File Systems Part 6

UBI/UBIFS

- UBI (unsorted block images)
  - supports multiple logical volumes on one flash device
  - performs wear-leveling across entire device
  - handles bad blocks
- UBIFS
  - file system layered on UBI
  - it really has a journal (originally called JFFS3)
  - page index kept in flash
    - no need to scan entire file system when mounted
  - compresses files as an option

Some information about how UBIFS works can be found at http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf. (UBI stands for Unsorted Block Images.)

Updates to the page index are done in a transactional fashion: they are stored in a journal, then updated in flash after being committed. The goal is not so much to have atomic updates, but to reduce the frequency of writes to flash. Thus a number of updates can be done via a single write.

Wednesday's Quiz

Suppose one used UBI/UBIFS on a disk rather than on a flash drive. Would it still be a usable file system?

a) no
b) yes, but some of what it does would be unnecessary and thus a waste of time
c) yes, and everything it does would be useful, even on a disk

Quiz 1

We've discussed the following HDD-based file-system topics:

1) seek and rotational delays
2) mapping file-location to disk-location
3) directory implementations
4) transactions
5) RAID

Which are not relevant for file systems on SSDs?

a) all
b) none
c) just one
d) just two

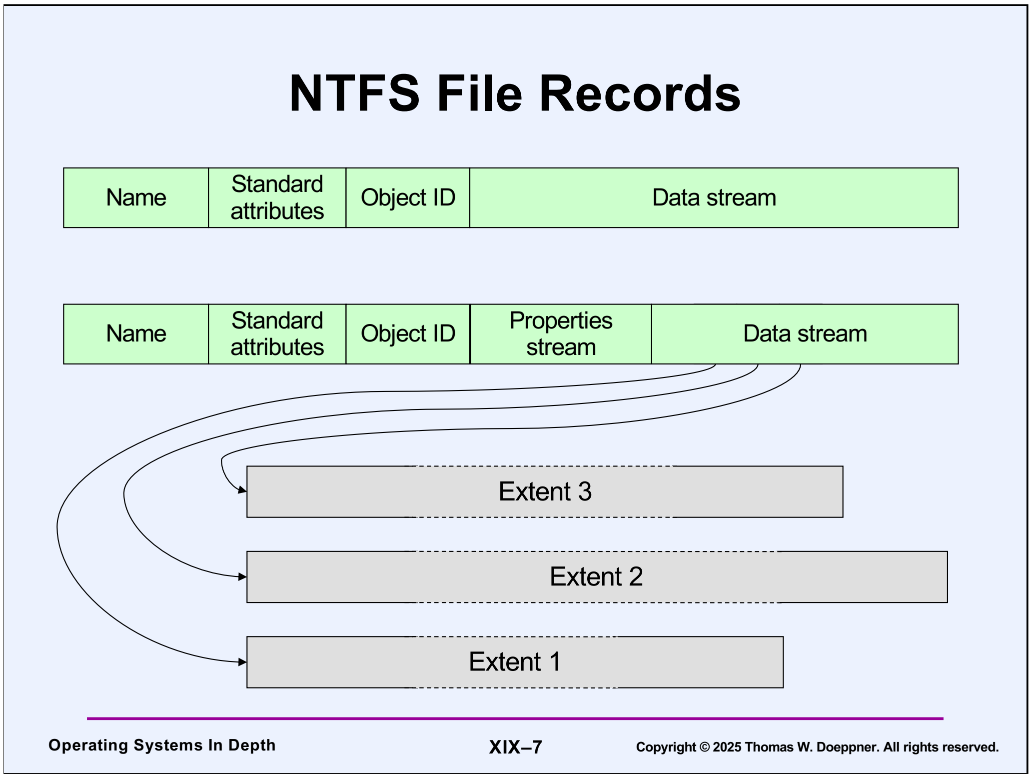
NTFS

- Journaled
  - normally redo
  - can do redo and undo simultaneously
- "Volume aggregation" options
  - spanned volumes
  - RAID 0 (striping)
  - RAID 1 (mirroring)
  - RAID 5
  - snapshots

Quiz 2

What is the benefit of doing redo and undo journaling simultaneously?

a) It makes the file system twice as reliable as it would be with just one
b) Since updates are committed before checkpointing, updates are less likely to be lost in the event of a crash
c) If the OS is running low on RAM, undo journaling makes it possible to reclaim RAM from cached blocks quickly
d) both b and c
e) none of the above



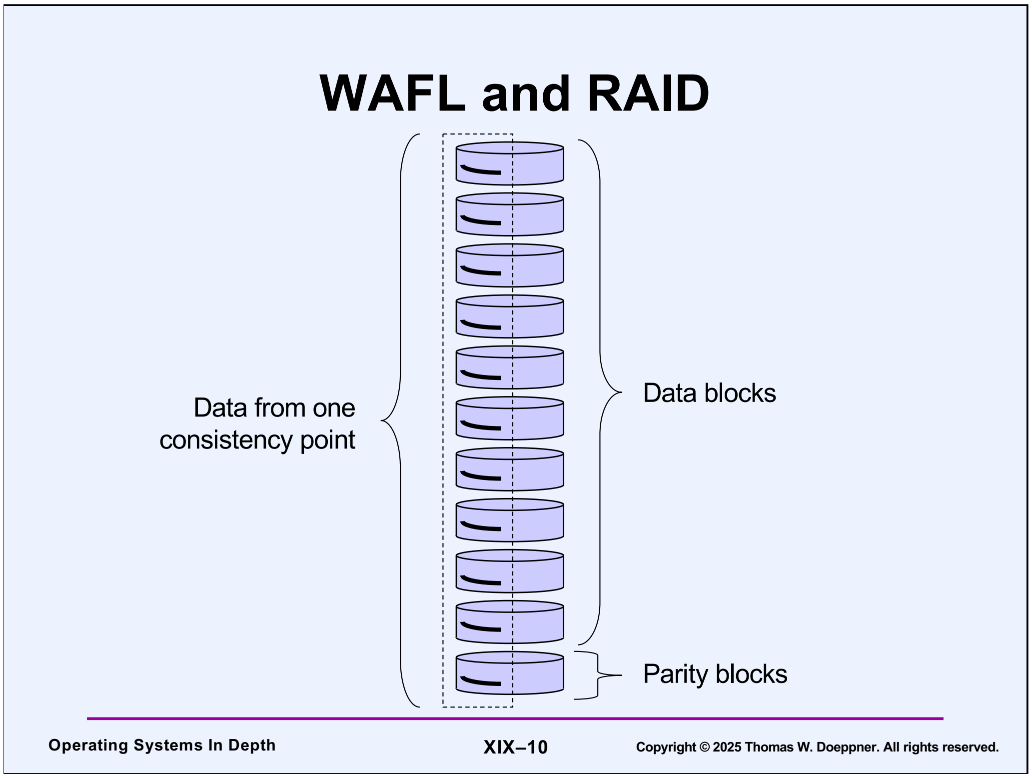
Two NTFS file records. The top one is for a small file fitting completely within the record. The bottom one is for a file containing two streams: one for some application-specific properties and one for normal data. The latter is too big to fit in the file record and is held in three extents.

Additional NTFS Features

- Data compression
  - run-length encoding of zeroes
  - compressed blocks
- Encrypted files

WAFL

- Runs on special-purpose OS
  - machine is dedicated to being a *filer*
  - handles both NFS and SMB requests
- Utilizes shadow paging and log-structured writes
- Provides snapshots



WAFL uses RAID level 4 and writes out changes in batches called consistency points. Each batch occupies some integral number of stripes — thus the parity disk is written to no more often than the data disks. Though blocks in a consistency point are from any number of files, they're written (using log-structured techniques) to the next contiguous locations on each disk.

Consistency Points … and Beyond

- Consistency points taken every ~10 seconds
  - too relaxed for many applications
    - NFS
    - databases
- Solution …

(battery-backed-up RAM)
(a.k.a. non-volatile RAM (NVRAM))

Between snapshots, changes are logged in NVRAM and recovered after a crash.

Quiz 2

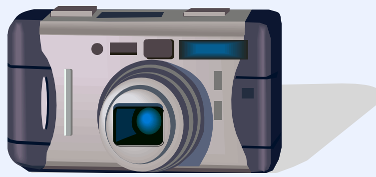
We have an 8-disk RAID 4 system (with a single parity disk). One of the disks goes bad – its entire contents are lost. Can we recover its contents using nothing but the contents of all the other disks?

a) no

b) yes

c) it depends upon whether the bad disk was the parity disk or was a data disk

Snapshots

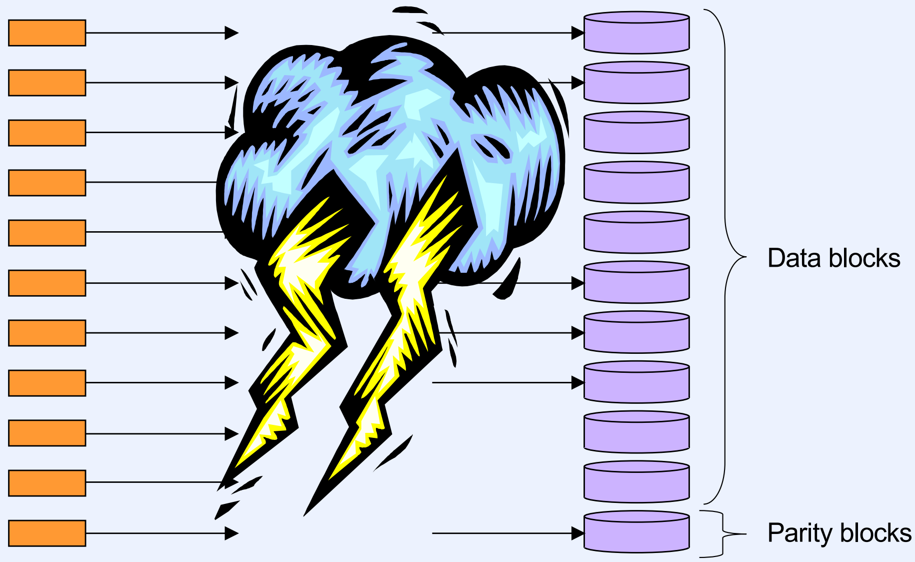


- Periodic snapshots kept of file system
  - made easy with shadow paging

Paranoia

- You think your files are safe simply because they're on a RAID-4 or RAID-5 system …
  - power failure at inopportune moment
    - parity is irreparably wrong
  - obscure bug in controller firmware or OS
    - data is garbage (but with correct parity!)
  - sysadmin accidentally scribbled on one drive
    - (profuse apologies …)
  - out of disk space
    - must restructure 16TB file system
  - out of address space
    - 2^{64} isn't as big as it used to be

Partial Writes



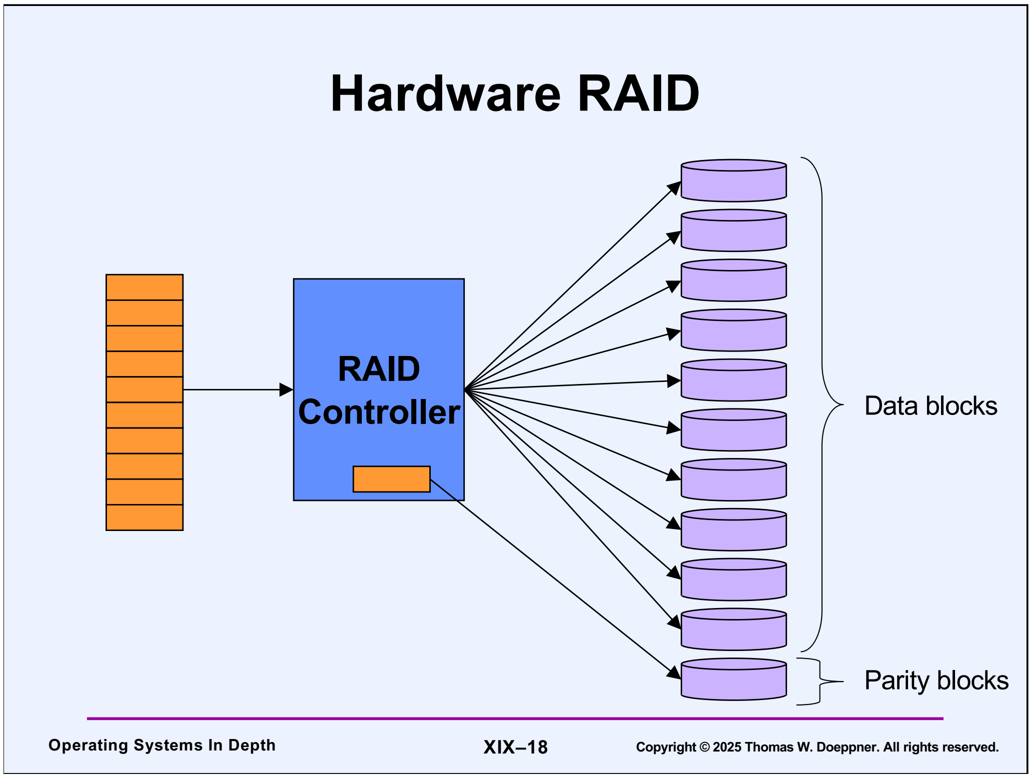
Quiz 3

NTFS employs journaling. Is it susceptible to the problem described in the previous slide? (Hint: it’s possible that parts of a transaction might be written in a single stripe.)

a) no

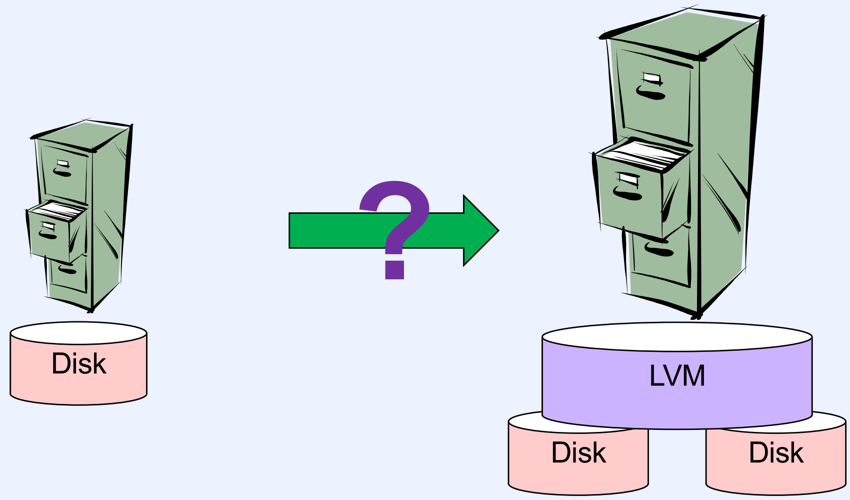
b) yes

Small Writes



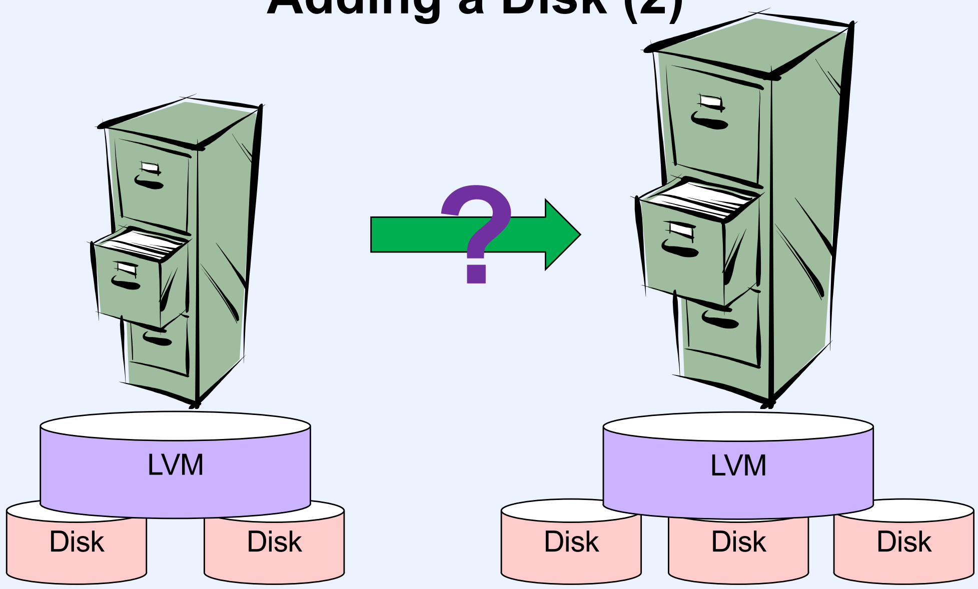
Most RAID systems on the market today do the parity computation in hardware. Thus parity is computed for whatever is given to the RAID controller. Garbage in, parity-protected garbage out.

Adding a Disk (1)



It is not trivial to expand a traditional file system simply by adding another disk, even with the use of a logical volume manager.

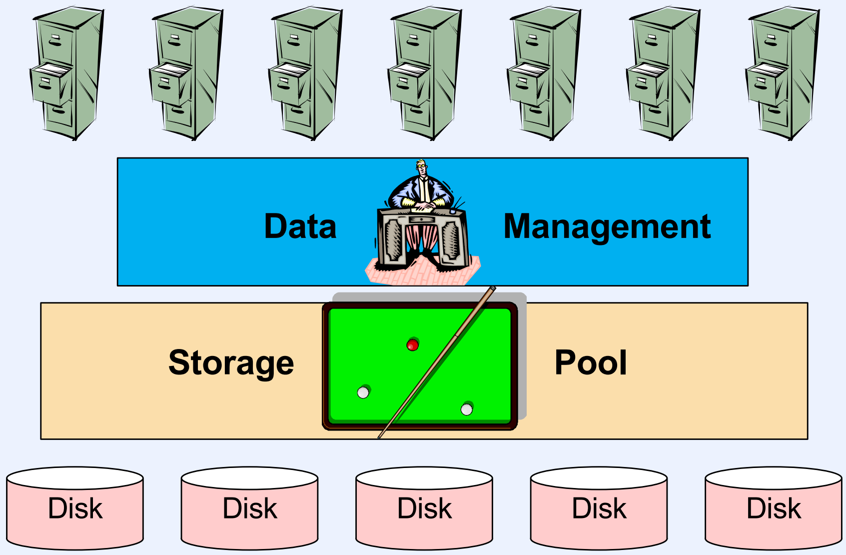
Adding a Disk (2)

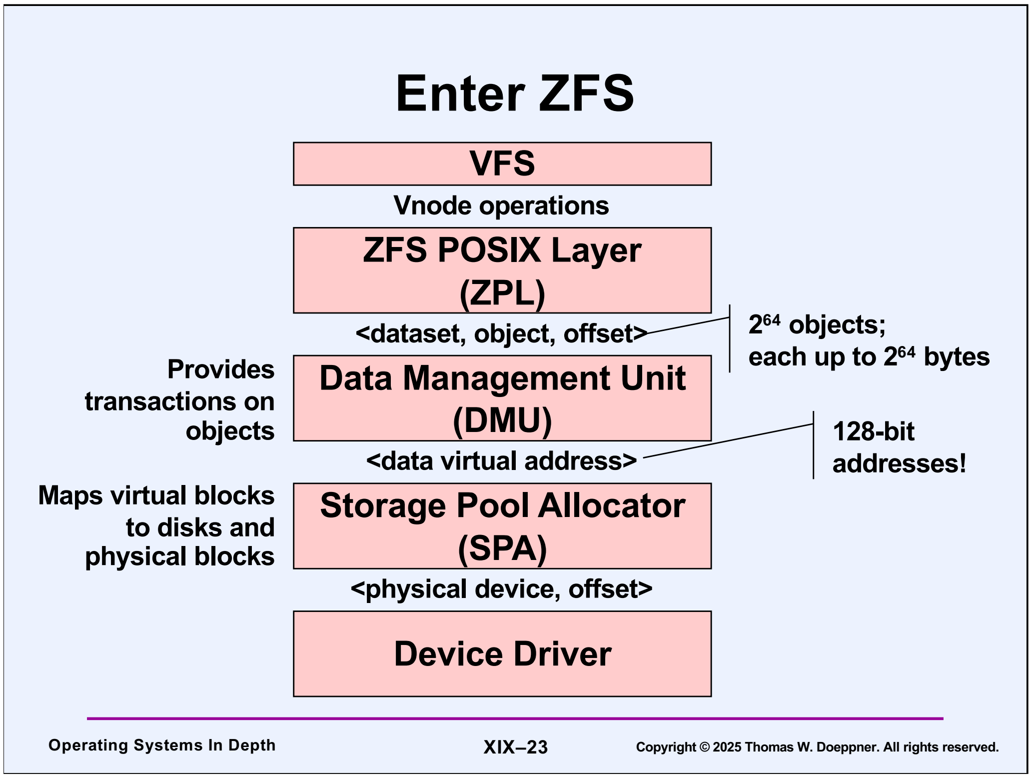


ZFS

The Last (?!) Word in File Systems

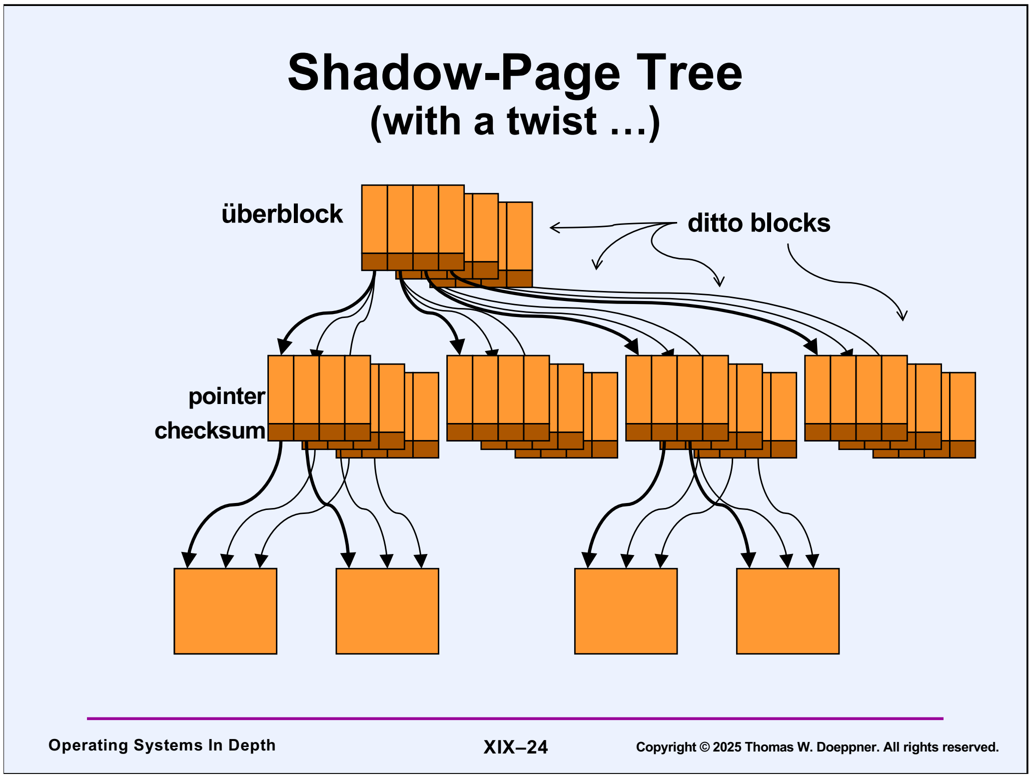
ZFS Layers





ZFS (formerly known as Zettabyte File System) is layered as shown in the slide. ZPL converts standard POSIX-style (Unix-style) operations into operations on objects within datasets. DMU batches these operations into transactions. Both ZPL and DMU operate in a large virtual address space and are oblivious to physical addresses. SPA provides a malloc-like interface for allocating space (it uses slab allocation) and is responsible for mapping the virtual address space into actual disk addresses. In doing so, it may employ spanning, mirroring, or a form of RAID, depending on the configuration.

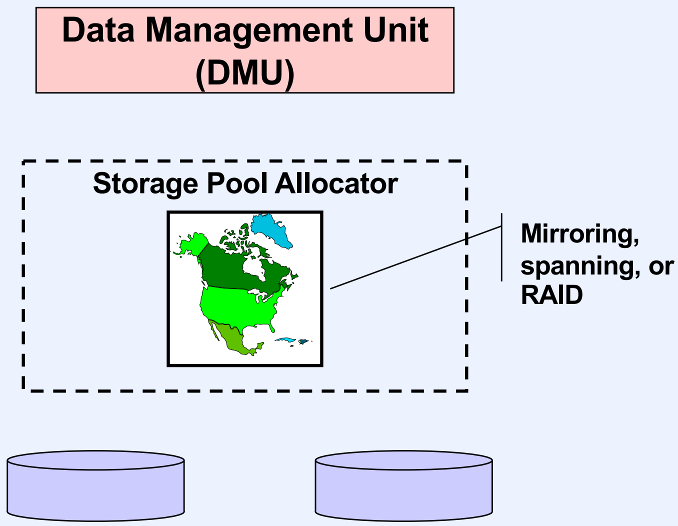
(A zettabyte is 2^{70} bytes. ZFS can handle file systems far larger than that.)



ZFS has a shadow-page tree (implemented in the DMU layer), as we saw in WAFL, but with two new features: each pointer contains a checksum of the block pointed to, and there are multiple copies of each interior node. If a block of data or meta-data becomes corrupted (even if someone accidentally scribbles on it), the checksum enables the system to detect the error. A bad interior node is a particular problem, since it could cause its entire subtree to be lost. Thus there are multiple copies of each (called **ditto blocks**), enabling repair.

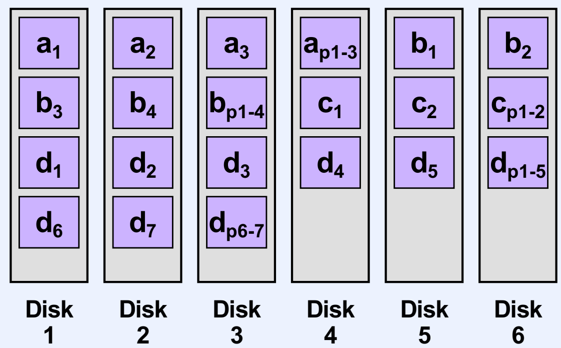
The root of the tree is called the **überblock**. The actual tree is taller than shown (i.e., has more layers of interior nodes).

Storage Pool Allocator



RAID-Z

Software Dynamic Striping



In RAID-Z, each block forms a stripe, regardless of the length, and each has its own parity. Here we are striping blocks a, b, c, and d on six disks. Block a consists of three sectors. It’s written to disks 1 through 3, followed by a parity sector written to disk 4. Similarly, block b consists of four sectors, which are written to the disks followed by a parity sector. Block d consists of seven sectors. So that we have enough parity to recover on the loss of a disk (since the stripe is longer than the number of disks), two parity sectors are supplied for it.

Thus small writes are handled efficiently — it’s not necessary to read the parity sector so as to modify it, since the block’s parity depends only on the block itself. The downside is that, to determine where the blocks are, one must traverse the metadata (the shadow-page tree).

Quiz 4

Compared with RAID 4, which of the following would be more time-consuming with RAID-Z?

a) adding a disk

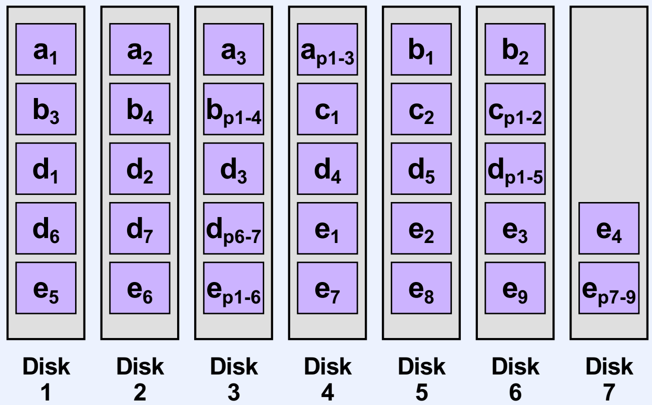
b) replacing a crashed disk

c) both

d) neither

RAID-Z

Adding a Disk



Adding a new disk to a RAID-Z system is easy. There's no parity to recompute; existing blocks are unchanged, but new blocks may be striped over old and new drives.

Scenarios

- Power failure at inopportune moment
  - "live data" is not modified
  - single lost write can be recovered
- Obscure bug in controller firmware or OS
  - detected by checksum in pointer
- Sysadmin accidentally scribbled on one drive
  - detected and repaired
- Out of disk space
  - add to the pool; SPA will cope
- Out of address space
  - 2^{128} is big
    - 1 address per cubic yard of a sphere bounded by the orbit of Neptune

And There's More …

- Adaptive replacement cache
- Advanced prefetching

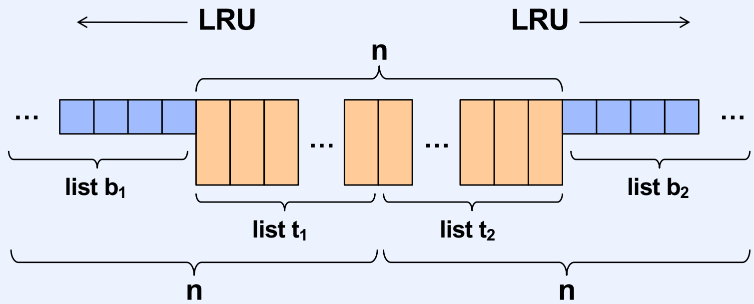
LRU Caching

- LRU cache holds n least-recently-used disk blocks
  – working sets of current processes
- New process reads n-block file sequentially
  – cache fills with this file's blocks
  – old contents flushed
  – new cache contents never accessed again

(Non-Adaptive) Solution

- Split cache in two
  - half of it is for blocks that have been referenced exactly once
  - half of it is for blocks that have been referenced more than once
- Is 50/50 split the right thing to do?

Adaptive Replacement Cache



t_1 ; b_1:

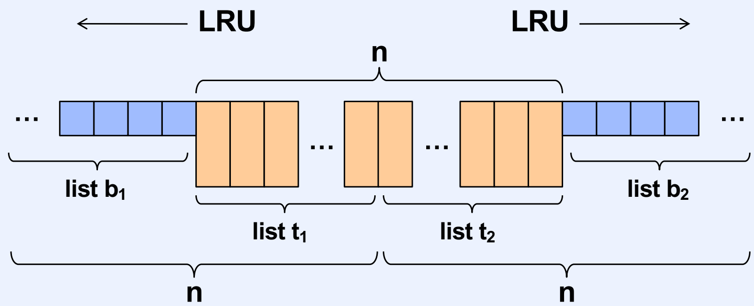
- LRU list of blocks referenced once
- t_1 list (most recently used) contain contents
- b_1 list (least recently used) contain just references

t_2 ; b_2:

- LRU list of blocks referenced more than once
- t_2 list (most recently used) contain contents
- b_2 list (least recently used) contain just references

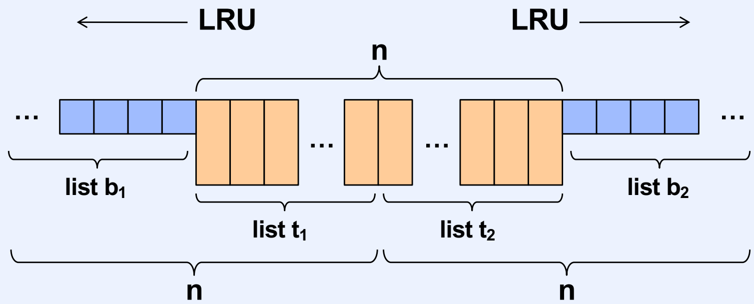
A description of the algorithm can be found at http://www.usenix.org/publications/login/2003-08/pdfs/Megiddo.pdf. A more detailed paper that includes an analysis can be found at http://www.usenix.org/publications/library/proceedings/fast03/tech/full_papers/megiddo/megiddo.pdf.

Adaptive Replacement Cache



cache miss:
- if t_1 is full
  - evict LRU(t_1) and make it MRU(b_1)
- referenced block becomes MRU(t_1)

Adaptive Replacement Cache



cache hit:

- **if in t_1 or t_2, block becomes MRU(t_2)**
- **otherwise**
  - **if block is referred to by b_1, increase t_1 space at expense of t_2**
  - **otherwise (referred to by b_1)**
    - **increase t_2 space at expense of t_1**
  - **if t_1 is full, evict LRU(t_1) and make it MRU(b_1)**
  - **if t_2 is full, evict LRU(t_2) and make it MRU(b_2)**
  - **insert block as MRU(t_2)**

Note that the size of the combined t_1 and t_2 lists doesn't change – it's n. Thus if there was a cache hit and the block was in t_1, it moves to t_2, increasing the size of t_2 and decreasing the size of t_1.

Quiz 5

Lists b_1 and b_2 do not contain cached blocks, but just their addresses. Why are they needed?

a) So that one can determine how much better things would be if the cache were twice as large

b) As placeholders so that when these blocks are read in, it's known where in the cache they would go

c) So that we would know, if the addressed block is referenced, whether it would have been in the cache if the corresponding t list were larger

d) They are used by the file system to help determine block reference patterns

Prefetch

- FFS prefetch
  - keeps track of last block read by each process
  - fetches block i+1 if current block is i and previous was i-1
  - chokes on
    - diff file1 file2

zfetch

- Tracks multiple prefetch streams
- Handles four patterns
  – forward sequential access
  – backward sequential access
  – forward strided access
    - iterating across columns of matrix stored by columns
  – backward strided access