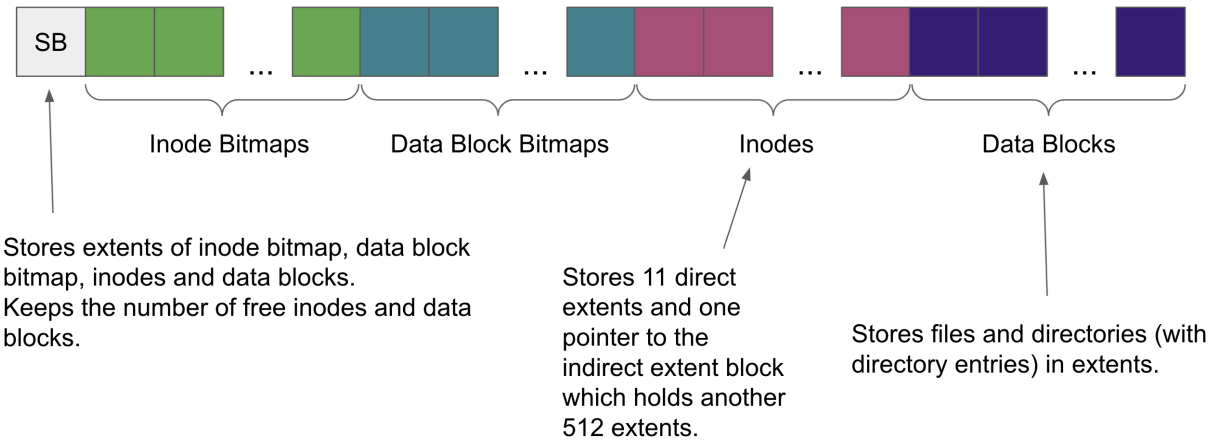
File System (OS)

Dan Lyu

June, 2021

Q1 Diagram and Metadata



Q2 Partitions

Suppose x is the total number of blocks available, the inode takes up n bytes with padding (depending on the final size), b is the block size (4096 bytes in the given header file) and i_0 is the total number of inodes given as parameter.

This means one block can hold $m = (b/n)$ inodes.

The total number of block bitmaps (in blocks): $b_m = \lceil x/8b \rceil$. This guarantees that the block bitmap blocks can hold all blocks in the FS.

The total number of blocks of inodes would be: $i = i_0/m$.

The total number of inode bitmaps (in blocks): $i_m = \lceil i/8b \rceil$.

The FS will first partition the superblock, inode bitmap and block bitmap. The indices will be stored in superblock as extents (start and count).

The FS will then allocate i blocks for inodes.

The rest of the blocks will all be data blocks.

Free inodes and data blocks will be tracked in their bitmap blocks. The extents of all bitmap blocks and inode blocks are stored in the superblock.

Q3 File Extents

The extents are stored in the inodes. One inode holds 11 direct extents as a list (88 bytes in total for 11 direct extents). The inode then holds a block index of an indirect block. Since the given block size is `4KB`, the indirect block can hold up to another `512` extents. The index of the indirect block will be `NULL` when the file has less than or equal to 11 extents.

Q4-5 Extents Allocation

The FS will first navigate to the corresponding inode of the file (Q10) and find its existing extents. When the action is done, the FS modifies the last modification timestamp in the inode and update the file size.

Method 1: The FS will seek the bitmap and find the closest free space that can hold the new extent, write the data and modify the inode to add the new extent.

Method 2: The FS will check the bitmap and if the last existing extent has free space right after it that can hold the indirect extent block. The FS will then allocate the extent there. If this is not the case, execute method 1.

Case 1: The file has no extent. The FS uses method 1.

Case 2: The file has less than 11 extents. The FS uses method 2.

Case 3: The file has 11 extents. The FS will check the bitmap and allocate a new indirect block in the closest location to hold the new extent. The FS then uses method 2 to exam the 11^th direct extent.

Case 4: The file has more than 11 extents. The FS will navigate to the indirect block and find the last extent. The FS then uses method 2.

Special Case: There doesn't exist any space for the new extent after checking the bitmap. The extent has to be broken up into multiple small extents. The FS will find the largest extents that are available and break the new extent into the least amount of extents then allocate them sequentially (from block index 0 to the last block). All extents information will be kept on the inode or its indirect extent block. If there's no enough space even after collecting all the small extents, throw an error.

The FS prioritizes file fragmentation over external fragmentation since it looks for a sequential block for the new extent (method 2). If the condition wasn't satisfied, it aims for less external fragmentation by filling in the holes. It always allocates a single extent unless there's not enough space.

Q6 Truncation and Deletion

The FS will navigate to the corresponding inode and find the last extents that need to be freed. It sets all their corresponding blocks' bits to 0 in the bitmap. The FS then modifies the last modification timestamp in the inode and update the file size.

If the file is to be removed, the FS will navigate to its containing directory's data block and compact the remaining valid entries in that extent. This fills in the removed entry.

The FS will then set all its occupying data blocks' bits and its inode bit to 0 in bitmap blocks.

Q7 Seeking to a specific byte

Find the block number (b_{num} = offset/blocksize) and the bytes offset in that block (b_{offset} = offset % blocksize). (From overview slides)

Navigate to the inode and check if the block number is in its direct extents. We add up all the blocks count in the direct extents, suppose the sum of all existing blocks is m.

> Yes: This means $b_{num} < m$. We get the extent starting index and find the block (*i.e.,* map the block number to the actual block index using extents). Navigate to the block and read the byte at b_{offset}.
>
> No: Navigate to the indirect block and find the actual block index by checking its extents. Then use b_{offset} to get the byte in the data block.

If the b_{num} is larger than the total number of blocks (all `counts` in direct extents + all `counts` in indirect extents block), return error.

Q8 Inodes

To allocate an inode: Check the bitmap and find the closest 0 bit. Navigate to the first inode block and offset the index to create a new inode in the table. Then set the bit in the bitmap to 1.

To free an inode: Set the inode bit to 0.

Q9 Directories

To allocate a directory entry: Create a new `a1fs_dentry` struct and add it to the list of entries if there's enough space. If not, create a new extent that's 1 block in size (how big should the extent be?), add its metadata to the inode of the directory and write the new entry to the first block in the extent. If the directory contains equal to or more than 11 extents, allocate a new indirect extent block or navigate to the indirect extent block and add a new extent there according to the methods mentioned in Q4.

To remove a directory entry: Shift all the directories that are right to the directory to the left to fill in the directory entry to remove. If there's no other directory entries in the whole extent, truncate the directory in the inode and mark the blocks in the removed extent to 0 in the bitmap.

Q10 Path

Navigate to the first inode, which should be the corresponding inode of the root directory. Read all its contents and find the desired directory entry (if the file's in a nested directory). The directory entry should point to its inode (`ino` in the struct). Then we read the directory entries in the extents using that inode. We repeat these steps until we reach the file in its containing directory data block. We use its directory entry metadata to navigate to the file's inode and read its data.

1. Part 1 Traces

Algo: FIFO	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
blocked-50	99.6911%	1939658	6011	5961	2646	3315
blocked-100	99.7881%	1941547	4122	4022	1700	2322
matmul-50	52.4492%	1217013	1103352	1103302	551200	552102
matmul-100	53.7949%	1248239	1072126	1072026	535751	536275
repeatloop-50	33.4507%	190	378	328	133	195
repeatloop-100	82.7465%	470	98	0	0	0
simpleloop-50	22.7206%	770	2619	2569	23	2546
simpleloop-100	24.0189%	814	2575	2475	12	2463

Algo: LRU	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
blocked-50	99.7477%	1940760	4909	4859	1993	2866
blocked-100	99.8141%	1942052	3617	3517	1304	2213
matmul-50	55.1396%	1279440	1040925	1040875	520001	520874
matmul-100	56.6329%	1314090	1006275	1006175	502761	503414
repeatloop-50	34.5070%	196	372	322	130	192
repeatloop-100	82.7465%	470	98	0	0	0
simpleloop-50	25.4352%	862	2527	2477	0	2477
simpleloop-100	25.4352%	862	2527	2427	0	2427

Algo: MRU	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
blocked-50	13.1380%	255622	1690047	1689997	816578	873419
blocked-100	21.6730%	421684	1523985	1523885	735889	787996
matmul-50	15.9283%	369595	1950770	1950720	965842	984878
matmul-100	22.3449%	518483	1801882	1801782	892158	909624
repeatloop-50	49.2958%	280	288	238	91	147
repeatloop-100	82.7465%	470	98	0	0	0
simpleloop-50	1.4163%	48	3341	3291	205	3086
simpleloop-100	1.9475%	66	3323	3223	201	3022

Algo: CLOCK	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
blocked-50	99.7491%	1940787	4882	4832	2009	2823
blocked-100	99.8015%	1941806	3863	3763	1523	2240
matmul-50	55.1394%	1279436	1040929	1040879	520004	520875
matmul-100	55.1721%	1280194	1040171	1040071	519673	520398
repeatloop-50	34.3310%	195	373	323	130	193
repeatloop-100	82.7465%	470	98	0	0	0
simpleloop-50	25.3762%	860	2529	2479	0	2479
simpleloop-100	25.3172%	858	2531	2431	1	2430

Algo: RAND	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
blocked-50	99.6067%	1938017	7652	7602	3496	4106
blocked-100	99.7451%	1940709	4960	4860	2140	2720
matmul-50	58.5619%	1358850	961515	961465	480469	480996
matmul-100	86.6919%	2011569	308796	308696	154098	154598
repeatloop-50	54.4014%	309	259	209	78	131
repeatloop-100	82.7465%	470	98	0	0	0
simpleloop-50	22.6025%	766	2623	2573	24	2549
simpleloop-100	24.1074%	817	2572	2472	12	2460

2. Analysis

blocked:

blocked is more memory aware since it keeps accessing all the frequently accesses addresses on line 194, which is the innermost loop of the entire program. This makes the memory to be focused on a relatively small region. The overall hit rate is thus high with LRU, FIFO and CLOCK. This is because:

1. Few other operations happen during the execution of the loop.
2. Most frequently accessed memory are being accessed in the innermost loop, causing them to stay in the memory until the entire loop finishes.

This tricks LRU, FIFO and CLOCK to keep the frequently accessed pages in the memory.
MRU has the worst performance because it keeps paging out the frequently used pages.
The hit rates are similar for different memory size using LRU, FIFO and CLOCK since the loop region is small and frequently accesses pages are preserved.

matmul:

matmul has more instructions in the outer loop, which might interrupt the focused accesses on instruction/data pages in the innermost loop, causing worse hit rates and way more evictions with all algorithms.
Both matmul and blocked are focused on a small region, which leads to worse performance using MRU.

repeat loop:

It scans the array in multiple repetitions and every scan must be done before the next scan can happen. This causes way more interruptions with a memory size of 50 since the memory isn't large enough to hold all elements and we will be constantly swapping out/in in every repetition and require evictions. We still have most (not all) parts of the scan in the memory since they hold the highest number of unique pages count. As such, the hit rate isn't really low.
However, with a larger memory size of 100, the memory is large enough to consume all the scan data/instruction pages and repetitions won't interrupt our scan. 100 seems to be enough for all the memory accesses required for the repetitive scan and we have an eviction count of 0 using LRU, MRU, CLOCK, RAND and FIFO.
MRU has a higher size-50 hit rate, which is probably because we are swapping out the polluted elements (array elements) constantly and keeping other parts intact which other algorithms could choose pages of non-polluted elements as victims.

simple loop:

We are scanning a size 2 array in a single run.
This causes all pages to be constantly brought in and we don't use these data pages after we are done with it. This leads to a low hit rate in LRU, FIFO and CLOCK.

MRU has even lower hit rate since we need to access the container array during the execution of every element write but some instruction/data patterns might have prevented the address of the outer array or its access instructions from being kept in the memory for our very next access (that happens right after it). We seem to be constantly paging in and out on line 18 in `simpleloop.c`.

3. Hand-created Traces

`Trace 1: 1 2 3 4 5 6 2 7 8 2 9 10 3 4 3 5 11 12 7 6 13 10 6 1 8 14 15 9 13 16 13 17 8`

Trace 1	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
lru-8	21.2121%	7	26	18	0	18
fifo-8	42.4242%	14	19	11	0	11
clock-8	33.3333%	11	22	14	0	14
optimal-8	45.4545%	15	18	10	?	?

`Trace 2: 1 2 3 4 2 4 5 6 3 7 8 6 5 6 9 10 7 6 11 12 5 6 10 12 13 10 7 14 12 15 8 9 13`

Trace 2	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
lru-8	48.4848%	16	17	9	0	9
fifo-8	54.5455%	18	15	7	0	7
clock-8	48.4848%	16	17	9	0	9
optimal-8	54.5455%	18	15	7	?	?

`Trace 3: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16`

Trace 3	Hit rate	Hit count	Miss count	Overall eviction count	Clean eviction count	Dirty eviction count
lru-8	0.0000%	0	48	40	0	40
fifo-8	0.0000%	0	48	40	0	40
clock-8	0.0000%	0	48	40	0	40
optimal-8	33.3333%	16	32	24	?	?

`Trace 3 Optimal:`

```
[0] Access: 1
['1', -1, -1, -1, -1, -1, -1, -1]
[1] Access: 2
['1', '2', -1, -1, -1, -1, -1, -1]
[2] Access: 3
['1', '2', '3', -1, -1, -1, -1, -1]
[3] Access: 4
['1', '2', '3', '4', -1, -1, -1, -1]
[4] Access: 5
['1', '2', '3', '4', '5', -1, -1, -1]
[5] Access: 6
```

```
['1', '2', '3', '4', '5', '6', -1, -1]
[6] Access: 7
['1', '2', '3', '4', '5', '6', '7', -1]
[7] Access: 8
['1', '2', '3', '4', '5', '6', '7', '8']
[8] Access: 9
['1', '2', '3', '4', '5', '6', '7', '9']
[9] Access: 10
['1', '2', '3', '4', '5', '6', '7', '10']
[10] Access: 11
['1', '2', '3', '4', '5', '6', '7', '11']
[11] Access: 12
['1', '2', '3', '4', '5', '6', '7', '12']
[12] Access: 13
['1', '2', '3', '4', '5', '6', '7', '13']
[13] Access: 14
['1', '2', '3', '4', '5', '6', '7', '14']
[14] Access: 15
['1', '2', '3', '4', '5', '6', '7', '15']
[15] Access: 16
['1', '2', '3', '4', '5', '6', '7', '16']
[16] Access: 1
['1', '2', '3', '4', '5', '6', '7', '16']
[17] Access: 2
['1', '2', '3', '4', '5', '6', '7', '16']
[18] Access: 3
['1', '2', '3', '4', '5', '6', '7', '16']
[19] Access: 4
['1', '2', '3', '4', '5', '6', '7', '16']
[20] Access: 5
['1', '2', '3', '4', '5', '6', '7', '16']
[21] Access: 6
['1', '2', '3', '4', '5', '6', '7', '16']
[22] Access: 7
['1', '2', '3', '4', '5', '6', '7', '16']
[23] Access: 8
['1', '2', '3', '4', '5', '6', '8', '16']
[24] Access: 9
['1', '2', '3', '4', '5', '6', '9', '16']
[25] Access: 10
['1', '2', '3', '4', '5', '6', '10', '16']
[26] Access: 11
['1', '2', '3', '4', '5', '6', '11', '16']
[27] Access: 12
['1', '2', '3', '4', '5', '6', '12', '16']
```

```
[28] Access: 13
['1', '2', '3', '4', '5', '6', '13', '16']
[29] Access: 14
['1', '2', '3', '4', '5', '6', '14', '16']
[30] Access: 15
['1', '2', '3', '4', '5', '6', '15', '16']
[31] Access: 16
['1', '2', '3', '4', '5', '6', '15', '16']
[32] Access: 1
['1', '2', '3', '4', '5', '6', '15', '16']
[33] Access: 2
['1', '2', '3', '4', '5', '6', '15', '16']
[34] Access: 3
['1', '2', '3', '4', '5', '6', '15', '16']
[35] Access: 4
['1', '2', '3', '4', '5', '6', '15', '16']
[36] Access: 5
['1', '2', '3', '4', '5', '6', '15', '16']
[37] Access: 6
['1', '2', '3', '4', '5', '6', '15', '16']
[38] Access: 7
['7', '2', '3', '4', '5', '6', '15', '16']
[39] Access: 8
['8', '2', '3', '4', '5', '6', '15', '16']
[40] Access: 9
['9', '2', '3', '4', '5', '6', '15', '16']
[41] Access: 10
['10', '2', '3', '4', '5', '6', '15', '16']
[42] Access: 11
['11', '2', '3', '4', '5', '6', '15', '16']
[43] Access: 12
['12', '2', '3', '4', '5', '6', '15', '16']
[44] Access: 13
['13', '2', '3', '4', '5', '6', '15', '16']
[45] Access: 14
['14', '2', '3', '4', '5', '6', '15', '16']
[46] Access: 15
['14', '2', '3', '4', '5', '6', '15', '16']
[47] Access: 16
['14', '2', '3', '4', '5', '6', '15', '16']
```