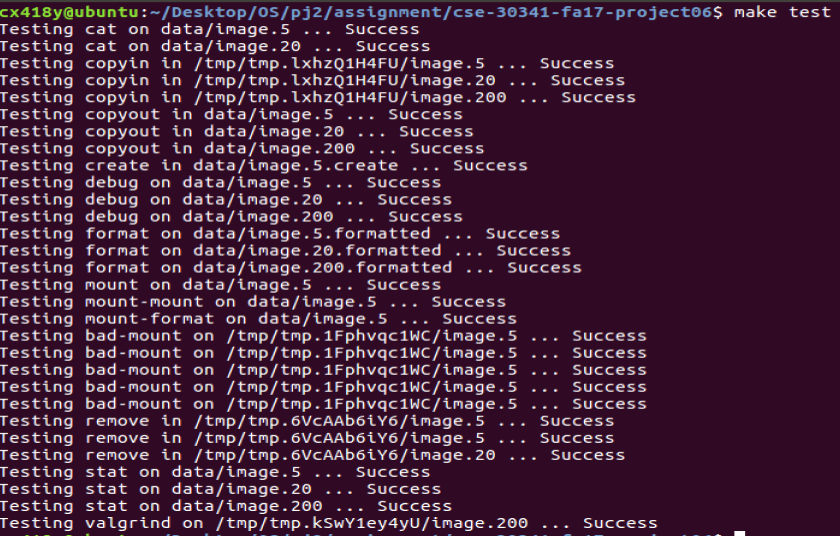
# PJ2实验文档

## 运行截图



## 代码实现

#### 1. static void debug(Disk \*disk)

该方法会扫描已挂载的文件系统，并打印出 inode 和块的组织方式。

磁盘的开头的第一块是超级块，首先读取超级块的内容并且判断超级块中的Magic字段是否有效并打印相关信息：

// Read Superblock  
 disk->read(0, block.Data);  
 printf("SuperBlock:\n");  
 // if Magic is valid  
 if (block.Super.MagicNumber == MAGIC\_NUMBER) {  
 printf(" magic number is valid\n");  
 } else {  
 printf(" magic number is invalid\n");  
 }

之后会读取inode块，根据inode块的valid位判断是否有效，如果有效则先读取5个direct指针指向的数据块，之后再读取indirect指针，读取间接块的内容，每个间接块有1024个指向数据块的指针，依次读取并打印：

// read direct block  
 for (unsigned int j = 0; j < POINTERS\_PER\_INODE; j++) {  
 if (block.Inodes[i].Direct[j] != 0) {  
 direct += " ";  
 direct += to\_string(block.Inodes[i].Direct[j]);  
 }  
 }  
 // read indirect block  
 indir = block.Inodes[i].Indirect;  
 if (indir != 0) {  
 disk->read(indir, inode\_block.Data);  
 for (unsigned int l = 0; l < POINTERS\_PER\_BLOCK; l++) {  
 if (inode\_block.Pointers[l] != 0) {  
 indirect += " ";  
 indirect += to\_string(inode\_block.Pointers[l]);  
 }  
 }  
 }

#### 2. static bool format(Disk \*disk)

此方法在磁盘上创建一个新的文件系统，销毁已存在的所有数据。为inode留出10%的块，清除inode表，并编写超级块。

// Write superblock  
 Block block;  
 memset(block.Data,0,disk->BLOCK\_SIZE);  
 block.Super.MagicNumber = MAGIC\_NUMBER;  
 block.Super.Blocks = disk->size();  
 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);  
 }

#### 3. bool mount(Disk \*disk)

此方法检查磁盘中的文件系统。如果存在，读取超级块，构建一个空闲块位图，并准备文件系统以供使用。

首先需要判断使用的disk是否已经被mount，如果已经被mount则直接返回false；

读取超级块，如果inode数不匹配、magic字段不对、或者为inode留出的块数不对均返回false：

// Read superblock  
 Block block;  
 disk->read(0, block.Data);  
   
 if (block.Super.MagicNumber != MAGIC\_NUMBER || block.Super.Inodes != block.Super.InodeBlocks \* INODES\_PER\_BLOCK || block.Super.Blocks < 0 || block.Super.InodeBlocks != ceil(.1 \* block.Super.Blocks)) {  
 return false;  
 }

之后复制元数据复制，之后创建一个空闲位图，除了第一个超级块以及其后的inode块为1，其余均为0：

free\_bitmap = std::vector<int> (num\_blocks,1);  
 free\_bitmap[0] = 0;  
 for (uint32\_t i = 0; i < num\_blocks; i++) {  
 free\_bitmap[i+1] = 1;  
 }  
 // inode blocks are not free  
 for (unsigned int i = 1; i < num\_inode\_blocks; i++) {  
 free\_bitmap[i] = 0;  
 }

之后根据inode读取数据块，过程与debug函数类似。

#### 4. ssize\_t create()

此方法创建长度为零的新 **inode**。成功返回inumber ，失败返回-1。

遍历每一个inode Block中的inode，寻找是否有空的inode，如果没有则返回-1.如果有则将该inode的valid位设为true，大小设置为0，将5个指向数据块的指针和1个指向间接块的指针均设为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's free to be written  
 if (!b.Inodes[inode].Valid) {  
 ind = inode + INODES\_PER\_BLOCK\*inode\_block;  
 break;  
 }  
 }  
 if (ind != -1) {  
 break;  
 }  
   
}

#### 5. bool remove(size\_t inumber)

此方法将删除由number指示的 **inode**。它应该释放分配给此**inode**的所有数据和间接块，并将它们返回到自由块映射。

这里实现辅助函数load\_inode()，传入inumber，如果inumber有效会从磁盘读取并inode；

函数开始会先加载inode，如果加载inode失败或者inode无效均会返回false；之后逐步将inode中direct block的指针、指向间接块的指针，间接块中的指针清空并将inode的size设为0，并保存：

// Free direct blocks  
 for (unsigned int i = 0; i < POINTERS\_PER\_INODE; i++) {  
 if (inode.Direct[i] != 0) {  
 free\_bitmap[inode.Direct[i]] = 1;  
 inode.Direct[i] = 0;  
 }  
 }  
 // Free indirect blocks  
 if (inode.Indirect != 0) {  
 free\_bitmap[inode.Indirect] = 1;  
 Block b;  
 disk->read(inode.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  
 inode.Indirect = 0;  
 inode.Valid = 0;  
 inode.Size = 0;

#### 6. ssize\_t stat(size\_t inumber)

此方法返回给定的inumber逻辑大小，以字节为单位。

Inode inode;  
 if (!load\_inode(inumber,&inode) || !inode.Valid) {  
 return -1;  
 }  
 return i.Size;

#### 7. ssize\_t read(size\_t inumber, char \*data, size\_t length, size\_t offset)

此方法从有效的 inode 读取数据。然后，它将length长度字节从 inode 的数据块复制到数据指针中，从 inode 中的offset开始。它应返回读取的总字节数。如果给定的 inumber 无效，或者遇到任何其他错误，则该方法返回 -1。

首先我们需要先根据inumber加载inode，之后调整读取的长度，应该为length和inode.size-offset的较小值。之后开始以块为单位读取，并且记录当前读取的块数，如果小于5则直接读取，否则则应该从间接块中读取：

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;  
 }  
 Block b;  
 disk->read(block\_to\_read,b.Data);  
 size\_t read\_offset;  
 size\_t read\_length;  
  
 if (read == 0) {  
 read\_offset = offset % disk->BLOCK\_SIZE;  
 read\_length = std::min(disk->BLOCK\_SIZE - read\_offset, length);  
 } else {  
 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;  
 }

#### 8. ssize\_t write(size\_t inumber, char \*data, size\_t length, size\_t offset)

此方法将数据写入有效的 inode，方法是将length长度字节从data指针复制到从偏移字节开始的 inode 的数据块中。它将在此过程中分配任何必要的直接和间接块。之后，它返回实际写入的字节数。如果给定的 inumber 无效，或者遇到任何其他错误，则返回 -1。

与读类似，只是需要新分配块，这里使用了辅助方法allocate\_free\_block：

int block = -1;  
 for (unsigned int i = 0; i < num\_blocks; i++) {  
 if (free\_bitmap[i]) {  
 free\_bitmap[i] = 0;  
 block = i;  
 break;  
 }  
 }  
  
 // need to zero data block if we're allocating one  
 if (block != -1) {  
 char data[disk->BLOCK\_SIZE];  
 memset(data,0,disk->BLOCK\_SIZE);  
 disk->write(block,(char\*)data);  
 }  
  
 return block;