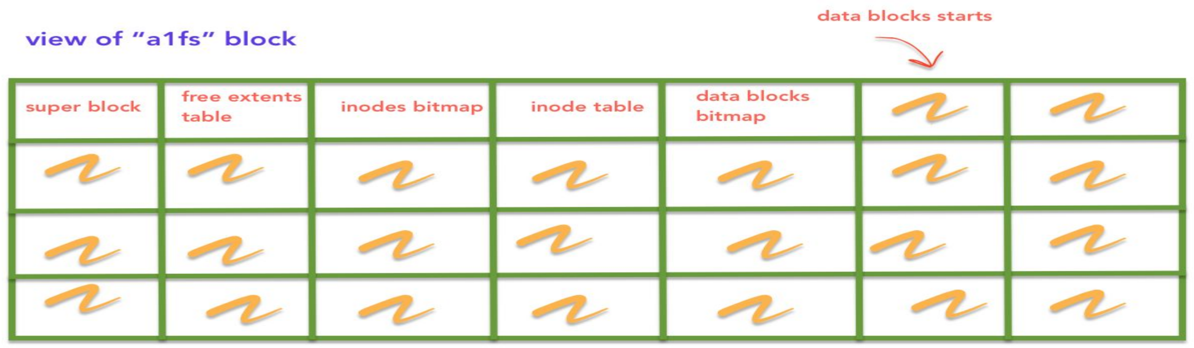

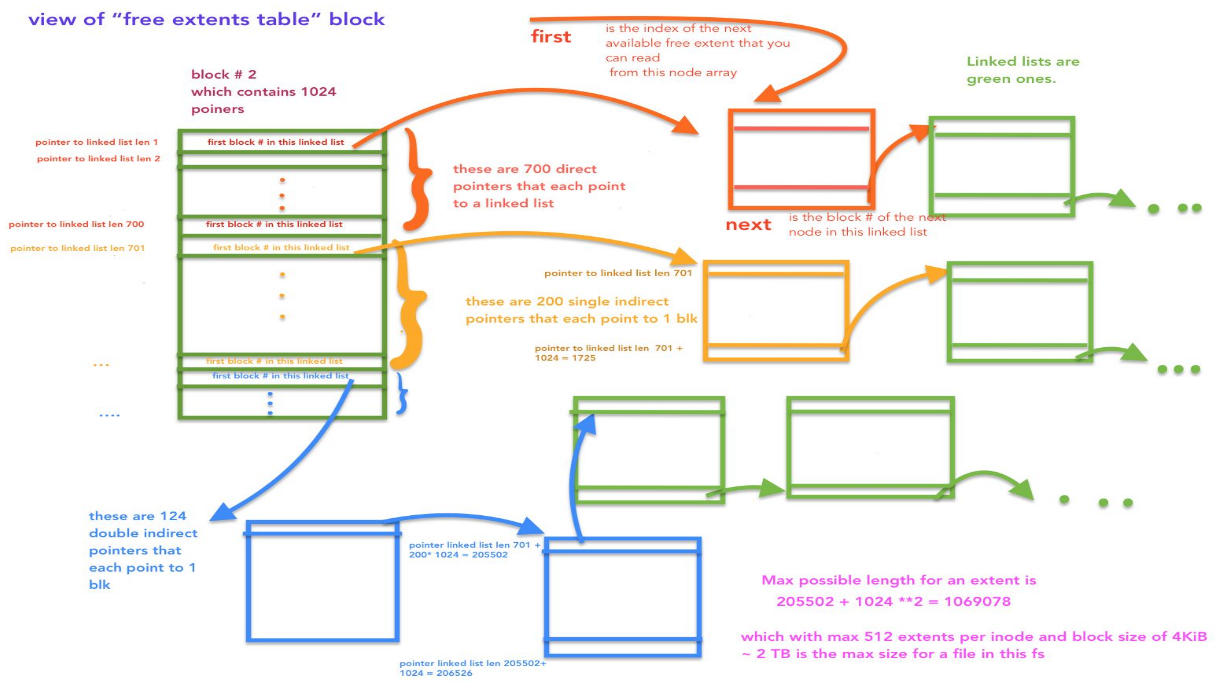
view of "free extents table" block
first
is the index of the next available free extent that you can read from this node array
Linked lists are green ones.
block # 2 which contains 1024 poiners
pointer to linked list len 1
pointer to linked list len 2
first block # in this linked list
these are 700 direct pointers that each point to a linked list
pointer to linked list len 700
first block # in this linked list
pointer to linked list len 701
first block # in this linked list
next
is the block # of the next node in this linked list
pointer to linked list len 701
these are 200 single indirect pointers that each point to 1 blk
pointer to linked list len 701 + 1024 = 1725
first block # in this linked list
first block # in this linked list
...
....
these are 124 double indirect pointers that each point to 1 blk
pointer linked list len 701 + 200* 1024 = 205502
pointer linked list len 205502+ 1024 = 206526
Max possible length for an extent is 205502 + 1024 **2 = 1069078
which with max 512 extents per inode and block size of 4KiB ~ 2 TB is the max size for a file in this fs

Let the input parameters to mkfs be: size of the fs >>> **FILE_SYS_SIZE** and number of inodes >>> **INODES_COUNT.** Then a1fs's layout will be as:

- keeping track of free data blocks: We will assign block #2 to be the **free extents table** which is an array whose index **i** will represent a linked list which represents the starting blocks of extents with length **i+1**. In each of these nodes, we let the first pointer(called **first**) to be the index of the next available free extent in this node and the last pointer(called **next**) to be a pointer to the next node in this linked list. To store longer pointers,out of 1024, 700 pointers go directly to a linked list, next 200 go to single indirect, then 124 go to double indirect blocks.
- Each inode struct contains: file mode, size of the file in bytes, modification time, links count, extents count, an **array with len 16 of pointers to struct extent** (i_extents[16]) to keep track of extents for the inode. Each of the pointers in this array point to an array of 32 struct extents(saved in the data blocks). Therefore, we can store 512 extents and have an inode of size of 128 bytes with padding.

- block #1 will be the super block with attributes as in the .h file
- Inode table and data block bitmap will be at the blocks after block 3, depending on the FILE_SYS_SIZE and INODES_COUNT.
- When the disk is formatted with the a1fs, only inodes are preallocated
- # of data blocks = FILE_SYS_SIZE - (sizeof(superblock) + sizeof(free extents table) + sizeof(inodes bitmap) + sizeof(inodes table) + sizeof(data bitmap)). If the inode count is greater than the # of data blocks, then we raise an error.

3) How will you store the information about the extents that belong to a file?
As mentioned above, each inode has an **array with len 16 of pointers to struct extent** that keep track of extents that make up a file.
4) How to allocate disk blocks to a file when it is extended?
When writing to a file, we calculate the amount of bytes needed. We try to find a free extent that has the same amount of blocks we need. Assume we want to write x blocks.

1) Go through the <u>free extents table</u> and look for an extent of size x. If we can find one, we use it for the file, and write the contents to it.
2) Else If there does not exist a free extent with len x available, in the <u>free extents table,</u> we check to see if we can find a <u>smallest free extent that is larger than x</u>. Then we can still write the whole file to one extent.
3) Else we try to allocate blocks starting at the largest block terminating at the end of the disk. We know the pointer to this extent from the super block called s_largest_extent. If s_largest_extent > x, use the blocks needed and then update the s_largest_extent in the super block.
4) Else if s_largest_extent < x, then look through the free extent table to find multiple extents that all add up to x. If you can't then raise a disk full error.

5) Explain how your allocation algorithm will help keep fragmentation low.
We will store all of our free extents as an array of linked lists which we have called the **free extents table**. The free extents table's indexes represents the length of the extents (as described above), and stores a pointer to either a linked list, or an indirect block that stores a linked list. To allocate blocks to files, we will try to find the one free extent which has the length of larger or equal to the one we need for our file. That way we write the blocks of the file sequentially as <u>one extent if possible</u>, and this allows us to use the free extents in the **free extents table**. Also once we truncate/delete a file, we will add the truncated/deleted blocks as an extent to the free extent block for the respective size(also we will merge free extents if possible before putting them in the free extents table). In case there are no available free extents with the size that we require for that file/directory, we write to the beginning of the largest extent terminating at the end of the disk. This sequential writes decreases fragmentation.
6) Describe how to free disk blocks when a file is truncated.
When a file is being truncated, we will need to delete/deallocate some/all extents of the file. To do that, we first need to apply that change in availability of extents in the <u>free</u>

extents table. We determine which blocks are being freed, then search through the data block bitmap to see if there is a free extent that comes immediately before or after the extents we are freeing. If we find one, we remove the found free extents from its associated linked list in the free extents table and merge it with the blocks we are deleting. Once we have determined our new free extent, we go to the appropriate index of free extents table that corresponds to the length of this extent, and add it to the beginning of its associated linked list. Once that is done, in the data bitmap, we set the bits corresponding to the blocks we just deleted to 0.

7) Describe the algorithm to seek a specific byte in a file.

Let the byte that we want to seek to be **b**. We know there is an array of pointers to struct extent in an inode. Each pointer points to an array of 32 struct extents and each of the extents has the property of starting point **x** and length **len**. First we need to determine which one these extents contain the byte b of the file. Let **i** = $ceiling(\, b \,/\, 4096 \,)$.
Go over the extents of the file one by one in order, and let **i = i - len** in each iteration. Stop that loop when **i < the len of an extent**. That extent is the goal extent that contains byte b. Let the goal extend be **E = (x, len)** . Then seek to **x**, and read **i** bytes.

8) Describe how to allocate and free inodes.

We use the same bitmap inodes for keeping track of the inodes, so the process will be similar to the simple file system. i.e: To create a new file/directory we need to allocate a new inode by going to inodes bitmap and look for the first bit that is 0, that bit number is the block number of where the new inode should be located (let that be block **blk**) and make it 1 in the bitmap. Then we go to the inode table (by finding out its block from the super block) and seek to block **blk** and assign the inode in that block for our new file/directory. To free an inode, we unset the bit index of it in inodes bitmap. In both cases, we should update inodes count in super block as well.

9) Describe how to allocate and free directory entries within the data block(s).

Allocating and freeing the directories is very similar to the one of the files. We allocate extents for a directory like in **question 4**. To free a directory, we will recurse until our base case is met: a directory with no sub-directories, only files. From there, we will delete all files, then return. Before the function returns on higher levels of the recursion, all sub-directories and files will be deleted. The process of deleting(and freeing) each dentry extent will be like the one of the file(in **question 6**).

10) Describe how to lookup a file given a full path to it, starting from the root.

Let the full path to look up be “/dir/file”. In the inode table, go to inode at index 0 which is the inode for the root directory(“/“) and find what block the extents of “/“ are located. Search through the extents in the extents array of “/“ one by one and look for an entry of the “dir” and find out its inode. Go to the inode of “dir” in the inodes table and find the location of its extents. Search through the extents in the extent array of “dir“ one by one and look for an entry for “file” and find out its inode.Go to the inode of “file” in inodes table and find the location of data extents of it. At this point we know where the extents of the file “file” are and so the ”file” is ready to be read or written to. Note that we can read the file consecutively by seeking a specific byte of it(in **question 7**).