A1a: File System Proposal

Author: Haoyan Wang

Student Id: 1004742638

UTORid: wangh278

Data Structures

```
/** a1fs superblock. */
typedef struct a1fs_superblock {
    /** Must match A1FS_MAGIC. */
    uint64_t magic;
    /** File system size in bytes. */
    uint64_t size;
    /** Number of data blocks. */
    uint32_t s_num_blocks;
    /** Number of inodes. */
    uint32_t s_num_inodes;
    /** Number of data bitmaps in contiguous blocks. */
    uint16_t s_num_data_bitmaps;
    /** Number of inode bitmaps in contiguous blocks. */
    uint16_t s_num_inode_bitmaps;
    /** Block number of the first inode bitmap. */
    a1fs_blk_t s_inode_bitmap;
    /** Block number of the first data bitmap. */
    a1fs_blk_t s_data_bitmap;
    /** Block number of the first data block. */
    a1fs_blk_t s_first_block;
    /** Block number of the root inode. */
    a1fs_blk_t s_root;
    /** Number of reserved blocks. */
    uint32_t s_num_reserved_blocks;
    /** Number of free inodes. */
    uint32_t s_num_free_inodes;
    /** Number of free data blocks. */
    uint32_t s_num_free_blocks;
} a1fs_superblock;
```

```
typedef struct a1fs_inode {
    /** File mode. */
    mode_t mode;
    /** Reference count (number of hard links). */
    uint32_t links;
    /** File size in bytes. */
    uint64_t size;
    /** Last modification timestamp.*/
    struct timespec mtime;
    /** Total number of extents. */
    uint32_t i_used_extents;
    /** Block pointer to an array of extents. */
    a1fs_blk_t i_ptr_extent;
    char extra[24];
} a1fs_inode;
```

Partitioning Disk Image

A1FS_BLOCK_SIZE	sizeof(a1fs_superblock)	sizeof(a1fs_inode)
4096	4096	64

Demonstration of Disk Layout

The space of a raw, unformatted disk image is allocated in the granularity of blocks of 4096 bytes. Note that we refer to the size of this image as S.

- Superblock

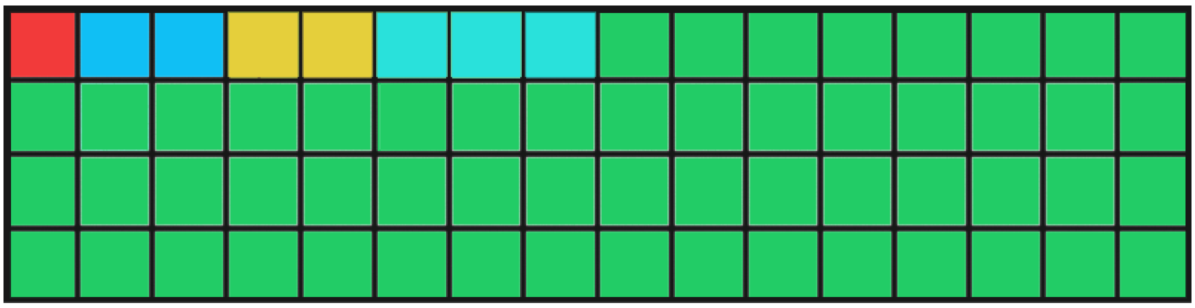
  The first block of the formatted disk is the superblock. It will store all of the data specified above.

- Inode bitmap and Inode table

  Given the desired number of inodes, INO, in this file system, we can compute the number of blocks to store the inode bitmap, $IB = \lceil INO/32,000 \rceil$, where the $32,000$ is the number of inodes one bitmap can represent; And the number of blocks to store the inode table, $IT = \lceil \frac{INO \times 64K}{4096K} \rceil$, where the $64K$ is the size of `a1fs_inode`.

- Data blocks and data bitmap

  Given the previous data, we can compute the number of available data blocks left, $N = \lfloor \frac{S}{4K} \rfloor - IB - IT - 1$. Next, we can compute the number of blocks to store data bitmaps. Number of data bitmap, $DB = \lceil N/32,000 \rceil$. The rest of the blocks are all available to store data, which there are $N - DB$. The following figure demonstrates the layout.



Superblock 🟥 **Inode bitmap** 🟦 **Inode table** 🟦 **Data bitmap** 🟨 **Data blocks** 🟩

Other Specifications

Location of Extents

Extents are stored separately from the inode. A block pointer `i_extent` in the inode points to where the extents are stored, and `i_used_extents` is the number of used extents. In this way, each inode refers to exactly $4096/sizeof(\text{a1fs\_inode}) = 512$ extents, as desired.

Allocate Space when Extending a File

1) The file system will access the superblock to check for space availability. If there is no enough free space, the system will throw an error. 2) If there is enough space, the system will firstly go over all of the extents of this file seeking available space right after the extent. In this way, the file could be extended in-place in a contiguous manner. 3) If it is impossible to extend in-place, the file system performs the *window-sliding* algorithm on the data bitmap to find the largest contiguous free slot. In this way, new extents will be created 4) If plus the new extents, the file excesses 512 extents, the system throws an error. 5) Otherwise, new extents are stored, the data blocks are written to, and the corresponding bits in the data bitmap are marked as used.

- How this allocation algorithm keeps fragmentation low?

  This algorithm prioritizes large, contiguous free blocks at the beginning of the image, and it does not break the file up to extents if not necessary. The file will take up the disk image sequentially and not scattered to many places. As a result, external fragmentation is avoided.

Free Disk Space

1) Starting from the last extent of this file, the file system will either i) free the entire extent, or ii) update the *end* of this extent to fit the truncated size. 2) The file system will mark the freed bits in the data bitmap as unused.

Seek a Specific Byte

1) The file system will access the inode of this file to obtain the file size, the number of extents, and a pointer to the block that contains extents. 2) If the byte is out of range of the file's size, an error is generated. 3) The file system cast `a1fs_extent *` to the block, and in a while-loop, it will go through all of the extents, and update the cumulative number of bytes from extents. 4) If the specific byte is in the range of the current extent, in a for-loop, the file system will go over all of the blocks in this extent to find the one that contains that byte. 6) The system will then cast a `char *` pointer to the block, and compute the offset from the start of the block to the byte, and access the byte by pointer arithmetic.

Allocate and Free Inodes

- Allocate inodes

  1) The file system will access the superblock to check for availability; If there is no space for a new inode, an error will be generated. 2) Otherwise, the file system traverses the inode bitmap to find a free slot in the corresponding inode table, and write to both of the bitmap and the table to create the new inode. 3) decrement the `s_num_free_inode` in the superblock. 4) update the `mtime` in inode. 5) Then, the system adds a new `a1fs_dentry` with the inode number and a null-terminated filename to the parent directory.

- Free inodes

  1) The file system will mark the corresponding bit in the bitmap as unused, 2) increment the `s_num_free_inode` in the superblock, and 3) free all (in bitmap) the extents associated with this inode, zero its reference count and set `ino` in directory entry to -1.

Allocate and free directory entries within the data block(s) that represent the directory

- Allocate directory entries

  The file system will firstly cast `a1fs_dentry *` to the block pointer. Then, in a while-loop, the system traverses the block to find a free space for the new directory entry. The criterion of free space is that the `ino` member is -1. Next, the file system will allocate a new inode to the new entry, and then store the null-terminated directory's name and the corresponding inode to the space.

- Free directory

  The file system will firstly cast `a1fs_dentry *` to the block pointer. Then, in a while-loop, the system traverses the directory entry and performs string comparison to find the desired directory entry. When the directory entry is found, the file system will free its inode as described in the previous section, null the filename, and set the `ino` in the struct `a1fs_dentry` to -1. If the directory is non-empty, an error will be generated.

File Lookup from Root Node

1) The file system access the inode of the root node, which is known to the superblock. 2) The extents of the root node are traversed, and for each block in every extent, the system will cast `a1fs_dentry *` to it, and then search for the desired child entries by comparing the file name. 3) After finding the desired child node, the system read the inode of the file, and if it is a directory, the system recursively traces down until it finds the file and returns the inode, or it returns `NULL`. For each recursive step, the system obtains the file's name by splitting the path at the first forward-slash /.