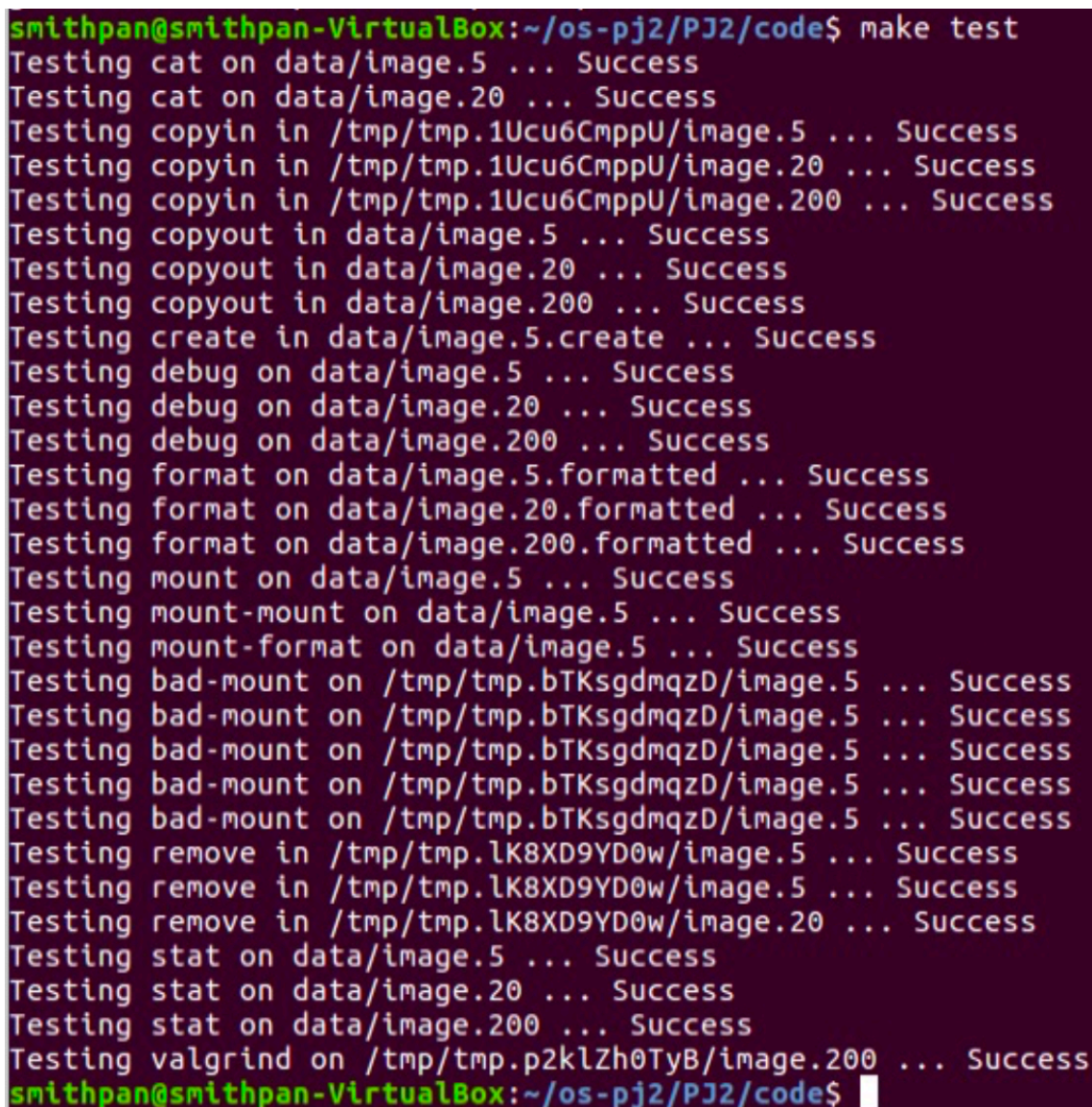


操作系统 PJ2 文档

潘星宇 18302010007

简述

本次 pj 需要实现一个简易的文件系统，系统运行的环境均由模拟完成，通过预先编写好的 shell 程序对于 file system 的各个操作进行调用和测试。下为测试样例通过截图



```
smithpan@smithpan-VirtualBox:~/os-pj2/PJ2/code$ make test
Testing cat on data/image.5 ... Success
Testing cat on data/image.20 ... Success
Testing copyin in /tmp/tmp.1Ucu6CmppU/image.5 ... Success
Testing copyin in /tmp/tmp.1Ucu6CmppU/image.20 ... Success
Testing copyin in /tmp/tmp.1Ucu6CmppU/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 ... Success
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.bTKsgdmqzD/image.5 ... Success
Testing bad-mount on /tmp/tmp.bTKsgdmqzD/image.5 ... Success
Testing bad-mount on /tmp/tmp.bTKsgdmqzD/image.5 ... Success
Testing bad-mount on /tmp/tmp.bTKsgdmqzD/image.5 ... Success
Testing bad-mount on /tmp/tmp.bTKsgdmqzD/image.5 ... Success
Testing remove in /tmp/tmp.lK8XD9YD0w/image.5 ... Success
Testing remove in /tmp/tmp.lK8XD9YD0w/image.5 ... Success
Testing remove in /tmp/tmp.lK8XD9YD0w/image.20 ... Success
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.p2klZh0TyB/image.200 ... Success
smithpan@smithpan-VirtualBox:~/os-pj2/PJ2/code$
```

实现

本次 pj 实际上即为面向接口编程，实现 fs.h 中规定的若干函数，包括 debug, format, mount, read, write，提供一个文件系统基本操作的支持。

读写 inode 的辅助函数

首先实现根据 inode number 找到对应的块号和偏置，并 inode 进行读写的功能。

```
// Load inode -----
bool FileSystem::load_inode(size_t inode_number, Inode* node) {
    size_t block_number = 1 + (inode_number / INODES_PER_BLOCK);
    size_t inode_offset = inode_number % INODES_PER_BLOCK;
    if (inode_number >= num_inodes)
        return false;

    Block block;
    disk->read(block_number, block.Data);
    *node = block.Inodes[inode_offset];
    return true;
}

// Save inode -----
bool FileSystem::save_inode(size_t inode_number, Inode* node) {
    size_t block_number = 1 + inode_number / INODES_PER_BLOCK;
    size_t inode_offset = inode_number % INODES_PER_BLOCK;
    if (inode_number >= num_inodes)
        return false;

    Block block;
    disk->read(block_number, block.Data);
    block.Inodes[inode_offset] = *node;
    disk->write(block_number, block.Data);
    return true;
}
```

对于 block 的读，由于需要的内存空间较大，为了避免不必要的内存分配，没有对 readBlock 进行封装

Debug

debug 函数主要通过 super block 中的内容对整个文件系统的信息进行打印，通过 inode 数量来遍历所有的 inode，包括会读取可能存在的 indirect inode

```
// Debug file system -----

void FileSystem::debug(Disk *disk) {
    Block block;
    // Read SuperBlock
    disk->read(0, block.Data);
}
```

```

printf("SuperBlock:\n");
if (block.Super.MagicNumber == MAGIC_NUMBER)
    printf("    magic number is valid\n");
else
    printf("    magic number is invalid\n");

printf("    %u blocks\n", block.Super.Blocks);
printf("    %u inode blocks\n", block.Super.InodeBlocks);
printf("    %u inodes\n", block.Super.Inodes);

// Read Inode blocks

// save the number of inode blocks
uint32_t inode_blocks_count = block.Super.InodeBlocks;
Block indirect_block;
for (unsigned int i = 0; i < inode_blocks_count; i++) {
    // reuse the block on stack
    disk->read(i + 1, block.Data);
    for (unsigned int j = 0; j < INODES_PER_BLOCK; j++) {
        if (!block.Inodes[j].Valid)
            continue;

        std::string direct;    // string to store direct blocks
        std::string indirect;  // string to store indirect blocks
        uint32_t indirect_block_number;
        for (unsigned int k = 0; k < POINTERS_PER_INODE; k++) {
            if (block.Inodes[j].Direct[k] != 0) {
                direct += " ";
                direct += std::to_string(block.Inodes[j].Direct[k]);
            }
        }
        indirect_block_number = block.Inodes[j].Indirect;
        if (indirect_block_number != 0) {
            disk->read(indirect_block_number, indirect_block.Data);
            for (unsigned int l = 0; l < POINTERS_PER_BLOCK; l++) {
                if (indirect_block.Pointers[l] != 0) {
                    indirect += " ";
                    indirect += std::to_string(indirect_block.Pointers[l]);
                }
            }
        }
        printf("Inode %u:\n", j);
        printf("    size: %u bytes\n", block.Inodes[j].Size);
        printf("    direct blocks:%s\n", direct.c_str());
        if (indirect.length() > 0) {
            printf("    indirect block: %u\n", indirect_block_number);
            printf("    indirect data blocks:%s\n", indirect.c_str());
        }
    }
}
}

```

```
}  
}
```

Format

format 函数负责格式化磁盘文件，首先需要检验磁盘是否已经挂载在系统中。随后会对超级块进行初始化设置，包括 inode 的数量，随后对所有 inode block 与 data block 进行置零

```
// Format file system -----  
  
bool FileSystem::format(Disk *disk) {  
    // Check if mounted  
    if (disk->mounted()) return false;  
  
    // Write SuperBlock  
    Block block;  
    memset(block.Data, 0, disk->BLOCK_SIZE);  
    block.Super.MagicNumber = MAGIC_NUMBER;  
    block.Super.Blocks = disk->size();  
    // set the number of inode blocks  
    block.Super.InodeBlocks = (size_t)((float)disk->size() * 0.1) + 0.5;  
    block.Super.Inodes = INODES_PER_BLOCK * block.Super.InodeBlocks;  
    disk->write(0, block.Data);  
  
    // Clear all other blocks  
    char clear[BUFSIZ] = { 0 };  
    for (size_t i = 1; i < block.Super.Blocks; i++)  
        disk->write(i, clear);  
  
    return true;  
}
```

Mount

mount 函数用于挂载磁盘，由于本次的实现中 bitmap 是存储在内存中的，因此挂载一个磁盘文件相当于需要扫描该磁盘文件，并在内存中生成对应的 free bitmap。具体实现中 bitmap 只用一个 `vector<int>` 实现

```
// Mount file system -----  
  
bool FileSystem::mount(Disk* disk) {  
    // Some validations ... ignore here  
    disk->mount();  
  
    // Copy metadata
```

```

num_blocks = block.Super.Blocks;
num_inode_blocks = block.Super.InodeBlocks;
num_inodes = block.Super.Inodes;
this->disk = disk;

// Allocate free block bitmap
free_bitmap = std::vector<int>(num_blocks, 1);

//set all blocks to free initially, 1 indicates true for free blocks
for (uint32_t i = 0; i < num_blocks; i++)
    free_bitmap[i] = 1;
// SuperBlock is not free
free_bitmap[0] = 0;
// inode blocks are not free
for (unsigned int i = 0; i < num_inode_blocks; i++)
    free_bitmap[1 + i] = 0;

for (uint32_t inode_block = 0; inode_block < num_inode_blocks; inode_block++) {
    Block b;
    disk->read(1 + inode_block, b.Data);
    // reads each inode
    for (uint32_t inode = 0; inode < INODES_PER_BLOCK; inode++) {
        // if it's not valid, it has no blocks
        if (!b.Inodes[inode].Valid)
            continue;

        uint32_t n_blocks = (uint32_t)ceil(b.Inodes[inode].Size / (double)disk->BLOCK_SIZE);
        // read all direct blocks
        for (uint32_t pointer = 0; pointer < POINTERS_PER_INODE && pointer <
n_blocks; pointer++)
            free_bitmap[b.Inodes[inode].Direct[pointer]] = 0;
        //read indirect block if necessary
        if (n_blocks > POINTERS_PER_INODE) {
            Block indirect;
            disk->read(b.Inodes[inode].Indirect, indirect.Data);
            free_bitmap[b.Inodes[inode].Indirect] = 0;
            for (uint32_t pointer = 0; pointer < n_blocks - POINTERS_PER_INODE;
pointer++) {
                free_bitmap[indirect.Pointers[pointer]] = 0;
            }
        }
    }
}

return true;
}

```

Create

create 函数通过创建一个新的 inode 来创建新文件，返回新的 inode 即可，这里需要找到空闲的 inode 的位置，实现方式为遍历

```
ssize_t FileSystem::create() {
    // Locate free inode in inode table
    int ind = -1;
    for (uint32_t inode_block = 0; inode_block < num_inode_blocks; inode_block++) {
        Block b;
        disk->read(1 + inode_block, b.Data);
        // reads each inode
        for (uint32_t inode = 0; inode < INODES_PER_BLOCK; inode++) {
            // if it's not valid, it's free to be written
            if (!b.Inodes[inode].Valid) {
                ind = inode + INODES_PER_BLOCK * inode_block;
                break;
            }
        }
        if (ind != -1)
            break;
    }
    // Record inode if found
    if (ind == -1)
        return -1;

    Inode i;
    i.Valid = true;
    i.Size = 0;
    for (unsigned int j = 0; j < POINTERS_PER_INODE; j++)
        i.Direct[j] = 0;
    i.Indirect = 0;
    save_inode(ind, &i);

    return ind;
}
```

Remove

remove 函数语义上为在文件系统中删除一个文件，实现上，首先将 direct block 与 indirect block 指向的数据块标记为 free，也就是修改 `vector<int>` 中对应的数从 0 变为 1，随后标记原来的 inode 为可用状态

```
// Remove inode -----

bool FileSystem::remove(size_t inode_number) {
    Inode node;
```

```

// Load inode information
if (!load_inode(inode_number, &node)) return false;
if (node.Valid == 0) return false;

// Free direct blocks
for (unsigned int i = 0; i < POINTERS_PER_INODE; i++) {
    if (node.Direct[i] != 0) {
        free_bitmap[node.Direct[i]] = 1;
        node.Direct[i] = 0;
    }
}

// Free indirect blocks
if (node.Indirect != 0) {
    free_bitmap[node.Indirect] = 1;
    Block b;
    disk->read(node.Indirect, b.Data);
    // Free blocks pointed to indirectly
    for (unsigned int i = 0; i < POINTERS_PER_BLOCK; i++) {
        if (b.Pointers[i] != 0) {
            free_bitmap[b.Pointers[i]] = 1;
        }
    }
}

// Clear inode in inode table
node.Indirect = 0;
node.Valid = 0;
node.Size = 0;
if (!save_inode(inode_number, &node))
    return false;
return true;
}

```

tat

stat 函数类似 Linux 中对应的命令，用于获取一个文件的信息。在该实现中，获取文件大小即可，也就是 inode 中的 size

```

// Inode stat -----

ssize_t FileSystem::stat(size_t inode_number) {
    // Load inode information
    Inode i;
    if (!load_inode(inode_number, &i) || !i.Valid)
        return -1;
    return i.Size;
}

```


Read

read 函数是一个通用读取接口，语义上的功能为读取一个指定文件从指定偏置开始的定长内容。实现上的流程如下

- (1) 载入对应 inode，并判断其合法性
- (2) 得到合法读取的长度，如果需要读取的数据超过了文件的长度，则截断
- (3) 通过循环不断读取块中的数据，这里需要判断是否会进入间接块
- (4) 将 data block 中的内容读取并复制之后，进行下一次循环，直到已经读到的长度等于需要读取的长度
- (5) 出现错误则返回 -1

```
// Read from inode -----
ssize_t FileSystem::read(size_t inode_number, char* data, size_t length, size_t offset)
{
    // Load inode information
    Inode inode;
    if (!load_inode(inode_number, &inode) || offset > inode.Size) return -1;
    // Adjust length
    length = std::min(length, inode.Size - offset);

    uint32_t start_block = offset / disk->BLOCK_SIZE;
    // Read block and copy to data; use memcpy
    // Get indirect block number if it will need it
    Block indirect;
    if ((offset + length) / disk->BLOCK_SIZE > POINTERS_PER_INODE) {
        // make sure direct block is allocated
        if (inode.Indirect == 0) return -1;
        disk->read(inode.Indirect, indirect.Data);
    }

    size_t read = 0;
    for (uint32_t block_num = start_block; read < length; block_num++) {
        // figure out which block we're reading
        size_t block_to_read;
        if (block_num < POINTERS_PER_INODE)
            block_to_read = inode.Direct[block_num];
        else
            block_to_read = indirect.Pointers[block_num - POINTERS_PER_INODE];

        //make sure block is allocated
        if (block_to_read == 0) return -1;

        //get the block -- from either direct or indirect
        Block b;
```



```

    disk->read(block_to_read, b.Data);
    size_t read_offset;
    size_t read_length;

    // if it's the first block read, have to start from an offset
    // and read either until the end of the block, or the whole request
    if (read == 0) {
        read_offset = offset % disk->BLOCK_SIZE;
        read_length = std::min(disk->BLOCK_SIZE - read_offset, length);
    } else {
        // otherwise, start from the beginning, and read
        // either the whole block or the rest of the request
        read_offset = 0;
        read_length = std::min(disk->BLOCK_SIZE - 0, length - read);
    }
    memcpy(data + read, b.Data + read_offset, read_length);
    read += read_length;
}
return read;
}

```

Write

Write 函数是一个通用写入接口，语义上的功能为在一个指定文件从指定偏置开始写入定长的内容。实现上的流程类似 read 函数，具体流程如下

- (1) 载入对应 inode，并判断其合法性
- (2) 得到合法读取的长度，同样要进行长度截断
- (3) 通过循环不断写数据，这里需要判断是否需要分配新的块以及是否会进入间接块
- (4) 将要写入的内容写入 data block 中，进行下一次循环，直到已经写入的长度等于需要写入的长度
- (5) 判断是否修改 inode.size 即文件长度，如果需要修改，则需要 save inode，即再写一次其对应的 block
- (6) 出现错误则返回 -1

```

// Write to inode -----

ssize_t FileSystem::write(size_t inode_number, char* data, size_t length, size_t
offset) {
    // Load inode
    Inode inode;
    if (!load_inode(inode_number, &inode) || offset > inode.Size) return -1;

    size_t MAX_FILE_SIZE = disk->BLOCK_SIZE * (POINTERS_PER_INODE *
POINTERS_PER_BLOCK);
    // Adjust length

```

```

length = std::min(length, MAX_FILE_SIZE - offset);

uint32_t start_block = offset / disk->BLOCK_SIZE;
Block indirect;
bool read_indirect = false;
bool modified_inode = false;
bool modified_indirect = false;

// Write block and copy data
size_t written = 0;
for (uint32_t block_num = start_block;
     written < length && block_num < POINTERS_PER_INODE + POINTERS_PER_BLOCK;
     block_num++) {

    // figure out which block we're reading
    size_t block_to_write;
    if (block_num < POINTERS_PER_INODE) {
        // Allocate block if necessary
        if (inode.Direct[block_num] == 0) {
            ssize_t allocated_block = allocate_free_block();
            if (allocated_block == -1)
                break;

            inode.Direct[block_num] = allocated_block;
            modified_inode = true;
        }
        block_to_write = inode.Direct[block_num];
    }
    else { // Indirect block
        // Allocate indirect block if necessary
        if (inode.Indirect == 0) {
            ssize_t allocated_block = allocate_free_block();
            if (allocated_block == -1)
                return written;

            inode.Indirect = allocated_block;
            modified_indirect = true;
        }

        // Read indirect block if hasn't been read yet
        if (!read_indirect) {
            disk->read(inode.Indirect, indirect.Data);
            read_indirect = true;
        }

        // Allocate block if necessary
        if (indirect.Pointers[block_num - POINTERS_PER_INODE] == 0) {
            ssize_t allocated_block = allocate_free_block();
            if (allocated_block == -1)

```

```

        break;

        indirect.Pointers[block_num - POINTERS_PER_INODE] = allocated_block;
        modified_indirect = true;
    }
    block_to_write = indirect.Pointers[block_num - POINTERS_PER_INODE];
}

//get the block -- from either direct or indirect
size_t write_offset;
size_t write_length;

// if it's the first block written, have to start from an offset
// and write either until the end of the block, or the whole request
if (written == 0) {
    write_offset = offset % disk->BLOCK_SIZE;
    write_length = std::min(disk->BLOCK_SIZE - write_offset, length);
}
else {
    // otherwise, start from the beginning, and write
    // either the whole block or the rest of the request
    write_offset = 0;
    write_length = std::min(disk->BLOCK_SIZE - 0, length - written);
}

char write_buffer[disk->BLOCK_SIZE];

// if we're not writing the whole block, need to copy what's there
if (write_length < disk->BLOCK_SIZE)
    disk->read(block_to_write, (char*)write_buffer);

// copy into buffer
memcpy(write_buffer + write_offset, data + written, write_length);
disk->write(block_to_write, (char*)write_buffer);
written += write_length;
}

// update inode size
uint32_t new_size = std::max((size_t)inode.Size, written + offset);
if (new_size != inode.Size) {
    inode.Size = new_size;
    modified_inode = true;
}

// save inode and indirect if necessary
if (modified_inode)
    save_inode(inode_number, &inode);
if (modified_indirect)
    disk->write(inode.Indirect, indirect.Data);

```

```
    return written;
}
```

总结

本次 pj 的要求与文档十分明确，测试用例也十分明了，个人感觉有两处不足的地方。其一在于测试用例的答案写死在 .sh 的脚本中，而非通过一个标准答案的二进制可执行程序实时产生的，这使得测试样例极大地与 data 中的 images 文件耦合起来，并且在项目一开始的时候也没有标准答案的程序可以对程序的行为进行实验。其二在于测试用例中规定了每个情况下读写块的次数，并要求学生的实现与标准实现的读写次数相同，这是有一些不合理的，如果实现的性能较好也许读写快的次数会少，而实现的不好则会出现冗余的读写，这都是有可能的，从功能的实现角度限制读写块次数不是很合理。