APFS

No clever or witty subtitle*

- But better: a free download! http://NewOSXBook.com/files/fsleuth.tar

About this talk

- Long overdue – I was supposed to be here last year ☹
  - Scheduling conflict and far east travel got in way. Thanks for having me again.
- Standing on the shoulders of giants:
  - APFS research of Kurt H. Hansen & Fergus Toolan (https://www.sciencedirect.com/science/article/pii/S1742287617301408)
- AAPL released the spec, but it's Javadoc/doxygen, and is pretty vague
  - Not anything like TN1150 (HFS+)
- Research was reverse engineering, and spec filled in missing pieces
  - Thanks to AAPL for at least not fully stripping apfs.kext and utilities (yet*)

* - If the next release of Darwin strips symbols, you'll know why.

APFS Features

The High Level View of APFS

APFS timeline

- New file system to replace venerable (15+ years) HFS+
  - Disappointed many who were expecting Apple to adopt ZFS
- Announced in 2016:
  - Initial MacOS 12 implementation was pretty bad:
    - Defined as “preview”
    - Full of incompatibilities with its own subsequent versions
    - No boot support (= EFI protocol)
  - Adopted first in iOS 10.3, full adoption in MacOS 13
    - iOS 11.3 moved to snapshot based mounts
  - Evolving further as Darwin progresses:
    - Darwin 18: Fragmentation
    - Darwin 19: purgeable files, volume groups, firmlinks

APFS features

- 64-bitness:
  - Support for ridiculous file sizes you'll never run into.
  - For-all-intents-and-purposes infinite number of files (2^{64} inodes)
  - Nanosecond-resolution timestamp since the Epoch (Jan 1st, 1970)
    - Y2K38 safe ☺

APFS features

- Built in volume management
  - R.I.P CoreStorage* and iOS's LwVM
  - Partition is now formatted as "Container"
  - Individual mountable filesystems are "Volumes"
  - All volumes share same container

```
Filesystem      Size   Used  Avail Capacity iused               ifree %iused  Mounted on
/dev/disk1s1   466Gi  399Gi   63Gi    87% 1753922 9223372036853021885     0%   /
devfs          221Ki  221Ki    0Bi   100%     764                   0  100%   /dev
/dev/disk1s4   466Gi  3.0Gi   63Gi     5%       4 9223372036854775803     0%   /private/var/vm
map -hosts       0Bi    0Bi    0Bi   100%       0                   0  100%   /net
map auto_home    0Bi    0Bi    0Bi   100%       0                   0  100%   /home
```

* - Goodbye, and Good Riddance!

APFS features

- Fast Directory Sizing
  - Directory totals are saved along with the directory's own inode
  - Allows for faster applications of du(1) and of Finder's Get Info
- Sparse file support
  - lseek(2) + SEEK_SET skipping over data
  - Using extents file system can store only actual data, working around "holes"

APFS features

- Cloning:
  - Rather than copy a file, maintain another reference to it
  - Any changes are stored as subsequent deltas
  - Proprietary system call `clonefileat(2)` (#462), pretty well documented

```
CLONEFILE(2)              BSD System Calls Manual             CLONEFILE(2)

NAME
     clonefile -- create copy on write clones of files

SYNOPSIS
     #include <sys/attr.h>
     #include <sys/clonefile.h>

     int
     clonefile(const char * src, const char * dst, int flags);

     clonefileat(int src_dirfd, const char * src, int dst_dirfd, const char * dst, int flags);

     fclonefileat(int srcfd, int dst_dirfd, const char * dst, int flags);

DESCRIPTION
     The clonefile() function causes the named file src to be cloned to the named file dst.  The cloned file dst shares its data blocks
     with the src file but has its own copy of attributes, extended attributes and ACL's which are identical to those of the named file
     src with the exceptions listed below

     1.   ownership information is set as it would be if dst was created by openat(2) or mkdirat(2) or symlinkat(2) if the current user
          does not have privileges to change ownership. If the optional flag CLONE_NOOWNERCOPY is passed, the ownership information is
          the same as if the the current user does not have privileges to change ownership

     2.   setuid and setgid bits are turned off in the mode bits for regular files.
```

APFS features

- Copy-on-Write
  - Contrary to other file systems, changes do not get written into same block
  - APFS is a Copy-on-Write filesystem
  - This makes APFS especially Flash Friendly (avoids P/E cycles wear)
  - Ensures much better resiliency in the face of possible crashes
  - Also makes APFS forensic friendly
  - Surprisingly, though – no undelete functionality provided by Apple

APFS features

- Snapshots:
  - Similar to well-known (and darn useful) virtual machine snapshots
  - Used by Time Machine, through the `tmutil(8)` command-line

```
localsnapshot
        Create new local Time Machine snapshots of all APFS volumes included in the Time Machine backup.

listlocalsnapshots mount_point
        List local Time Machine snapshots of the specified volume.

listlocalsnapshotdates [mount_point]
        List the creation dates of all local Time Machine snapshots.

        Specify mount_point to list snapshot creation dates from a specific volume.

        Listed dates are formatted YYYY-MM-DD-HHMMSS.

deletelocalsnapshots date
        Delete all local Time Machine snapshots for the specified date (formatted YYYY-MM-DD-HHMMSS).

thinlocalsnapshots mount_point [purge_amount] [urgency]
        Thin local Time Machine snapshots for the specified volume.

        When purge_amount and urgency are specified, tmutil will attempt (with urgency level 1-4) to reclaim purge_amount in bytes by thinning snapshots.

        If urgency is not specified, the default urgency will be used.
```

  - Maintained by `fs_snapshot(2)` system call

```
FS_SNAPSHOT_CREATE(2)      BSD System Calls Manual      FS_SNAPSHOT_CREATE(2)

NAME
     fs_snapshot_create -- create read only snapshot of a mounted filesystem

SYNOPSIS
     #include <sys/attr.h>
     #include <sys/snapshot.h>

     int
     fs_snapshot_create(int dirfd, const char * name, uint32_t flags);

     int
     fs_snapshot_delete(int dirfd, const char * name, uint32_t flags);

     int
     fs_snapshot_list(int dirfd, struct attrlist * name, void * attrbuf, size_t bufsize, uint32_t flags);

     int
     fs_snapshot_rename(int dirfd, const char * old, const char * new, uint32_t flags);

     int
     fs_snapshot_mount(int dirfd, const char * dir, const char * snapshot, uint32_t flags);

     int
     fs_snapshot_revert(int dirfd, const char * name, uint32_t flags);

DESCRIPTION
     The fs_snapshot_create() function, for supported Filesystems, causes a snapshot  of the Filesystem to be created. A snapshot is a read only copy of the filesystem frozen at
     a point in time.  The Filesystem is identified by the dirfd parameter which should be a file descriptor associated with the root directory of the filesystem for which the
     snapshot is to be created.  name can be any valid name for a component name (except . and ..).  The fs_snapshot_delete() function causes the named snapshot name to be
     deleted and the fs_snapshot_rename() function causes the named snapshot old to be renamed to the name new.  Available snapshots along with their attributes can be listed by
     calling fs_snapshot_list() which is to be used in exactly the same way as getattrlistbulk(2).  The flags parameter specifies the options that can be passed. No options are
     currently defined.
_
```

APFS features

- Additional features (inherited from VFS) are:
  - Extended Attributes
    - Arbitrary key/value combinations, viewable through `ls -@`
  - Transparent File Compression
    - chattr(1) compressed, `ls -O`
    - Files compression metadata is in (invisible) com.apple.decmpfs extended attribute
    - Small files compressed directly into attribute value; larger files compressed on disk
  - Resource forks
    - com.apple.ResourceFork extended attribute (`ls -@`)
    - Also accessible through *filename*`/../namefork/rsrc` (yes, seriously)
    - Ensures compatibility with MacintoshFS, from 20 years ago*

* - Also, great way to hide data, if you're malware..

APFS features

- APFS Fuses two of Apple's strongest encryptions:
    - FileVault 2 ("Full Disk Encryption")
        - Required to mount the volume
        - Remains in memory for lifetime of mount
        - Hardware accelerated on iOS and Macs with new T2 chip that's popping up everywhere
    - NSFileProtectionClass ("Per File/Class Encryption")
        - Required to access a file
        - One of four* protection classes
            - D: Unprotected (UID only – still requires on device access)
            - C: protected First Unlock
            - B: unless file is open (ECC)
            - A: unless unlocked

* - Technically, five, but I'm ignoring class F here

APFS (Darwin 19) features

- Volume groups:
  - Light LVM management allowing grouping of related volumes
  - More volume roles (Data, Baseband, Preboot, installer)

- Firmlinks:
  - com.apple.fs.firmlink xattr, SF_FIRMLINK BSD flag
  - Conceptually, hardlinks across volumes
  - Used in MacOS to keep / read-only and mount writable /Volumes/Data
  - WWDC ‘19 Session 710

Forensic hurdles

- T2 (or iOS) complicates access to encrypted containers and volumes
  - MacOS: requires use of Target Disk Mode
    - Not possible if firmware password is set
    - Would still need user's password/recovery key
  - https://www.apple.com/mac/docs/Apple_T2_Security_Chip_Overview.pdf
    - Can image raw device if code execution is possible
      - Root + com.apple.rootless.restricted-block-devices entitlement
  - iOS: requires acquisition to be performed only by code on the device
    - Prerequisite: Jailbreak
    - Entitlement: com.apple.private.security.disk-device-access

Forensic hurdles

- APFS supports asynchronous TRIM
  - Deleted files can be purged from SSD using NAND TRIM
  - This effectively makes files unrecoverable

- T2 (or iOS) complicates access to encrypted containers and volumes
  - Can’t simply desolder NAND/SSD and dump
  - MacOS: disables external drive boot, leaving only Target Disk Mode
    - Not possible if firmware password is set
    - Would still need user’s password/recovery key
  - https://www.apple.com/mac/docs/Apple_T2_Security_Chip_Overview.pdf

Forensic hurdles

- NSFileProtectionClass Levels C and greater require device unlock
  - Can’t do much about that.

- Keychain protected data is SEP-enforced
  - Bypassable if on the device using the ShaiHulud method or target injection.

Apple's APFS tools

Binary	Purpose
apfsd(8)	APFS Volume Management Daemon. Invoked automatically to maintain mounted volumes.
apfs.util(8)	Extremely limited APFS file system utility
apfs_condenser	MacOS 14 – shrink/defrag containers (won't even output command line arguments)
apfs_invert	Apparently inverts container and volume (not brave enough to try this yet)
apfs_stats	Gets human readable statistics for IORegistry. Invoked by sysdiagnose(8)
fsck_apfs(8)	APFS file system checker; Invoked automatically when fsck(8) detects APFS
hfs_convert(8)	Converts HFS+ volumes to APFS
`mount_apfs(8)`	APFS file system mounter; Invoked with –t apfs (or when APFS is detected)
`newfs_apfs(8)`	Format a block device to create an APFS container and/or add volumes to an existing one
`slurpAPFSMeta`	Dumps APFS metadata from an APFS volume. Useful for debugging..

Let’s get technical

The Low Level view of APFS

Ignorance was bliss. You might want to space out/Insta-Message-Snap-Post instead at this point

How does APFS really work?

- Don’t ask. You don’t need to know.
- It’s the best file system. Ever*.
- It Just Works.™
- But if you want to know more: http://NewOSXBook.com/tools/fsleuth

* - ZFS advocates might disagree. But they’re just BSD-folk. This is Darwin. The very name of the OS shows how evolved it is.

General file system nomenclature

Term	Meaning
Block	Atomic unit of disk space. Usually 512-8,192 bytes. APFS uses 4,096
Extent	Sub unit of a block, used when files are smaller than a block size so as to save space
File	A mapping of a logical name to a set of blocks and/or extents
Contiguity	A File (or free space) spanning sequential blocks. May impact (non-SSD) disk I/O performance
Fragmentation	Unallocated/freed blocks in non-contiguous chunks arising over time from file creation/deletion
SuperBlock	A special block on disk, usually at fixed location(s), providing file system metadata
Inode	Index node – metadata (block allocation, permissions, unique identifier) of file in file system.
`fsck(8)`	A command you don't want to find yourself executing.

A good file system must provide an optimal allocation of blocks (= less wasted space as possible), ensuring maximum contiguity (= minimal fragmentation), reliability, and recoverability, while minimizing I/O overhead.

APFS file system blocks

- A given block in an APFS file system may be:
  - **Free:** contents may be zeroed out, or left over from previous generation
  - **File data:** contents may be fragment of some file data stream
  - **APFS object:** One of specific types used by APFS for its metadata.

- APFS objects are easily recognizable by a Fletcher 64 checksum
  - Highly efficient, SIMD-capable
  - If checksum is valid, it's an object
  - Else likely some stream fragment (or corrupt anyway)
  - Caveat: Zero and all 0xFF blocks

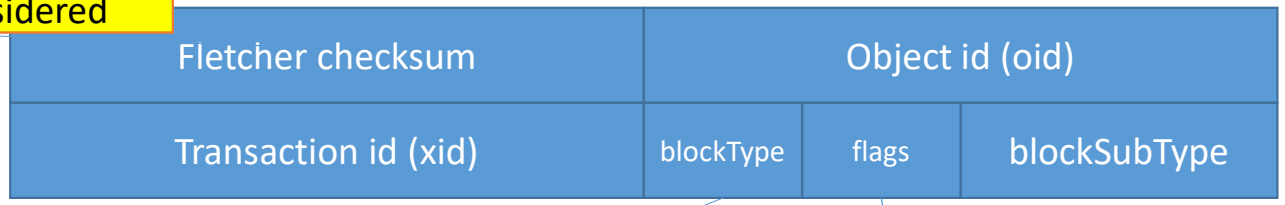
```
uint64_t fletcher64(const uint32_t *data,
                    size_t cnt,
                    uint64_t init) {
        size_t i;

        uint64_t sum1 = init & 0xFFFFFFFFU;
        uint64_t sum2 = (init >> 32);

        for (k = 0; k < cnt; k++) {
                sum1 += data[i];
                sum2 += sum1;
        };

        sum1 = sum1 % 0xFFFFFFFF;
        sum2 = sum2 % 0xFFFFFFFF;
        return ((sum2) << 32) | (sum1);
}
```

APFS Objects

- All object nodes start with a 32-byte header:

Fast checksum, must be valid for block to be considered

64-bit ID indexed by the object map



Allows versioning and checkpoints for objects

#	
1	NXSB (Container)
2	B-Tree root node
3	B-Tree non-root node
12	Object Map
13	APSB (Volume)

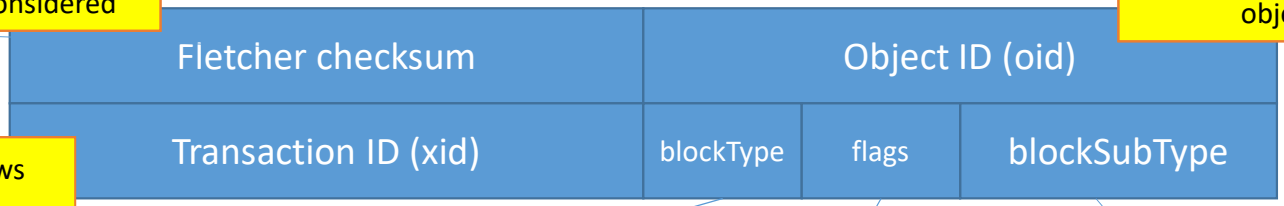
Some 26 object types presently defined – these are the common ones

Flags indicate storage method of object

#	
0x0	Virtual
0x80..	Ephemeral
0x40..	Physical

Fast checksum, must be valid for block to be considered

64-bit ID indexed by the object map



Transaction ID allows versioning and checkpoints for objects

The Block Types identify the type of object contained

Flags indicate storage type of object and additional properties

Block subtypes draw from same space as block types, but are commonly used when the block type is a B-Tree node

#	Block/Object Type
1	NXSB (Container)
2/3	B-Tree root/non-root node
5-9	Space Manager objects
11	Object Map
12	Checkpoint Map
13	APSB (Volume)
17/18	Reaper/Reap List
20	EFI Jumpstart (boot info)
22-23	Fusion Write Back Cache
24	Encryption Rolling Info
25,27	General Bitmap Tree/Block

Flag	Meaning
0	Virtual
0x8000	Ephemeral
0x4000	Physical
0x2000	No header
0x1000	Encrypted
0x0800	Transient

#	Block/Object Sub Type
10	Extent List Tree
11	Object Map
14	File System Tree
15	Block Reference Tree
16	Snapshot Metadata Tree
19	Object Map Snapshot
21	Fusion Middle Trees
26	General Bitmap Tree

APFS Objects

- Objects can be stored by one of three methods:
  - **Physical** objects are stored at a physical 64-bit block address
  - **Ephemeral** objects are stored on disk, but change during mount
  - **Virtual** objects may “move about” disk and address needs to be looked up

- An **object map** is used to look up physical addresses of virtual objects
  - Object map is a B-Tree
  - Container Object Map for global (container-scope) objects
  - Per-Volume Object Map for local (volume-scope) objects

To B or not to B(-Tree)

- B-Trees are fundamental data structures in modern file systems
- Used by HFS+, and unsurprisingly also in APFS (similar node format)
  - Allows for quick conversion of apfs_hfs_convert
- Enable efficient lookup of nodes in logarithmic time – $O(\log_b(n))$
  - 100 files – O(7) operations (for b=2)
  - 1,000,000 files – O (20) operations (for b=2)
  - 1,000,000,000 files – O(30) operations (for b=2)
  - In practice b is higher than 2 (e.g. 5), making operations even more efficient.

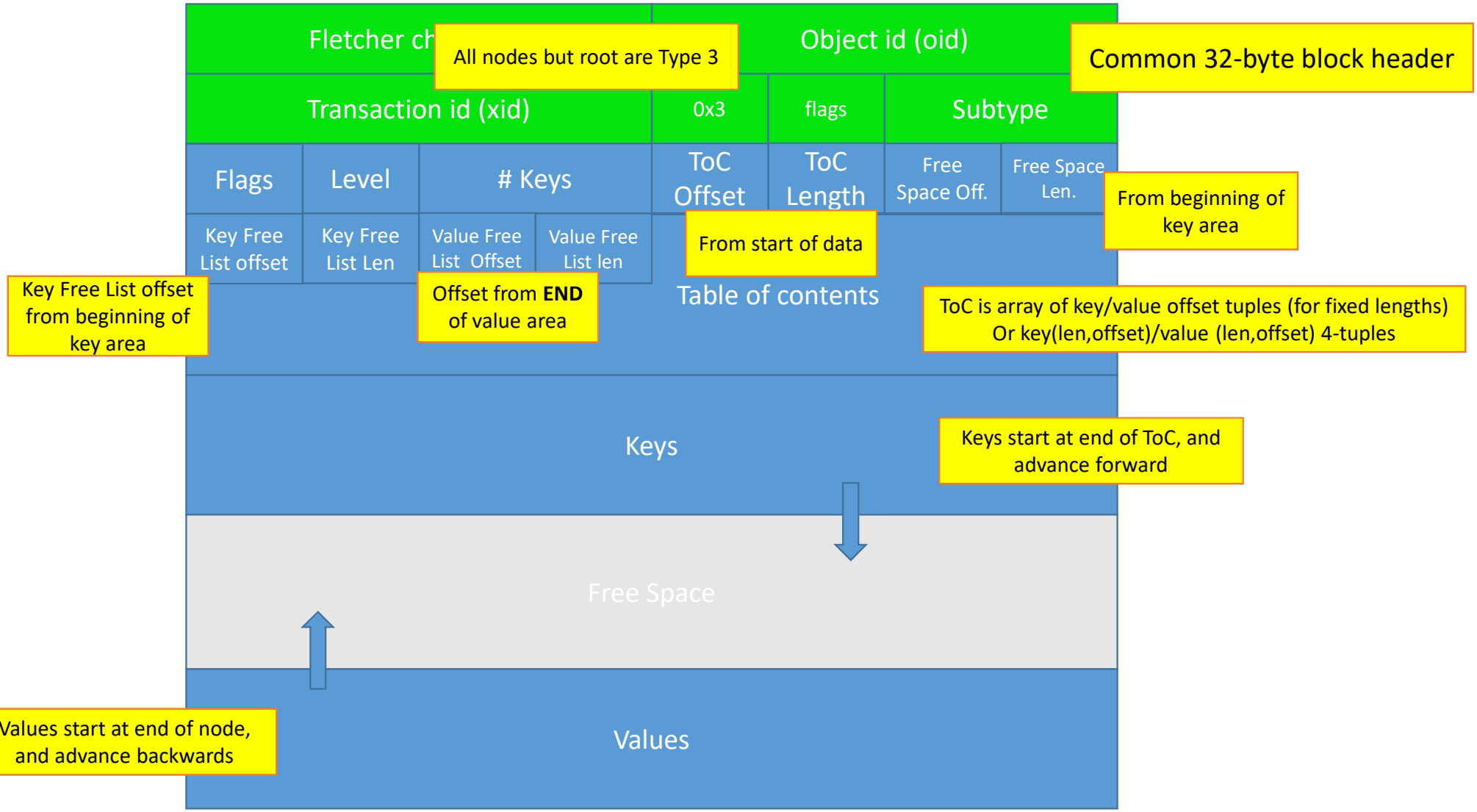
Don’t just B. B+

- APFS B-Tree are specific types called B+ Trees, which satisfy:
  - Every node can have a large number of children
  - Internal nodes index the smallest keys in their children
  - Insertion, deletion and search are all $O(\log_b n)$
  - Caveat: APFS implementation tree are not sibling linked.

B-Tree Nodes

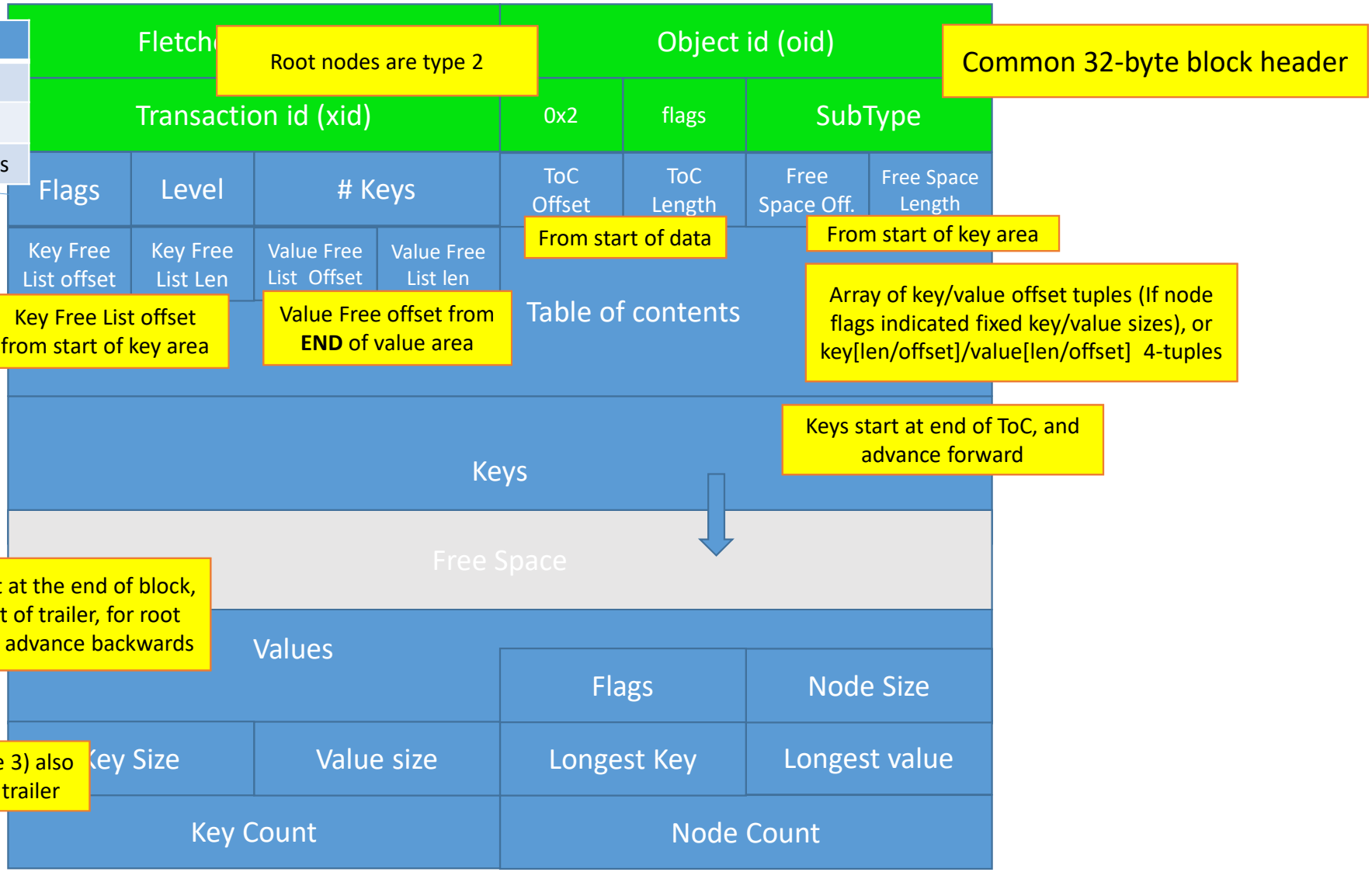
- B-Tree node format bears some similarities to that of HFS+
  - Because A) it works and B) it makes for really fast conversion
- Nodes are of block type “2” (root) or “3” (non-root) nodes
  - Contain fixed size header
  - Contain a “table of contents” (ToC) indicating keys, values and free space
  - Keys start in sequential order after ToC
  - Values start at end of block, reverse sequential order
  - Free space is in middle, fragmentation eventually managed by a free list
  - Root nodes also have a small trailer information blob

The APFS B-Tree Leaf/Middle Node



The APFS B-Tree Root Node

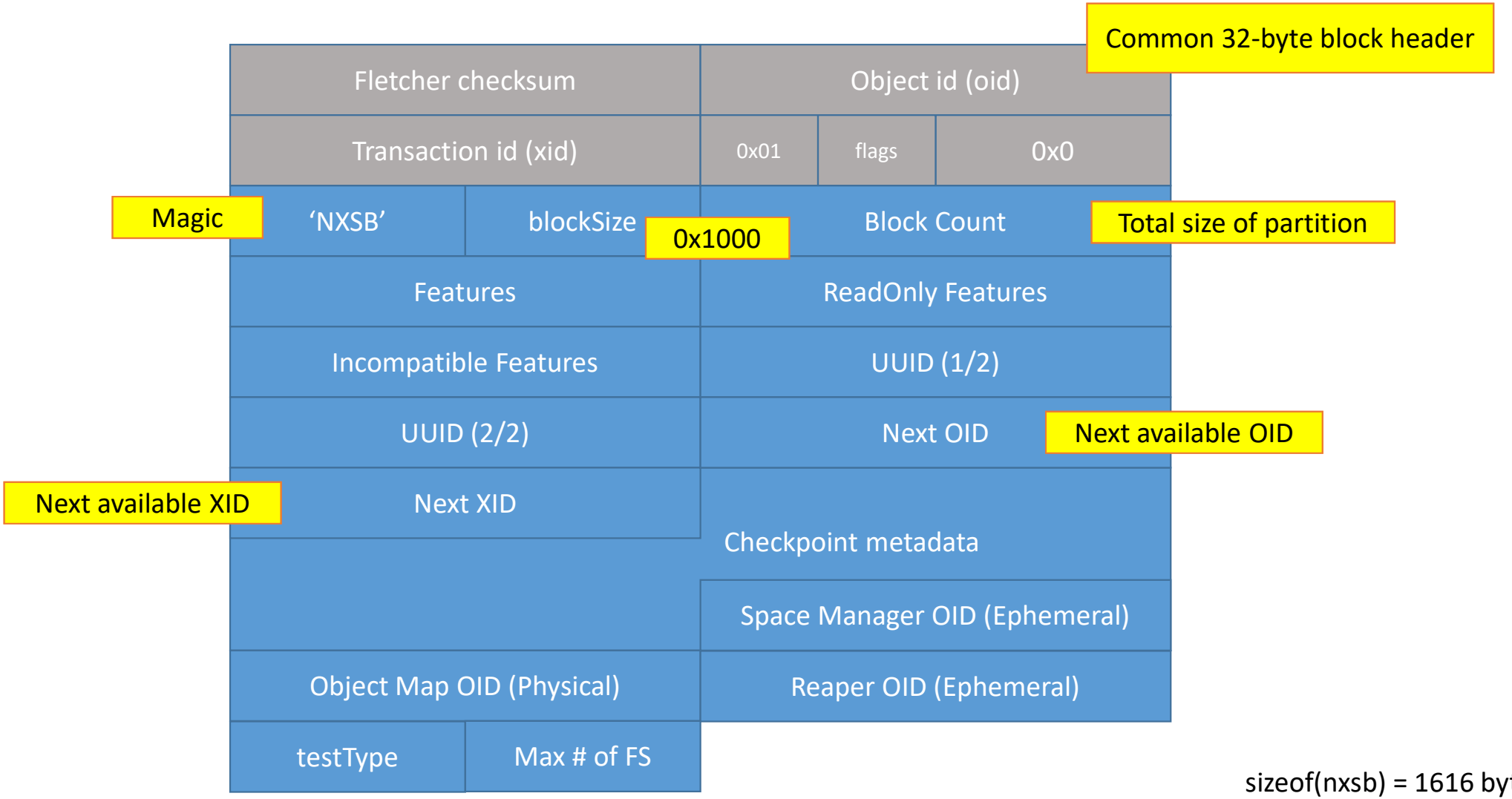
Flag	Meaning
0x1	Root Node
0x2	Leaf Node
0x4	Fixed Key/Values



APFS Containers

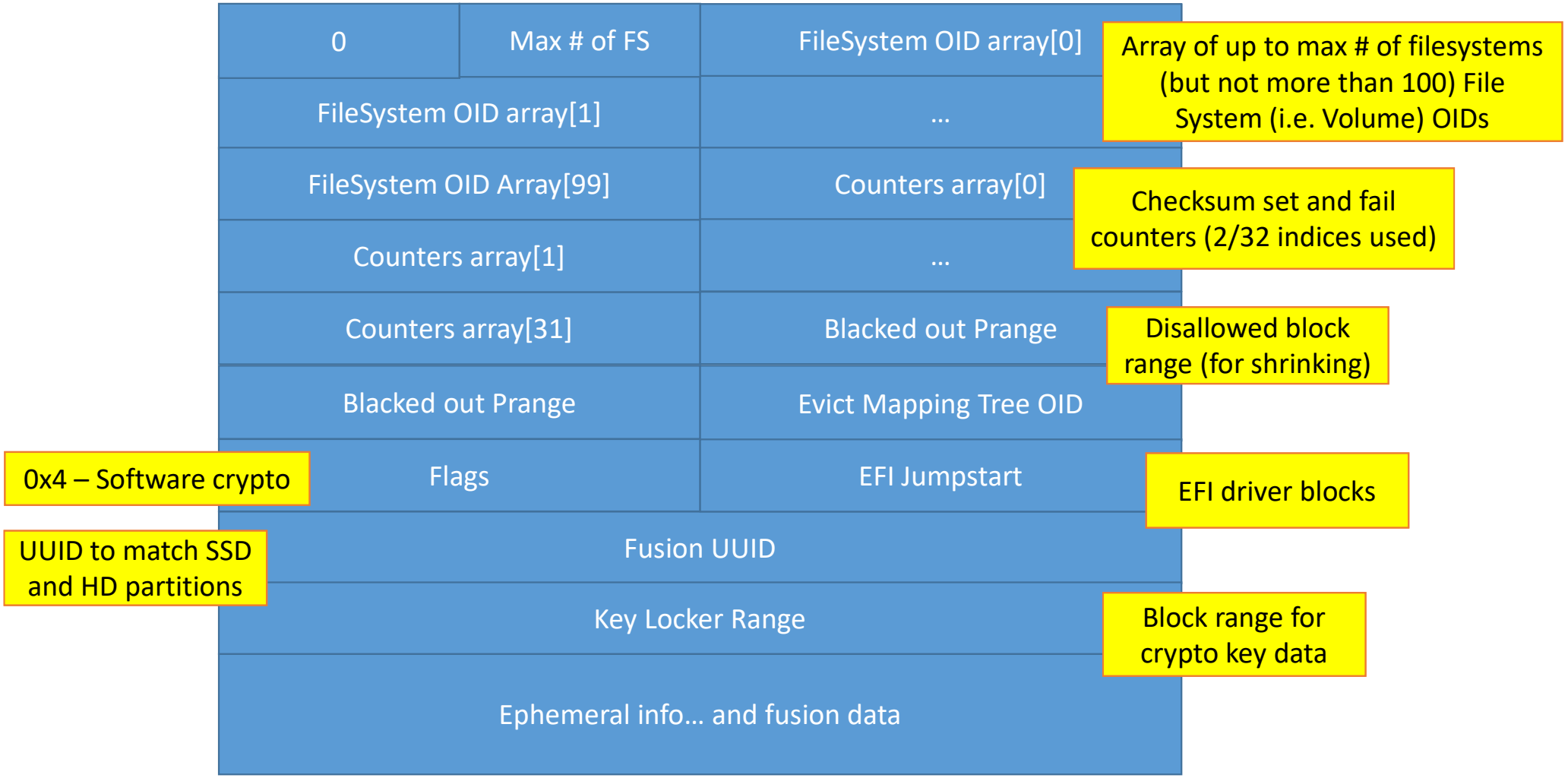
- The container (“nx”) is the top level object of the partitioned space
  - Contains one or more volumes (“apfs”)
  - Effectively acts as a logical volume manager
    - All volumes see and expand into the same free space
    - Single Space Manager (“spaceman”) handles block allocation
  - Container holds global object map

The APFS Container Superblock (NXSB)



sizeof(nxsb) = 1616 bytes

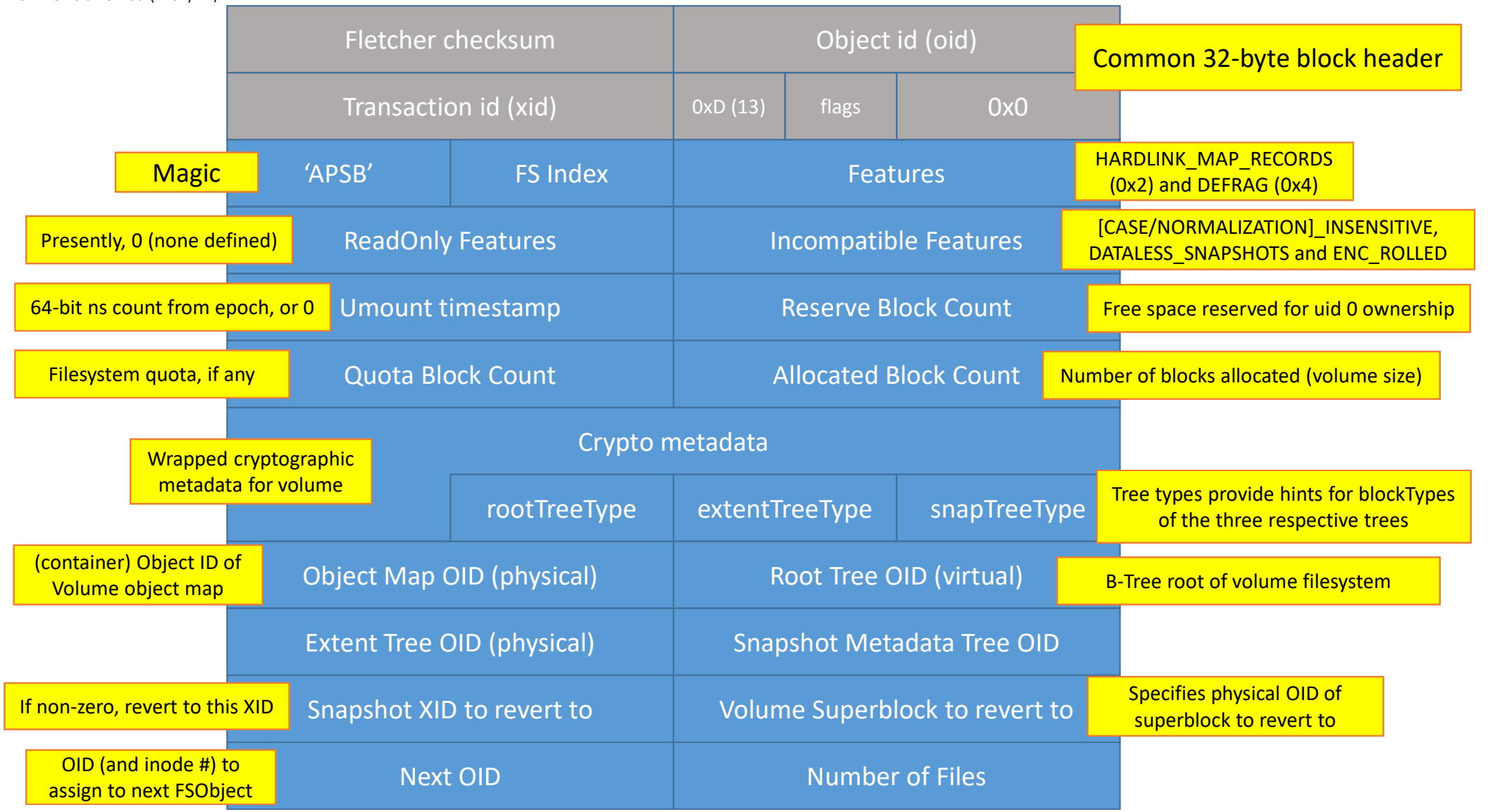
The APFS Container Superblock (NXSB)



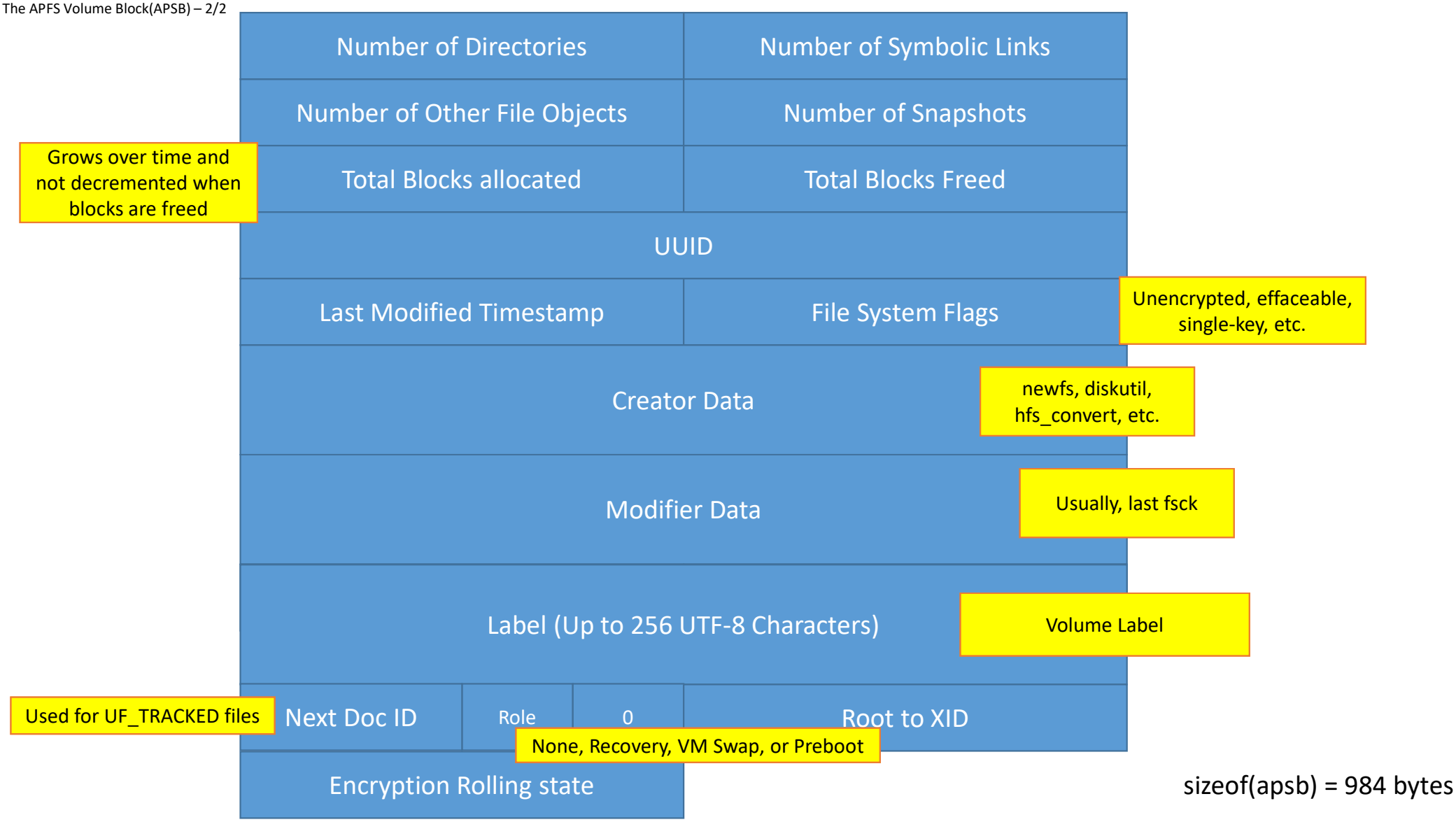
APFS Volumes

- The Volume (“apsb”) represents a mountable file system
  - Contains its own object map
  - Tied to a given xid (checkpoint)
- Changes frequently!
  - Every filesystem level change (add/remove file object, quotas, etc)
  - Deliberate snapshots

The APFS Volume Block(APSB) – 1/2



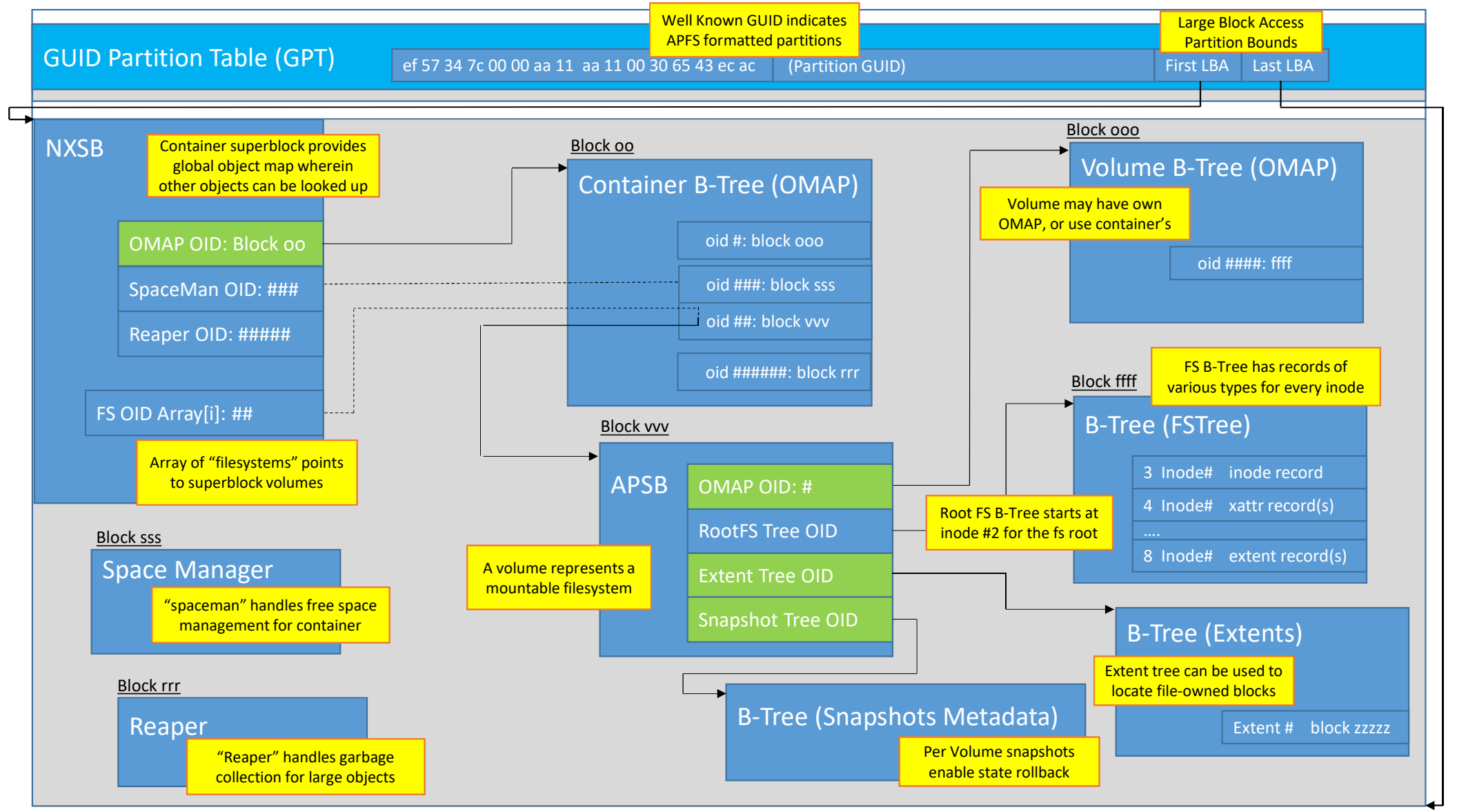
The APFS Volume Block(APSB) – 2/2



sizeof(apsb) = 984 bytes

Mounting

- Container superblock at 0x0 consulted
- Locate Checkpoint area to find other (previous) superblocks
- Find superblock with highest XID (may be the original superblock)
- Get Object Map of container
- Read fsOid array to see which volumes are in container
- Lookup fsOid in Object Map to get physical block
- Get Root Tree object ID from Volume’s OMAP
- Root file system will Be a Btree of type 2 (root) and subtype 14 (fstree)


GUID Partition Table (GPT)
Well Known GUID indicates APFS formatted partitions
ef 57 34 7c 00 00 aa 11 aa 11 00 30 65 43 ec ac
(Partition GUID)
Large Block Access Partition Bounds
First LBA
Last LBA
NXSB
Container superblock provides global object map wherein other objects can be looked up
OMAP OID: Block oo
SpaceMan OID: ###
Reaper OID: #####
FS OID Array[i]: ##
Array of "filesystems" points to superblock volumes
Block oo
Container B-Tree (OMAP)
oid #: block ooo
oid ###: block sss
oid ##: block vvv
oid ######: block rrr
Block ooo
Volume B-Tree (OMAP)
Volume may have own OMAP, or use container's
oid ####: ffff
Block vvv
APSB
OMAP OID: #
RootFS Tree OID
Extent Tree OID
Snapshot Tree OID
A volume represents a mountable filesystem
Root FS B-Tree starts at inode #2 for the fs root
Block ffff
FS B-Tree has records of various types for every inode
B-Tree (FSTree)
3 Inode# inode record
4 Inode# xattr record(s)
....
8 Inode# extent record(s)
Block sss
Space Manager
"spaceman" handles free space management for container
Block rrr
Reaper
"Reaper" handles garbage collection for large objects
B-Tree (Snapshots Metadata)
Per Volume snapshots enable state rollback
B-Tree (Extents)
Extent tree can be used to locate file-owned blocks
Extent # block zzzzz

Locating files

- Using Volume's Root Tree B-Tree (as found from Volume's omap)
- Every object is keyed by a 64 bit value:

Type(s)	Purpose
1/11	Snapshot Metadata/Name
2/8	Physical/File extent
3	Inode
4	Xattr
5/12	Sibling/Sibling Map
6	Data Stream (file contents)
9	Directory record (dentry)
10	Directory statistics

- 60 least significant bits provide inode #

Locating files

- Files, directories and symlinks MUST have Inode records
- It's not uncommon for one file system object to have multiple entries:
  - Symlinks MUST also have xattr (com.apple.fs.symlink)
  - Files usually have DSTREAMs, may have Extents and may have XATTR
    - If com.apple.decmpfs is present, it may actually hold file stream for small files
  - Directories commonly have records (their dentries), stats, and xattrs
  - Snapshots must have both metadata and name records.
- File metadata reconstructed by walking over all records with same id.
- Sibling dentries will be with same id, to which name is concatenated.
  - Dentry will reveal file/dir id, which will point to inode and any additional records.

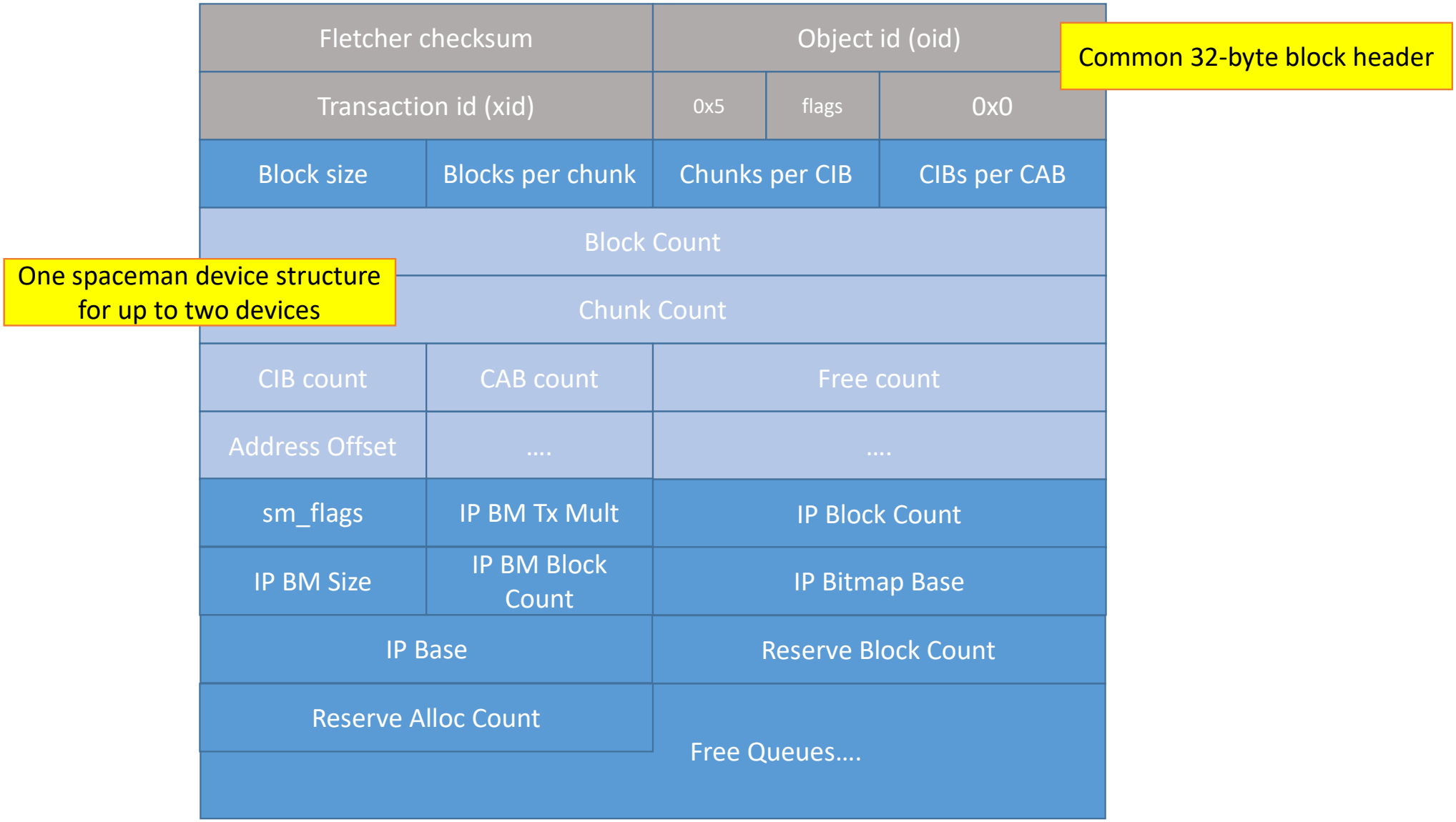
Locating files

- Inode record will appear first
- If file is compressed, it will have `com.apple.decmpfs` xattr
  - If file is small enough, content is embedded in attribute
  - Otherwise, `com.apple.ResourceFork` holds compressed content in data stream

- If file is uncompressed, it will have one or more extent (type 8) records
  - Extent record defines first block, size, and number of blocks for extent

SpaceMan

- The container uses a Space Manager (‘spaceman’) for all volumes
- POORLY DOCUMENTED in Apple’s reference
- Painful to work with..
- Space manager tracks container free space using:
  - CIB: Chunk Info Block – containing bitmaps for contiguous chunks
  - CAB: CIB Address Blocks – grouping together CIB bitmaps
  - Internal Pool (IP) Bitmap


Fletcher checksum
Object id (oid)
Common 32-byte block header
Transaction id (xid)
0x5
flags
0x0
Block size
Blocks per chunk
Chunks per CIB
CIBs per CAB
Block Count
One spaceman device structure for up to two devices
Chunk Count
CIB count
CAB count
Free count
Address Offset
....
....
sm_flags
IP BM Tx Mult
IP Block Count
IP BM Size
IP BM Block Count
IP Bitmap Base
IP Base
Reserve Block Count
Reserve Alloc Count
Free Queues....

Volume Encryption information

- Hardware (T2) encryption is entirely transparent
- Software encryption is fairly well documented (rev2 of spec)
- Open Source decryption implementation in BlackBag SleuthKit plugin

- Short version:
  - Volume is encrypted with random key (VEK)
  - VEK unwrapped with a key-encryption-key (KEK)
  - KEK unwrapped by user's password or one of several recovery keys.

Take aways

- Whether or not you like it, APFS is here to stay
  - Will stick around for longer than HFS+ did, and might outlive some of us..
- Reference doc – better late than never, but really, too little.
  - Wait for MOXiI Volume II – it fills the gaps in Apple's (incomplete) document
- APFS CAN support undelete, but Apple doesn't want outside snapshots
  - fsleuth will change that.
  - Still won't be useful outside forensics due to entitlement (or disable SIP…?)

Resources

- Apple's (finally-released-but-quite-disappointing) APFS reference:
  https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf
- Seminal APFS research of Kurt H. Hansen & Fergus Toolan
  https://www.sciencedirect.com/science/article/pii/S1742287617301408
- Jtsylve (BlackBag) – SleuthKit APFS (Open Source!)
  https://github.com/blackbagtech/sleuthkit-APFS/
- All this and further, even gorier details: *OS Internals, Volume II
  - Dedicated chapters on VFS and APFS
- Technologeeks.com – fsleuth Pro – When Darwin 19 comes out..

Questions?

有问题吗？

Habētisne Quaestiōnēs?

?שאלות

One more thing!

- As a personal thank you to Dr. Schatz, and appreciation for having me:
  - QiLin now has optimized-for-forensics build, with ZERO* filesystem impact
  - Sterile binary, no need to worry about extra code
  - No $#%#$%# Cydia to defile your /var filesystem (root left read-only)
  - Built-in server with quasi-shell environment!
  - Closed source, but modular and F-R-E-E at http://NewOSXBook.com/QiLin/
  - LiberiOS will be silently updated to include this (to avoid Reddit zealots)

  - Feature requests welcome: http://newosxbook.com/forum/viewforum.php?f=15

* - outside its own app host, that is..