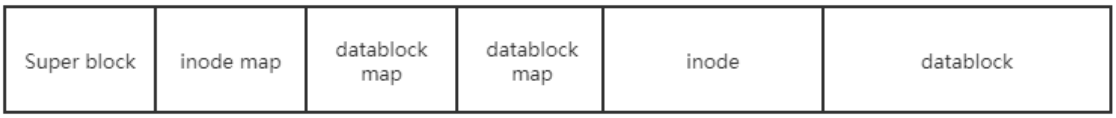
实验报告

系统整体架构

1. 块设备组织结构



如图所示，整个系统的架构分为六个主要部分。

第一部分，super block，占据一个block，记录了文件系统具有了data block的数量，可以使用的data block的数量以及其他有关于文件系统的基本信息。

第二部分，inode map，占据一个block，使用bit map位向量的形式记载了有哪些inode已经被占用，有哪一些是空闲的。最多可以支持4096*8=32768个inode的信息表示。

第三部分，datablock map，占据两个datablock，使用bit map位向量的形式记录了有哪些data block已经被使用，有哪一些空闲。最多可以支持4096*8*2=65536个data block的占用信息。

第四部分，inode块，总共分配了640个block 给这一部分。由于我设计的单个inode的大小是64B，因此一个块可以装得下64个inode，最多可以有40960个inode。这里受限于inode map的大小，最多只能有32768个inode被使用，所以可以支持的文件及目录数量也是32768，符合实验要求。

第五部分，datalbock数据区，总共占据了64800块。每一个块大小4kb，总共具有64800*4kb=259200kb=250mb+3200kb空间。具有超过250mb的真实可用的空间，复合实验的设计要求。

最后还剩下92个block没有使用，属于unused region，可以跟据以后的需要补充到某一模块。

2. 文件信息节点及文件目录结构

inode结构体的定义如下：

```
struct inode {//size=4+4(padding)+8+8+8+8+2+22=64
  mode_t mode;//4+4(padding)
  off_t size;//8
  time_t atime;//8
  time_t mtime;//8
  time_t ctime;//8
  unsigned short single_ptr;//2
  unsigned short direct_ptr[DIRECT_PTR];//22
};
```

其中，mode指文件模式，size指文件的大小，atime、ctime、mtime分别是指访问时间，状态改变的时间和修改的时间，以上这些内容都是同结构体stat相对应的，为的是在使用fs_getattr调用的时候可以方便的跟据inode信息快速填充完一个stat结构体。

使用了11个直接指针（direct ptr），一个一级间接指针（single ptr）。当文件的大小不超过11*4kb=44kb的时候可以只使用直接指针进行存储。受inode内含指针数量的限制，该文件系统所支持的最大的文件大小是

$$11 \times 4kb + 2048 \times 4kb = 8mb + 44kb$$

而实验要求只需要支持单个文件最大大小为8mb，所以符合实验要求。

这个地方需要搞清楚的是，虽然直接指针和一级间接指针都是指针，但是这并不等价于必须要求他们真的必须是指针类型，只要他们可以实现完成指针的功能即可。无论是直接指针还是一级间接指针，他们的作用是存储这个inode对应的文件的datablock所在的位置，换句话说，就是disk之中的第几个块里存放的是这个inode的数据。而我们的硬盘只有65536个block，所以要存"第几个"只需要一个unsigned short就恰好可以办到，而且一个unsinged short类型只使用两个字节，大大地节省了inode需要的空间大小，优势明显。实际上在我看来，这个地方直接使用指针类型反而无法提供指针的功能，因此我们的空间是以块为单位进行组织的，并且只有通过disk_read和disk_write两个接口访问，使用指针是不可能解引用访问硬盘的。

文件目录的结构体定义如下：

```
struct directory_entry {//size=24+2+2+4=32
  char filename[24];//24
  unsigned short inodeno;//2
  unsigned short padding1;//2
  int padding2;//4
};
```

目录的功能相当于字典，提供了一种文件名字和inode编号之间的映射关系，所以一条目录最重要的内容就是文件名字以及inode no。此外，在结构体中还添加了两个填充物：padding 1 padding2。其目的是将一条entry结构体的大小设置为32，从而使一个block恰好可以容纳128条目录项。

另外还需要想清楚的一点是，虽然我们常用目录树来表示文件系统之中目录的层次关系，但是在实现的时候，不需要真的使用一个实际的树来组织全部的文件和目录。每一个目录只需要记录自己下一级的文件名与inode no之间的对应关系，而他的子目录之中的内容不是由自己来负责，是由自己的子目录来负责的。

通用函数设计

通用函数指的是一些常见的接口，将需要被反复使用的代码和小的功能抽取出来，包装成一个小的函数。主要的功能接口包括位图的更改、查询操作，inode指针的修改、计数工作，以及从文件的绝对路径到文件的inode编号的转换操作。现将这些函数介绍如下：

1. 位图更改操作

实验中将所有的位更改操作都包装成了一个通用的函数，需要进行置位或者擦除的时候全部都是用这个函数来完成。函数如下图所示：

```
// op =  0(clear) or 1 (malloc)
// base =IM_BASE or DM_BASE
// no is the no of bit to operate
void map_operation(int op, int base, int no) {
  int map_index = base + no / BIT_PER_BLOCK;
  int map_offset = no % BIT_PER_BLOCK;
  int byte_index = map_offset / 8;
  int byte_offset=map_offset%8;
  char buf[BLOCK_SIZE + 1];
  memset(buf, 0, BLOCK_SIZE);
  disk_read(map_index, buf);
  if(op)
    buf[byte_index] |= 1 << byte_offset;
  else{
```

```
    buf[byte_index] &= ~(1 << byte_offset);
    if(base==DM_BASE){
      char buf1[BLOCK_SIZE+1];
      memset(buf1,-1,BLOCK_SIZE);
      disk_write(DATA_BASE+no,buf1);
    }
  }
  disk_write(map_index,buf);
}
```

首先计算每一个要操作的位处于哪一个block之中（只有操作datablock map才需要计算）并读取相应的块；然后计算这个位处于第几个byte、处于这个byte之中的第几位；接着再用一些常见的位运算完成置1或清0，最后将这个块写回即可。

2. 位图查询操作

每当需要申请新的inode或者是新的data block时，都不可避免地要使用位图查询寻找空间位置。位图查询分为两种情况：查询空闲的data block，或者是查询空闲的inode，分别用不同的函数进行实现，现将查询空闲data block的核心代码展示如下所示：

```
int find_free_block(int need,int* freelist){
  char buf[BLOCK_SIZE + 1];
  int i,j=0,k=0;
  memset(buf, 0, BLOCK_SIZE);
  if(disk_read(DM_BASE, buf)){
    printf("disk_read error when find free block\n");
  }
  for (i = 0; i < BLOCK_SIZE;i++){
    if(buf[i]==(char)0xff)
      continue;
    if (need<=0)
      break;
    for (j = 0; j < 8; j++)
      if (!(buf[i] & (1 << j))) {
        freelist[k] = 8 * i + j;
        k++;
        need--;
      }
  }
  memset(buf, 0, BLOCK_SIZE);
  if(disk_read(DM_BASE + 1, buf)){
    printf("disk_read error when find free block\n");
  }
  for (i = 0; i < BLOCK_SIZE; i++) {
    if (buf[i] == (char)0xff)
      continue;
    if (need<=0)
      break;
    if (8 * i + BIT_PER_BLOCK > DATA_BLOCK_NUM) {
      printf("not enough free blocks\n");
    }
    for (j = 0; j < 8; j++)
      if (!(buf[i] & (1 << j))) {
        freelist[k] = 8 * i + j+BIT_PER_BLOCK;
        k++;
        need--;
      }
```

```
    }
}
```

参数列表之中，need代表需要寻找的空闲data block数量，freelist是一个数组，该函数将将找到的datablock的编号全部记录到freelist之中，因此不再需要函数返回值。函数实现原理比较简单，遍历map block之中的每一个字节，如果发现有0就添加到freelist之中，找到需要的数量

查询空闲inode的函数与该函数非常类似，唯一的区别是由于几乎每次查询inode的时候都只需要找到一个空闲inode即可，所以相应的函数find_free_inode()不接受任何参数，找到一个空闲inode就返回编号。不再进行展示。

3. 路径转换inode编号

由文件的绝对路径转换到文件的inode编号可以说是整个文件系统的实现之中最为核心，也使用最广泛的通用函数，几乎每一个功能都离不开这个操作。函数展示如下：

```
int path2inodeno(const char *path) {
  int pathlen = strlen(path);
  char mypath[MAXPATHLEN];
  char filename[24];
  int i,j,k;
  strncpy(mypath, path, MAXPATHLEN);
  if(pathlen==1 && path[0]=='/'){
    return 0;
  }
  if (path[pathlen - 1] == '/')
    pathlen--;
  mypath[pathlen] = '\0';
  int now_inodeno = 0;//root directory inodeno
  int start_slash = 0;//path must begin with /
  int end_slash = 0;
  for (i = start_slash + 1; i < pathlen;i++){
    if(mypath[i]=='/'){
      end_slash = i;
      for (j = start_slash + 1,k=0; j < end_slash;j++,k++)
        filename[k] = mypath[j];
      filename[k] = '\0';
      now_inodeno = p_inodeno2c_inodeno(filename, now_inodeno);
      if(now_inodeno==-1)
        return -1;
      start_slash=end_slash;
      i = start_slash;
    }
  }
  for (j = start_slash + 1,k=0; j < pathlen;j++,k++)
    filename[k] = mypath[j];
  filename[k] = '\0';
  now_inodeno = p_inodeno2c_inodeno(filename, now_inodeno);
  if(now_inodeno==-1)
    return -1;
  return now_inodeno;
}
```

整个函数的原理是：用now inode记录当前所在的目录，初始化为0表示根目录。对路径进行字符串的解析，每次遇到‘/’，同上一个‘/’之间的内容就代表一层目录的名称filename，提取出来并结合当前的now inode使用p_inodeno2c_inodeno（）函数在now inode所对应的目录之中找到filename对应的inode no。将返回结果赋给now inode no，并继续在路径之中寻找下一个‘/’，直至将路径扫描完毕。整个过程层层递进。

p_inodeno2c_inodeno(childname, parent_inodeno)的实现原理比较简单，因此没有再报告之中展示，总体思路就是由父目录的inode no找到父目录，然后一项一项便利地查询所有的目录项，知道扎到childname并返回其对应的inodeno。遍历需要先便利直接指针，如果找不到的话再遍历间接指针。

有几个细节上需要注意的点：路径是根目录的话整个路径之中只有一个‘/’，所以需要单独处理。如果使用strcpy来处理字符串进行截取操作，strcpy是不会专门在字符串的结尾加'\0'的！用strncpy进行处理，并设置一个超出字符串实际长度的的拷贝尺寸，这样的话会在拷贝结束之后在末尾自动添加'\0'的。

4. inode指针操作

inode指针操作，分为两个层面，修改操作和计数操作。所谓的修改操作，就是指增加或者减少一个inode之中所包含的指针。所有需要更改文件大小尺寸的功能都需要使用这个操作。跟据增加指针还是减少指针，分别实现了add_ptr 以及 rm_ptr。这两个函数的具体实现方面大同小异，仅仅指在细微的地方上有差别。所以只将add_ptr展示如下：

```
void add_ptr(int inodeno,int append_block,int* freelist){
  struct inode _inode=inodeno2inode(inodeno);
  int ptr_cnt[2]={0},i,j,tempblock=freelist[append_block];
  if(freelist==NULL || !append_block) return;
  count_ptr(_inode.size,ptr_cnt);
  for(i=ptr_cnt[0],j=0;i<DIRECT_PTR;i++){
    if(!append_block)
      break;
    _inode.direct_ptr[i]=freelist[j];
    append_block--;
    map_operation(1,DM_BASE,freelist[j]);
    j++;
  }
  if(append_block){
    if(_inode.single_ptr==(unsigned short)-1){
      _inode.single_ptr=tempblock;
      map_operation(1,DM_BASE,tempblock);
    }
    char ptr_buf[BLOCK_SIZE+1];
    memset(ptr_buf,0,BLOCK_SIZE);
    disk_read(DATA_BASE+_inode.single_ptr,ptr_buf);
    unsigned short * pptr=(unsigned short*)ptr_buf;
    for(i=0;i<PTR_PER_BLOCK;i++)
      if(*(pptr+i)==(unsigned short)-1)
        break;
    for(;i<PTR_PER_BLOCK;i++){
      if(append_block<=0)
        break;
      *(pptr+i)=(unsigned short)freelist[j];
      map_operation(1,DM_BASE,freelist[j]);
      j++;
      append_block--;
    }
    disk_write(DATA_BASE+_inode.single_ptr,ptr_buf);
  }
```

```
  int inode_index=inodeno/INODE_PER_BLOCK+INODE_BASE;
  int inode_offset=inodeno%INODE_PER_BLOCK;
  char buf[BLOCK_SIZE+1];
  memset(buf, 0, BLOCK_SIZE);
  disk_read(inode_index, buf);
  struct inode *inode_ptr = (struct inode *)buf;
  *(inode_ptr + inode_offset) = _inode;
  disk_write(inode_index,buf);
}
```

这个函数在调用之前一般需要先调用一次find_free_block，由此获得freelist，然后作为参数传递给该函数。其实整体思路就是读出inodeno对应的inode，然后修改结构体之中的指针内容即可。函数之所以看起来冗长，同样是因为指针的结构：如果direct ptr还没有用完，就首先填满direct ptr，如果还不够的话，再动用间接指针；如果direct ptr已经用完了，就直接在间接指针的指针块之中寻找第一个为空的位置，顺着把增加的指针往后写即可。

有一种特别的情况，就是把direct ptr填满了还不够。这个时候需要在额外找一个data block作为间接指针存放直接指针的空间。为了应对这种情况，find_free_block()会多申请一个块，将这个多余的块作为备用的间接指针的指针块。

而inode指针的计算，就是指通过inode之中的size信息，也就是该文件的大小信息，计算出该文件的inode之中一共使用了几个直接指针，有没有使用间接指针（实际上来说，通过inode的size大小不仅可以知道有没有使用间接指针，而且还可以知道间接指针所指向的直接指针使用了几个）。由于比较简单，所以也不在此进行赘述了。

5. 硬盘读写

因为使用块设备来模拟硬盘，所有的读写操作都是需要依靠disk_read 和disk_write()来完成，以block作为最小读写单位。在这种情况下无论是读取inode，读取目录项还是读取indirect ptr块中的一个直接指针，都需要一定的特殊处理。

已知inode编号，读取一个inode的方法如下图所示。首先跟据一个块所能容纳的inode 的数量来计算这个inode处于哪一个块之中，然后再计算这个inode是这个块之中的第几个inode。声明一个块大小的char数组buf，将这个块读入到buf之中，在对buf进行类型转换为inode*类型即可。

```
int inode_index=file_inodeno/INODE_PER_BLOCK+INODE_BASE;
int inode_offset=file_inodeno%INODE_PER_BLOCK;
char buf[BLOCK_SIZE+1];
memset(buf,0,BLOCK_SIZE);
disk_read(inode_index,buf);
struct inode* inode_ptr=(struct inode*)buf;
```

读取一个目录项的方法为首先初始化一个buf，然后使用一个结构体directory_entry 指针，将buf进行类型转换：

```
char buf[BLOCK_SIZE+1];
memset(buf,0,BLOCK_SIZE);
disk_read(dir_inode.direct_ptr[i]+DATA_BASE,buf);
struct directory_entry *dir_ptr = (struct directory_entry *)buf;
```

读取一个indirect ptr所对应的块之中所有的指针如下图所示，由于指针实际上是unsigned short，所以需要将ptr_buf进行类型转换成为unsigned short类型的指针。

```
unsigned short *pptr;
char ptr_buf[BLOCK_SIZE + 1];
memset(ptr_buf, 0, BLOCK_SIZE);
disk_read(dir_inode.single_ptr+DATA_BASE, ptr_buf);
pptr = (unsigned short *)ptr_buf;
```

这三个硬盘读写的接口由于跟上下文联系非常密切，如果用函数来封装的话会需要传递较多的参数，而且还会增加函数调用的代价。所以这三个没有封装成为函数的形式，一定程度上导致代码重复率较高，代码量比较庞大。

功能函数设计

功能函数设计就是指实验要求中fs_getattr fs_readdir fs_read fs_write等函数的具体实现，许多实现中都依靠了前面介绍的通用函数。

1. mkfs() & fs_getattr()

mkfs进行初始化工作，主要的任务是是构造一个statvfs的结构体并将这个结构体写入到super block之中，设置根目录的inode 并及时修改inode map。在调试的过程中，往往可以在mkfs的时候先在根目录底下写几个文件，用于读和写的调试。

fs_getattr()也相对比较简单，只需要调用path2inode（）函数，由路径得到inode编号，在读出inode，将其中的信息对应地填入到stat结构体之中即可。如果fs_getattr()没有找到的话，需要返回-ENOENT。这一点是特别重要的，因为，创建新文件是必须由-ENOENT来触发的。当操作涉及到一个没有创建过的文件的时候，依然会先调用fs_getattr()，这个时候返回-ENOENT，就会出发mk_dir 或者是mk_nod操作。如果不返回的话，会报错找不到需要的文件。

2. fs_readdir()

读目录其实是一种遍历工作，同fs_read()的实现原理大致上是一样的，只不过fs_readdir()相对于fs_read()的特殊之处在于fs_readdir读取的是目录，由此需要对内容按照目录项的格式进行解析；而且读取的参数是固定的，从头开始读offset=0，读取全部的内容。

遍历需要分成两部分，对直接指针的遍历和对间接指针块的遍历。首先跟据路径，使用函数path2inode（）找到目录对应的inode，遍历它的direct ptr数组，如果仍未读完，就读取出single ptr对应的block，一个一个地读取其中的直接指针，再把直接指针指示的块读出来扫描遍历一遍。

最后，所读取目录的inode需要修改其ctime，atime等属性，需要在修改完之后将inode 写回到原来的位置。

3. fs_read() & fs_write()

这两个函数其实同fs_readdir()是差不多的，他们的本质都是“遍历”。不过，由于读写模式的不同，所以并不是全部遍历，而是从offset处开始“遍历”地读或者写size个byte。而所有地“遍历”扫描问题，其实都需要考虑从直接指针到间接指针的过渡问题。

最初实现的时候想专门就多种情况进行分类讨论：①从直接指针开始读/写，结束于直接指针块；②从直接指针开始读/写，结束于间接指针的指针块；③从间接指针开始读/写，最终结束于间接指针.....

后来进行代码优化的时候，发现这样的讨论使得代码的复杂度增加许多，同时会含有许多的隐含bug，所以改变了代码思路，不分类讨论，而是整体一个流程，首先跟据offset，计算是从第几个块开始读或者写的，从直接指针开始进行，如果没有完成或者offset已经超出了直接指针的容量，就从间接指针的指针块开始，不再分类讨论终止于什么位置，而是读/写完规定的size之后就终止。

4. fs_mknod() & fs_mkdir()

这两个功能本质是一样的，所以自己构造了一个函数mkdir_mknod(const char *path,int is_dir)两个函数基本都是依靠直接调用这个函数来完成的。

主体分为三个部分，第一个部分是首先构造新文件的inode，在inode map之中寻找一个空闲的inode，将mtime，ctime等各种内容初始化，跟据inode的编号写回到对应的位置。第二个部分是在父目录之中添加一个目录项，记录名字的对应的inode编号。为了达到这一目的，首先需要找到目录之中的最后一个目录项并在其后添加。

需要考虑许多种情况：①最后一个目录项出现在indirect ptr部分，特殊地，最后一个目录项可能出现在最后一个块的尾部，这个时候需要为indirect ptr新增加一个direct ptr；②最后一个目录项出现在direct ptr的部分，一个特殊情况是这个目录项出现在一个块的结尾，需要新使用一个direct ptr；还有一个特殊情况是这个目录项恰好出现在最后一个direct ptr指向的块尾部，需要分配间接指针块并增加一个直接指针。

最后一个部分是读出父目录的inode，修改其中的ctime，mtime和size信息再次写回。

5. fs_rmdir() & fs_unlink()

第一步，从父目录之中删除该节点对应的目录项，这个功能由于以后还会用到，所以包装成了一个接口，命名为rm_name()。因为我们在目录之中查找某一个特定的目录项的时候是从头开始遍历查找的，找到一个位置的inode编号为(unsigned short)-1就认为已经查找完了并结束。所以在删除的时候是不能简单地将对应的位置内容置成(unsinged short)-1,而是找到目录之中的最后一个，用最后一个目录项的内容复制到要删除的目录项进行覆盖，并将最后一个目录项置为(unsigned short)-1。这样做有一些特殊的情况需要考虑，①该目录只有一个目录项；②要删除的目录项恰好就是最后一个。最后，修改更新父目录的inode并最终写回。

第二步，处理datablock。将这个文件inode之中全部的datablock，包括direct ptr以及indirect ptr之中的所有普通块都置为(unsigned short)-1，并在datablock map之中进行相应的处理。

第三步，处理inode，将inode所在的块读取出来，将这个inode 所在的位置全部置为(unsigned short)-1，然后再写回去。

6. fs_rename()

第一步，解析原有的路径，获得父目录以及原有的文件名，从父目录之中删除该节点对应的目录项，这一步同上述fs_unlink()的第一步是完全相同的，只需要再次调用rm_ptr()函数即可。

第二步，解析新的路径，获取新的父目录以及新的文件名。同readdir一样，在父目录之中遍历一遍，查找有没有重名的文件，如果有重名的文件，就自动修改新的文件名从而进行区别。

第三步，在新的父目录底下添加一个新的目录项。要添加一个新的目录项，首先是要找到最后一个，然后在后边追加。需要考虑几种情况:①最后一个目录项处于indirect ptr之中。特殊的，恰好处于最后一个datablock的尾部，需要新添加一个datablock；②最后一个目录项位于direct ptr之中。一种情况是处于一个block的尾部，需要新动用一个direct ptr；一种情况是处于第11个datablock的尾部，需要开始使用indirect ptr并分配块；还有一种情况是当前的目录为空，所以需要为该目录对应的inode分配一个direct block

第四步，处理新的父目录的inode，修改ctime、mtime、尺寸等信息。

7. fs_truncate() & fs_utime() & fs_statfs()

fs_truncate()首先根据文件的路径找到其对应的inode，然后分为两种情况进行处理：尺寸增大和尺寸减小。这两种情况分别计算需要增加的块和需要减少的块的数量，然后交由专门的函数接口add_ptr()和rm_ptr()进行处理即可,最后将inode进行写回。

fs_utime()根据文件路径找到对应的inode。读取上来之后修改其中的尺寸，atime，ctime，mtime等信息，然后写回即可。

fs_statfs()的作用是为了了解文件系统整体的情况。直接读取super block，i将其中的信息返回即可。

遇到的问题 & 解决方案

1. 由于硬盘被封装成了块结构，所以只能整块的读和写。读上来的内容一定是(void*)类型的，需要进行指针类型转换。我写过的一个bug是因为指针的加法是考虑了随指向对象的size的，而不是一个字节一个字节地加。
2. 读硬盘之前我往往都是声明一个char buf[BLOCK_SIZE+1]，然后用这个buf来存硬盘上某一块的内容。但我曾经写过的一个bug是将一个buf反复读取了多个硬盘的内容，读取上来之后进行类型转换后就赋给了一个指针，以为只要有这个指针就可以了，结果把buf用新的内容覆盖掉了。
3. 一个inode之中的direct_ptr和single_ptr存的到底是什么容易搞混。一个写过的bug是因为分不清是全局的第几个块，还是datablock区的第几个块，两者在数值上相差了644
4. 对于inode的操作往往有很多函数来完成，这些函数会修改inode并写回。用完这些函数之后，如果还需要使用inode，就必须重新读取一遍，否则数据的一致性会出现问题。
5. 无论是删除一个文件，还是rename一个文件，将他从父目录之中删除之前，一定要先获得他的inode，这是很重要的，如果顺序反了，就会找不到inode了。
6. 在使用14.sh进行测试的时候，遇到了一个很奇怪的bug，调用find_free_block()函数在return之前使用printf检验没有错误，但是就是无法正常的return回来。后来在Menci的帮助下，才发现这种情况一般是因为在函数里边写内存的时候有越界现象，从而导致了函数的返回地址被覆盖，返回的时候跳到了奇怪的地方导致的bug
7. 由于弄错了结构体的尺寸导致过一次bug，结构体的size是要加上padding的，而且很多不常见的类型，如mode_t,off_t等等，在window上和linux上的尺寸大小是不一致的。所以不能在windows本机上用sizeof输出单个类型的大小，放到服务器上是不一样的！