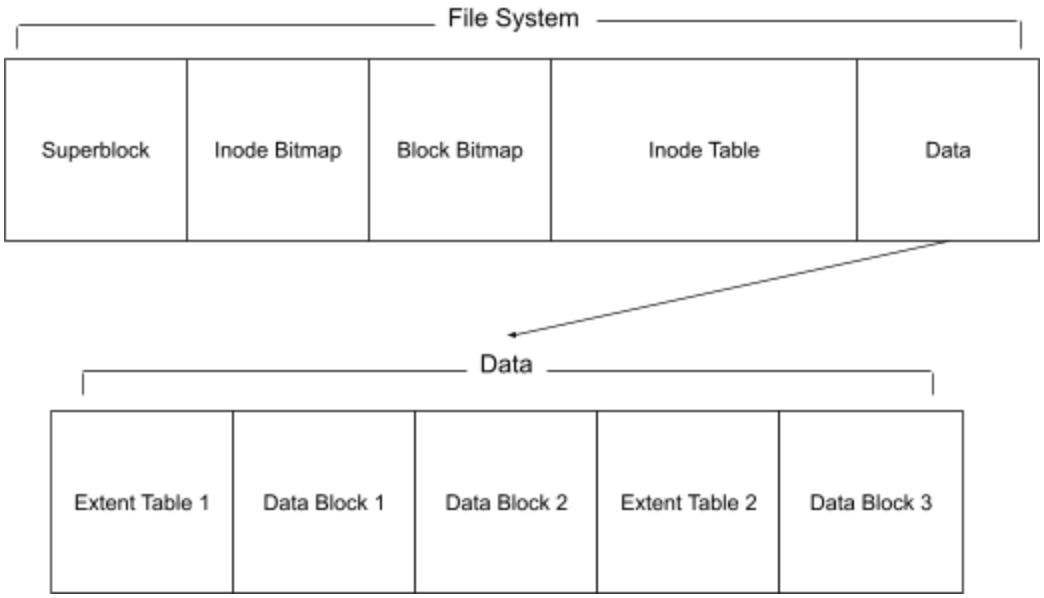
CSC369 Assignment a1a: File System Proposal



How are we partitioning space?:

We will partition space in the filesystem using bitmaps. The inode bitmap will keep track of inodes that are in-use/free within the inode table. The block bitmap will keep track of 4096 byte blocks that are in-use/available within the entire file system. In the data section, we will be able to store extent tables, which are dedicated 4K blocks containing extents of a single inode, or data blocks, which contain file or directory information. The size of the block bitmap tells us the size of the entire disk image. Similarly, the size of the inode bitmap, gives us an indication of how many inodes may exist.

How are we storing information about extents that belong to a file? Where is it stored?:

When a new directory or non-empty file is created, we assign an additional block before the data, called the extent block (which is set to all 0s before being written to), this stores up to 512 extents (or start, length pairs corresponding to block data) for a given file/directory. The inode of each non-empty file points to its own dedicated extent block. Both the extent block and its allocated data blocks, will show as occupied (value of 1) within the block bitmap.

How are we allocating disk blocks to a file when it is extended?

1) Find the file's inode, to get the location of its extent block.
2) Calculate how many blocks are needed for extension (stored in a buffer as variable n)
3) Find the last extent in the extent block, and one block after its data, call it *last*.
4) Set $x = n$ (the number of blocks we are trying to write)
5) Given x and block bitmap range $[b_{start}, b_{end}]$. Search between $[last, b_{end}]$ finding the smallest index i pointing to x consecutively free blocks. If index i exists, write x blocks

starting at i. Otherwise, search again from $[b_{start}, last - 1]$ picking the largest index i pointing to x consecutively free blocks. If index i exists, write x blocks starting at i - x.

6) On a successful write operation, set $n = n - x$. and if $i == last$ then it is an adjacent write, so *modify* the last extent in the table, else, append a new extent to the extent table.
7) If $n == 0$, all blocks are written and we are done.
   Otherwise if, $x == 1$ and index i is invalid, it means there is no more space in the file system, so return an error.
   Otherwise, set $x = max(1, x - 1)$ and recursively **repeat** steps 5-7.

How does the allocation algorithm keep fragmentation low?:

Our allocation algorithm enforces a dynamic programming strategy, finding the largest section of consecutively free blocks near the last extent. By prioritizing contiguous block allocation, we decrease fragmentation in our file system. In the best case, n blocks starting from $last$ happens to be free and we modify the last extent instead of adding a new one. In the worst case, there are no n consecutive free blocks in the entire system, then we look for $n - 1$ consecutive blocks or fewer until we have the largest consecutive blocks j. We then add one extent for these j blocks, and recursively find n-j consecutive blocks to allocate.

How are disk blocks freed when a file is truncated (reduced in size) or deleted?:

Truncate:

1) Find the inode of this file and its parent directory, name it *parent_inode*
2) Buffer the delete commands, so each time we know how many bytes we need to delete, or don't buffer and delete one byte at a time, which might be inefficient.
3) Find the old length of this file, and find a new length. Calculate how many blocks might be free again
4) Edit the extent table. We know how many blocks to reduce, so from the end we iterate each extent, reduce its length and possibly remove the extent, and free the corresponding blocks, at last we set the freed blocks to 0. Then set all bits in the extent entry to 0.
5) Edit the inode of the file to update metadata

Delete a file and free inodes as entry of directory:

1) Find the inode of this directory
2) Find the parent directory of this file, name it *parent_inode*
3) Iterate over the extent table of the file, for each extent, free the corresponding blocks in the block bitmap, reset the blocks, and remove the extent
4) When the file's extent table is empty, free the block on bitmap, and remove the table
5) In *parent_inode*, find its extent table, and iterate over the extents to find the *dentry* struct with same name as the file, taking note of its inode number
6) Remove the inode from inode bitmap and free the inode from inode table
7) Fill the *dentry* struct with dummy values (i.e. set inode to -1) to unattach it from the parent.

Delete a directory and free inodes as entry of directory:

1) Find the inode of the directory and its parent directory, name it *parent_inode*
2) Iterate over the extent table of the directory, and extract its *dentry* structs.

3) If no *dentry* structs exist, proceed to remove the directory by following steps 4-7 above. Otherwise, throw an error saying that we cannot remove a non-empty directory.

How do we seek a specific byte in a file?:

1) Lookup the file, find its inode
2) Calculate the block number and byte number of this byte in the file.
3) Iterate the extents, subtract their length from the block number, until we find the exact extent of the target.
4) Find the beginning of the extent, then find the exact block of the target.
5) Find the exact byte of the target in this block

How do we allocate inodes?

1) Find the parent directory of this directory/file, name its inode *parent_inode*
2) Iterate the inode bitmap from the beginning to find the first free one.
3) Create a new directory entry for the inode that we need (name, type, inode number)
4) Add this entry to the first unused *dentry* struct (*dentry* with dummy value i.e. inode == -1) in *parent_inode*'s data block. If there is no more space for writing, we need to first allocate more space to the parent using our file extension algorithm, before attempting another write.
5) Create a new inode about this directory, and write it to the inode table.
6) Set the inode bitmap for this entry to 1

How do we free inodes?:

1) Find the inode and its parent directory of this file, name it *parent_inode*
3) Iterate over the extent table of the file, for each extent, free the corresponding blocks in the block bitmap, then remove the extent.
4) When the extent table is empty, free the block on the bitmap, and remove the table
5) In *parent_inode*, iterate over its extents to find the struct with same name as the file, taking note of its inode number
6) Remove the inode from the inode bitmap, then free the inode from the inode table
7) Set dummy variables in the *dentry* of the inode being removed. (i.e. inode = -1)

How do we lookup a file given a full path to it, starting from the root directory?

1) Turn the file path into a list of "/" separated strings called *pathlist* (/foo/bar is ["foo", "bar"])
2) Set *inode* to the root inode (first inode in the inode table), and $i = 0$.
3) If $i == length(pathlist)$, then print the contents of *inode*'s extents.

   Otherwise if *inode* is a directory (by checking of metadata) loop through the extents of *inode*, reading its *dentry* structs.

   a) If a *dentry* struct is valid (has positive inode number) and its name is equal to *pathlist[i]*, set *inode* to the inode of the *dentry* struct, update $i = i + 1$ then exit the loop and **repeat** step 3.
   b) In case the loop completes and no directory entries match *pathlist[i]*, then the directory attempting to be read does not exist, so return an error message.

   Else, we are trying to change directory into a file, so return an error message.