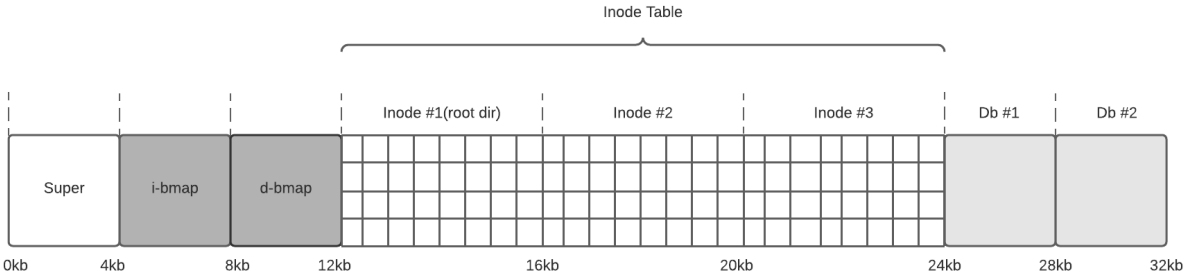
File System Proposal

Diagram:



Explanation of Partition: As we can see with the diagram, the method for partitioning is very simple and directly inspired from the “Simple File System” that we looked at in class, this is because for our purposes the gained performance does not make up for the complexity in our implementation if we are to go a file system more modern file systems that use concepts of groups and cylinders. The number of inodes per block for now is 32 but this may change if I make any changes to the a1fs.h file down the line. The following gives a brief overview of the significance for each block:

- Superblock
    - Keeps track of information like the location of inode table, number of inodes, etc.
    - Important fields which tell the OS information about the specific file system in order to mount it
- i-bitmap and d-bitmap
    - A bitmap which tells us which inode and data block is and isn’t in use
    - Important so that we don’t overwrite files and can allocate space for a file
- Inode table
    - This area of our file system is made up of inodes
    - The size of this table is dictated by parameters provided by the user who will specify the size of the disk and the number of inodes which is not reflected in the diagram
- Data blocks
    - Store actual file data
    - Could be a indirect block which then points to more data blocks
    - Could be of the type dir_entry

Inode Structure: The key thing about the structure of the inode I will be implementing my file system with is that it will use an extent based approach to store the data. The advantage to this method is that it improves efficiency because data is allocated in contiguous blocks so we don't have to constantly have to seek across the disk. In terms of where and how the extent is stored. We know that an extent is stored in the form (int, int) which is 4 + 4 = 8 bytes. We are given that they're can be a maximum of 512 extents for each inode. 8 * 512 = 4096 which is exactly the size of 1 block. So keep a pointer to a block which stores the extents(called an indirect block). On top of this I have decided to store 10 direct extent pointers inside the inode for space efficiency as an additional block should not be used unless needed as it may cause internal fragmentation. The actual data structure used is a nested list and will be in the form [data block, length].

Extend File: when the file is extended(a write syscall is called by the user) the size of the write is passed on, so we know exactly how many more blocks we will need to allocate. This is where the bitmap comes into play. We find a contiguous array of the desired length of block. In particular we are always trying to limit file fragmentation before external fragmentation, so if we have a choice between two contiguous blocks to use for our extent we choose the one that limits file fragmentation. Here are the rough steps that our algorithm will take:

1. Identify the length of the extent(denote this as n)
2. Seek to the last data block of the file
3. If there are n blocks following the last block we can write to the blocks and make the appropriate changes to the structures(best option to reduce file fragmentation)
    a. If this is not the case we try to reduce external fragmentation so we try to find n contiguous blocks in the range(beginning data block, last data block)
    b. Otherwise there is no way to avoid fragmentation and that is fine, we find any available extent

Truncate: We simply flip the bits not in use back to 0 in the data bitmap. This will indicate that the blocks can now be used by another inode. We will also have to modify the inode and data blocks in order to reflect the changes(eg. Reduce inode size attribute or remove the file dir_entry from the parent directory).

Seek: We need a specific algorithm to seek some part of a file. We can apply the fact that we cannot delete from the middle of the file. This makes this operation easier to perform. The following describe our algorithm will take:

1. Calculate the number of blocks into the file that byte is
2. Loop over the extents until we hit an extent in which contains the block we are looking for
3. Calculate the location of the data block we are looking for based off the extent
4. Follow the location of the data block and index it using bitwise operators to get the byte we are looking for

Create and Unlink File: The main ideas for these two operations are how to **allocate** an inode and how to **deallocate** an inode. For allocation we must first find an unused inode using our inode bitmap and flip the bit in the bitmap to 1 to indicate that it is now being used. For deallocation, we find the bit in the bitmap which represents the inode we want to de-allocate and flip it to 0. The rest of these operations just involve updating all the relevant structure's fields like removing the file from the dir_entry of the parent.

Search/Lookup: This operation is very important and used by many other operations. We need to be able to search our file systems for a specific file or directory. We do this by starting at the root inode representing the root of the file system. This information is stored in the superblock. We then have a field inside the inode structure telling us the type of the inode(directory, file, symbolic link) and we use this information to keep traversing the path until we have found the specific file we are looking for. It is worthy to explain how traversing a directory works. The inode of a directory has an indirect block which has a list of **dir_entry data blocks**. We use this information to traverse the path.