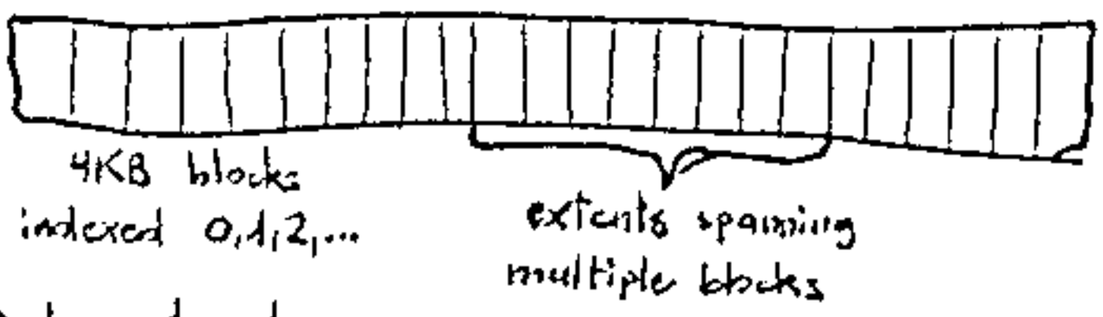


Data structures specification:

header contains:
- root node extent address

root node contains:
- dictionary of names mapped into extent addresses

file node contains:
- dictionary of content addresses mapped into extent addresses

Another data structure:

directory nodes contain:
- dictionary of names mapped into inode numbers (IDs)

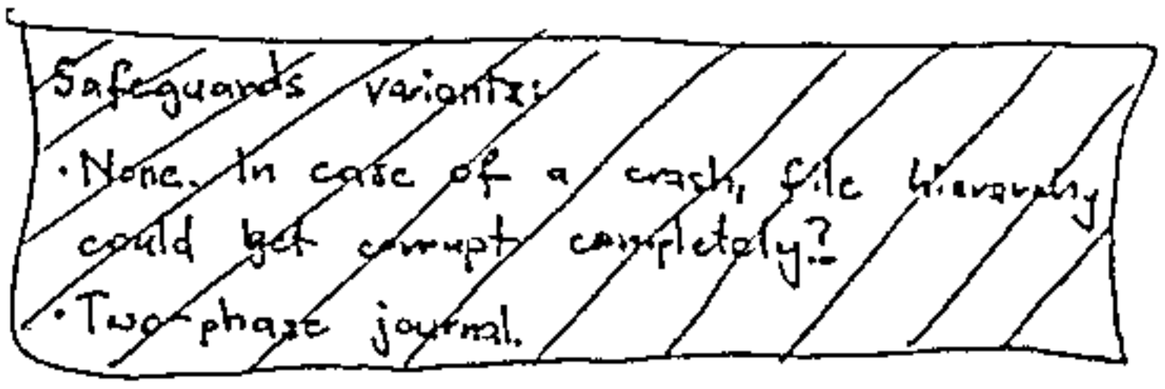
file node contains:
- unique inode number (ID)
- dictionary of content ranges mapped into inode numbers
- size (≥ content ranges total)

content node contains:
- unique inode number
- array of bytes (file contents)
- size (≤ extent size)

global table contains:
- dictionary of inode numbers mapped into extent addresses

extent defined as:
- address of beggining
- address of ending

from that, also computable:
- length



Two major strategies:

All nodes are immutable. After file is closed (all it's handles) the file node has to be stored on disk, then it's parent has to be updated. But parent directory node is also immutable, and so on up to root node. If every buffer write is mentioned in journal, file node ~~can be updated~~ can be updated in-memory and it's on-disk version can be updated with delay safely.

=either immutable or journaled=

All nodes are ~~semi-immutable~~. Buffer writes ~~are stored in~~ ~~point to~~ new extents. After file is closed, updated file node is stored in a new extent. Then parent node is ~~updated~~ overwritten with a two-phase protocol.

(Don't like it, too complicated, for two-phase p too also anyway

Estimated global table size:
- 1 million files / directories
- 20 fragments per file (~~frag~~ freg= extent)
- 2×64bit addresses per extent

total 160 million bytes
without compression or variable-length coding could be put into 80 or 40 MB easily

Journal data structure:

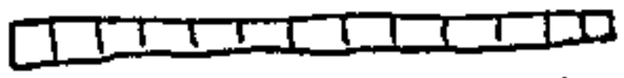
entries 0, 1, 2, ...

first entry contains:
- pointer to root node
- pointer to global table

following entries contain:
(file write case)
- file node ID
- content range (buffer length)
- & content hash (buffer checksum)
- content extent address
- { whole entry checksum, entry type (for file write) } always present (in every case)

Recovery scenario:
Read first entry, if checksum is wrong then raise critical failure.
Then read next (and next...) entry, verify its checksum. If checksum is wrong, stop, ignore rest of entries. If entry contains some data checksums, these also must be verified. If all entries were verified correctly, filesystem was fully recovered, otherwise filesystem was partially recovered (guaranteeing weak ACID).

~~Steghide~~ overview?



array of data blocks + bitmap of used blocks

Initialization procedure:

- Pick some amount of blocks (5-10% maybe), at random locations, fill them with random data, mark them as used.

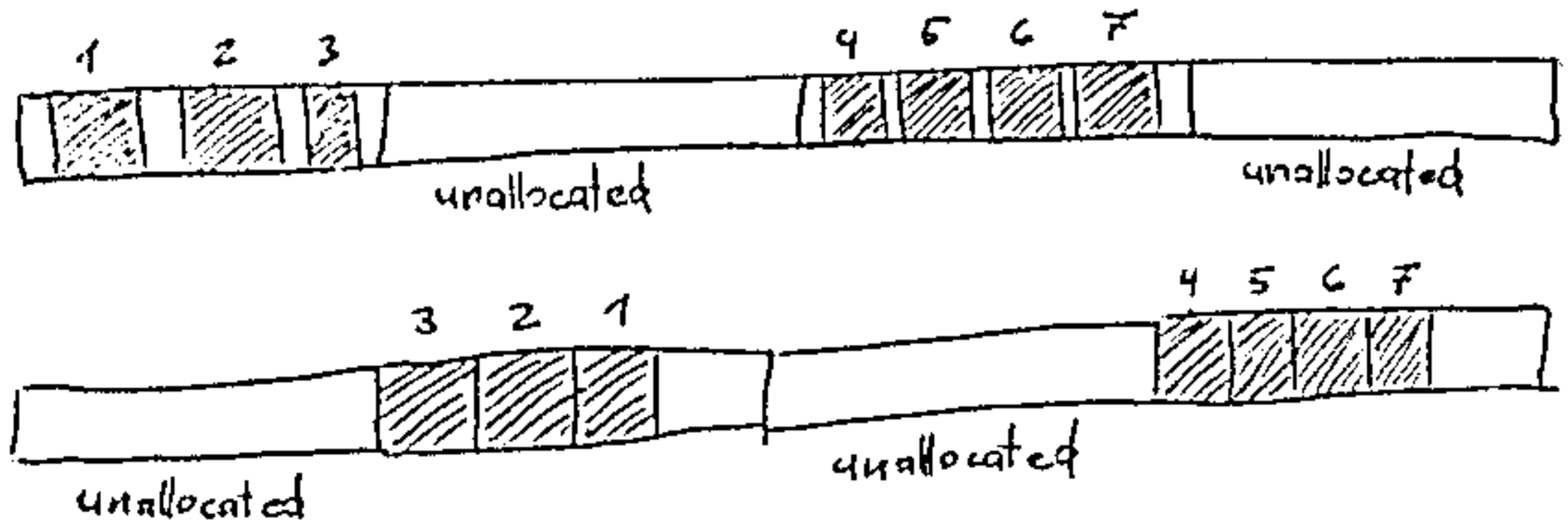
Usage scenario:

(2). Allocate new blocks from random unused locations, write encrypted data into them, mark blocks as used. Directory inodes are stored same like files.

Denial scenario:

- Unspecified (arbitrary) amount of blocks is always "used but not associated to any file". User can always claim that unassociated blocks were just wasted during initialization.

(1) · User selects multiple passwords. Every file and directory is encrypted with one of user's passwords. If a passwod remains undisclosed, then files encrypted with it remain "unassociated" or "unreachable". Yet, these files occupy blocks that are marked as used in the bitmap.

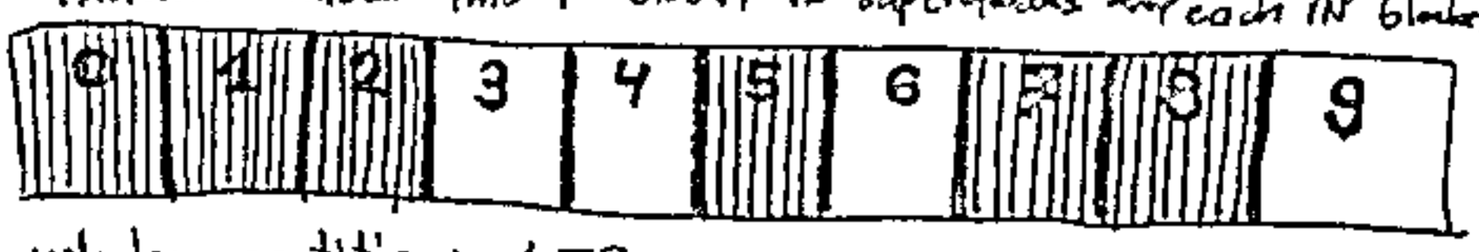


Online defragmentation strategy:

- Pick an allocated superblock worth defraging (maximizing some heuristic)
- Read the superblock as a whole
- Compact the superblock in memory (possibly reorder extents within the superblock)
- Store the superblock as a whole
- Delay deallocation of previous superblock till new superblock was synced on disk

1 TB = 1e12 bytes

1 TB = 4KB x 256 M



whole partition: 1 TB

~~256·256~~

$256 \cdot 10^6$ blocks

superblocks: $16 \cdot 10^3$ superblocks

each $16 \cdot 10^3$ blocks 4KB

superblock = 64 MB

$\frac{1s}{9ms} = 111$ seeks

1,08 MB read/written within 9 ms period (assuming 120 MB/s) throughput

read a superblock in ($\leq 1s$) then write compactly in ($< 1s$), if superblocks are highly fragmented or low utilisation (online defragmentation)

4GB file then requires at minimum 64 superblocks

every superblock can be allocated to at most one profile (password)

Alternative strategy:

Partition ~1TB ~10^{12} bytes

assuming $\sqrt{N} \cdot \sqrt{N}$ division, $\sqrt{N} \cdot \sqrt{N}$ superblocks

each superblock has 10^6 bytes, then a huge file 4GB requires 4000 superblocks (too fragmented)

FileSystem (per profile) data structures:

- UsedSegments : set of ~~ints~~ extents
- ~~InodeExtents~~ : dictionary of ints to extents
- InodeExtents
- FreeExtents : min-heap of extents
- UtilizationOfSegments : dictionary of ints to floats/ints

File preallocation strategy:

- When a file ~~is first created~~ ~~its size is increased~~ requires more space, allocate a new whole segment (16MB?) at a time.
- When file gets closed, only the last segment will not be fully utilized. At that point, it should be moved and joined with existing segments if possible.

At most one segment will need to be moved to finalize closing.

Since reading files is more frequent than writing, last segment should be immediately ~~not~~ defragmented (moved out).