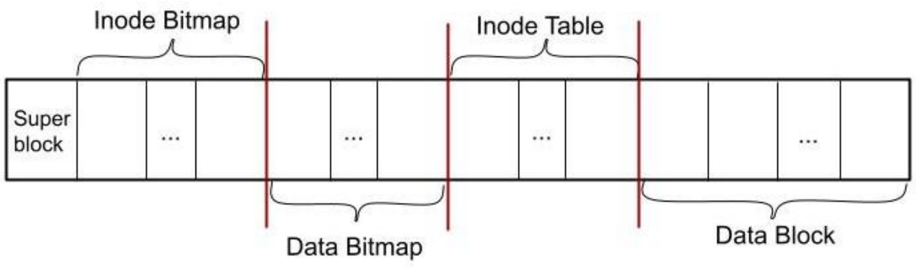
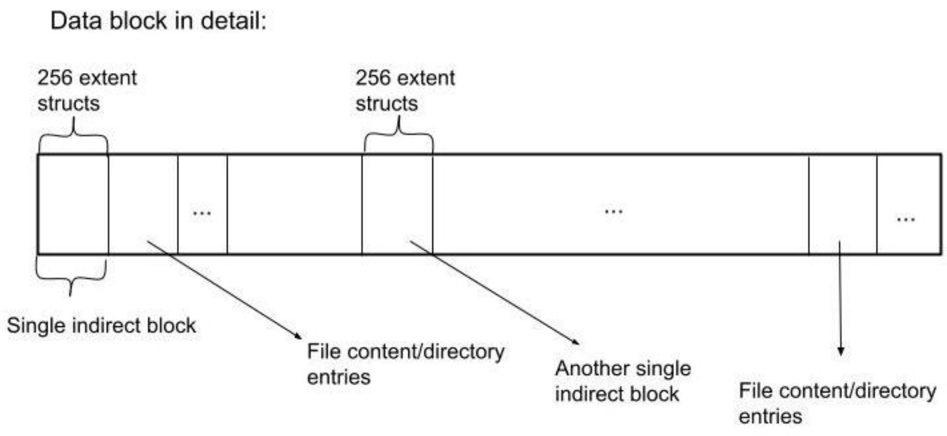
May 25, 2021 **CSC369 A1FS Proposal** Ruoming Ren
Steven Yuan

1. The layout of our disk image: (see the header file for more information)




Data block in detail:
256 extent structs
256 extent structs
...
Single indirect block
File content/directory entries
Another single indirect block
File content/directory entries

Note that single indirect blocks and actual file/directory contents scatter across the data block.

2. A description of how you are partitioning space.

Solution: The beginning of the disk contains a superblock, an inode bitmap that tracks free inodes, a data bitmap that tracks free data blocks, an inode table that contains metadata of files, and finally the data block that consists of the actual contents of files, directory entries, and single indirect blocks that consist of extent structs. Each inode stores 2 pointers to single indirect blocks.

3. A description of how you will store the information about the extents that belong to a file.

Solution: The information about the extents are stored in inodes and data blocks - each inode has 2 pointers each of which points to a data block that consists of 256 extent structs. Note that two blocks contain 512 extent structs which are 16 bytes each as defined in the a1fs header file. In other words, **there are at most 2 indirect blocks for each file.**

4. Describe how to allocate disk blocks to a file when it is extended.

Solution: To reduce file fragmentation, we will use the following algorithm:

1) Find the last extent of a file(since user could only add things at the end of the file), check if there is enough space to allocate blocks immediately before or after it (*reason: extending an extent can make the file look less spread out*);

2) If we cannot find such a position, try to find the first spot in the data block that has enough space for the whole extent. If not, try to find the first spot in the data block that has enough space for extent with its length - 1; if the spot is found, we split the extent to 2 extents(the first extent's length is equal to original extent's length - 1, the second extent consists the rest of blocks in the original extent), and allocate the appropriate blocks for the first extent. Then, do the step 2 recursively to the second extent(*reason: a single extent should be as larger as possible*);

3) If there is no such spot that we are seeking in step 2, we try to seek the spot that has enough space for the extent with length - 2 and repeat the rest of steps in step 2. If we still cannot find the spot, continually reduce the length by 1 until a desired position is found. If there is no place even for a extent with length equal to 1 block, the file system is full(However, we shouldn't be able to reach this situation, since before extending the file, we should know from the superblock if the file system is large enough to accommodate the operation);

4) When creating a new file, allocate its blocks at the first position that has enough space; if it cannot be done, we repeat the strategy above (*reason: allocate a single extent if possible to avoid having different chunks of a file all over the place*)

5. Describe how to free disk blocks when a file is truncated (reduced in size) or deleted.

   Solution: If an entire block is truncated, we flip the corresponding bit in the block bitmap to 0; if only parts of the block are truncated, we reduce the number of meaningful bytes(a field in a1fs_extent) in the corresponding block and leave the bitmap as it is.
   If a file is deleted, we flip the corresponding bit in the inode bitmap to 0 and flip all the corresponding bits in the data bitmap to 0.
   Finally, we reduce the size of the file by subtracting the number of bytes truncated/deleted from the size field in the corresponding inode.

6. Describe the algorithm to seek to a specific byte in a file.

   Solution: Through the pointer in the inode described in question 3, we can find the corresponding array of extent structs in the data block. After dividing the offset by the block size, we get the quotient which is the block number and the remainder which is the offset within the block. Then, we iterate through the structs: at each iteration, we add the count field in each extent struct to a variable *sum*, and compare the sum to the quotient - if *sum* is less than or equal to the quotient, we subtract *sum* from the quotient and store the difference in a variable *diff*; if at any point the sum is greater than the quotient, we can deduce that the extent struct in the previous iteration is what we are looking for - so we break out of the loop. The variable *diff* can tell us which block from the start of the extent we should be focusing on. And finally, we find the corresponding block in the data block and add the remainder to find the byte of interest.

7. Describe how to allocate and free inodes.

   Solution: A new inode will be allocated at the first empty spot in the inode table, which can be found by traversing the inode bitmap; then, we flip the corresponding inode bit to 1.
   We free an inode by fliping its corresponding bit in the inode bitmap to 0.

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

   Solution: Similar to file allocation, when a directory entry is created, we add its entry struct to some data block pointed at by an extent struct in an indirect block and increment num_dir_entry. When a directory entry is removed, we replace the directory entry with the last directory entry in the data block and decrement num_dir_entry.

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

   Solution: Suppose we want to lookup /dir/f1, where dir is a directory and f1 is a file. First, we read superblock to get the location of the root inode. From the root inode, we can track its data block to find the directory entry that has the name dir and get the inode number for that entry. Then, we go to the inode table to find the inode of dir. Again we can get the entry block of dir that corresponds to f1. After the inode number of f1 is obtained, we once again head back to the inode table to find the inode of f1. Finally, we get the metadata of f1 in its inode.