Operating System I - Project I - Simple File System

17302010002 黄元敏

Design

1. To implement `Filesystem::debug`, you will need to load the file system data structures and report the **superblock** and **inodes**.
    - How will you read the superblock?
        - The superblock contains four numbers, the magic number, the number of blocks, the number of inode blocks and the number of inodes. They can be read in using the `Data` block type in the defined union `Block` and then be interpreted easily using the `Super` block type.
    - How will you traverse all the inodes?
        - From `Super.InodeBlocks`, we can get the number of inode blocks. Also, we know that from the second block, the inode blocks begin. Hence, we only need to have a loop which traverses from the `second` block to the `InodeBlocks + 1`th block.
        - Then, in each inode block, we know that there are `INODES_PER_BLOCK = 128` inodes, so here is the second loop which goes from the first inode to the 128th one within one inode block .
        - With the two loops above, we can go through every inode in the given disk.
    - How will you determine all the information related to an inode?
        - Within each inode, the first thing we need to determine is the validity, which we can achieve from `Inode.Valid`.
        - Then, the size of a inode is defined in `Inode.Size`.
    - How will you determine all the blocks related to an inode?
        - The direct blocks can be found using a loop built according to `POINTERS_PER_INODE = 5`. We just need to concatenate all the non-zero block numbers written in each direct array element.
        - Whether the inode contains an indirect block can be told by whether `Inode.Indirect` equals zero. Therefore, if `Inode.Indirect` is a non-zero number, it points to an indirect block. If so, we can read in the block using `Disk::read` and then go through the `POINTERS_PER_INODE = 1024` block pointers in the block to find non-zero data block pointers.
2. To implement `FileSystem::format`, you will need to write the superblock and clear the remaining blocks in the file system.
    - What pre-condition must be true before this operation can succeed?
        - `Disk::mounted` must be false, otherwise, do nothing and return false.
    - What information must be written into the superblock?
        - According to the definition of superblock, we should construct a union `Block` and give it a `MagicNumber = MAGIC_NUMBER`, a `Blocks = disk->size()`, a

`InodeBlocks` which is rounded from 10% of `Blocks` and `Inodes = InodeBlocks * INODES_PER_BLOCK`.
        - One thing to mention is that before we set the fields in `SuperBLock`, we should first use `memset` to set all the bits in the block to 0 because the four fields in `SuperBlock` **only occupy 16 bytes instead of 4096**.
    - How would you clear all the remaining blocks?
        - To utilize the `Disk::write` function, we can use a loop from the second block to the last one to clear each of the remaining blocks one by one. For each block, we can write a block filled with `0`, created using `memset` function.

3. To implement `FileSystem::mount`, you will need to prepare a filesystem for use by reading the superblock and allocating the free block bitmap.
    - What pre-condition must be true before this operation can succeed?
        - The disk hasn't been mounted.
    - What sanity checks must you perform?
        - The `MagicNumber` should be right.
        - The `InodeBlocks` should equal `Ceil(0.1 * Blocks)`.
        - The `Inodes` should equal `InodeBlocks * INODES_PER_BLOCK`.
    - How will you record that you mounted a disk?
        - Call `Disk::mount` which will update the `Disk::Mounts` attribute.
    - How will you determine which blocks are free?
        - I will build a bit map using `std::vector`, where contains `Blocks` of int values, each one of which represents whether a block is free or occupied. In my design, there are two const int values defined in `fs.h`: `FREE = 1`, `OCCUPIED = 0`.
        - The super block and inode blocks are `OCCUPIED`.
        - Go through the inode blocks, compute how many data blocks needs to be occupied to store the `inode.Size` of data. Then, go through the direct and indirect (if needed) blocks and mark `Ceil(inode.Size / disk->BLOCK_SIZE)` blocks to be `OCCUPIED`.
        - **If there is a indirect block, it should also be marked as `OCCUPIED`.**
        - The work is done by `initialize_free_blocks()`.
4. To implement `FileSystem::create`, you will need to locate a free inode and save a new inode into the inode table.
    - How will you locate a free inode?
        - This is a relatively basic operation which only calls for a double loop to go through every inode block of a disk then every inode of an inode block. The program doesn't pursue performance, so we can simply do a linear scan and record the first invalid inode we find which would be the free inode we are looking for.
        - If no empty inode was found, -1 should be returned.
    - What information would you see in a new inode?
        - The new inode should hava `Valid = 1`, `Size = 0`, `Direct[0 ~ (POINTERS_PER_INODE - 1)] = 0` and `Indirect = 0`.
    - How will you record this new inode?
        - I implemented `FileSystem::save_inode` as instructed. The function should compute the block index and inode index according to the `inumber` passed in.
        - Then, it should ~~read in the corresponding block and~~ modify the selected inode.

> In improved version, it doesn't need to read the block again, instead, the block will be passed in as a parameter.

- Finally, it should write the whole block back onto the disk.

5. To implement `FileSystem::remove`, you will need to locate the inode and then free its associated blocks.
    - How will you determine if the specified inode is valid?
        - Firstly, I implemented `FileSystem::load_inode` as suggested, which is quite similar to `FileSystem::save_inode` except for the value assignment to `Inode *` and the return value `node->Valid`.
        - By utilizing the `load_inode` function, I can load the indicated inode gracefully with `inumber`, of which the validity is determined by its `Valid` value. When we want to remove an inode, the `Valid` should surely be 1 instead of 0. According to the definition of `load_inode`, the validity can be determined according to the return value of the function.
    - How will you free the direct blocks?
        - Just go through the `Direct` array using a loop and mark all corresponding element in vector `bitMap` as `FREE`. The actual value of `Direct` array doesn't need to be reset because it would be done when `create` is called. A lazy strategy can be used here to improve performance.
    - How will you free the indirect blocks?
        - This work is just like freeing direct blocks except that we should make sure that the inode does have indirect blocks by examining `inode.Indirect != 0`.
    - How will you update the inode table?
        - The only value needs to be changed is `inode.Valid` due to the lazy strategy. After setting `Valid = 0`, just call `save_inode` to write the inode back to disk.
6. To implement `FileSystem::stat`, you will need to locate the inode and return its size.
    - How will you determine if the specified inode is valid?
        - Just make sure that `inode.Valid == 1`, otherwise, return -1.
        - Again, `load_inode` is a useful helper.
    - How will you determine the inode's size?
        - The size of an inode can simply be retrieved by reading `inode.Size`.
7. To implement `FileSystem::read`, you will need to locate the inode and copy data from appropriate blocks to the user-specified data buffer.
    - How will you determine if the specified inode is valid?
        - Again, use `load_inode` and make sure that `inode.Valid == 1`.
    - How will you determine which block to read from?
        - For me, the data blocks related to an inode can be seen sequentially, although they may come from direct or indirect pointers.
        - From the `offset`, we can tell how many blocks are completely skipped (`skipBlocks`) and how many bytes still need to be skipped (`remainder`) in the next block to come. When looping over the sequence of data blocks, the first few of blocks will be skipped, and the skip count will be updated, until it comes to the first valid block to be read (when skip count is decreased from `offset / disk->BLOCK_SIZE` to `0`).
        - From the `length`, we can know how many bytes of data we need to read in total. In my implementation, I will loop over the sequence of data blocks related to the

inode. Each time I read a data block, I will read `min(disk->BLOCK_SIZE, rlength)` bytes of data, where `rlength` is the remaining length to read, in case that too many bytes of data is read than needed. After reading, I will update the remaining length to read by `rlength -= bytes_read` and turn to scan next data block until `rlength == 0`.

- How will you handle the offset?
    - The offset mainly gives me two pieces of information.
        1. It determines the number of blocks to skip and the remainder bytes to skip in the first block to be read as mentioned above.
        2. `min(length, inode.Size - offset)` gives me the actual bytes needed to be read to avoid reading too many bytes which may be out of the scope of the indicated inode.
- How will you copy from a block to the data buffer?
    - For a specific block's data copy, I will utilize `memcpy`. From `offset` and `rlength`, I can know how many bytes of data to copy and which byte to start as described above.
    - The only thing left to be determined is where should the data be written to in `char *data`. So I keep a variable `size` to record how many bytes of data I have already copied. Then, the destination offset would be `size`.
    - After finishing copy, `size` can be used to indicate how many bytes of data were copied.

8. To implement `FileSystem::write`, you will need to locate the inode and copy data the user-specified data buffer to data blocks in the file system.
    - How will you determine if the specified inode is valid?
        - Again, use `load_inode` and make sure that `inode.Valid == 1`.
    - How will you determine which block to write to?
        - This is similar to `FileSystem::read`, use the relationship between `offset` and `disk->BLOCK_SIZE` so that we can tell how many blocks and another bytes to skip.
        - Then, we can loop over the sequence of blocks (direct first, then indirect) until find the first block to write in.
        - The write loop should continue until all `length` of bytes are written or there is no free block to allocate or we meet the end of the inode.
    - How will you handle the offset?
        - This work is the same as the one in `read`: `skipBlocks` and `remainder`.
    - How will you know if you need a new block?
        - When meeting empty blocks during the loop, we can call `allocate_free_block` to get a free block assigned.
        - It is the same no matter the empty block is in direct array or indirect block.
    - How will you manage allocating a new block if you need another one?
        - We can simply loop over the `bitMap` vector we maintained and find the first index marked as `FREE`. That block can be the new block.
        - Before returning the index, the element should be set to `OCCUPIED`.
    - How will you copy from a block to the data buffer?
        - Similar to `read`, `size` and `bytesToWrite = min(disk->BLOCK_SIZE, rlength)` tell me what place at the `data` pointer to write and how many bytes to write at a time.
        - Each time I copy from a block to the data buffer, I use `memcpy` function.

        - After finishing copy, the `size` variable would be the total bytes written.
    - How will you update the inode?
        - There are three fields that may be changed during the writing process, they are `Size`, `Direct` and `Indirect`.
        - I will follow a lazy strategy to minimize the times of calling `Disk::write`, which means I will do all the updating things just before return.
        - I have a `size` variable which keeps tracking how many bytes are written. Therefore, `inode.Size += size` is all I need.
        - The `Direct` array is changed during write process. When it is before writing, `Direct` is surely ready.
        - The `Indirect` is the same as `Direct`.

Test Result

The test result is shown below. The failed tests are all caused by **performance incoherence**. In all the cases, my implementation has **better** performance than the test requires.

> hnoodles@hnoodles-UX410UQK:~/17302010002/OperatingSystemI/SimpleFileSystem$ make test
>
> g++ -g -gdwarf-2 -std=gnu++11 -Wall -Iinclude -fPIC -c -o src/library/fs.o src/library/fs.cpp
>
> ar rcs lib/libsfs.a src/library/disk.o src/library/fs.o
>
> g++ -Llib -o bin/sfssh src/shell/sfssh.o -lsfs
>
> Testing cat on data/image.5 ... Success
>
> Testing cat on data/image.20 ... **Failure**
>
> --- /dev/fd/63 2020-05-16 22:20:04.810435945 +0800
>
> +++ /dev/fd/62 2020-05-16 22:20:04.810435945 +0800
>
> @@ -143,7 +143,7 @@
>
> 0 bytes copied
>
> 0 disk block writes
>
> -20 disk block reads
>
> +21 disk block reads
>
> 27160 bytes copied
>
> 9546 bytes copied
>
> Abraham Clark
>
> Testing copyin in /tmp/tmp.xoaZh20yYg/image.5 ... Success
>
> Testing copyin in /tmp/tmp.xoaZh20yYg/image.20 ... Success
>
> Testing copyin in /tmp/tmp.xoaZh20yYg/image.200 ... Success
>
> Testing copyout in data/image.5 ... Success

```
Testing copyout in data/image.20 ... Success
Testing copyout in data/image.200 ... Success
Testing create in data/image.5.create ... Files /dev/fd/63 and /dev/fd/62 differ
False
Testing debug on data/image.5 ... Success
Testing debug on data/image.20 ... Success
Testing debug on data/image.200 ... Success
Testing format on data/image.5.formatted ... Success
Testing format on data/image.20.formatted ... Success
Testing format on data/image.200.formatted ... Success
Testing mount on data/image.5 ... Success
Testing mount-mount on data/image.5 ... Success
Testing mount-format on data/image.5 ... Success
Testing bad-mount on /tmp/tmp.3dZos4qHQG/image.5 ... Success
Testing bad-mount on /tmp/tmp.3dZos4qHQG/image.5 ... Success
Testing bad-mount on /tmp/tmp.3dZos4qHQG/image.5 ... Success
Testing bad-mount on /tmp/tmp.3dZos4qHQG/image.5 ... Success
Testing bad-mount on /tmp/tmp.3dZos4qHQG/image.5 ... Success
Testing remove in /tmp/tmp.ry27gDmNee/image.5 ... False
--- /dev/fd/63  2020-05-16 22:20:05.022429932 +0800
+++ /dev/fd/62  2020-05-16 22:20:05.022429932 +0800
@@ -38,5 +38,5 @@
 Inode 2:
     size: 0 bytes
     direct blocks:
-17 disk block reads
+25 disk block reads
 8 disk block writes
Testing remove in /tmp/tmp.ry27gDmNee/image.5 ... False
--- /dev/fd/63  2020-05-16 22:20:05.030429705 +0800
+++ /dev/fd/62  2020-05-16 22:20:05.030429705 +0800
@@ -54,5 +54,5 @@
 Inode 2:
     size: 965 bytes
```

```
    direct blocks: 4
-23 disk block reads
+27 disk block reads
 10 disk block writes
Testing remove in /tmp/tmp.ry27gDmNee/image.20 ... False
--- /dev/fd/63  2020-05-16 22:20:05.038429478 +0800
+++ /dev/fd/62  2020-05-16 22:20:05.038429478 +0800
@@ -41,5 +41,5 @@
    direct blocks: 4 5 6 7 8
    indirect block: 9
    indirect data blocks: 13 14
-31 disk block reads
-11 disk block writes
+41 disk block reads
+18 disk block writes
Testing stat on data/image.5 ... Success
Testing stat on data/image.20 ... Success
Testing stat on data/image.200 ... Success
Testing valgrind on /tmp/tmp.0xOvTFPQph/image.200 ... Success
```

Let's walk through those "failures" one by one.

Cat on data/image.20

This one is because my implementation used less reads.

```
...
-20 disk block reads
+21 disk block reads
...
```

Create in data/image.5.create

This one only gives me a `False` with no more information. Thus I modified the test shell code to seek more details. Here is the actual result.

```
Testing create in data/image.5.create ... False
--- /dev/fd/63  2020-05-16 18:03:27.829294066 +0800
+++ /dev/fd/62  2020-05-16 18:03:27.829294066 +0800
@@ -524,5 +524,5 @@
Inode 127:
```

> size: 0 bytes
>
> direct blocks:
>
> -134 disk block reads
>
> +261 disk block reads
>
> 127 disk block writes

You can see that this is also because my implementation has better read performance.

Remove in /tmp/tmp.ry27gDmNee/image.5

1. Again, mine have better read performance.

   > ...
   >
   > -17 disk block reads
   >
   > +25 disk block reads
   >
   > ...

2. Same as above.

   > ...
   >
   > -23 disk block reads
   >
   > +27 disk block reads
   >
   > ...

Remove in /tmp/tmp.ry27gDmNee/image.20

In this case, my implementation is better in both read and write.

> ...
>
> -31 disk block reads
>
> -11 disk block writes
>
> +41 disk block reads
>
> +18 disk block writes

Bonus

In this section, I will explain the main optimization I have done to improve my performance(, though it's not requested).

1. Load_inode & Save_inode

Besides `size_t inumber` and `Inode *node`, I added two more parameters `Block *block` and `bool needRead` to these two functions.

The reason why I made the change is that I found that in some cases such as `create`, the inode block has already been located, so it's unnecessary to read it from the disk again when saving a node.

Also, in `remove`, we can use the `Block *` to store the inode block we read in `load_inode` and reuse it in `save_inode` to decrease the time of calling `Disk::read`.

2. Allocate_free_block

In common sense, allocating a free block may come with cleaning it at once (use `memset` to set all its bits to 0). However, it doesn't need to be like this. Instead, it can be treated in a way like `malloc`, not `calloc`.

In some cases, we know that we are going to write a whole block of data to a newly allocated block, where allocating a "clean" block is unnecessary. Actually in those cases, all we care about is that the block is `FREE`. This happens in `write`.

Even though in some cases, for instance, when we want to allocate a block as indirect block, or when the new block isn't going to be written wholly, we can still explicitly call `memset` to do the cleaning work on demand.

In this way, we can have less `Disk::write` operations.