ZFS Copy on Write

Why Copy on Write?

- With many traditional file systems, updated data is overwritten in place. Open a saved Word doc, change a few lines, and save. When you save, your updates just overwrites the old data on the disk with the new, updated data.
- What it the system dies (crashes, loses power, explodes) halfway through this write? This can cause issues… you'll have half of old file and half of new file.
- If it's a more important data block (something saving essential system data), the whole file system could be corrupted!
- ZFS Copy on Write ***NEVER*** overwrites any data in place! It always writes the updated data to a new spot on the disk then makes a pointer to that new data. The pointer to the old data is gone, effectively freeing that space on the disk.
- ZFS' copy on write system allows the storage to perform "**atomic writes**", i.e., all data is effectively written to the system is done in a single instant!
- ZFS transitions from one consistent state to another consistent state without ever passing through an inconsistent state! Meaning, a sudden power loss or crash won't leave file system corrupted!

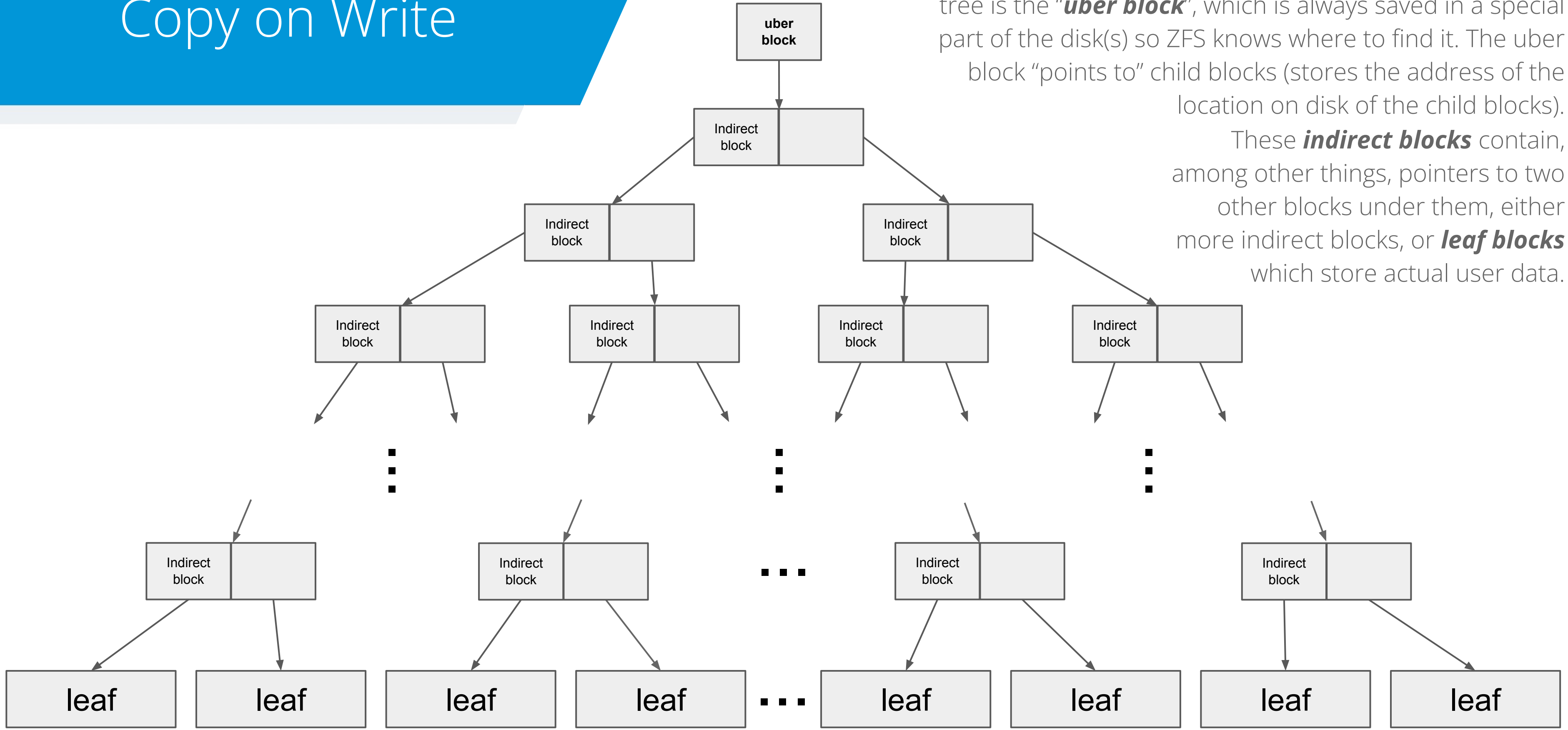
ZFS and TrueNAS Data Protection

OpenZFS

- **Copy on Write (CoW):** Data blocks (including metadata) are never overwritten in-place; ZFS writes updated blocks to an empty space on disk and creates new pointers to that data.
- **Atomic Writes:** CoW allows the filesystem to transition between consistent states without ever passing through an inconsistent state thus avoiding potential corruption from a sudden power loss or system crash.
- **Scheduled Snapshots:** ZFS can preserve the filesystem state at scheduled intervals to make rollbacks and recovery simple. The system only tracks changed data for better storage efficiency.
- **Scheduled Replication:** Snapshots can be replicated to a second system at the block level for extra protection.
- **Block Checksums:** All data and metadata blocks are checksummed when they are written and every time they are read.
- **Multiple Copies of Important Metadata:** ZFS stores at least two copies of all its metadata on disk to further reduce risk of total pool failure. Some metadata is so important that ZFS stores up to four copies on disk.
- **Automatic Pool Scrubs:** Checksums are also recalculated automatically on a monthly basis to preserve pool health.
- **Pre-fail Disk Replacement:** ZFS monitors disk health and will attempt to mark a disk as failed before it dies completely.
- **Granular Alert Configuration:** TrueNAS includes 70 different alertable events all with configurable severity levels.

iXsystems™

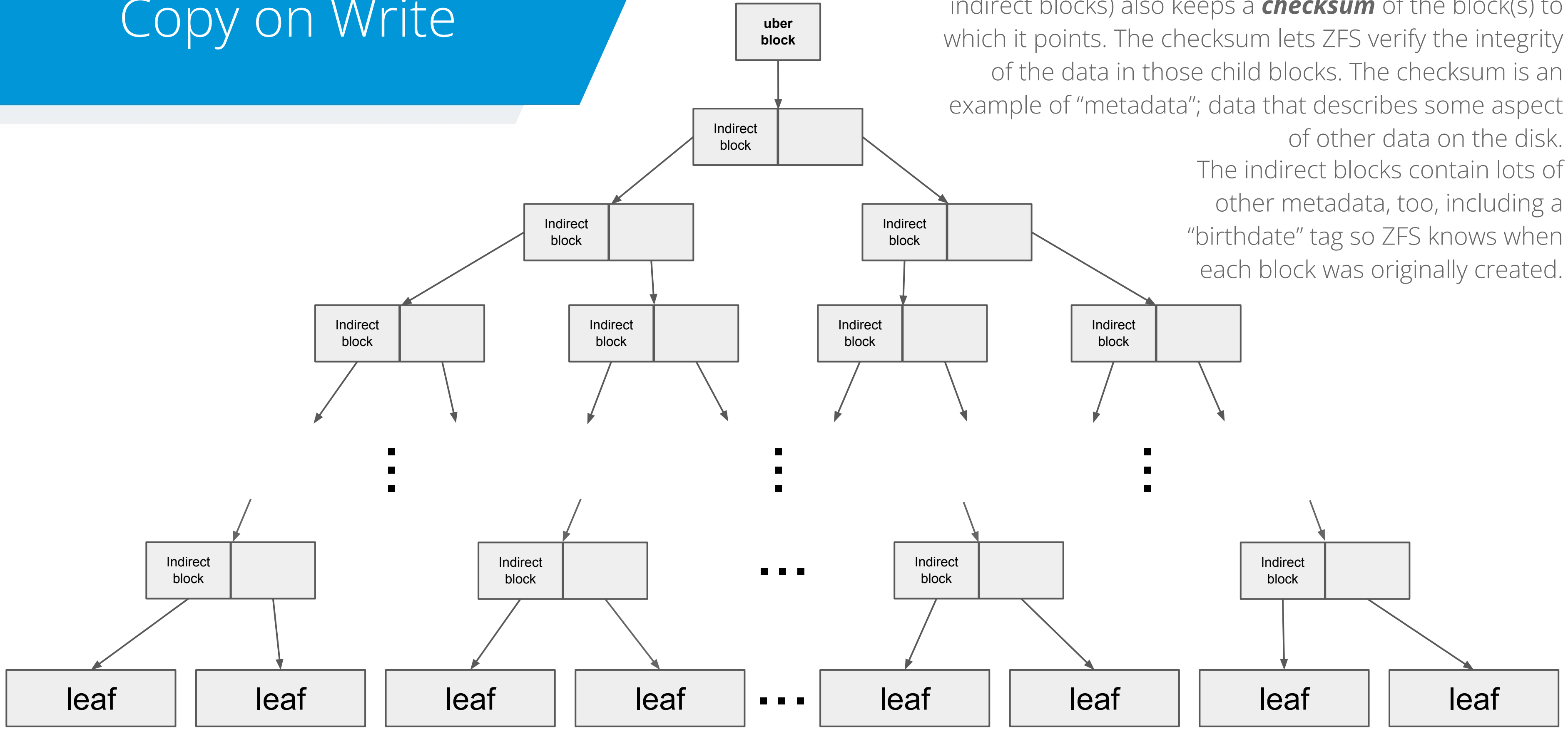
Copy on Write



ZFS keeps all data in a "tree" of blocks. At the top of the tree is the "***uber block***", which is always saved in a special part of the disk(s) so ZFS knows where to find it. The uber block "points to" child blocks (stores the address of the location on disk of the child blocks). These ***indirect blocks*** contain, among other things, pointers to two other blocks under them, either more indirect blocks, or ***leaf blocks*** which store actual user data.

iXsystems

Copy on Write

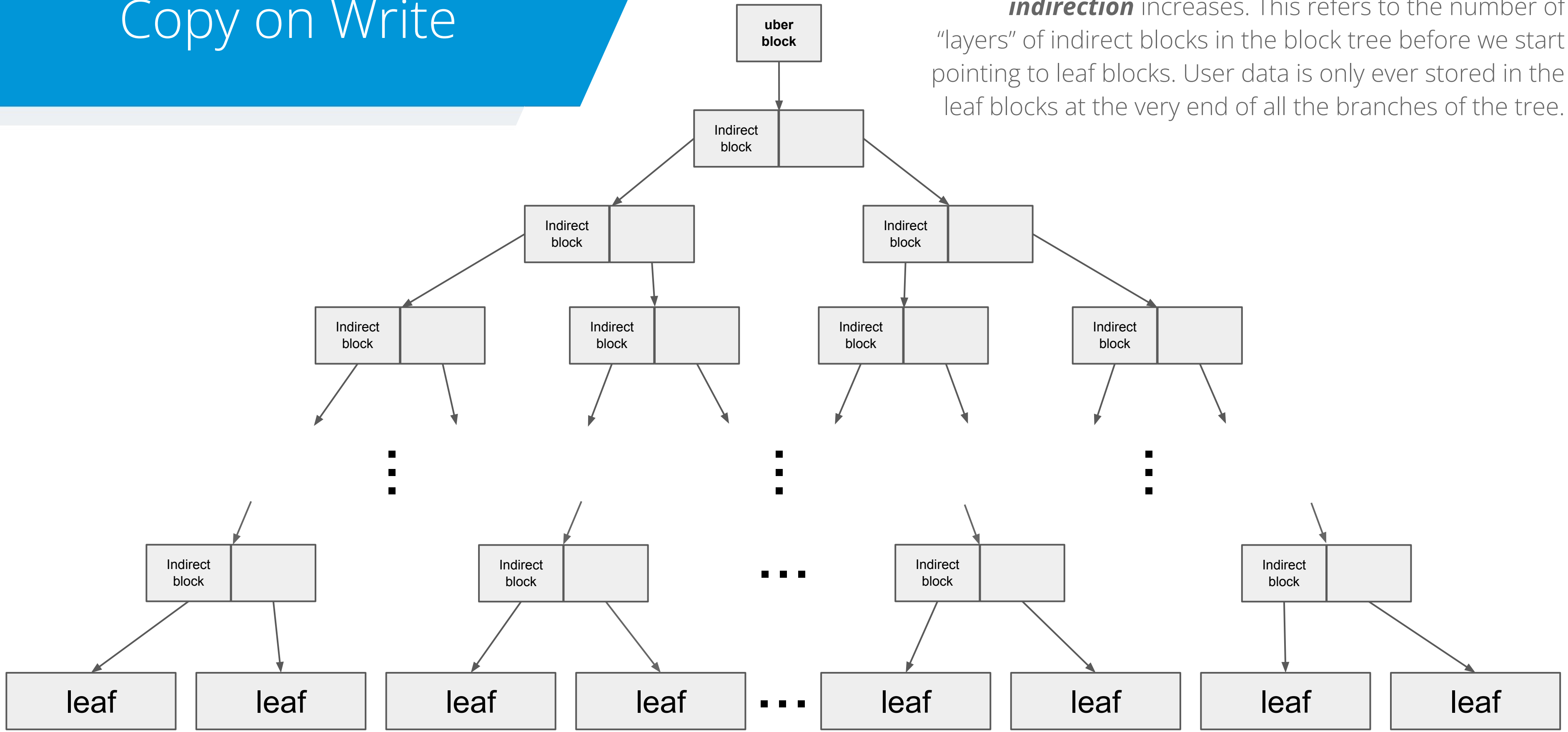


Every block that contains a pointer (the uber block and all indirect blocks) also keeps a ***checksum*** of the block(s) to which it points. The checksum lets ZFS verify the integrity of the data in those child blocks. The checksum is an example of "metadata"; data that describes some aspect of other data on the disk.

The indirect blocks contain lots of other metadata, too, including a "birthdate" tag so ZFS knows when each block was originally created.

Copy on Write

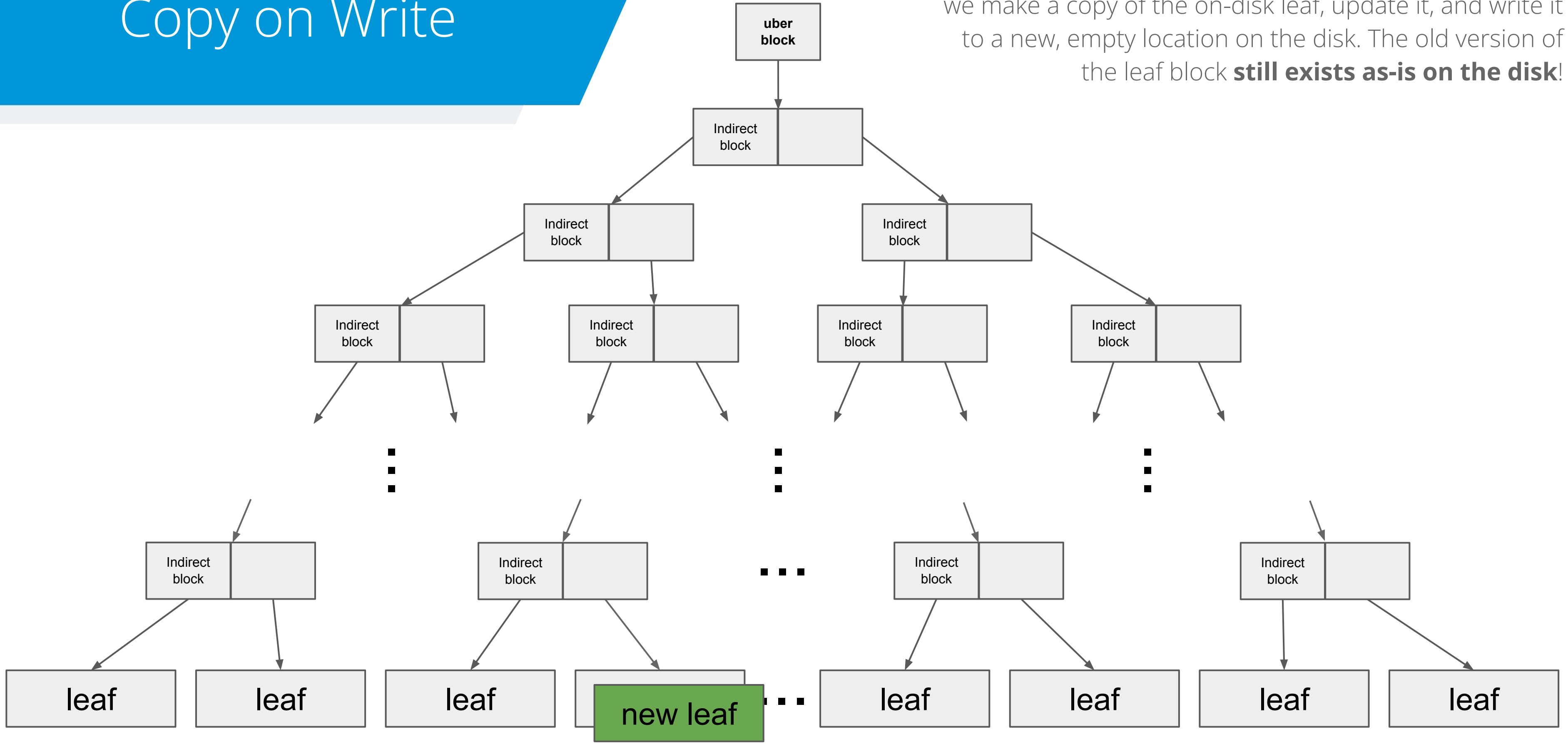
As the amount of user data grows, the ***level of indirection*** increases. This refers to the number of "layers" of indirect blocks in the block tree before we start pointing to leaf blocks. User data is only ever stored in the leaf blocks at the very end of all the branches of the tree.



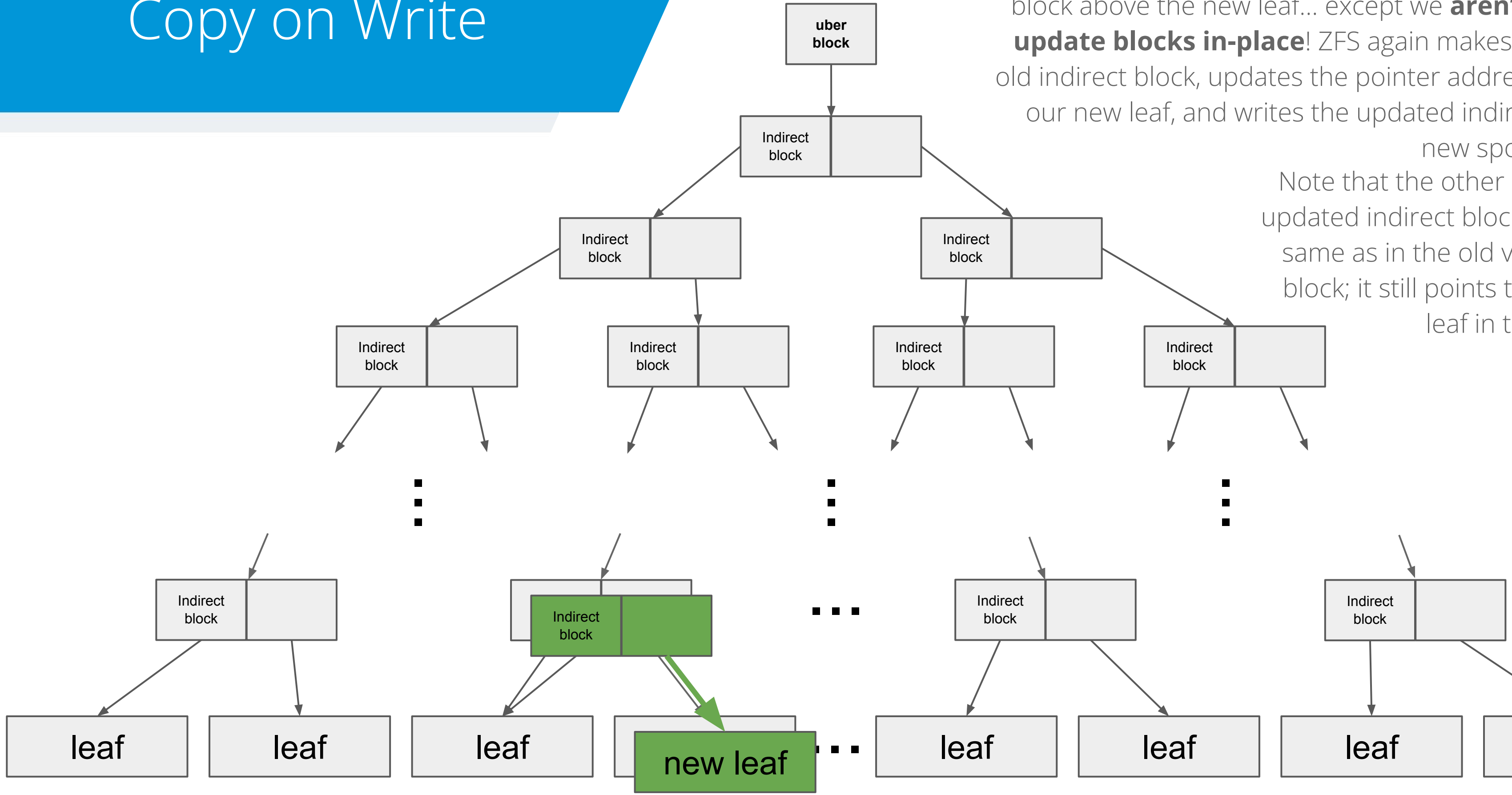
iXsystems

Copy on Write

When we want to "update" an existing block in the tree, we make a copy of the on-disk leaf, update it, and write it to a new, empty location on the disk. The old version of the leaf block **still exists as-is on the disk**!



Copy on Write



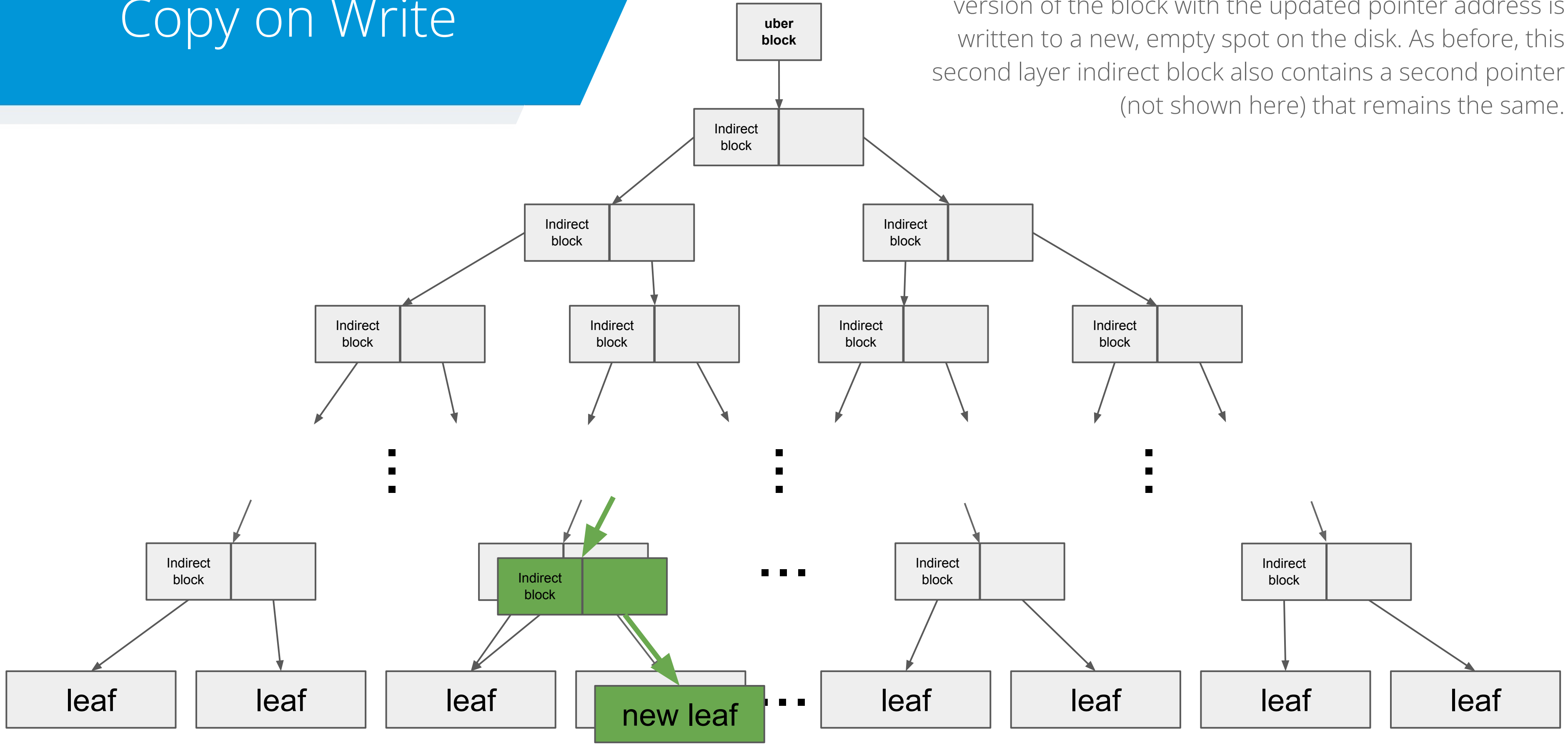
ZFS now needs to update the pointer in the first indirect block above the new leaf… except we **aren't allowed to update blocks in-place**! ZFS again makes a copy of the old indirect block, updates the pointer address to point to our new leaf, and writes the updated indirect block to a new spot on the disk.

Note that the other pointer in the updated indirect block remains the same as in the old version of that block; it still points to some other leaf in the block tree.

iXsystems

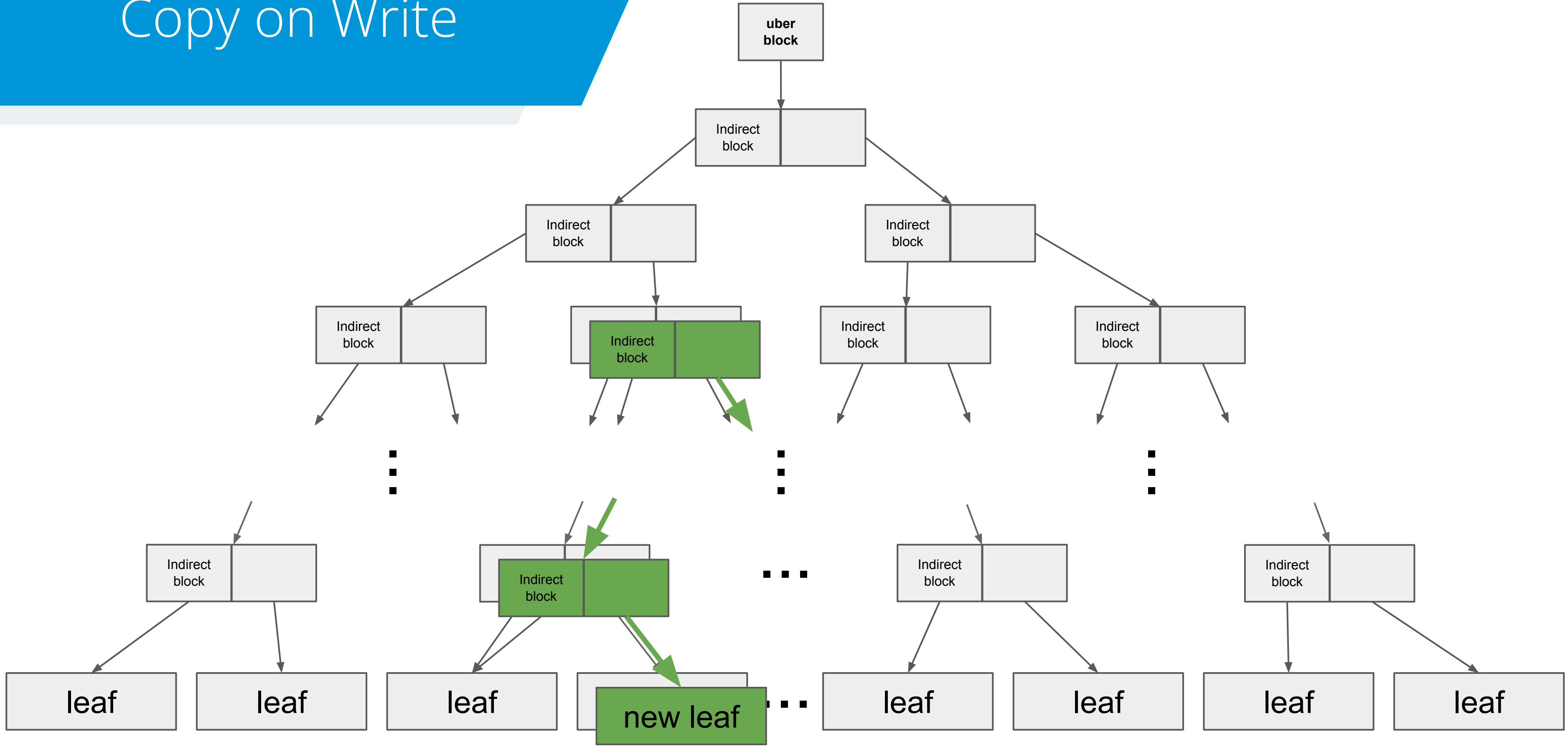
Copy on Write

This repeats for the second layer indirect block. The new version of the block with the updated pointer address is written to a new, empty spot on the disk. As before, this second layer indirect block also contains a second pointer (not shown here) that remains the same.



Copy on Write

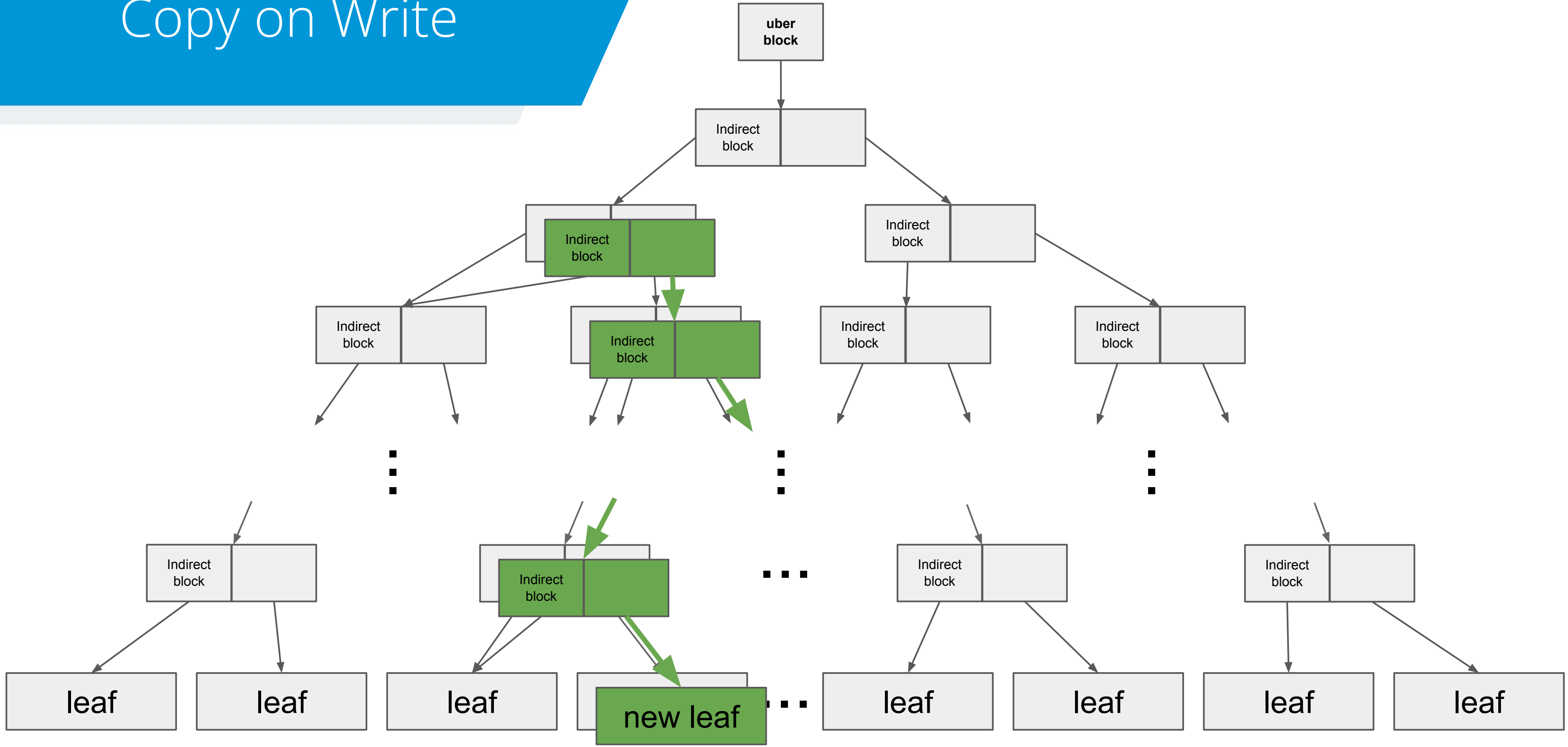
And up the tree we go…



iXsystems

Copy on Write

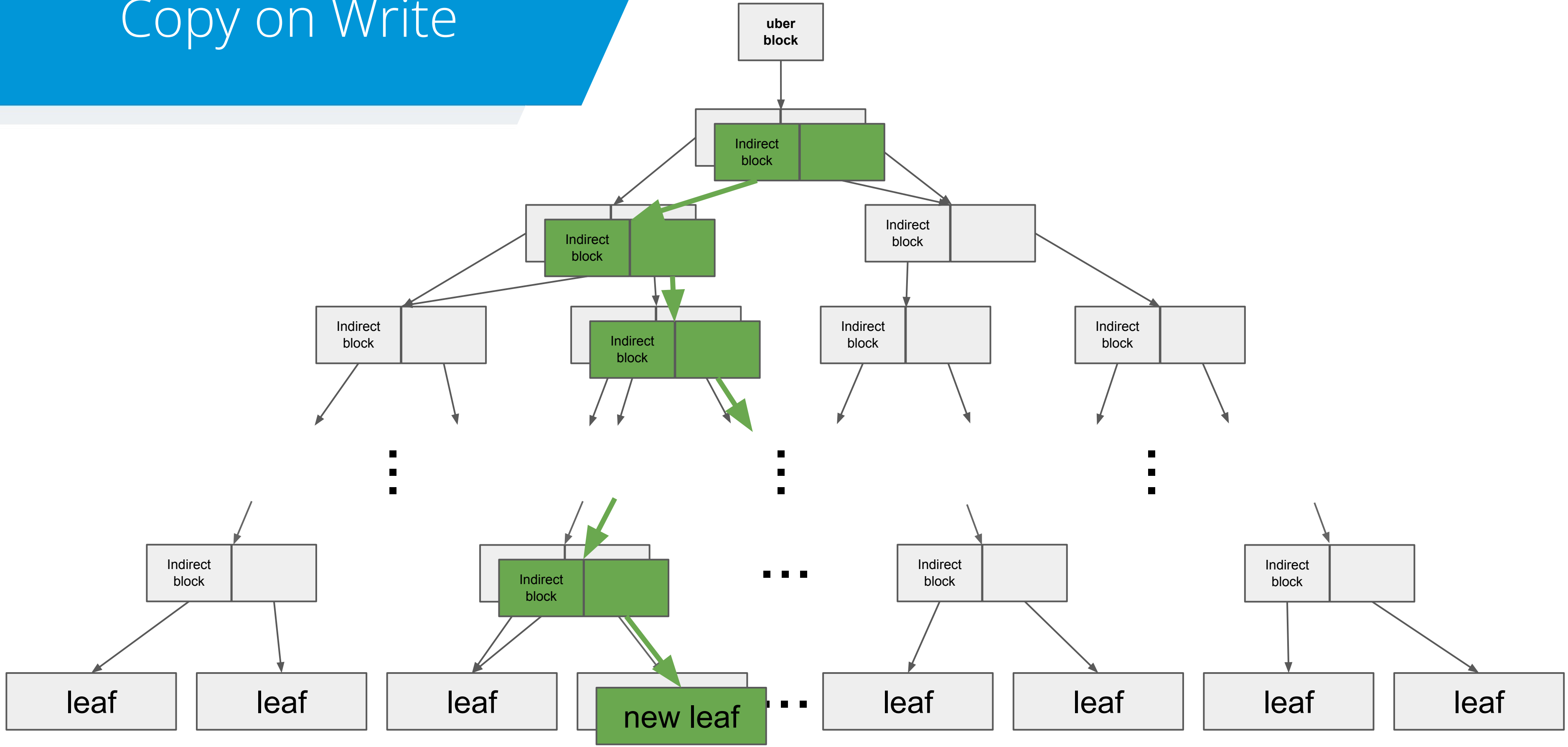
And up the tree we go...



iXsystems

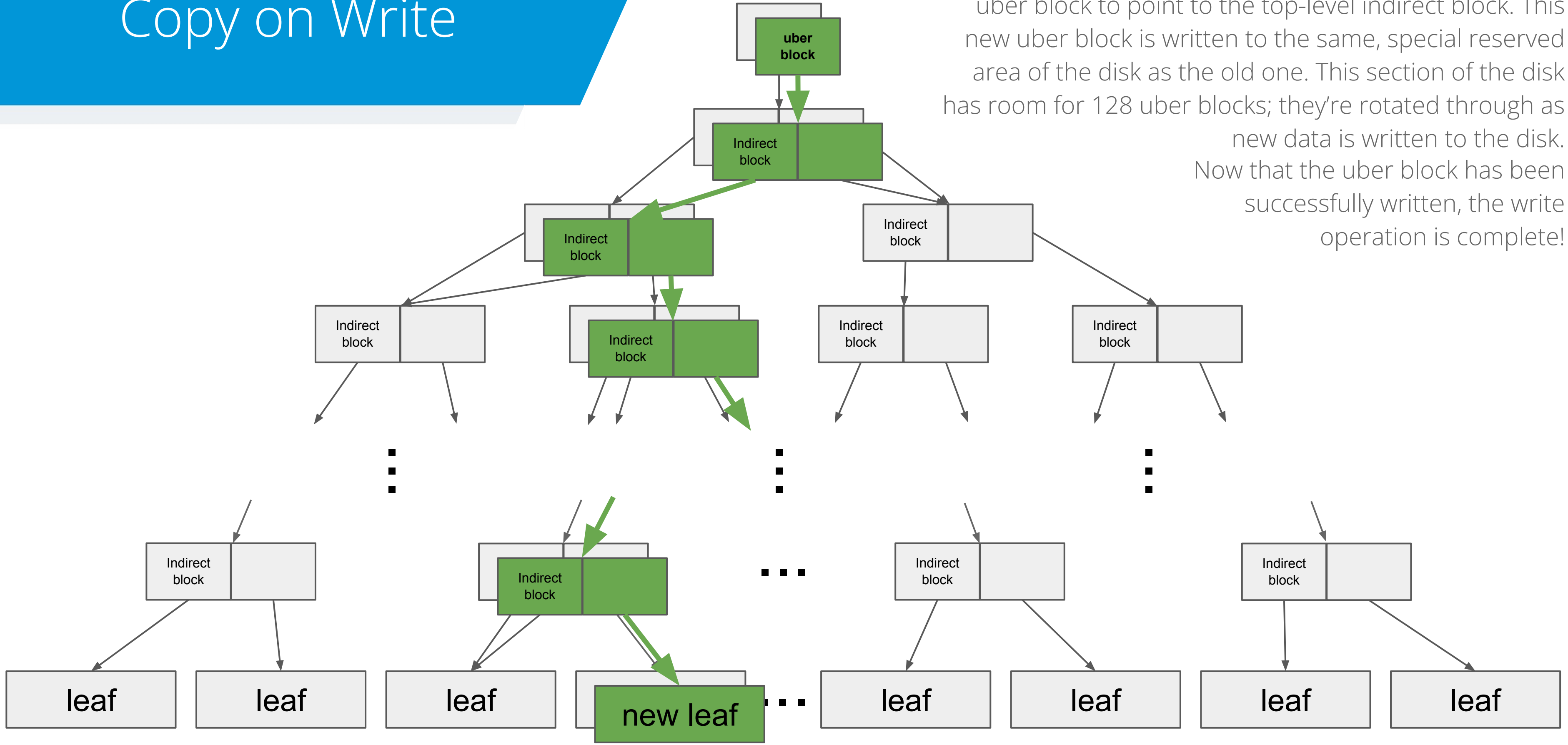
Copy on Write

And up the tree we go…



iXsystems

Copy on Write

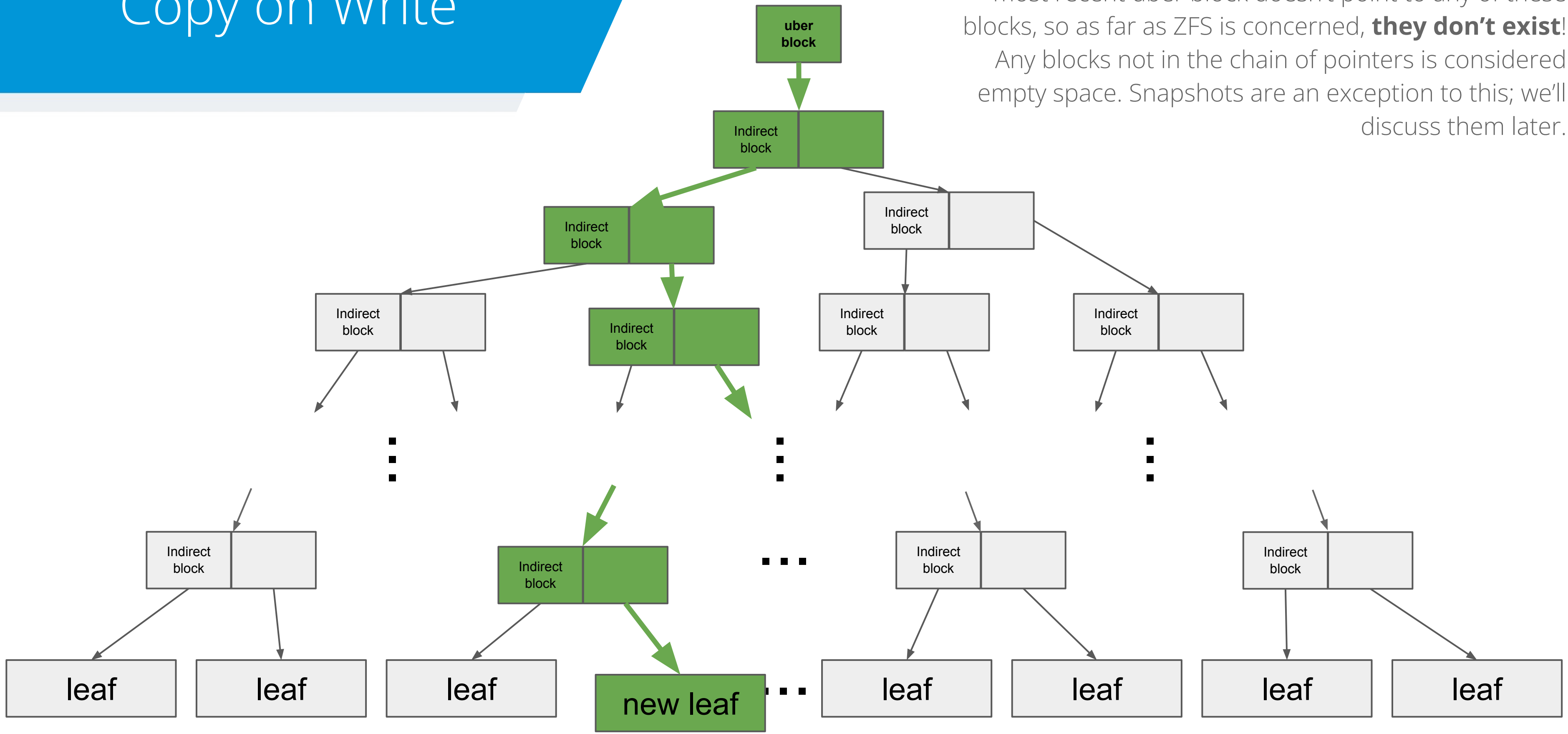


At the very end of the write operation, we write a new uber block to point to the top-level indirect block. This new uber block is written to the same, special reserved area of the disk as the old one. This section of the disk has room for 128 uber blocks; they're rotated through as new data is written to the disk.
Now that the uber block has been successfully written, the write operation is complete!

iXsystems

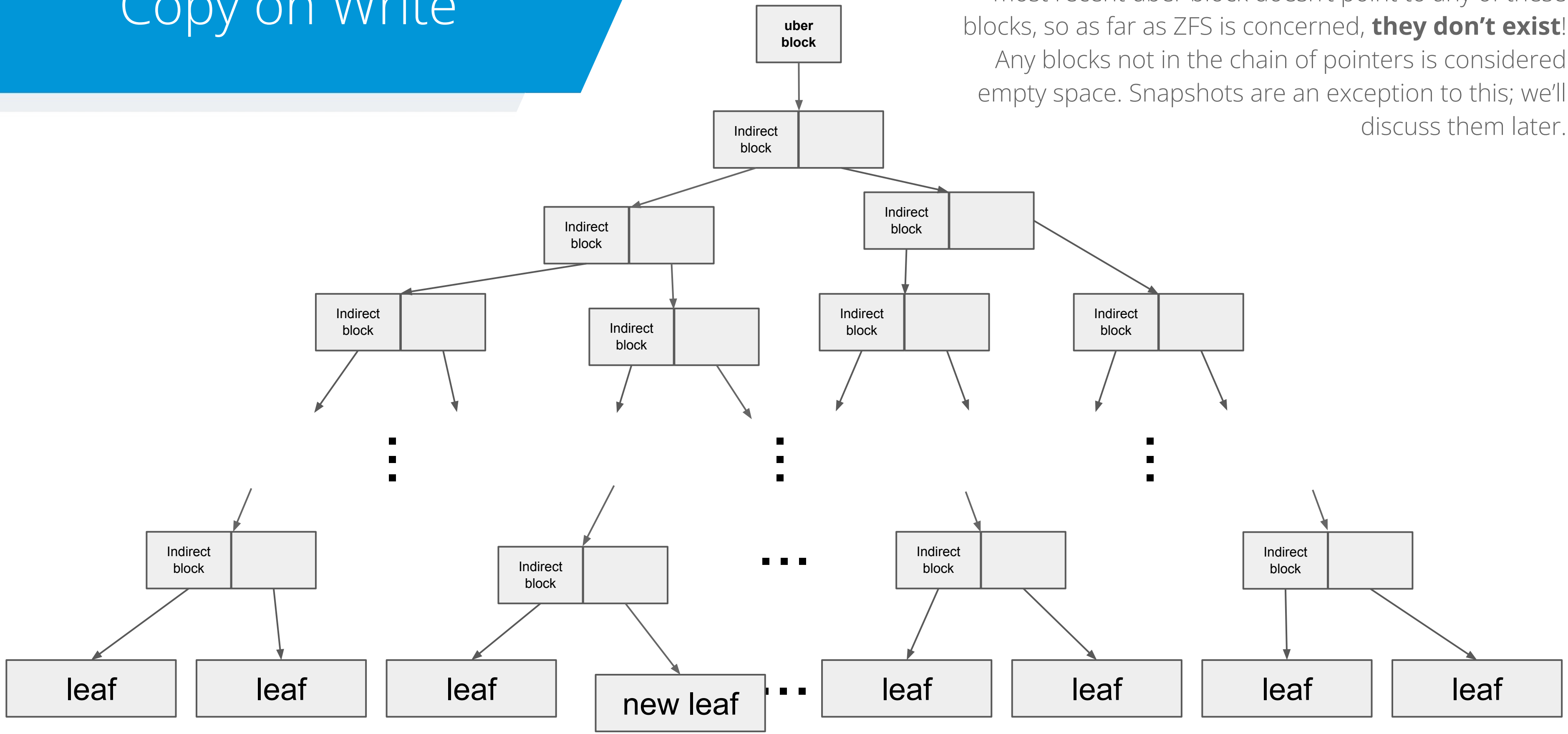
Copy on Write

We can free up the old version of all the blocks now. The most recent uber block doesn't point to any of these blocks, so as far as ZFS is concerned, **they don't exist**! Any blocks not in the chain of pointers is considered empty space. Snapshots are an exception to this; we'll discuss them later.



Copy on Write

We can free up the old version of all the blocks now. The most recent uber block doesn't point to any of these blocks, so as far as ZFS is concerned, **they don't exist**! Any blocks not in the chain of pointers is considered empty space. Snapshots are an exception to this; we'll discuss them later.



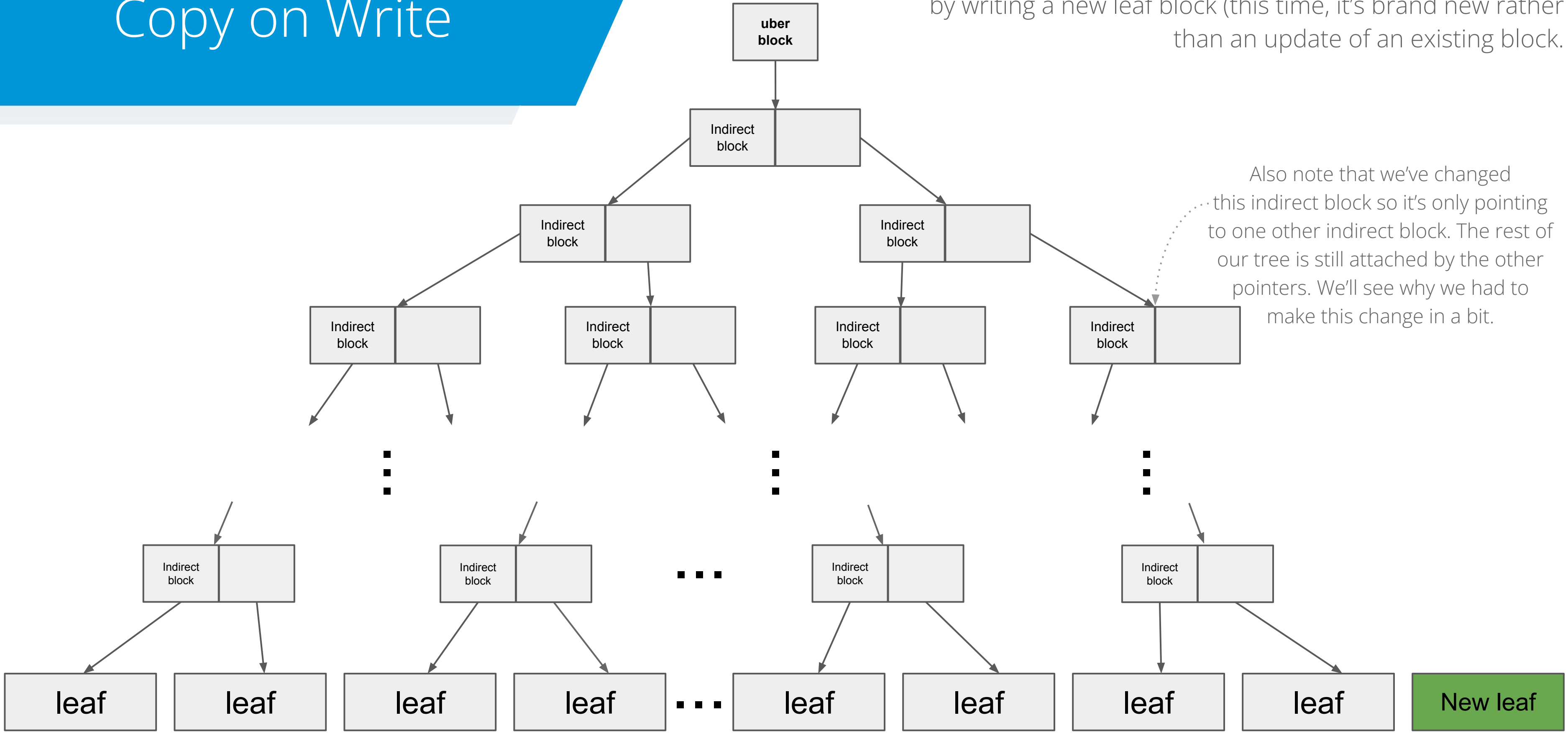
So that's updating an existing block.

*What about **writing a new block**?*

Copy on Write

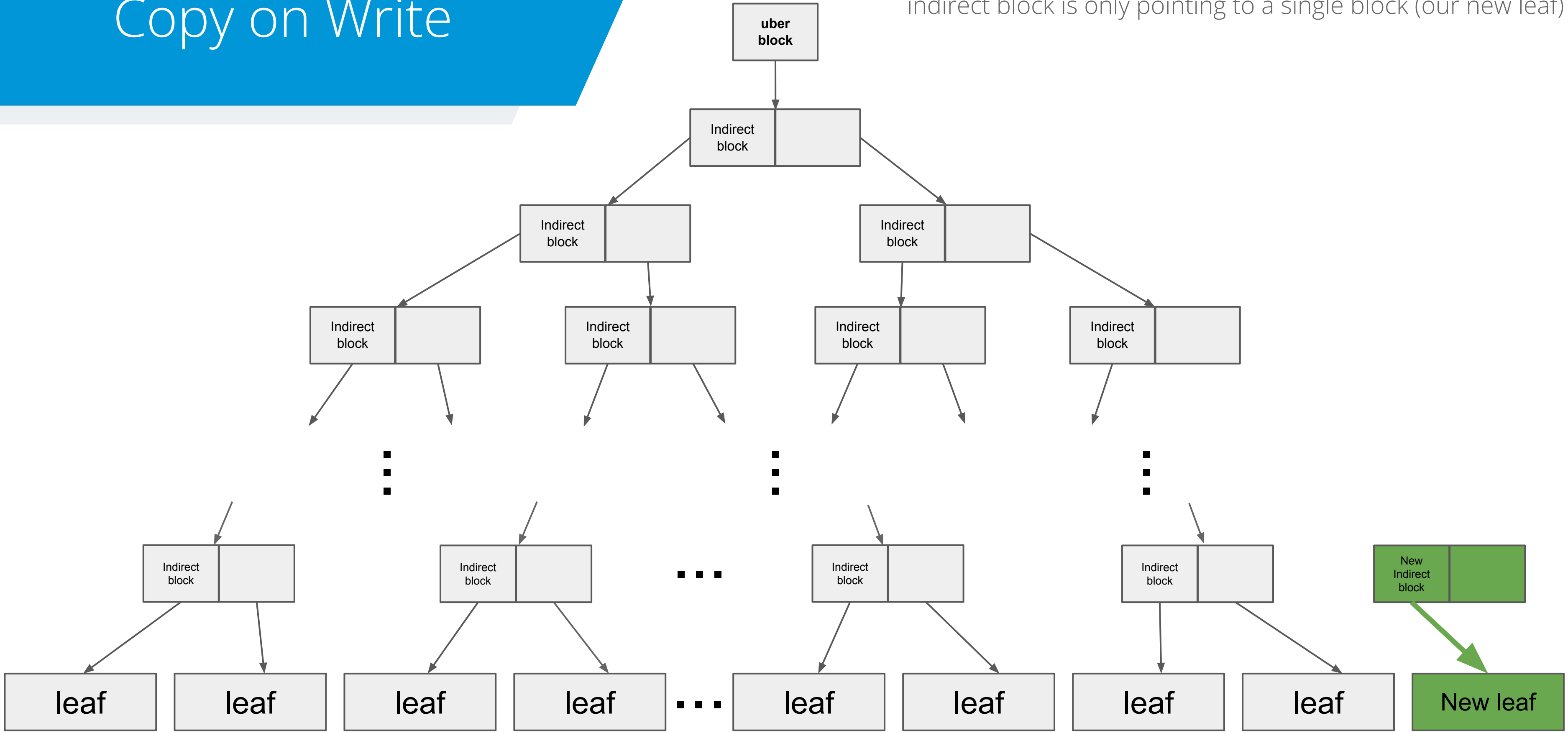
ZFS will add a new branch to the tree! As before, we start by writing a new leaf block (this time, it's brand new rather than an update of an existing block.

Also note that we've changed this indirect block so it's only pointing to one other indirect block. The rest of our tree is still attached by the other pointers. We'll see why we had to make this change in a bit.



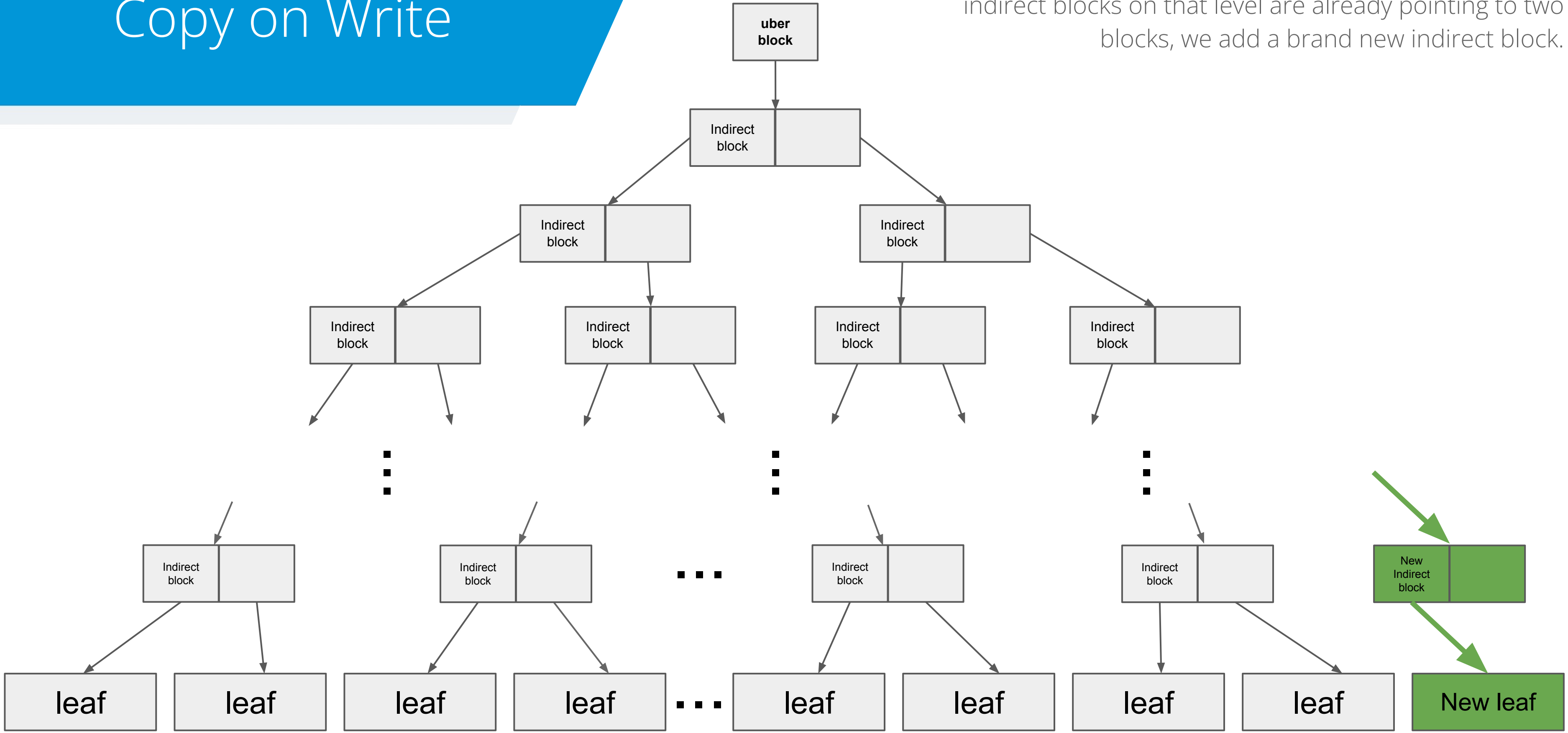
Copy on Write

A new indirect block is added for this leaf. Note that the indirect block is only pointing to a single block (our new leaf)



Copy on Write

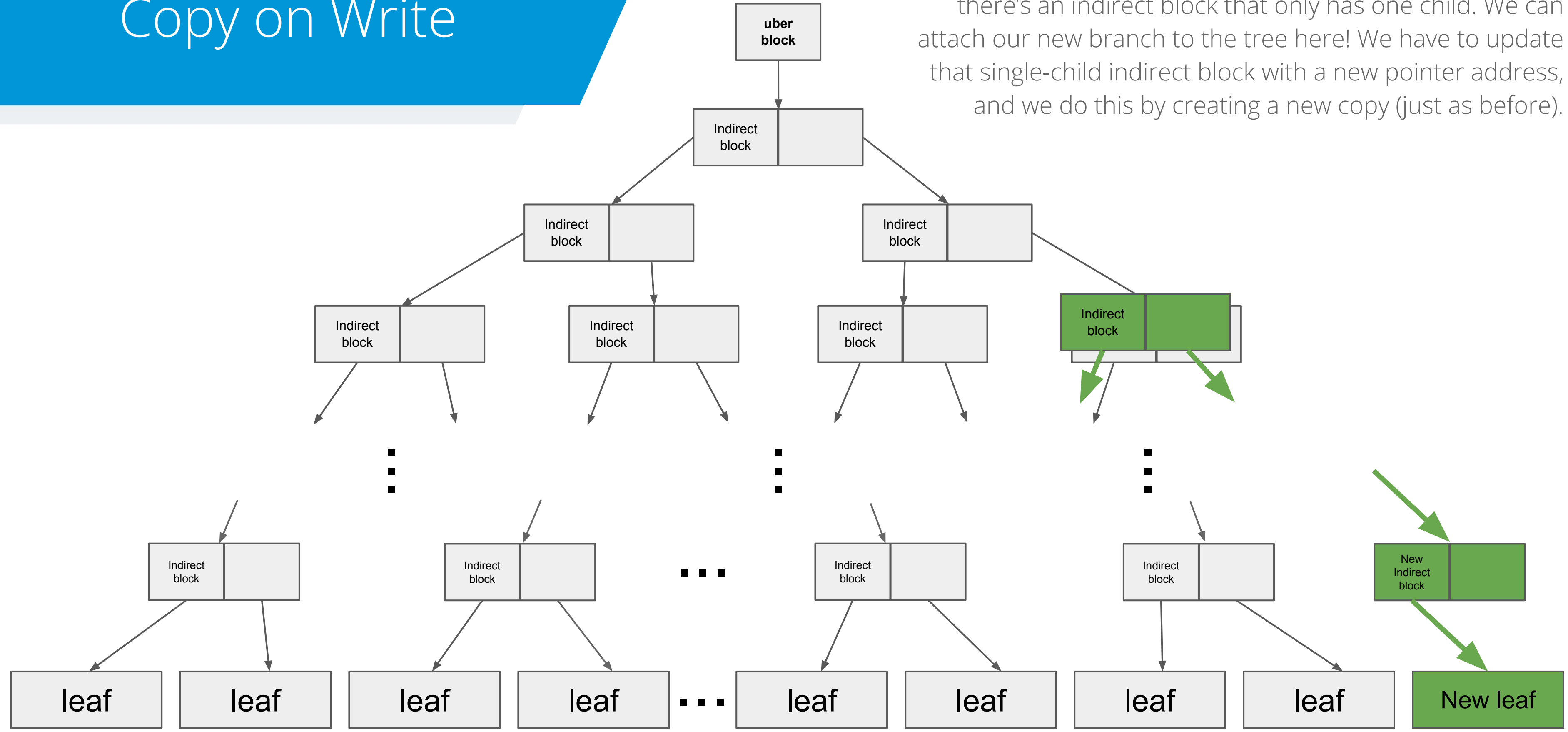
We keep working our way up the block tree. If all the other indirect blocks on that level are already pointing to two blocks, we add a brand new indirect block.



iXsystems

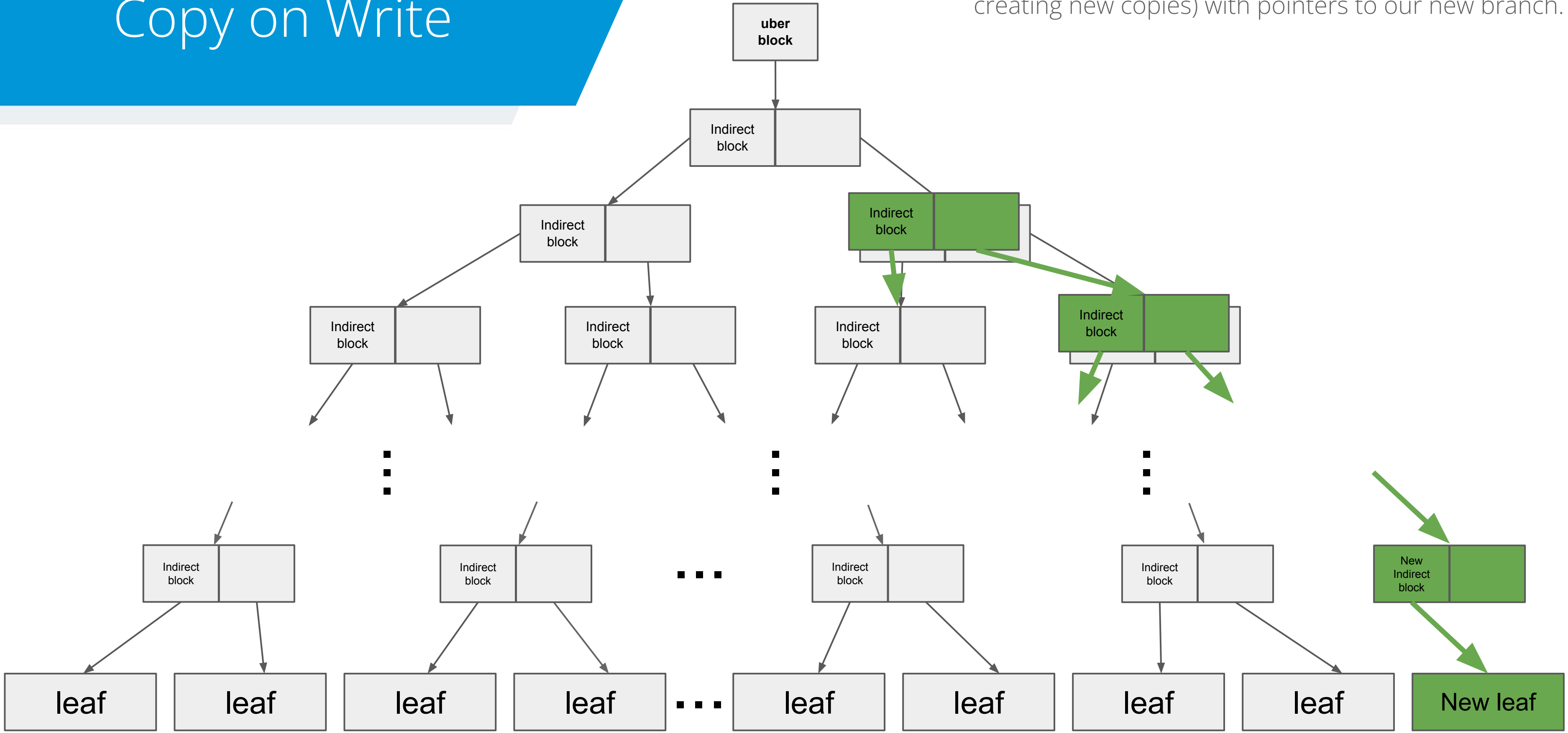
Copy on Write

As we work our way up, we might come to a level where there's an indirect block that only has one child. We can attach our new branch to the tree here! We have to update that single-child indirect block with a new pointer address, and we do this by creating a new copy (just as before).



Copy on Write

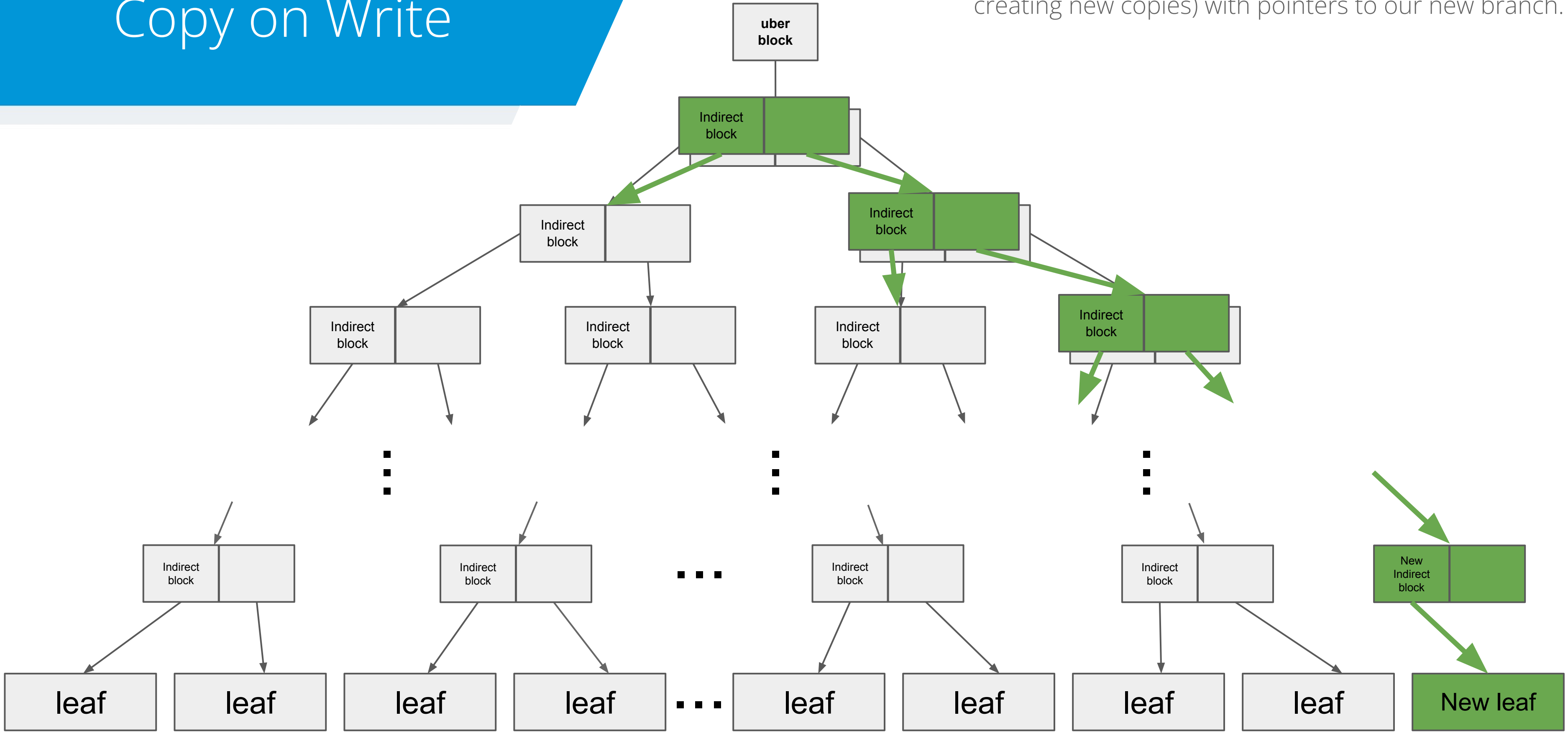
Just as before, we keep updating parent indirect blocks (by creating new copies) with pointers to our new branch.



iXsystems

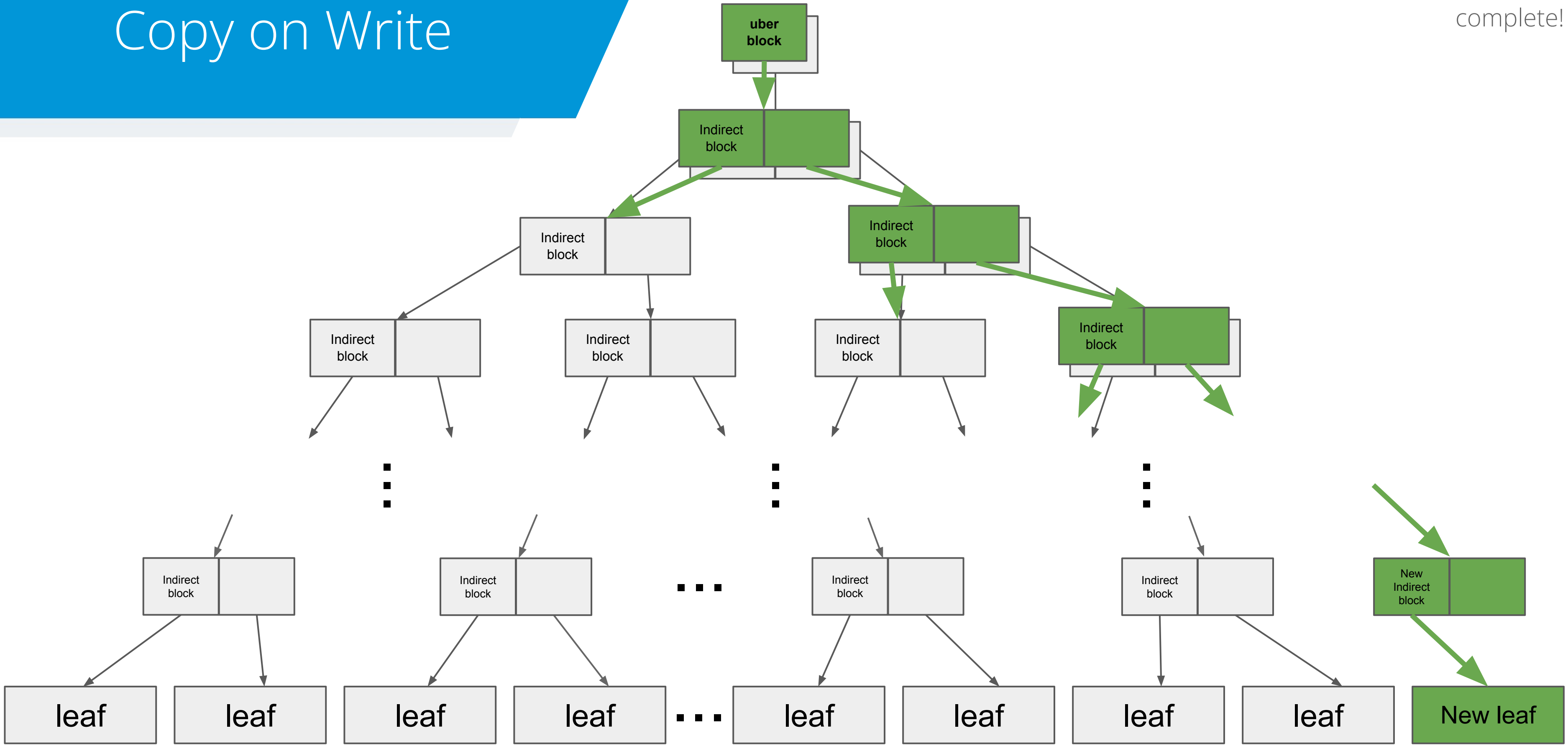
Copy on Write

Just as before, we keep updating parent indirect blocks (by creating new copies) with pointers to our new branch.



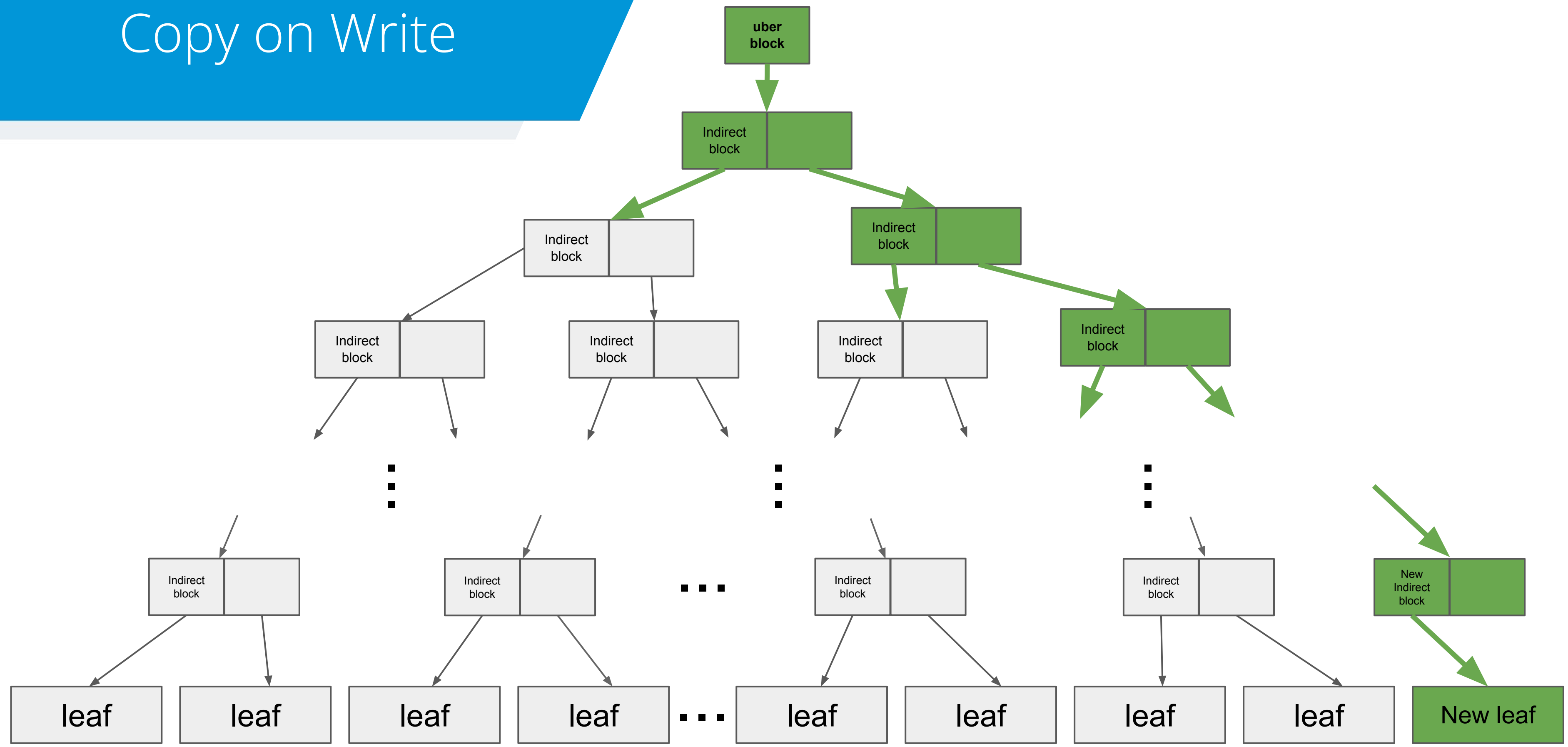
Copy on Write

Once the new uber block has been written, the write is complete!



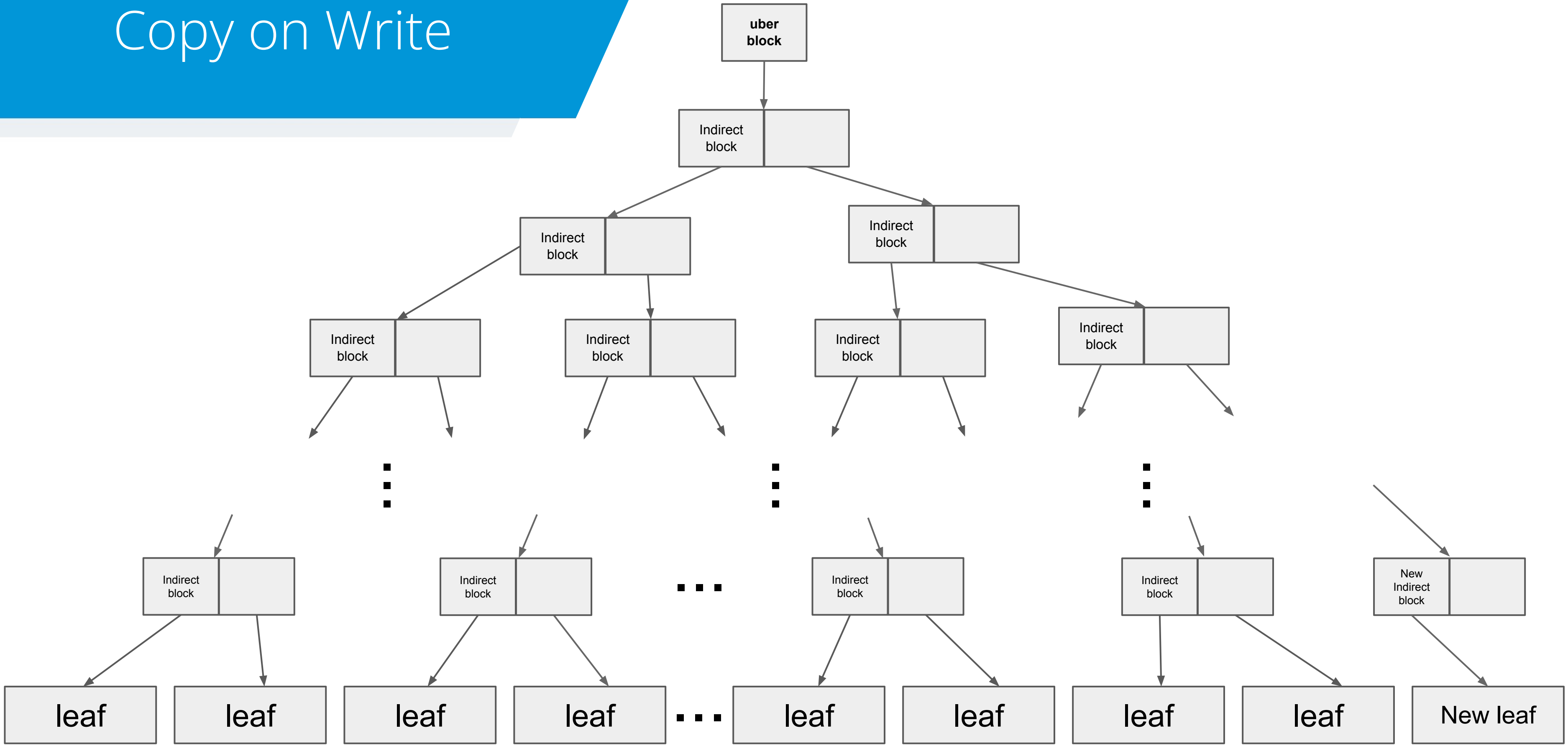
Copy on Write

We can now remove all the unreferenced indirect blocks.

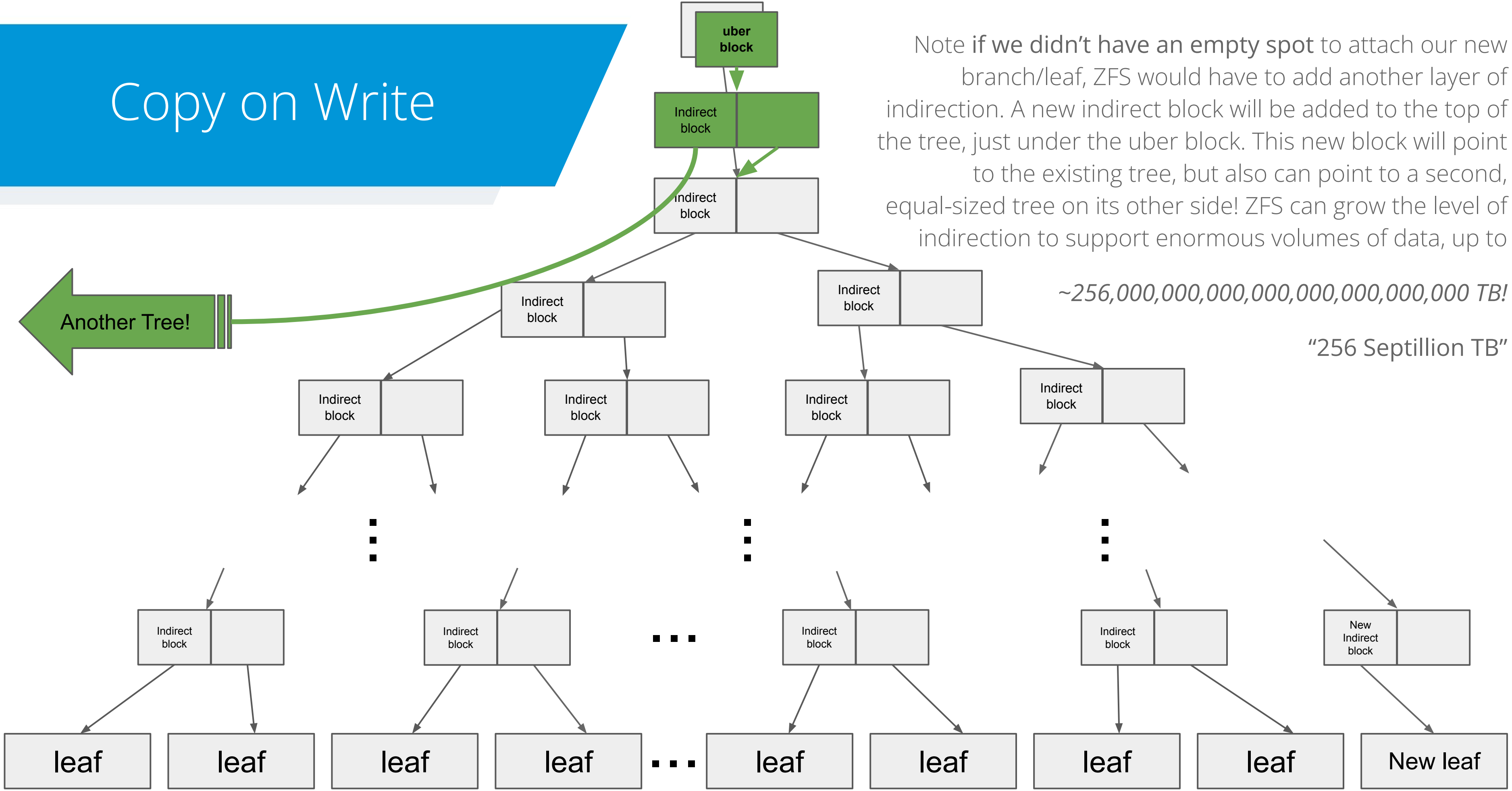


Copy on Write

New block added!



Copy on Write



Note **if we didn't have an empty spot** to attach our new branch/leaf, ZFS would have to add another layer of indirection. A new indirect block will be added to the top of the tree, just under the uber block. This new block will point to the existing tree, but also can point to a second, equal-sized tree on its other side! ZFS can grow the level of indirection to support enormous volumes of data, up to

~256,000,000,000,000,000,000,000,000 TB!

"256 Septillion TB"

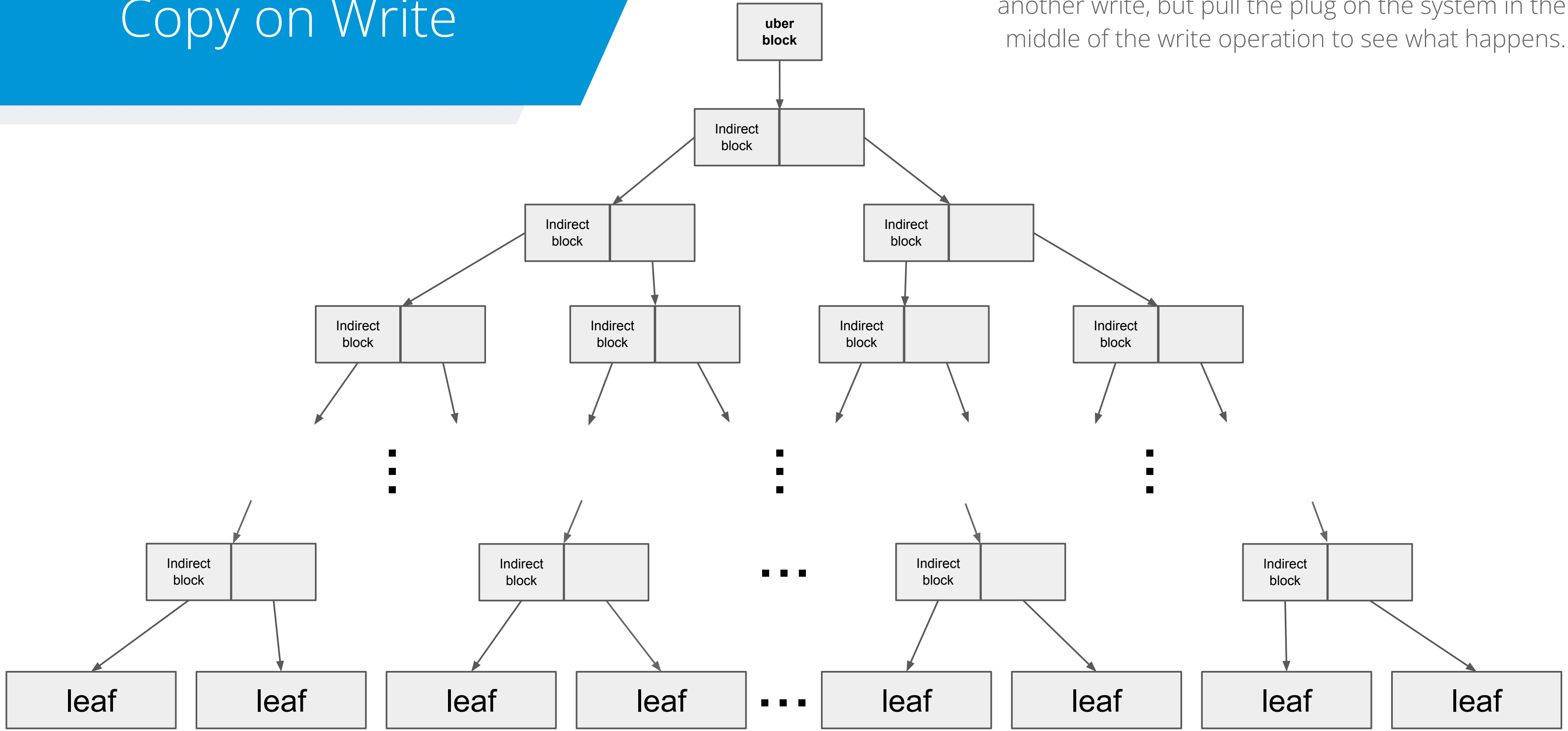
iXsystems

Okay, neat. But why bother with all this?

How does this protect against ***data corruption****?*

Copy on Write

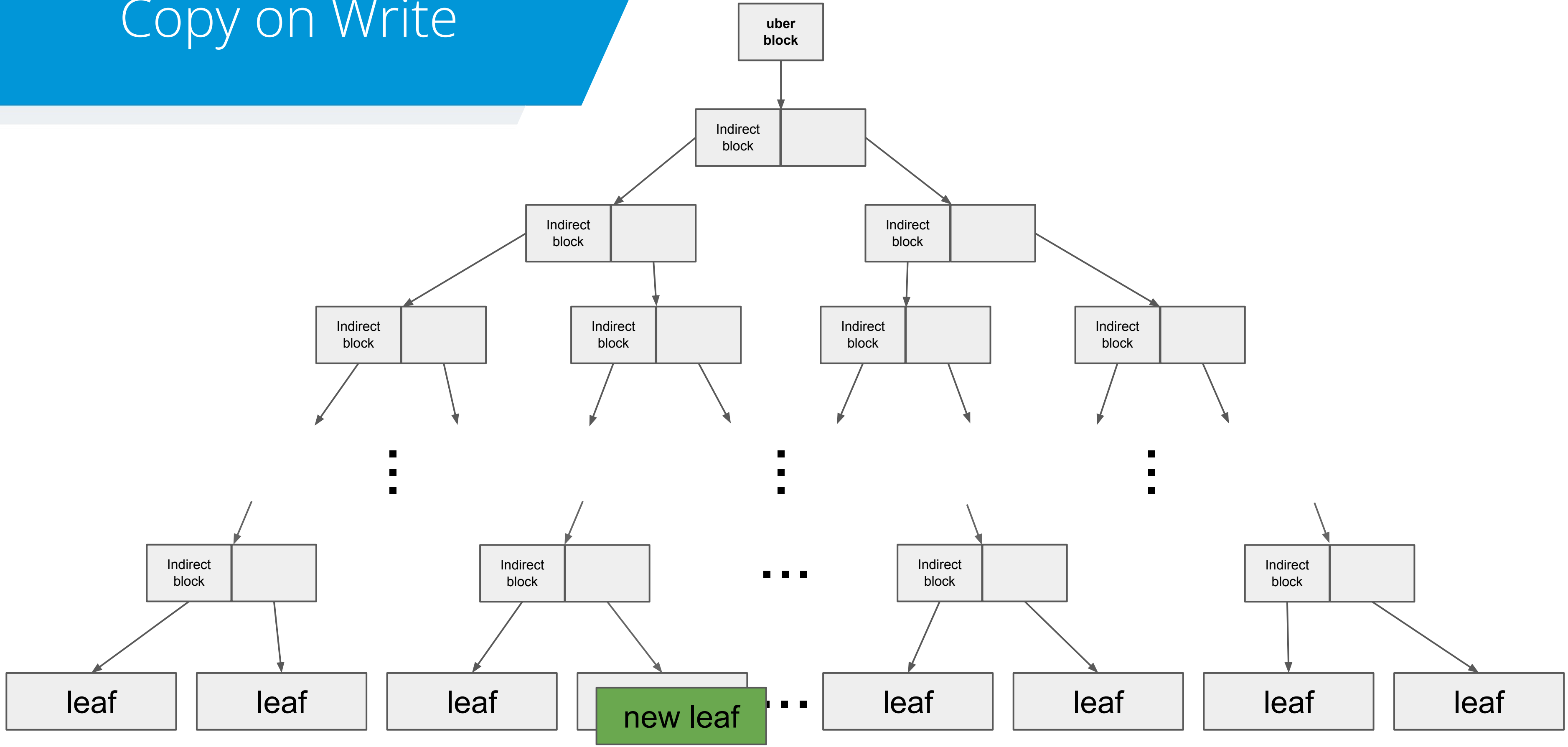
Let's go back to our original block tree. We'll go through another write, but pull the plug on the system in the middle of the write operation to see what happens.



iXsystems

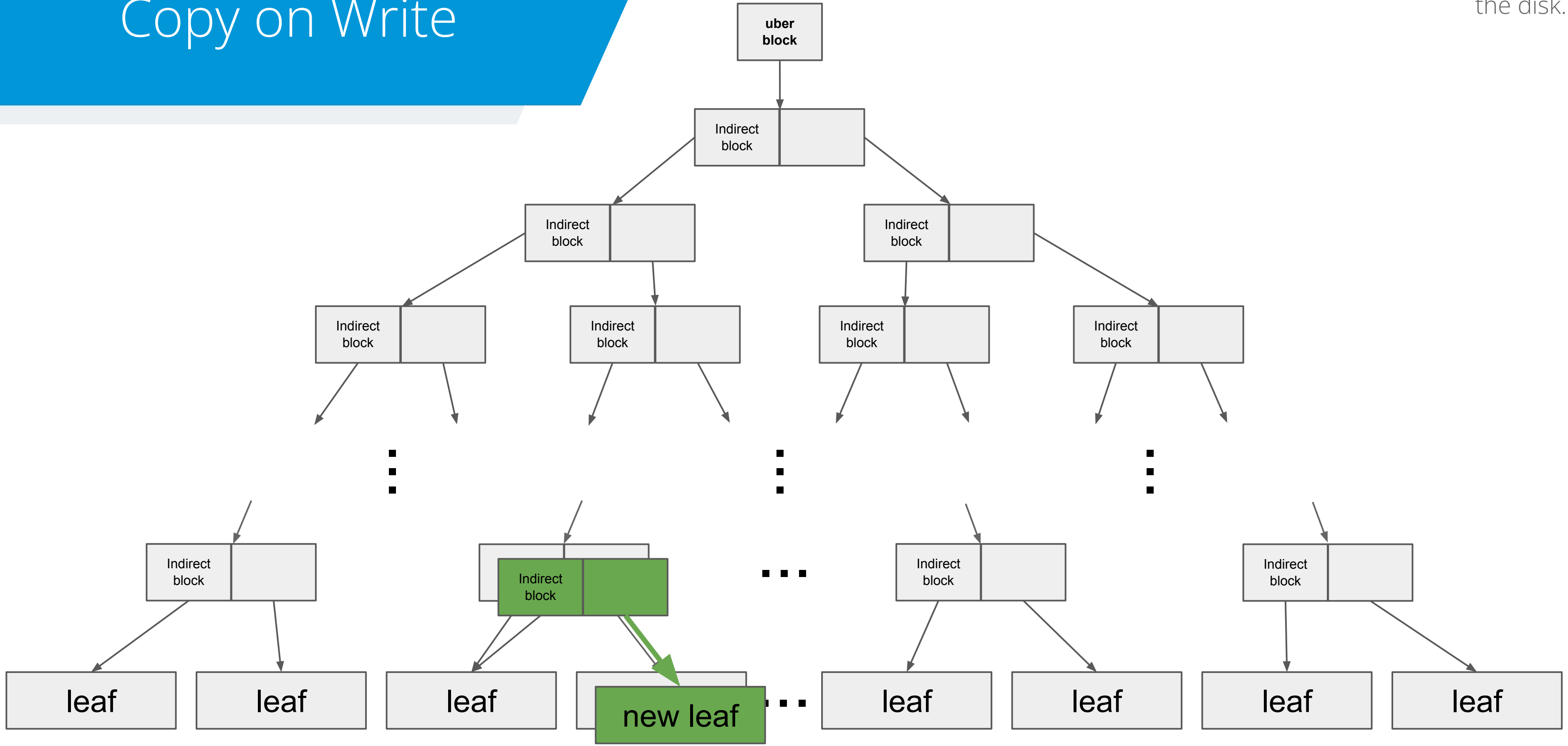
Copy on Write

We start by writing our update to a new leaf block.



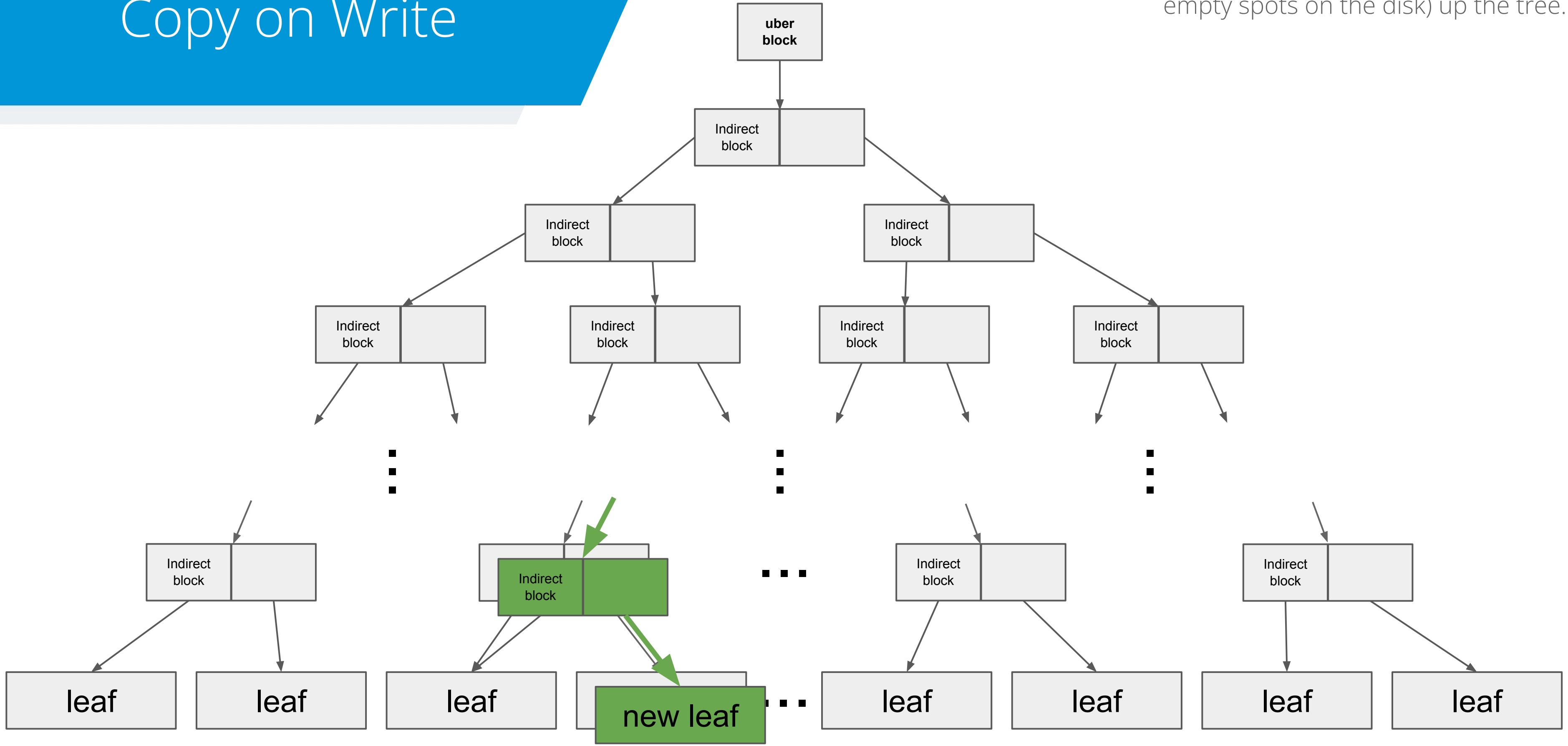
Copy on Write

We write our updated indirect block to a free space on the disk.



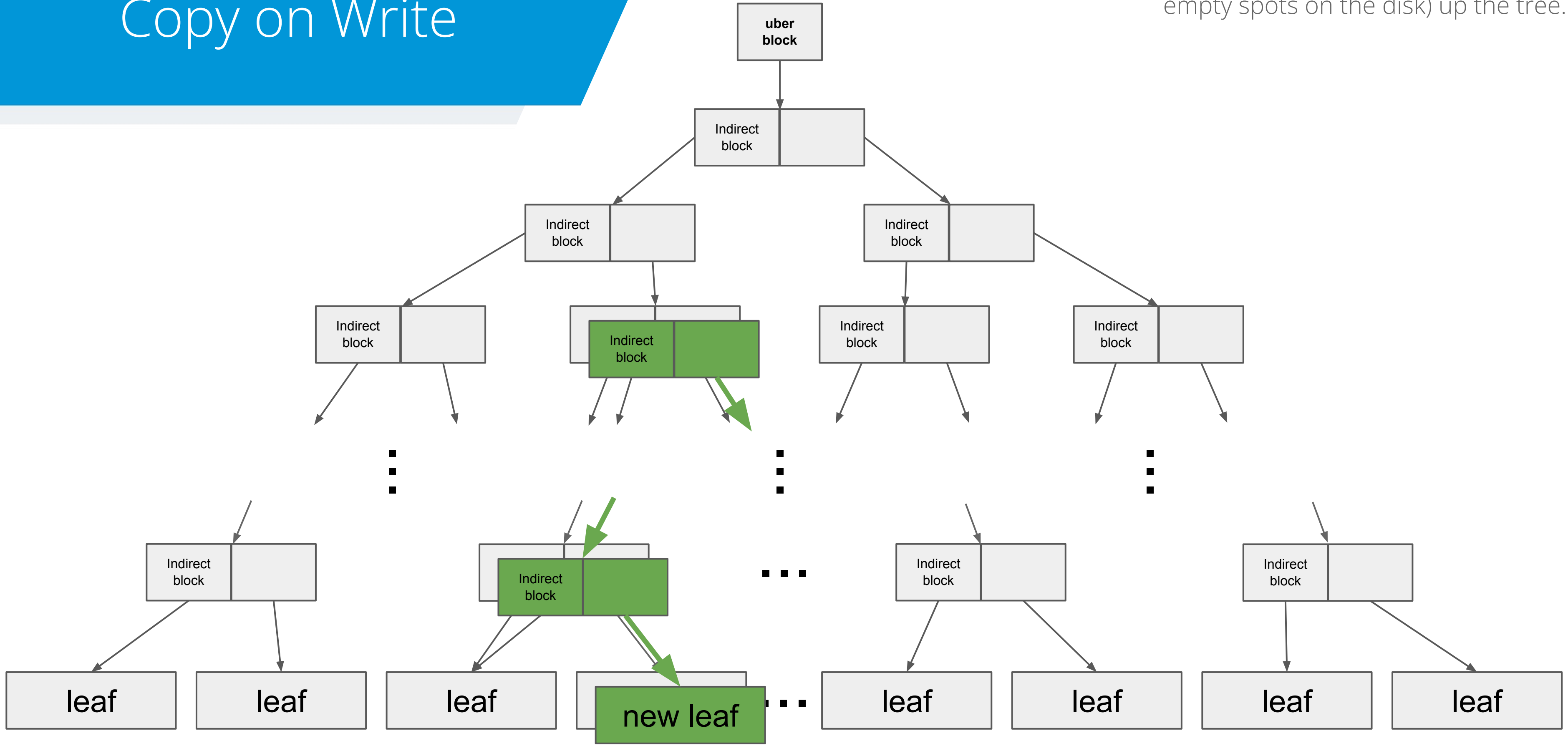
Copy on Write

And we continue writing our updated blocks (each to new, empty spots on the disk) up the tree.



Copy on Write

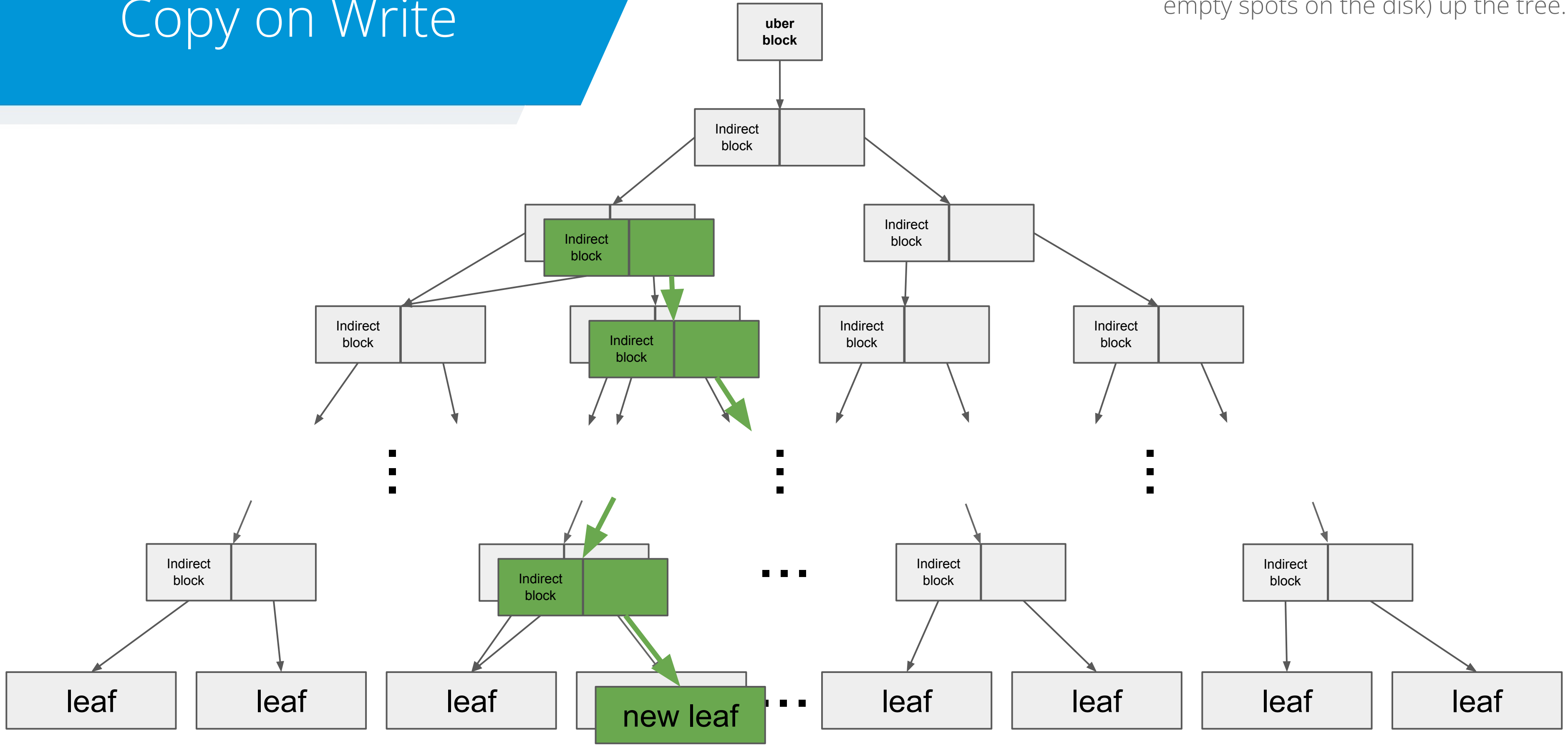
And we continue writing our updated blocks (each to new, empty spots on the disk) up the tree.



iXsystems

Copy on Write

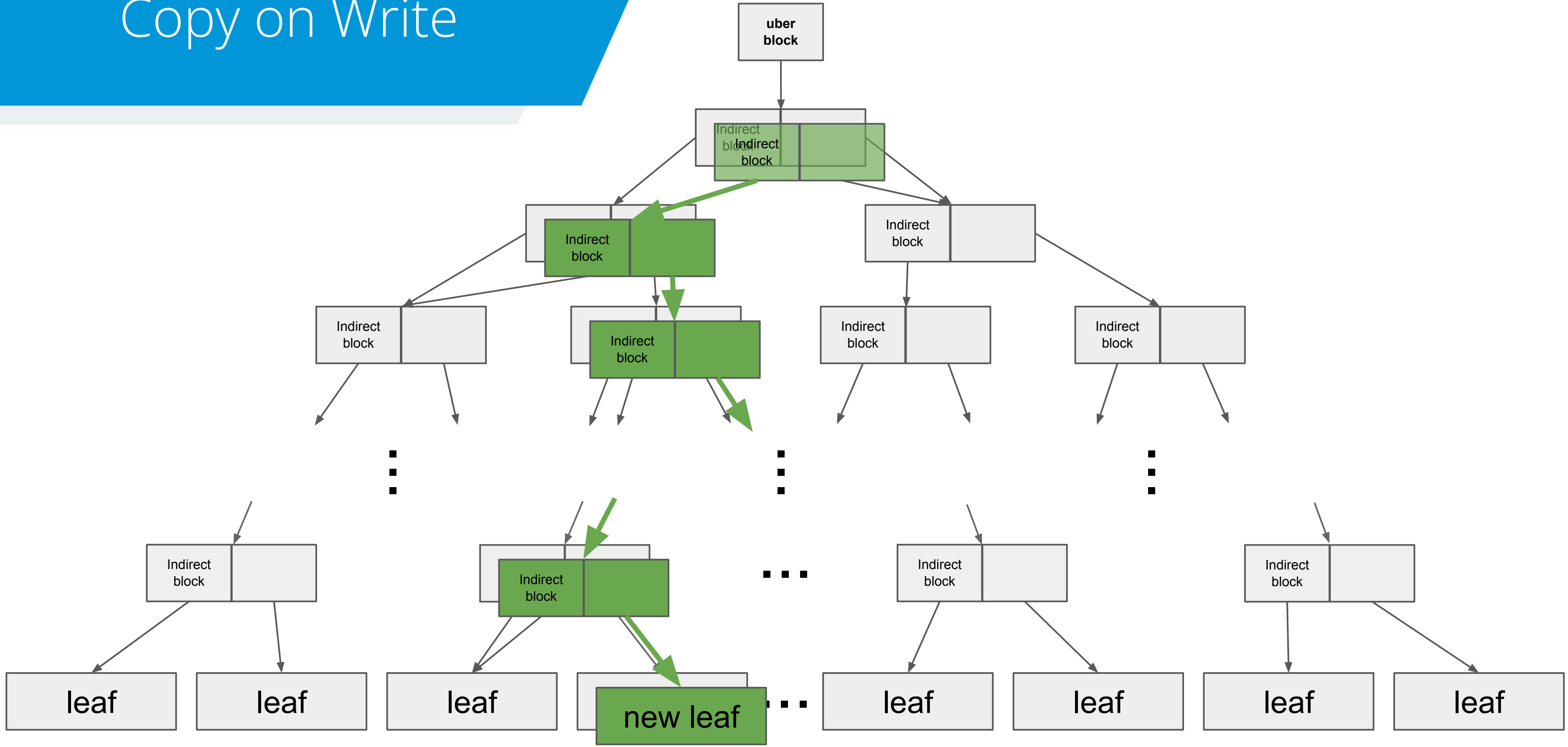
And we continue writing our updated blocks (each to new, empty spots on the disk) up the tree.



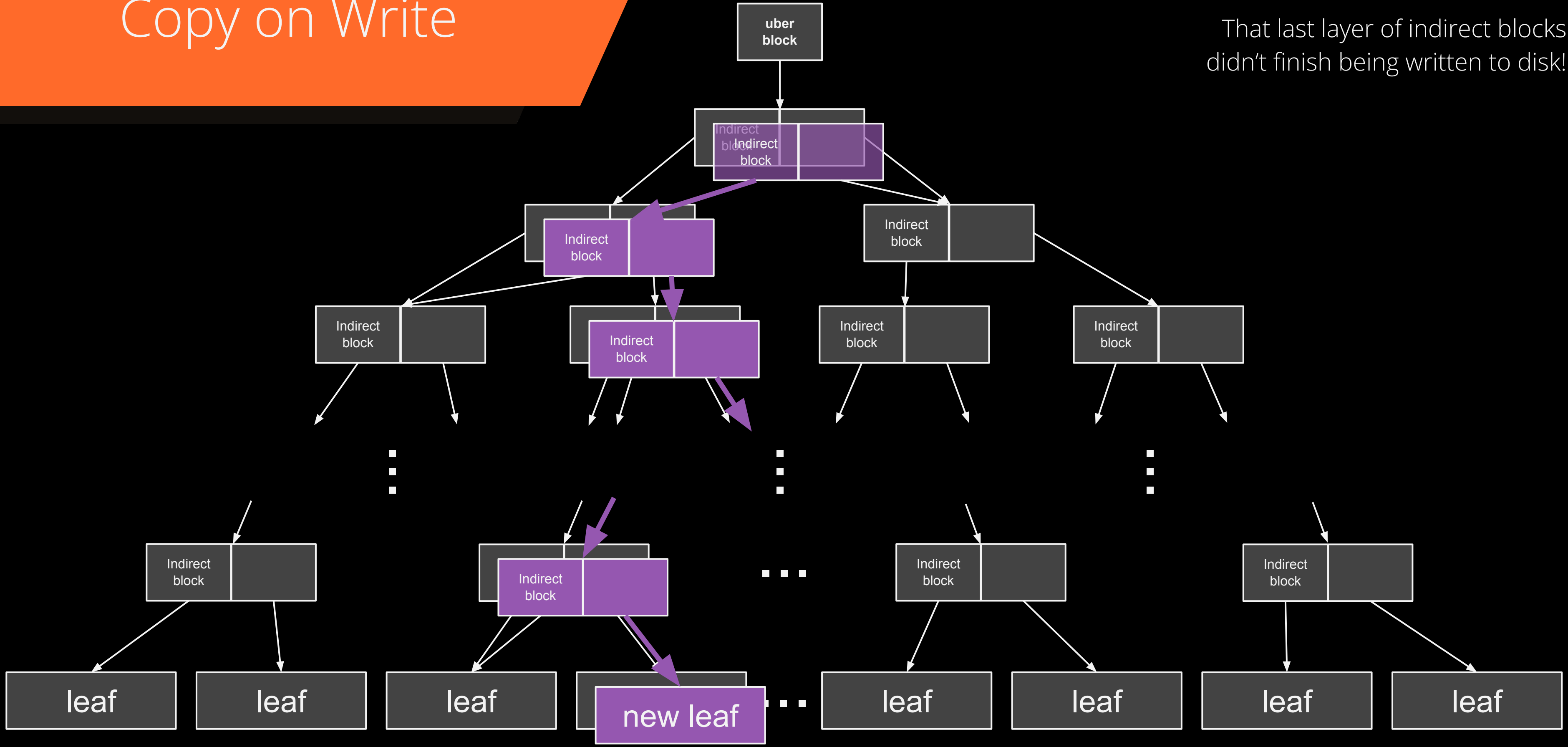
iXsystems

Copy on Write

We start writing the last level of indirect blocks…



Copy on Write



We pull the plug here!

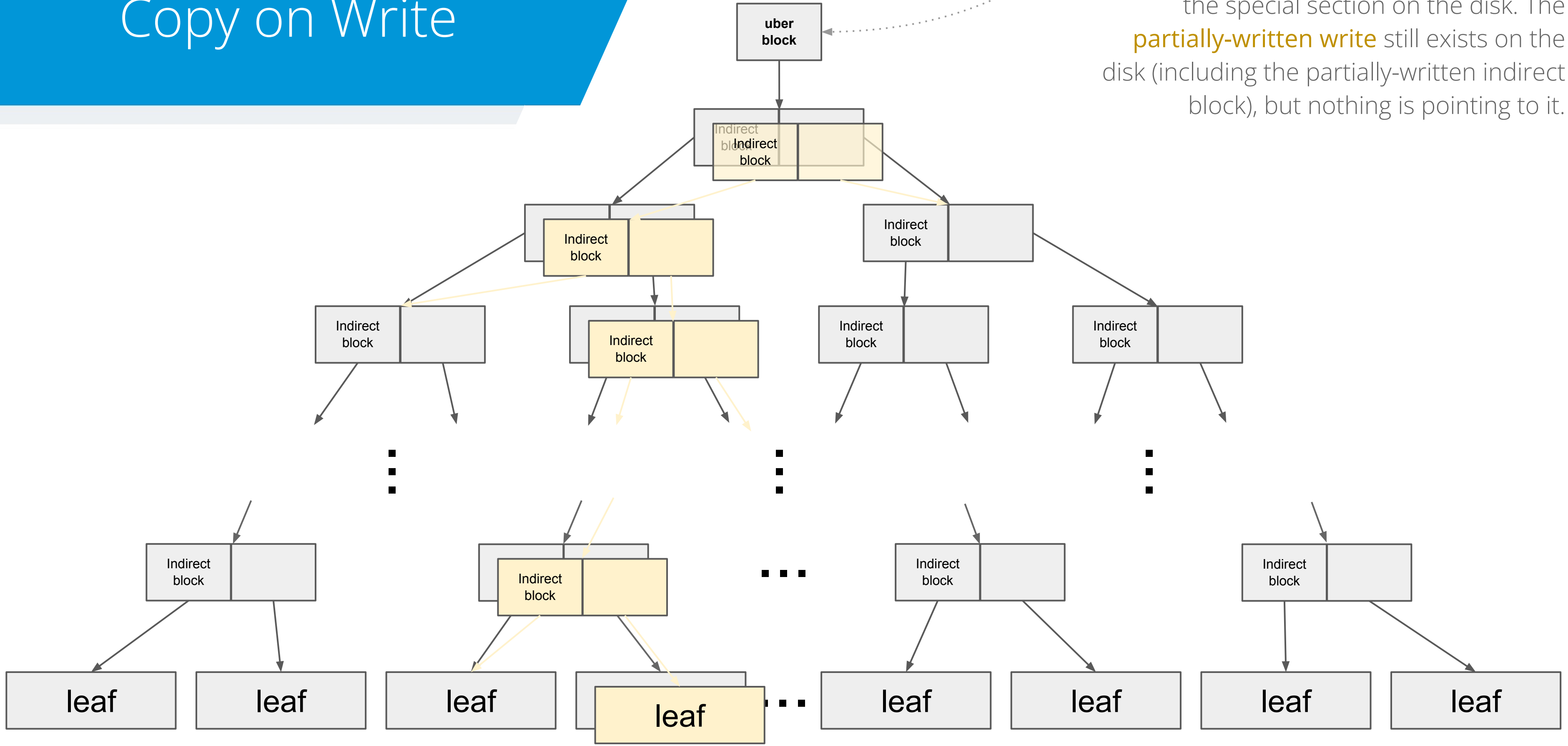
That last layer of indirect blocks didn't finish being written to disk!

Our System Reboots…

When ZFS loads, it looks in the special uber block area on disk. ZFS looks at the birthdate tag on every valid uber block it finds to locate the most recently-created one. This uber block is the head of our last valid block tree.

Copy on Write

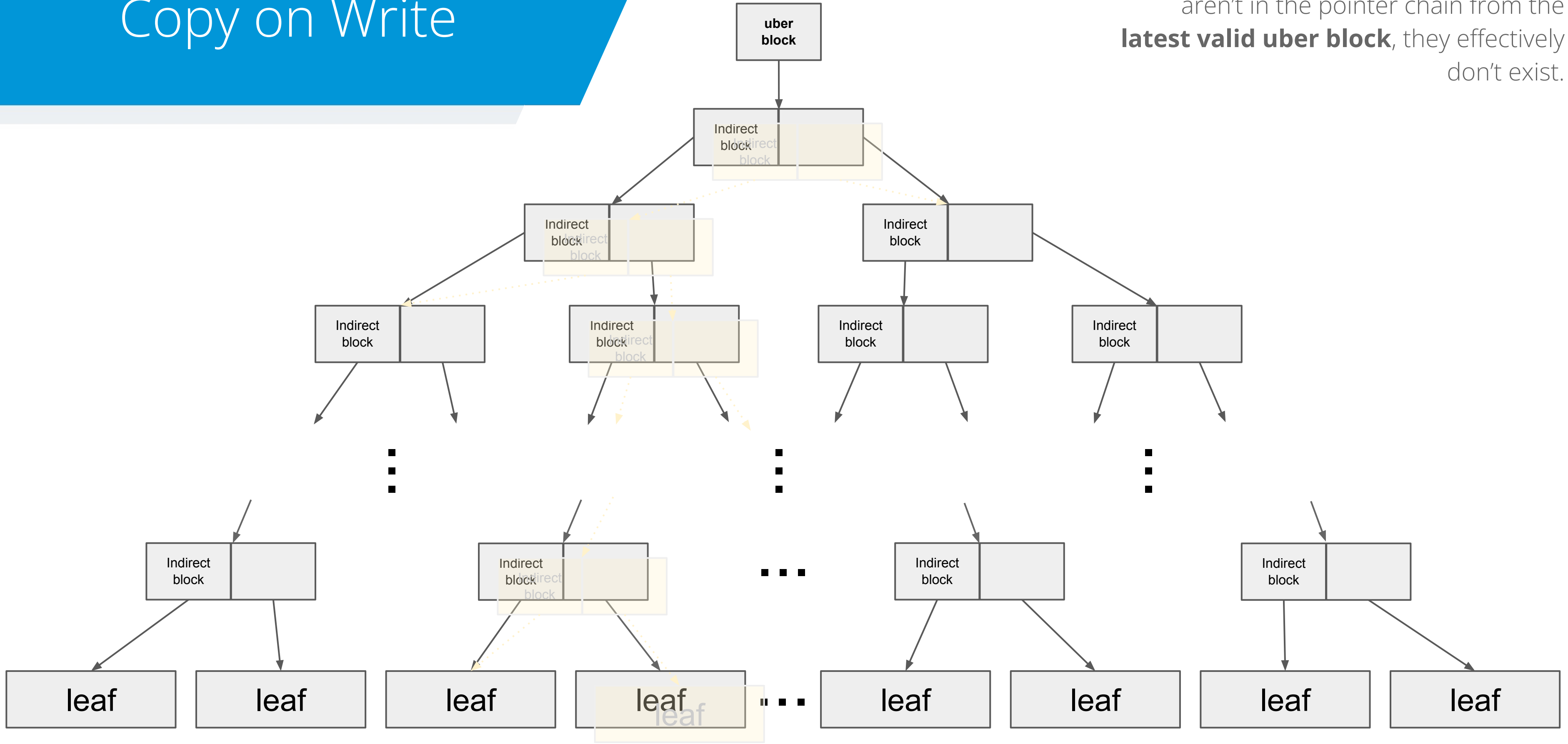
ZFS finds the **latest valid uber block** in the special section on the disk. The partially-written write still exists on the disk (including the partially-written indirect block), but nothing is pointing to it.



iXsystems

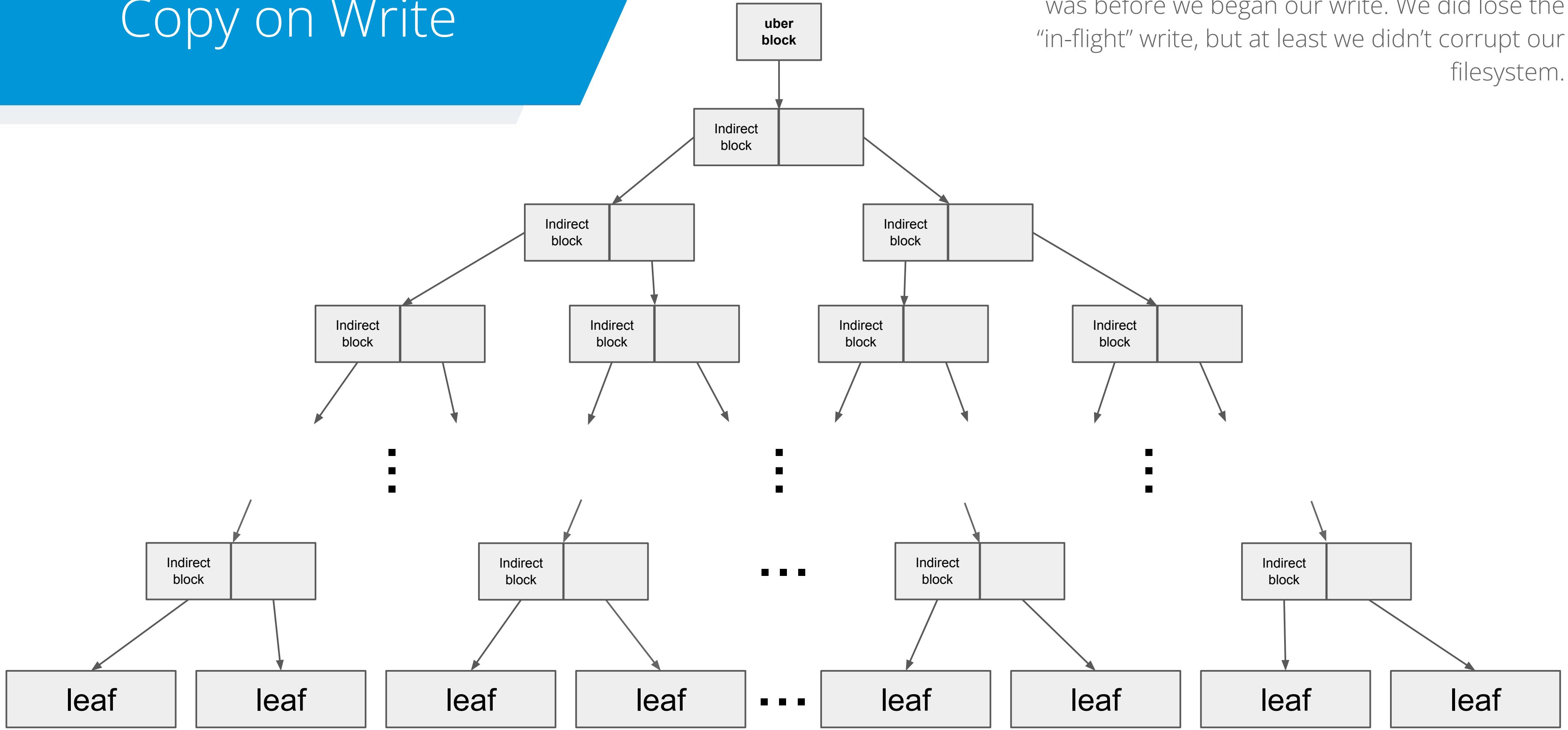
Copy on Write

Because all the new blocks we wrote out aren't in the pointer chain from the **latest valid uber block**, they effectively don't exist.

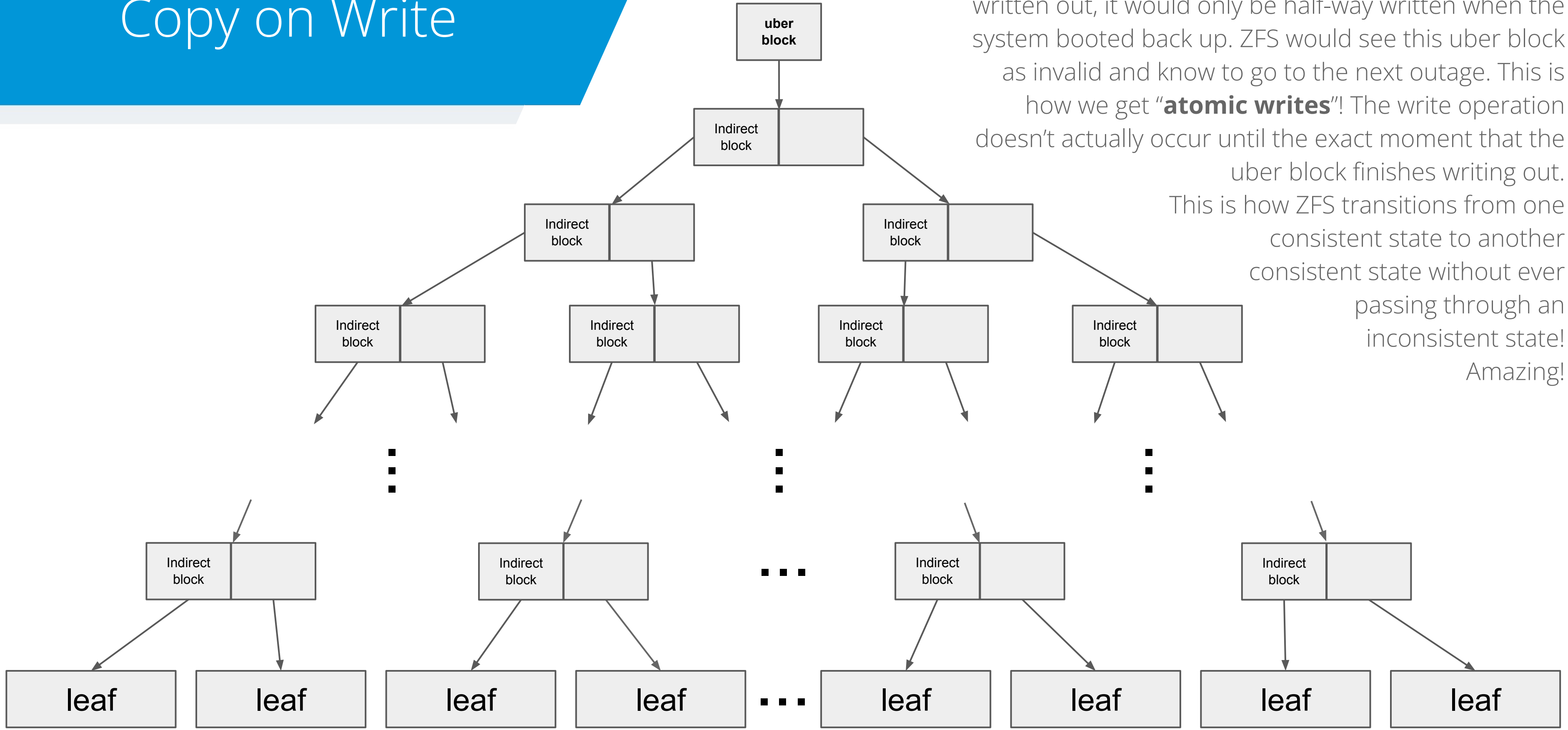


Copy on Write

Our file system is essentially "rolled back" to where it was before we began our write. We did lose the "in-flight" write, but at least we didn't corrupt our filesystem.



Copy on Write



If the outage occurred while the uber block was being written out, it would only be half-way written when the system booted back up. ZFS would see this uber block as invalid and know to go to the next outage. This is how we get "**atomic writes**"! The write operation doesn't actually occur until the exact moment that the uber block finishes writing out. This is how ZFS transitions from one consistent state to another consistent state without ever passing through an inconsistent state! Amazing!

iXsystems

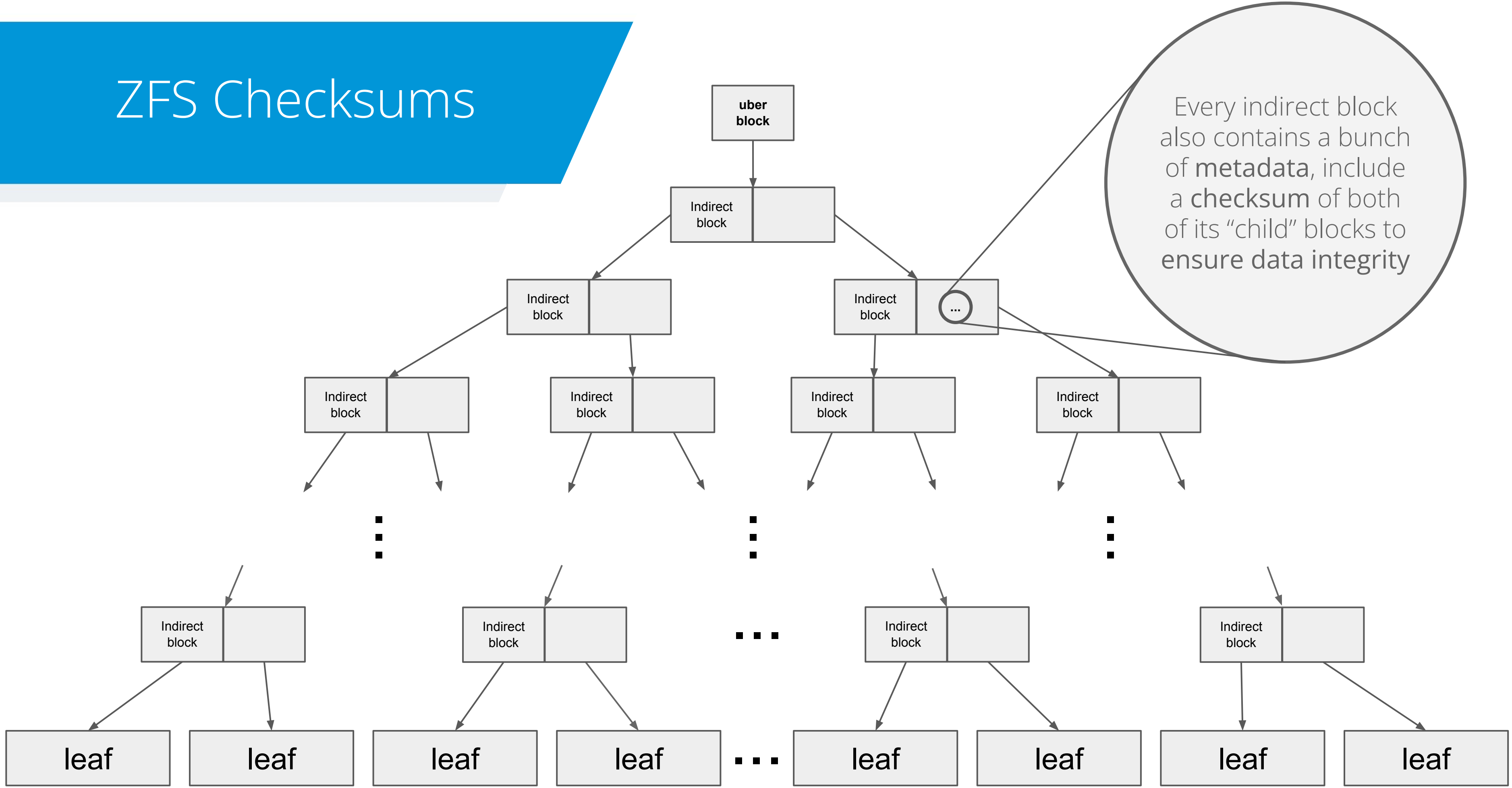
Actual ZFS Data Structure

- The actual on-disk data structure of ZFS is considerably more complicated than what was shown in the previous slides.
- There are several different types of blocks in the tree between the uber block (which is still always at the top) and the first indirect block below that
- This additional structure at the top of the tree lets ZFS manage lots of different datasets, zvols, snapshots, clones, etc.
- Imagine the uber block pointing to the start of a complex tree structure and one of the blocks at the bottom of that tree structure points to the rest of the diagram we had. Other bottom blocks on that top tree structure point to zvols, snapshots, clones, each with their own tree.
- To complicate things even more, each indirect block can actually contain thousands of block pointers and each of those block pointers sometimes contain pointers to 2 or even 3 locations of the data on the disk!
- For a full, detailed description of ZFS's on-disk structure, see the book ***The Design and Implementation of The FreeBSD Operating System*** (Chapter 10)!

ZFS Checksums

How do we know the data on our disks hasn't changed since we wrote it?

ZFS Checksums



What is a Checksum?

- Uses a “hash function” to generate a unique (long, hex) number for a given data input
- If input data changes even slightly, the hash output will totally change!
- Hash function output is always a fixed length regardless of input length
- Designed to avoid hash “collisions”, i.e., two different inputs that produce the same output number

- MD5 hash of the text “**iXsystems believes that Open Source technology has the power to change the world through its process of open and collaborative innovation.**”:

  `1fe6e8a3549cd5642d3c0769d034af65`

- MD5 hash of the text “**iYsystems believes that Open Source technology has the power to change the world through its process of open and collaborative innovation.**”:

  `d5c8e32d0dca8c248bb15b0ed4780be1`

- MD5 hash of the text “**hi**”:

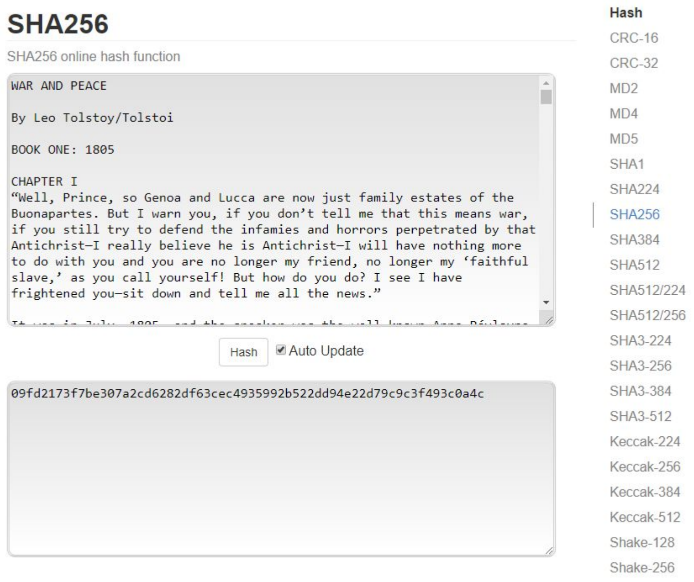
  `49f68a5c8493ec2c0bf489821c21fc3b`

iXsystems™

Another Checksum Example

- Checksums can be used to quickly verify the integrity of large quantities of data.
- For example, we want to store the entire text of the Leo Tolstoy's 1,225-page novel *War and Peace*
- This is obviously a very important work, so we want to ensure the text isn't corrupted in any way while it's stored on our system.
- To do this, we take a checksum of the full text with the very popular SHA256 hash function (using the website https://emn178.github.io/online-tools/sha256.html)
- We get this hash for the text:

`09fd2173f7be307a2cd6282df63cec4935992b522dd94e22d79c9c3f493c0a4c`



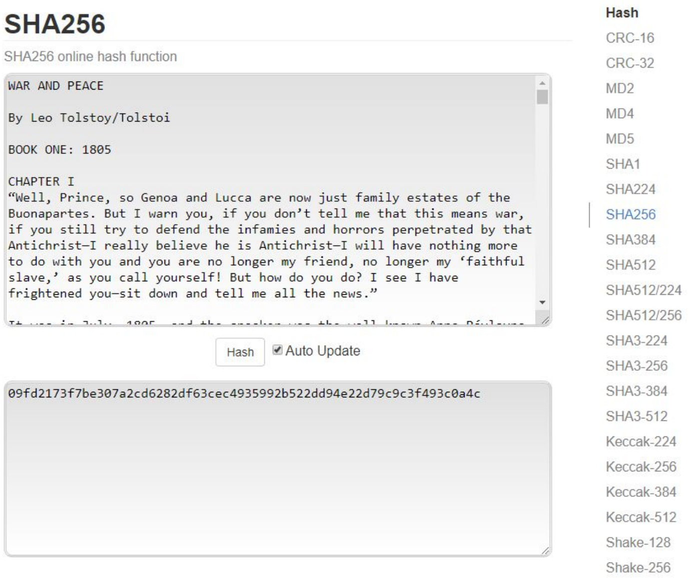
Another Checksum Example

- This hash value…

09fd2173f7be307a2cd6282df63cec4935992b522dd94e22d79c9c3f493c0a4c

…is effectively a fingerprint of the text we gave it.

- If any of the 562,486 words or 3,201,582 characters changes even a little bit (or even a comma becomes a period) the has value will change entirely… Let's make a tiny change and see what happens!



Another Checksum Example

- This hash value…

09fd2173f7be307a2cd6282df63cec4935992b522dd94e22d79c9c3f493c0a4c

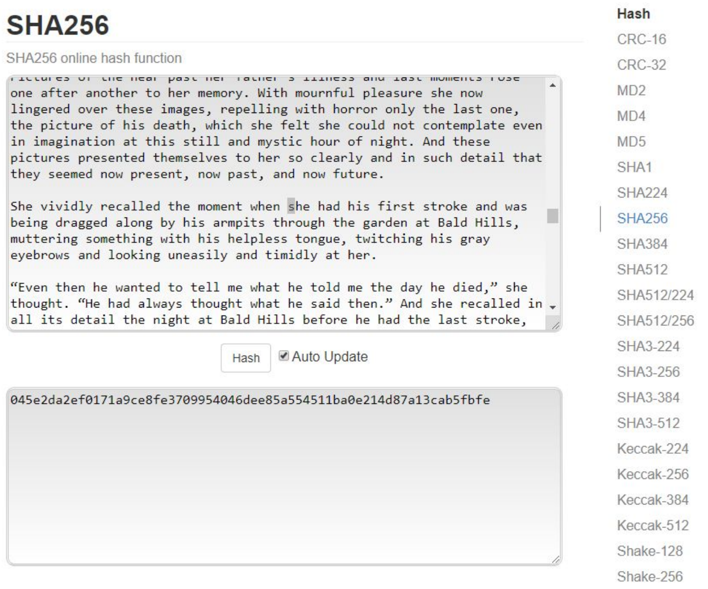
…is effectively a fingerprint of the text we gave it.

- If any of the 562,486 words or 3,201,582 characters changes even a little bit (or even a comma becomes a period) the has value will change entirely… Let's make a tiny change and see what happens!

- We add an "s" in front of a "he" somewhere in Book 10, Chapter XII and the hash value changed to:

045e2da2ef0171a9ce8fe3709954046dee85a554511ba0e214d87a13cab5fbfe

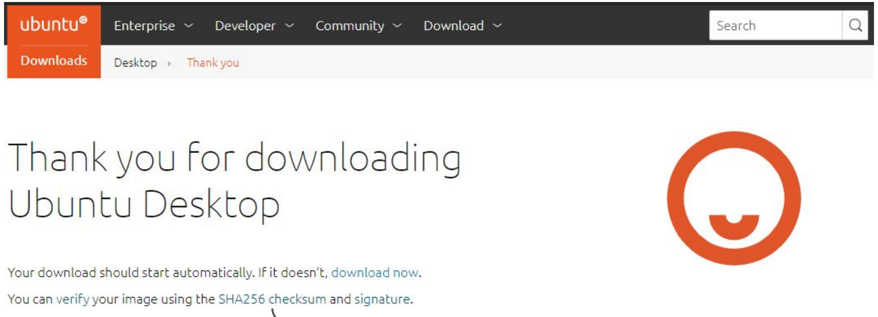
- If we accessed this saved text 20 years after we saved it to our system to share it with our children, we would re-compute the hash value on the data we read and if it doesn't match the hash value we computed and saved originally, we would know the data was somehow corrupted!



iXsystems

Other Checksum Uses

- Hash functions are also used to verify the integrity of large file downloads.
- For example, if you want to download the ISO file to install Ubuntu Linux, you can make sure the download didn't mess up by running a hash checksum on your downloaded file and comparing it to the hash values that Ubuntu lists on its website for the download.
- If your checksum doesn't match theirs, the file you downloaded somehow got corrupted!




```
22580b9f3b186cc66818e60f44c46f795d708a1ad86b9225c458413b638459c4 *ubuntu-18.04.2-desktop-amd64.iso
ea6ccb5b57813908c006f42f7ac8eaa4fc603883a2d07876cf9ed74610ba2f53 *ubuntu-18.04.2-live-server-amd64.iso
```

When data is read from ZFS...

- ZFS checks all checksums against data that will be read for every block in the tree (including uber block, all indirect blocks, and the leaf block)
- If checksum of any block in the tree doesn't match, ZFS will...
    a. Attempt to fetch the block from redundant disk(s) (i.e., mirror disk or parity data in RAID-Z), performing same checksum verification along the way
    b. Correct the corrupted blocks with the data from the verified redundant disk(s)
    c. Make a note that there was a data corruption error on one of the disks
    d. Serve up the data to whatever process requested it
- If there are no redundant disks, ZFS will generate an error that faults the pool and will not serve up any data