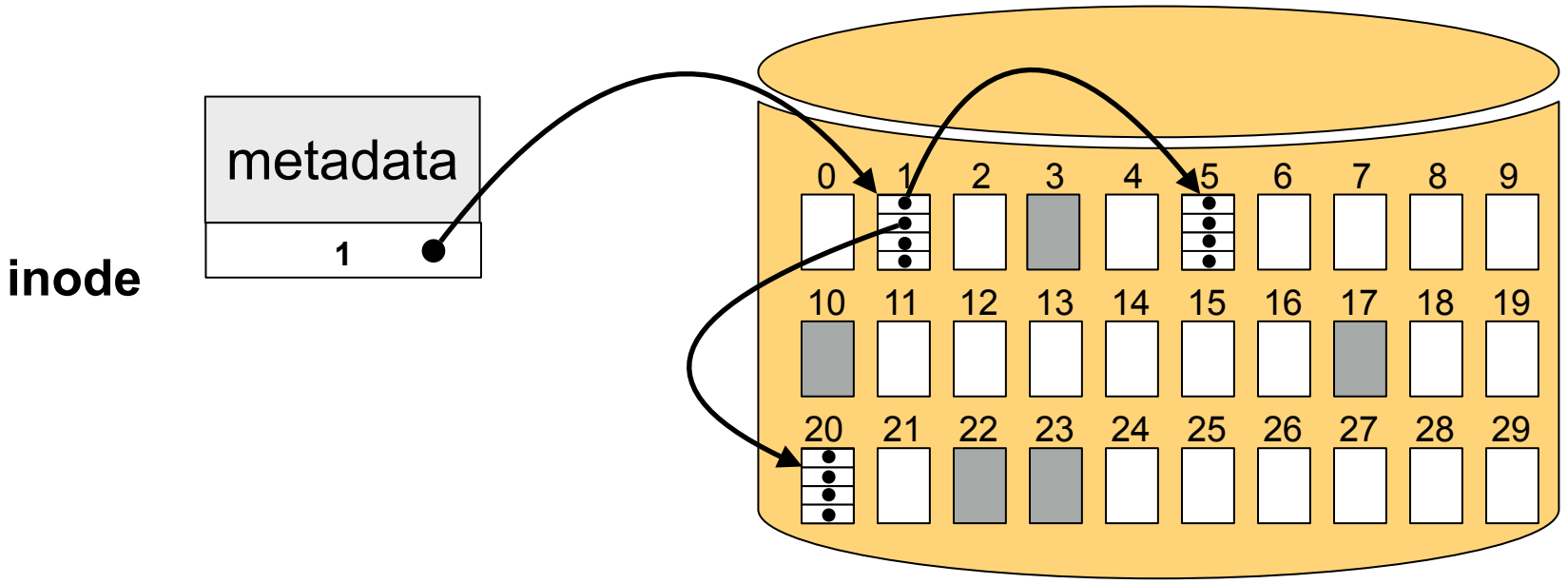
A Hierarchical Index?

- **Another solution:** Same idea as hierarchical page tables
  - The inode points to an index, which points to indices, etc.



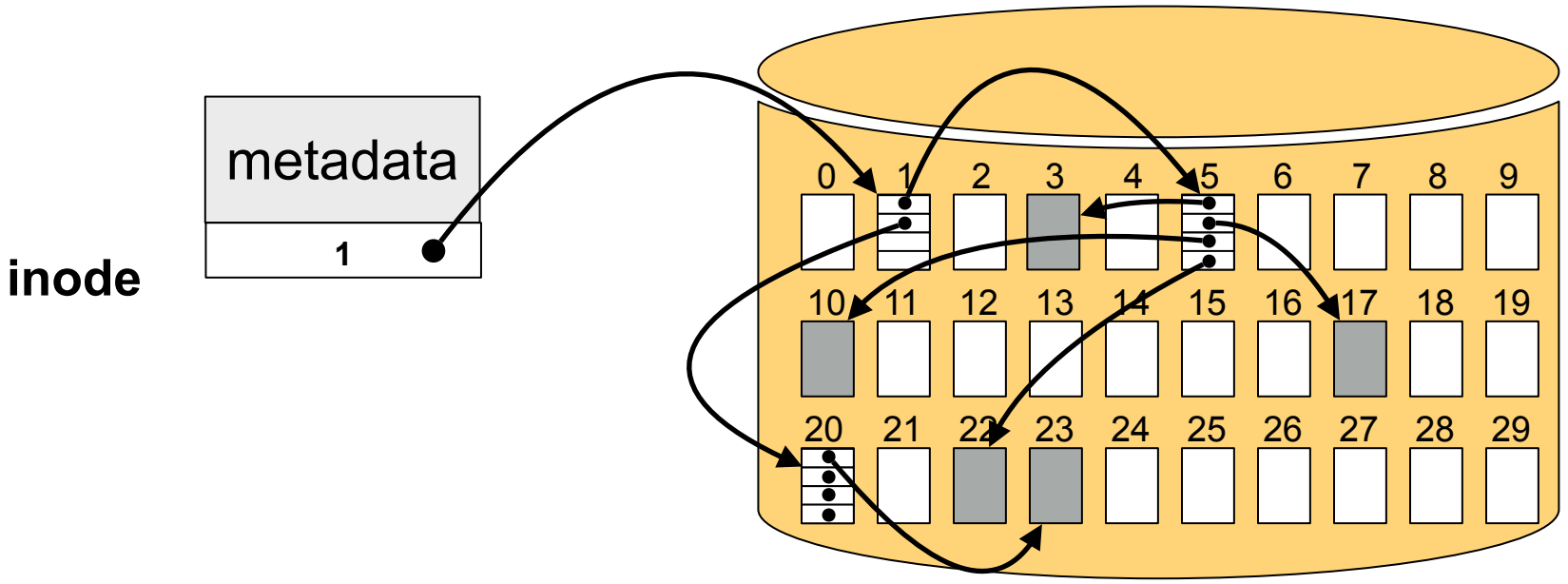
A Hierarchical Index?

- **Another solution:** Same idea as hierarchical page tables
  - The inode points to an index, which points to indices, etc.



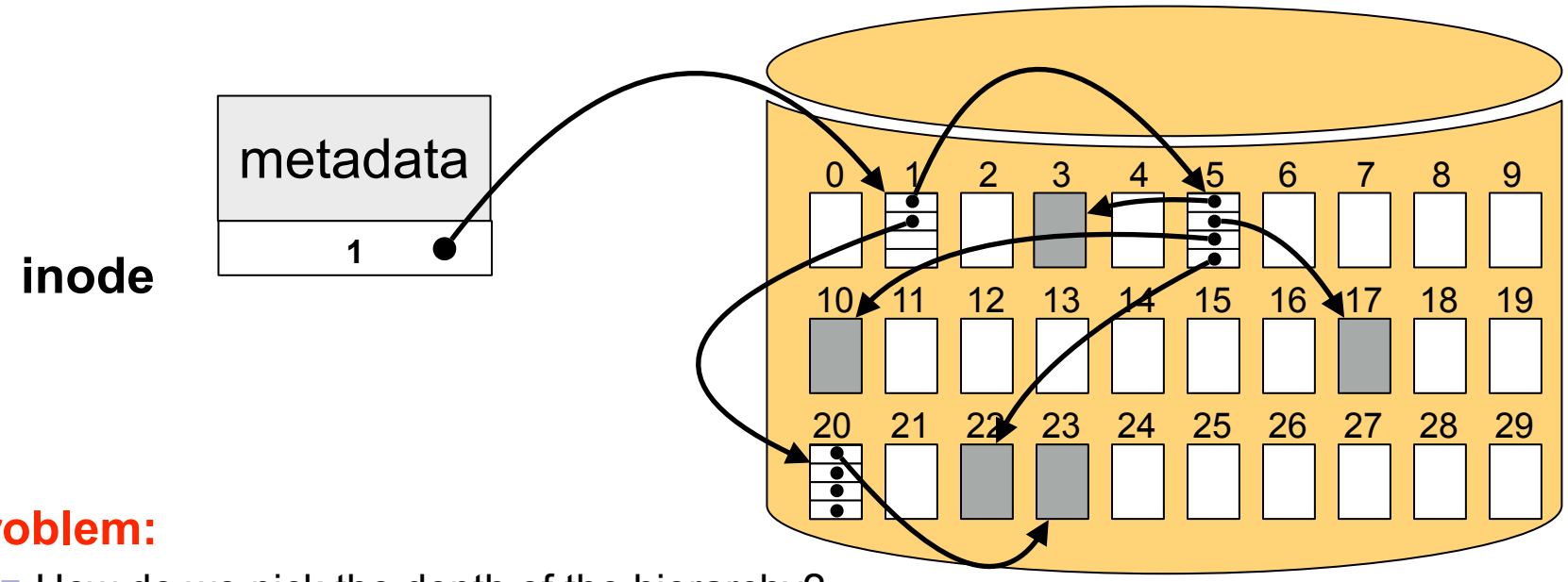
A Hierarchical Index?

- **Another solution:** Same idea as hierarchical page tables
  - The inode points to an index, which points to indices, etc.



A Hierarchical Index?

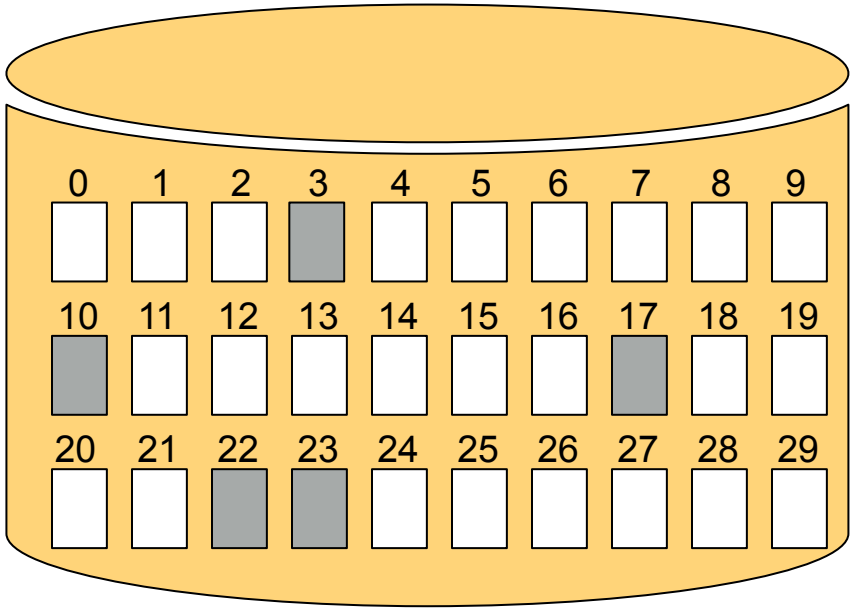
- **Another solution:** Same idea as hierarchical page tables
  - The inode points to an index, which points to indices, etc.



- **Problem:**
  - How do we pick the depth of the hierarchy?
  - Say we pick depth 10 because we want to accommodate large files
  - Then to access a 1-block file we need to access 10 blocks
    - We just made our disk 10x slower for small files!!!
    - And most files are small in practice, so we need to be fast for them!
  - Once again, we have a small file / big file problem….

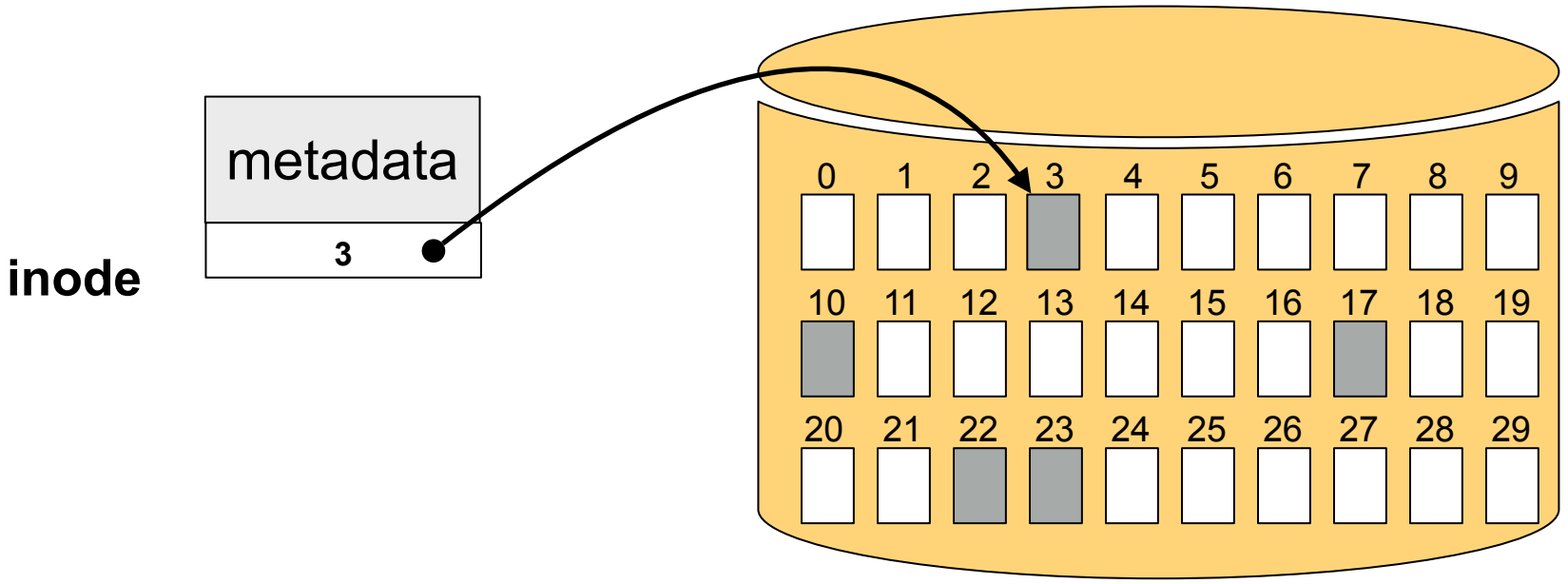
Multi-level Index

- Combine previous solutions into one: multi-level index



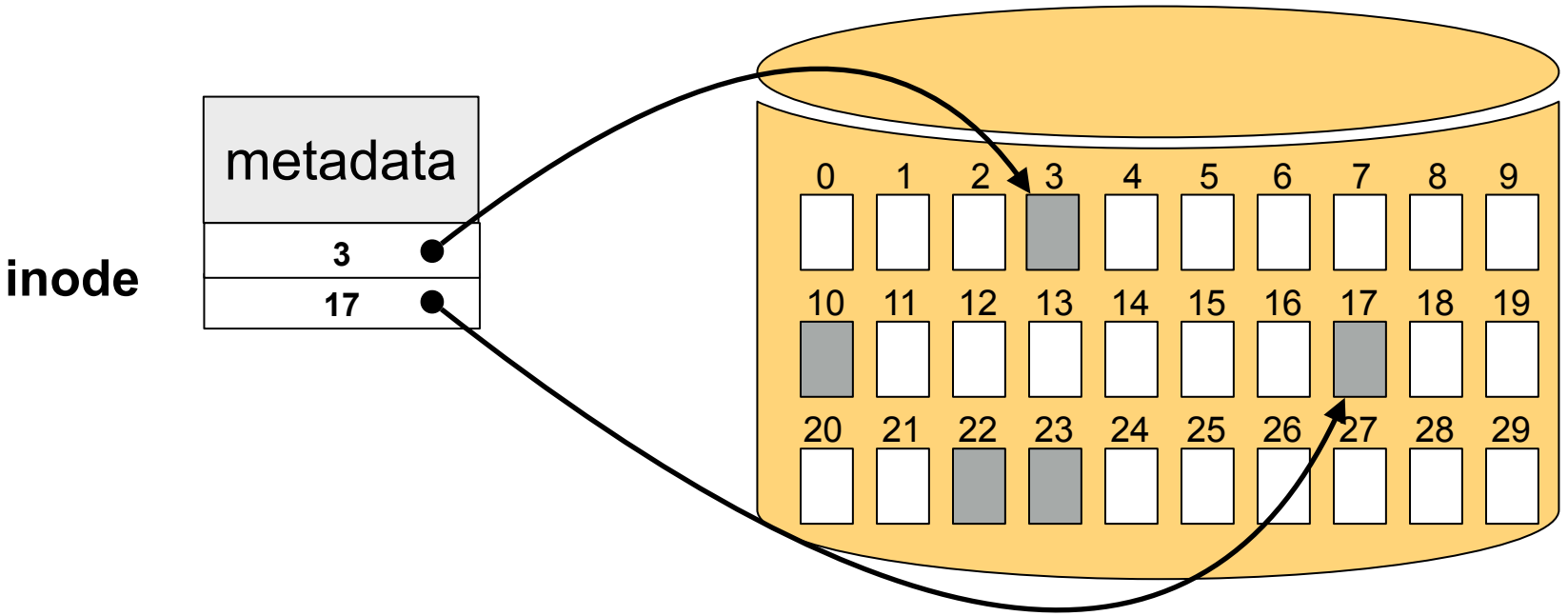
Multi-level Index

- Combine previous solutions into one: multi-level index



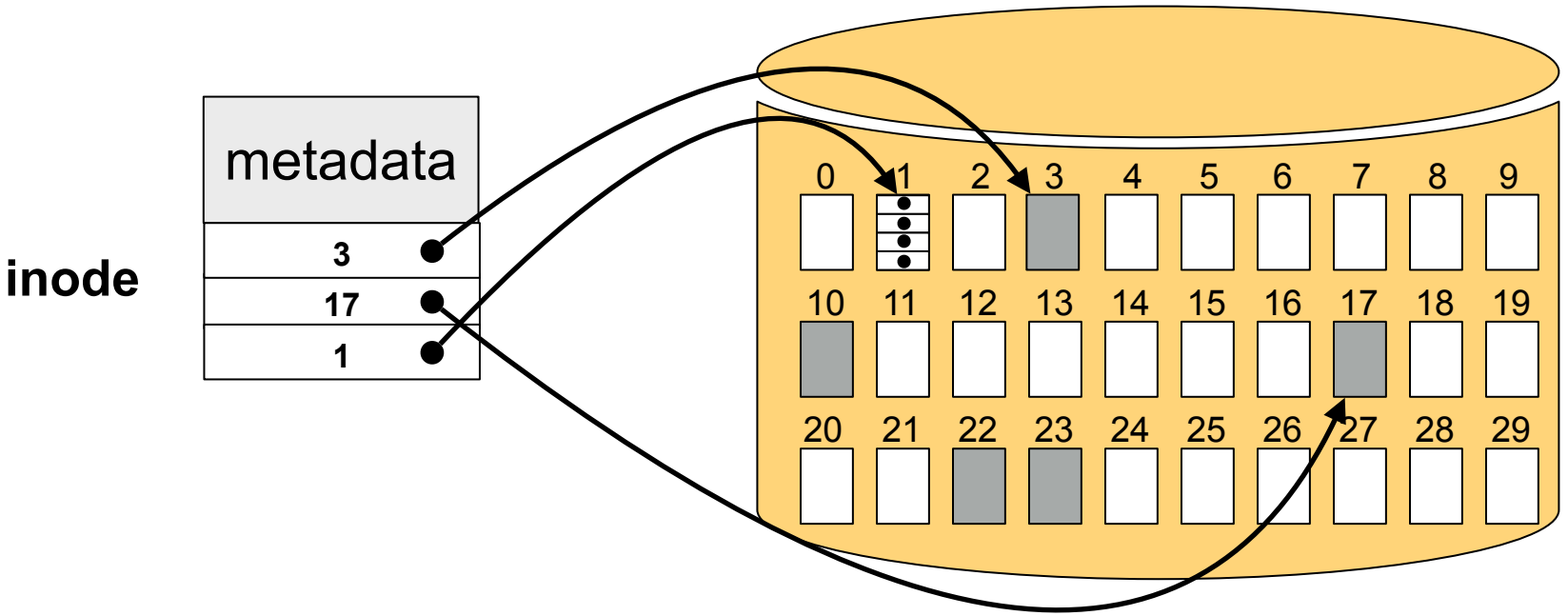
Multi-level Index

- Combine previous solutions into one: multi-level index



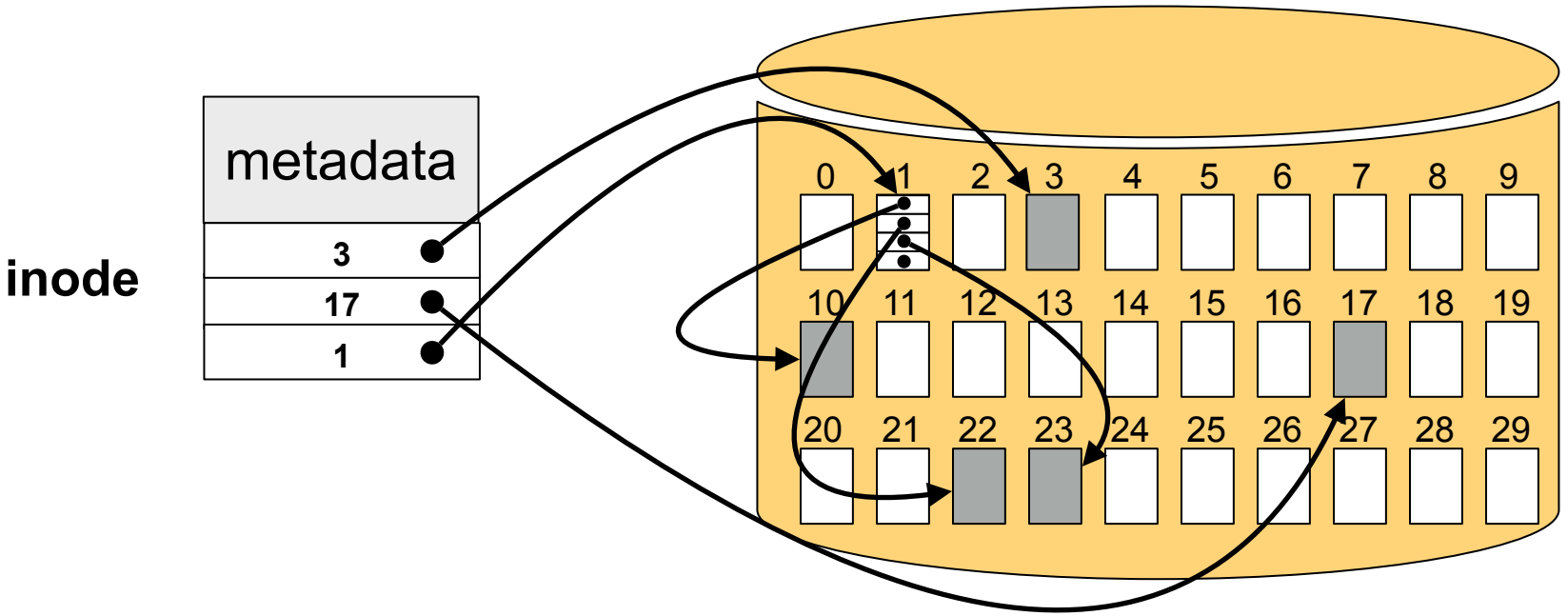
Multi-level Index

- Combine previous solutions into one: multi-level index



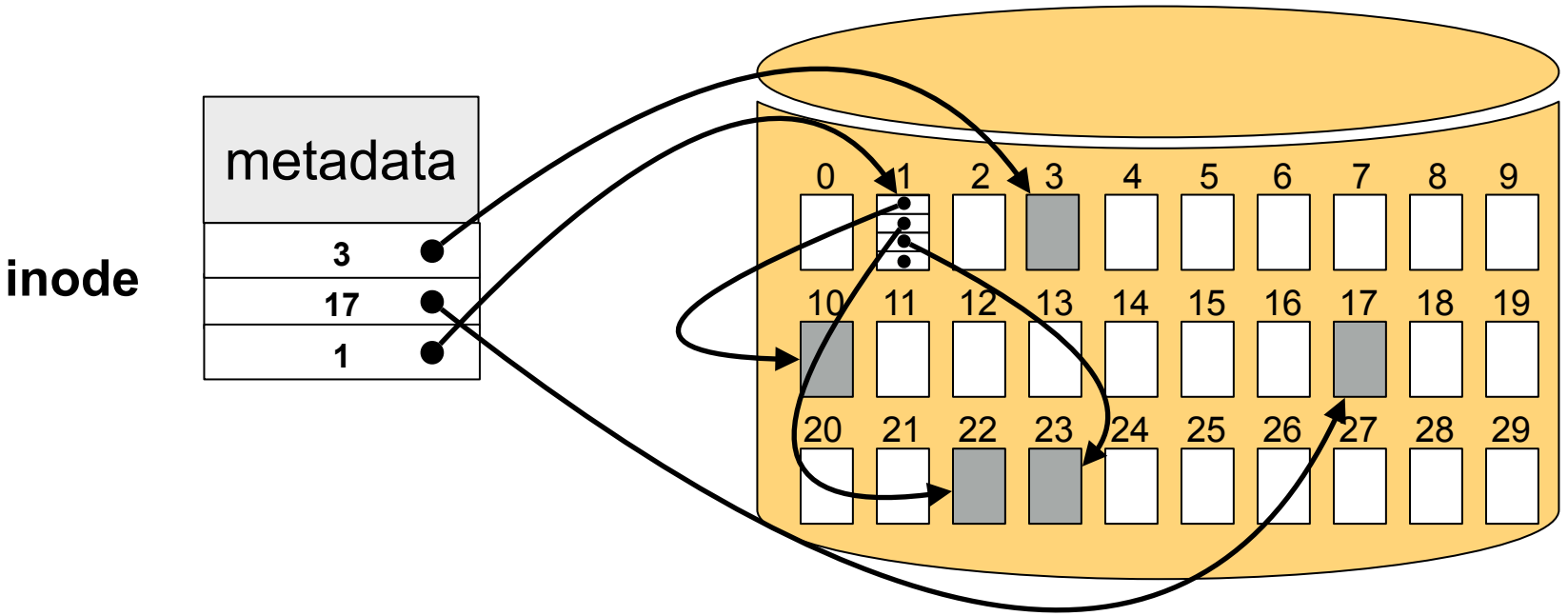
Multi-level Index

- Combine previous solutions into one: multi-level index



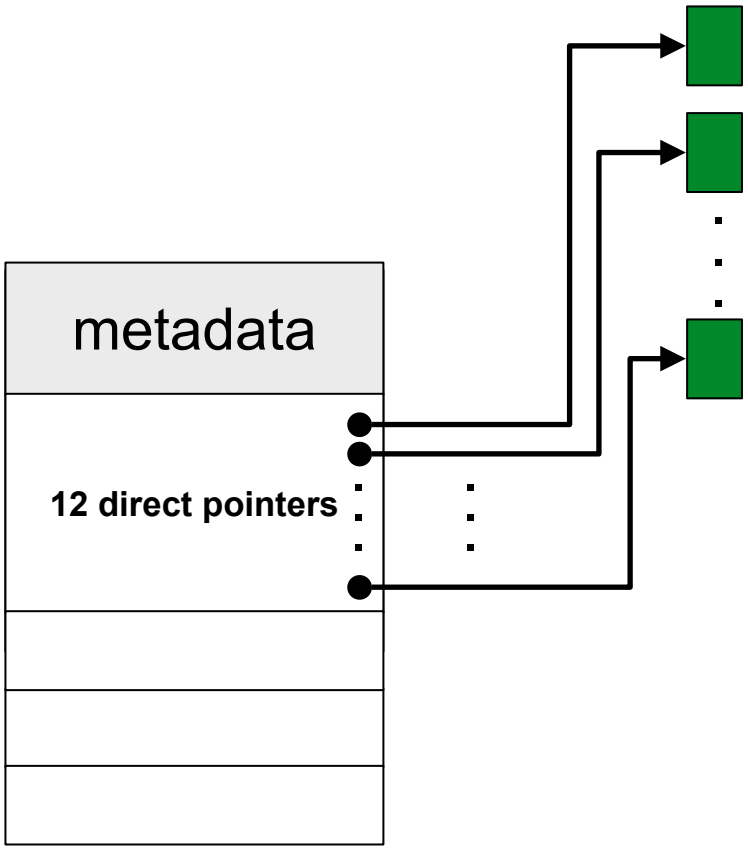
Multi-level Index

- Combine previous solutions into one: multi-level index

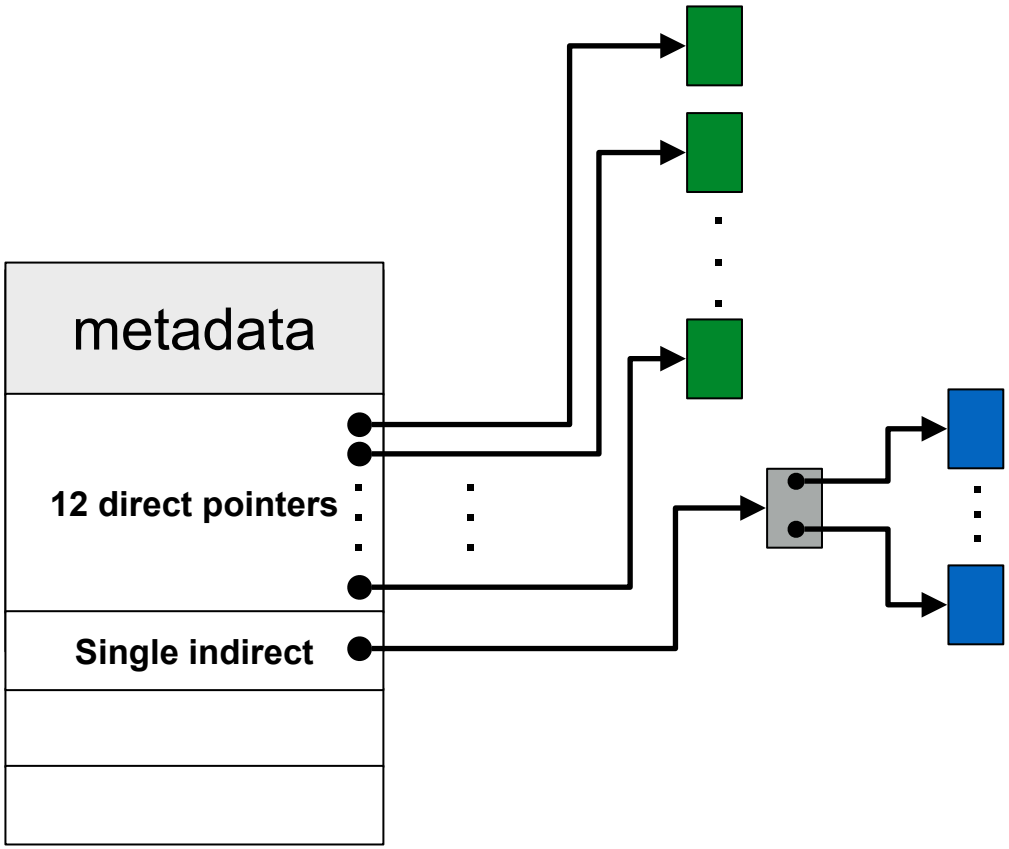


- In the above example: 2 “direct” blocks and 3 “single-indirect” blocks
- Many file system implementations use this idea

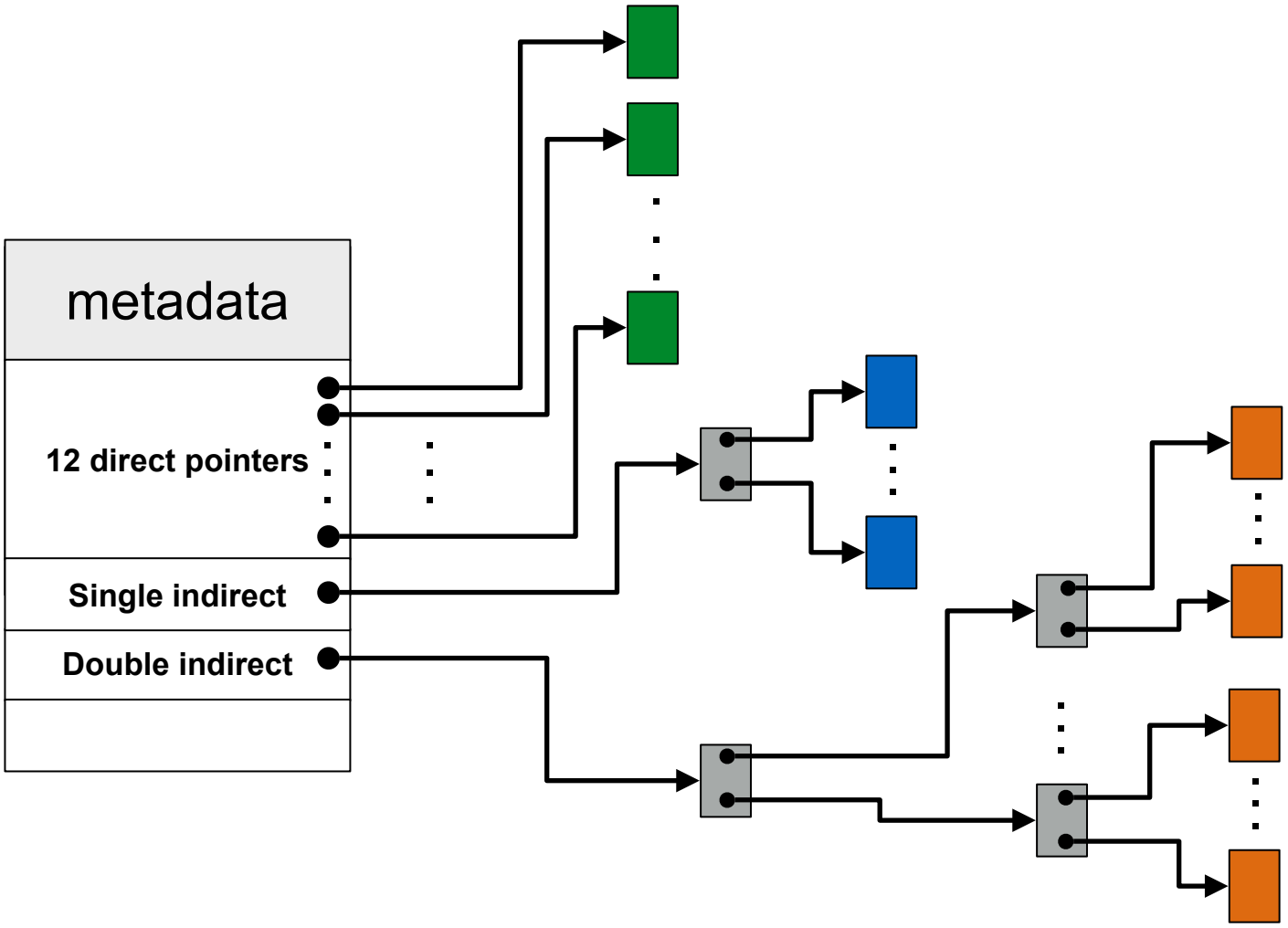
The UNIX inode



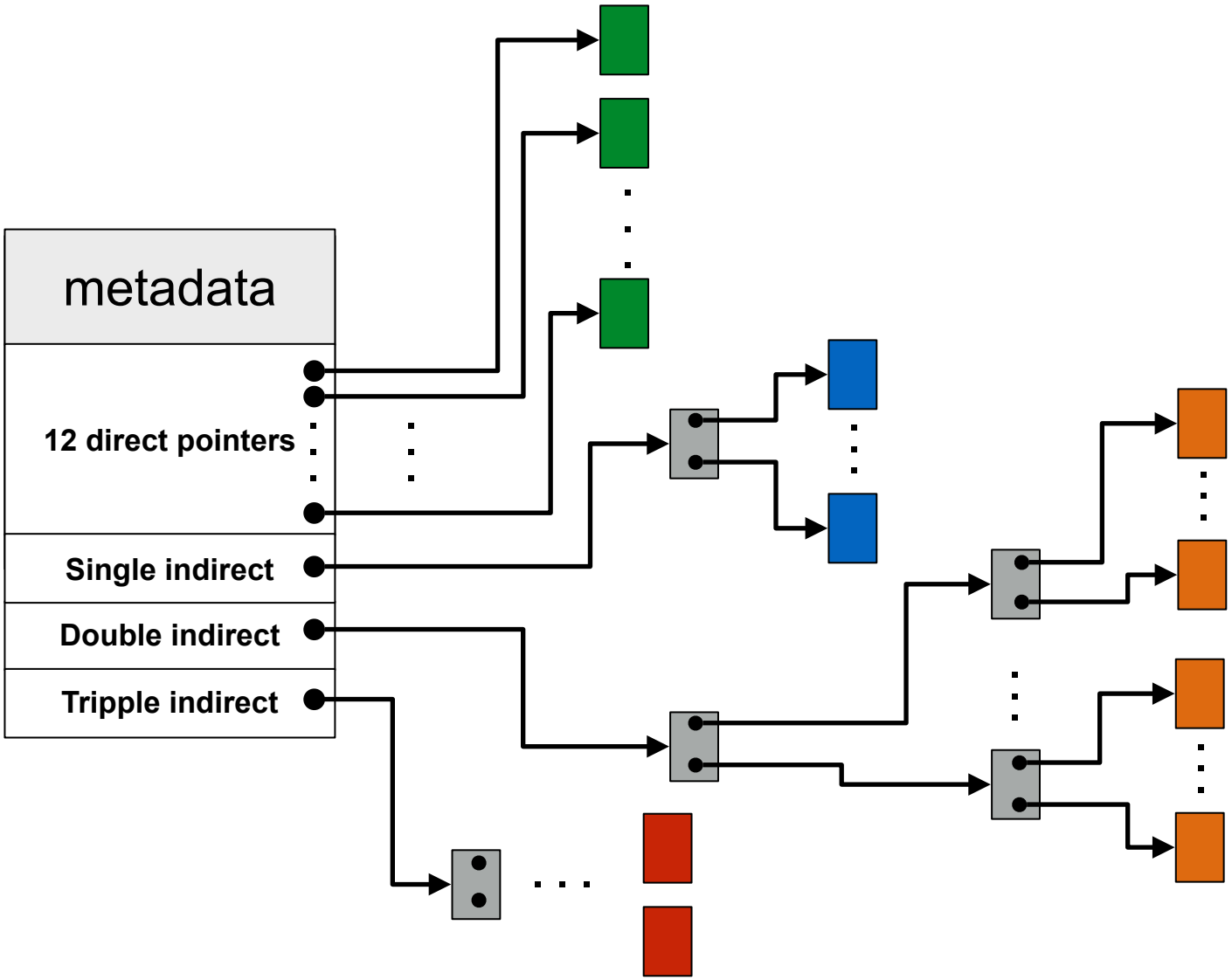
The UNIX inode



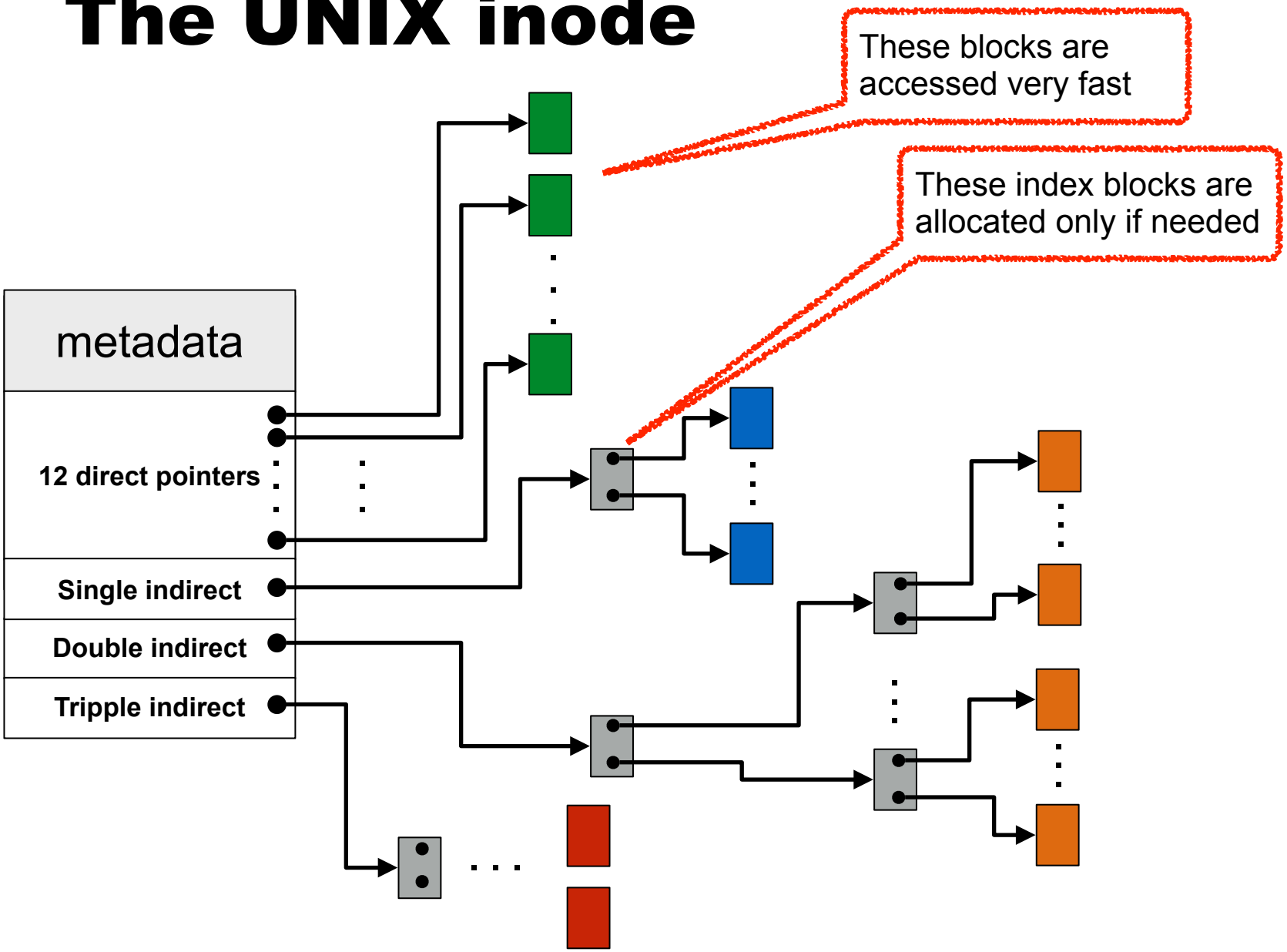
The UNIX inode



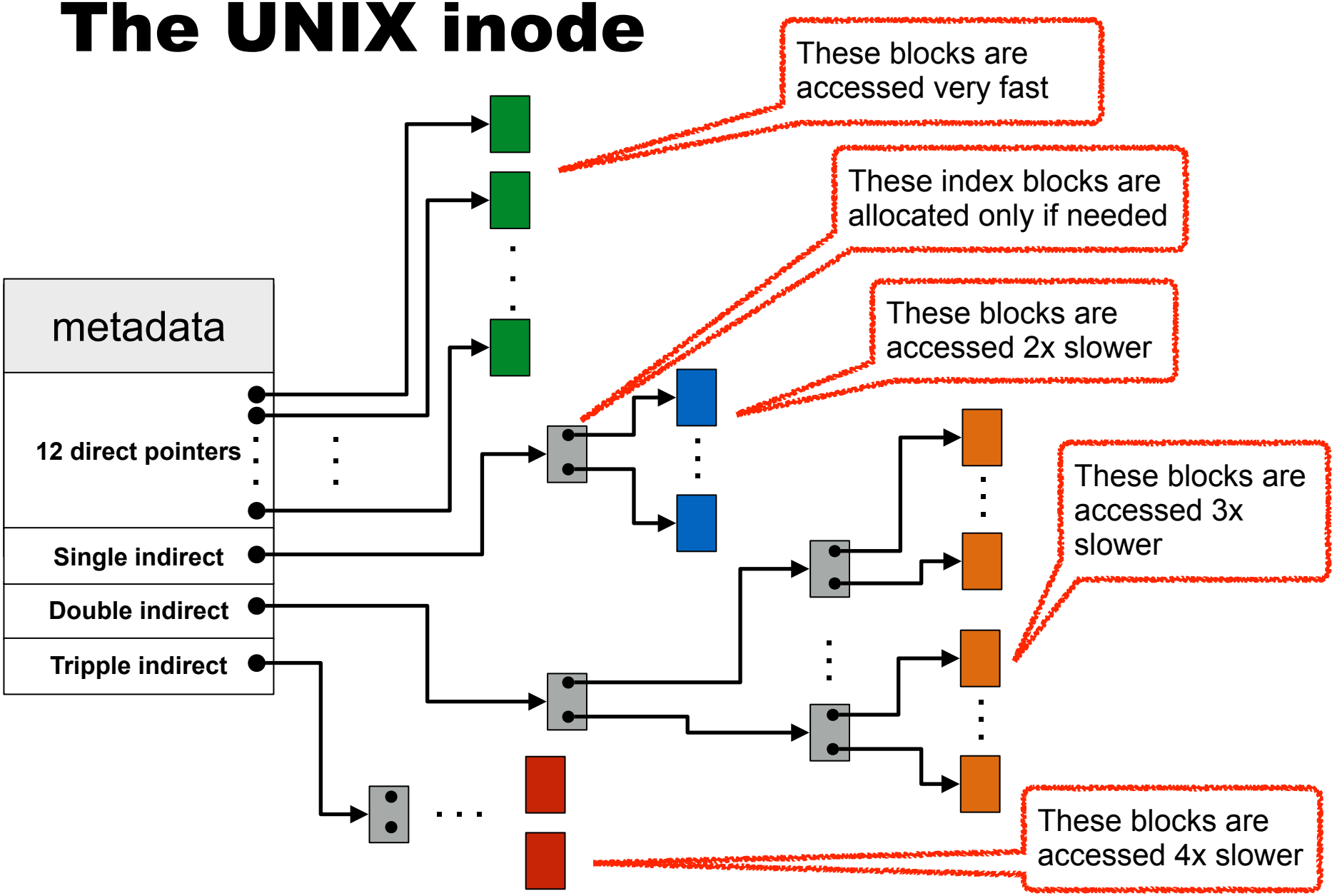
The UNIX inode



The UNIX inode

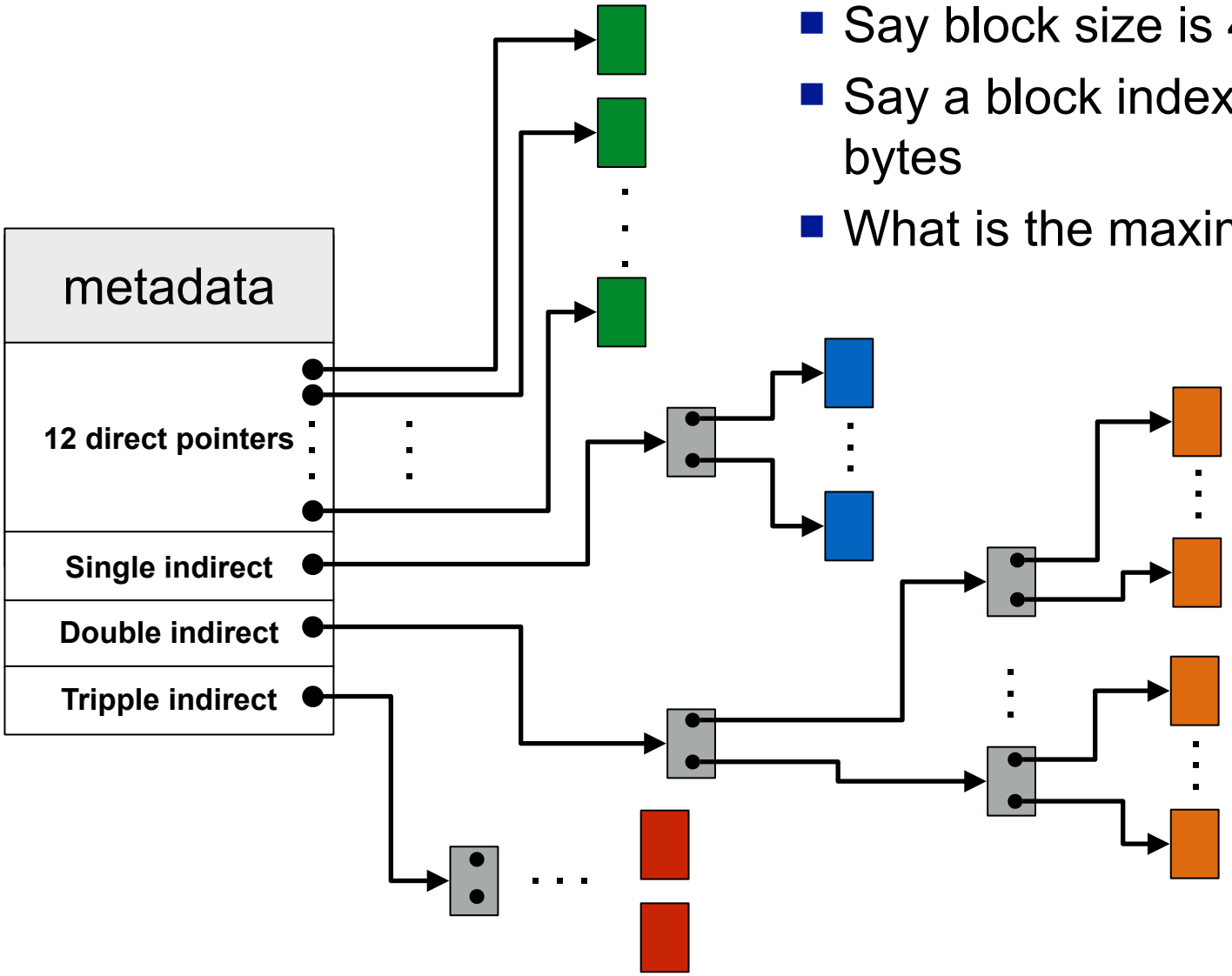


The UNIX inode

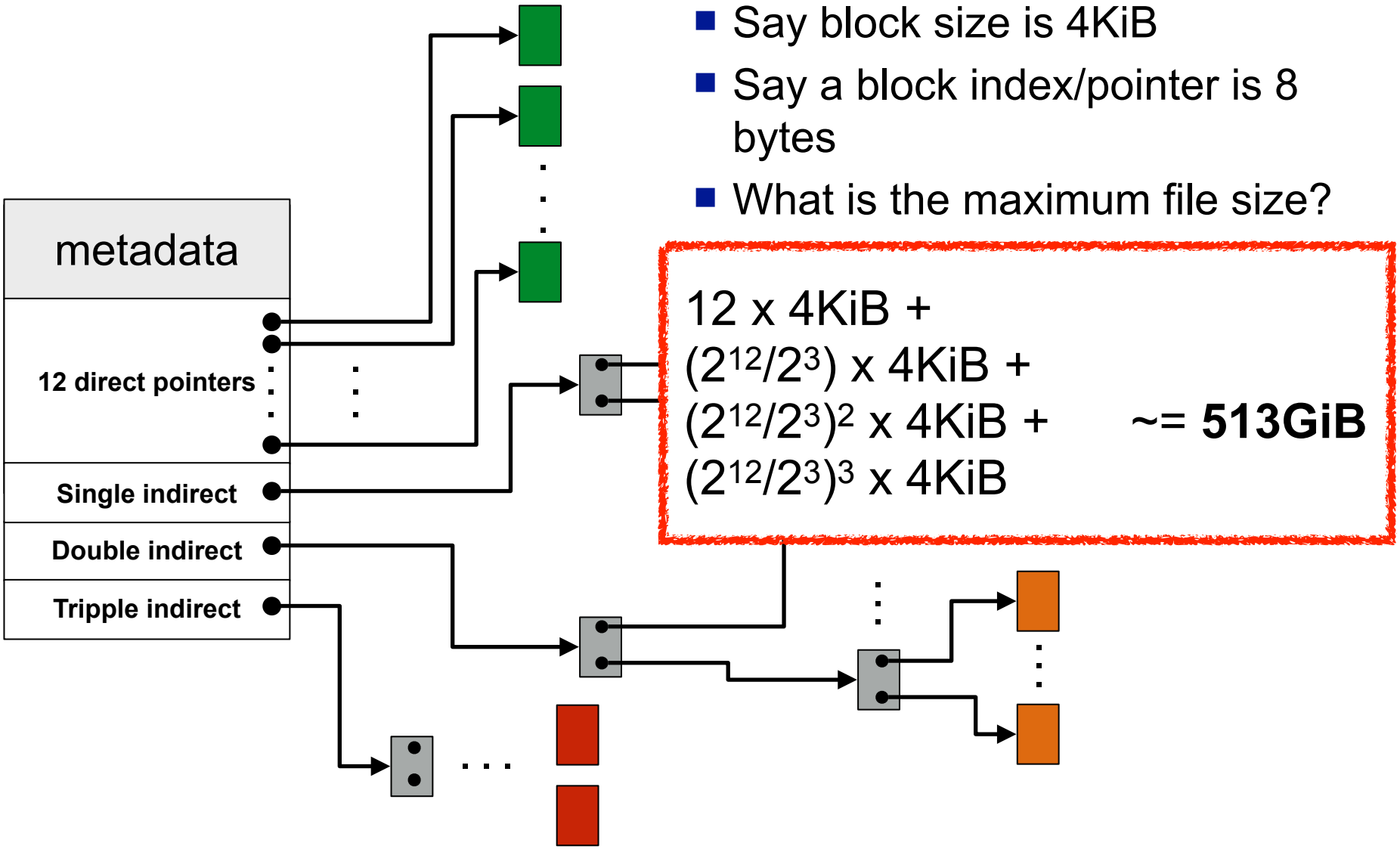


The UNIX inode

- Say block size is 4KiB
- Say a block index/pointer is 8 bytes
- What is the maximum file size?



The UNIX inode



- Say block size is 4KiB
- Say a block index/pointer is 8 bytes
- What is the maximum file size?

$12 \times 4\text{KiB} +$
$(2^{12}/2^3) \times 4\text{KiB} +$
$(2^{12}/2^3)^2 \times 4\text{KiB} +$
$(2^{12}/2^3)^3 \times 4\text{KiB}$

~= **513GiB**

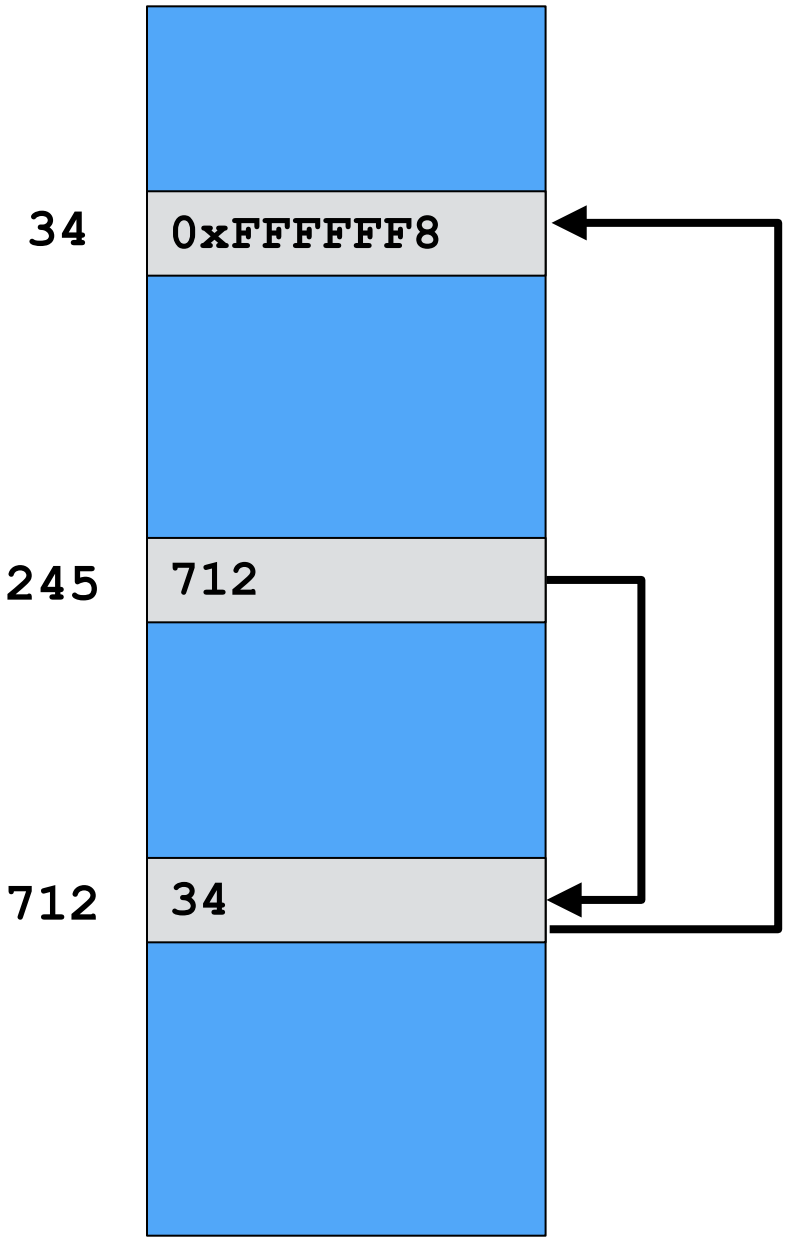
And now, for something completely different… FAT

- What we've described so far is a pretty standard UNIX approach
- An old, but still used today, filesystem on Windows is FAT (File Allocation Table)
  - NTFS is more recent, and uses a different kind of table
- The simple idea of FAT is that there is a table stored on disk, that is loaded (at least partially) in RAM upon boot
  - Since it's in RAM, accessing the table is fast!
- The table keeps track of clusters of contiguous file blocks, in a linked list manner
- Each entry in the table is for a cluster, and that table entry is the index of the next cluster
  - A "cluster" is simply some fixed number of disk blocks
- Finding free space is simple: free clusters are organized in a linked list, and one just need to find the first entry in the table that contains a zero
- Let's see this on a picture…

The FAT table

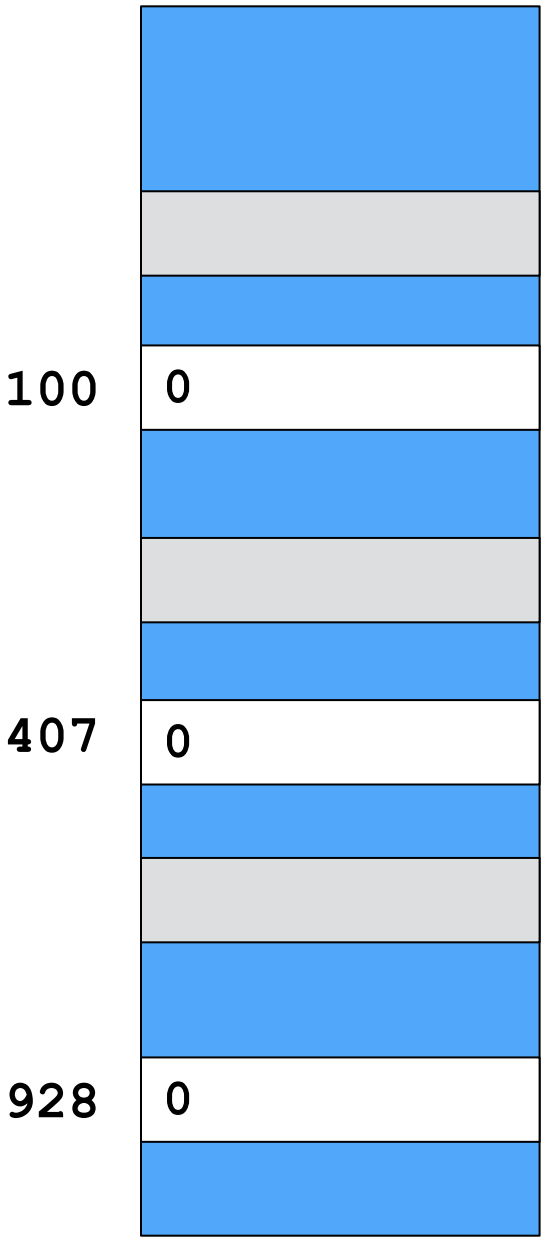
foo.txt	...	245

- A file entry in the file system just contains one FAT table index
- This is the index of a cluster
- The entry in the table for that cluster contains the index of the next cluster
- The last entry contains some reserved code that means “last cluster”



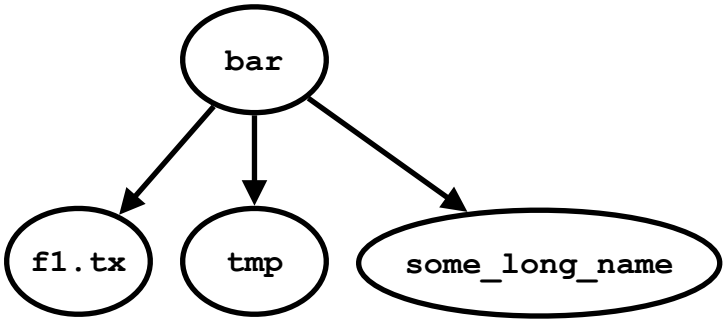
The FAT table

- Free clusters simply have an entry set to 0 in the table
- Finding free space means finding the first entry in the table that has value 0
- This is a O(n) search, but in memory
- Not super efficient
- Not great for fragmentation
- NTFS remedies this
  - Using a bitmap!



Directories

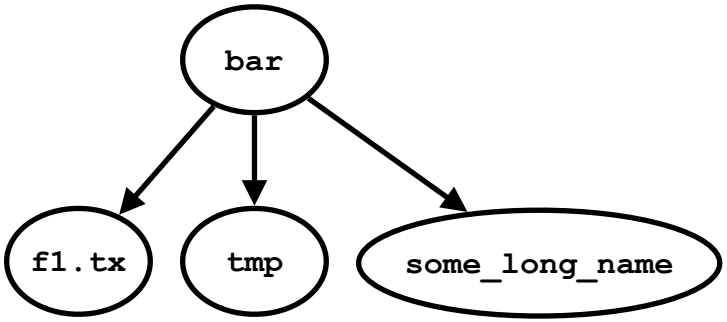
- A directory is described in an inode, **just like a normal file**
- But its content is a list of key-value pairs:
  - A user-level name (and perhaps a length)
  - An inode reference
- Each directory has two additional entries: "." and ".."
- For instance, a directory content on disk could be (encoded in binary):



inode#	rclen	strlen	Name
24	20	1	.
72	20	2	..
0	40	??	?????
23	20	6	f1.txt
189	20	3	tmp
121	40	14	some_long_name

Directories

- A directory is described in an inode, **just like a**
- But its content is a list of key-value pairs:
  - A user-level name (and perhaps a length)
  - An inode reference
- Each directory has two additional entries: "." a
- For instance, a directory content on disk could be (encoded in binary):



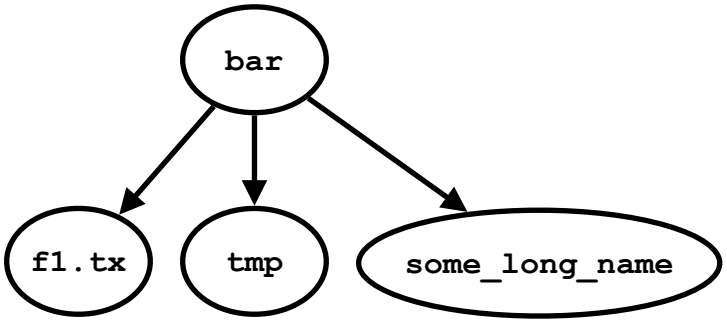
inode#	rclen	strlen	Name
24	20	1	.
72	20	2	..
0	40	??	?????
23	20	6	f1.txt
189	20	3	tmp
121	40	14	some_long_name

Length of this record, which is a multiple of some integer (here: 20). This means that each record has some unused bytes. But it simplifies the implementation if records are all multiples of the same integer

Directories

- A directory is described in an inode, **just like a**
- But its content is a list of key-value pairs:
  - A user-level name (and perhaps a length)
  - An inode reference
- Each directory has two additional entries: "." a
- For instance, a directory content on disk co d be (encoded in binary):

An inode number 0 means: this slot isn't used. This happens after a file is deleted. This empty record can then be reused later when a new file is created in the directory



inode#	rclen	strlen	Name
24	20	1	.
72	20	2	..
0	40	??	?????
23	20	6	f1.txt
189	20	3	tmp
121	40	14	some_long_name

Isn’t a Linear List O(n)??

- You have likely noted that in the previous slides we say that the directory contains a list of entries for its content
- This means we we have to do a linear search for a name in that list, which is O(n)
- If a directory has a lot of entries, then this can take a long time
- There are file systems that use better data structures for logarithmic-time searching (e.g., a B-tree)
- Everything you learn in 311 comes into play here
  - But the measure of complexity should be the number of disk blocks read/written, not some number of compute operations
- There are many, many, many file systems out there that have used or currently use all kinds of data structures
- But for now, let’s stick to our simple list…

Opening a Path

- Now that we understand how directories are stored, it's easy to see how to navigate the directory hierarchy to find a file
- Say the user does: open("**`/home/henric/ics332/file_system.pdf`**")
  - In the superblock find the address of the inode for "**`/`**" and load this inode into RAM
  - Load the data blocks pointed to by this inode until an entry for "**`home`**" is found, and then load that inode into RAM
  - Load the data blocks pointed to by this inode until an entry for "**`henric`**" is found, and then load that inode into RAM
  - Load the data blocks pointed to by this inode until an entry for "**`ics332`**" is found, and then load that inode into RAM
  - Load the data blocks pointed to by this inode until an entry for "**`file_system.pdf`**" is found, and then load that inode into RAM
  - FINALLY: access the data blocks points to by that inode, which is the file content we wanted
  - Assuming that each directory content fits in a single block, **this is 10 block loads before we can load the first data block of the file!!**
- This is a lot of I/O!!!

Opening a Path

- The previous slide is the reason why we have and **`open()`** system call, instead of something like:
  - **`read("/home/henric/ics332/file_system_implementation.pdf", 12)`**
  - **`write("/home/henric/ics332/file_system_implementation.pdf", data, 48)`**
  - **`lseek("/home/henric/ics332/file_system_implementation.pdf", 56)`**
  - …
- Furthermore, all (good) file systems cache path translations
  - i.e., the address/index of the inode for file "**`file_system_implementation.pdf`**" is remember after it's closed, just in case it's opened again later
  - A "software cache" managed using LRU
    - Like a TLB, but in software

Data Block Caching

- Most file systems implement some form of caching
  - Remember that disk controllers also implement their own caching
- When you read a (clean) block that you've read recently, likely you will get it from an in-memory cache rather than from the Disk
- When you write a block, likely it won't go to disk but stay in an in-memory cache
  - It could be written later whenever the disk is idle
  - Or it could never be written at all if the program re-writes it
    - Imagine a program that every 1ms writes one different byte in the block
    - This program should only write the block back to disk once its done!
  - And if the system crashes, you've lost data!
- Caching is the one idea that occurs EVERYWHERE in this course

Consistency Checking

- The File System shouldn't lose data or become inconsistent
- It's a fragile affair, with data structure pointers all over the place, and data/metadata cached in memory
- An abrupt shutdown can leave an inconsistent state
  - The system was in the middle of updating some pointers
  - Part of the cached data/metadata was never written back to disk
- One approach: perform consistency checking
- Consistency can be checked by scanning all the metadata
  - Takes a long time, occurs upon reboot if necessary
  - A "is it necessary to do the check?" bit is kept up-to-date by the system
  - Unix: **`fsck`**, Windows: **`chkdsk`**

- Overall philosophy: we allow the system to be corrupted, and we later attempt repair

Journaling

- Issue with consistency checking:
  - Some data structure that is damaged may not be repairable
  - Human intervention is needed
  - Checking a large file system takes a very long time
- Another option: Log-based transaction-oriented FS (Journaling)
  - Whenever the file system metadata is about to be modified, the sequence of actions, or *transaction*, to perform is written to a circular log and all actions are marked as “pending”
  - Then the system proceeds with the actions asynchronously, marking them as completed along the way
  - Once all actions in a transaction are completed, the transaction is “committed”
  - If the system crashes, we know all the pending actions in all non-committed transactions, so we can perform an undo
  - Writing to the log is overhead, but it’s sequential writing to the log file, and (on HDD) sequential writing is fast

Conclusion

- File Systems are considered part of the OS, but implementations are developed outside of the OS
  - It's an OS thing, but it's not part of the Kernel code
- File Systems are a huge topic and we only scratched the surface here
  - If you're into it: OSTEP Chapters 42, 43, 45, 48, 49, 50
  - There is a lot of research and development in this area (especially for Distributed File Systems)
- What we covered in this modules gives you the basics from which you can, if needed/desired, work towards becoming a file system expert