lab4拓展实验实验报告

PB17111636 徐宇鸣

实验目标

1、支持长文件名的FAT 16文件系统 在之前的实验中，我们只关注了短文件名(8+3格式)的实现，这种这种方式有很大的局限性。 要求:参考相关资料，实现能够支持长文件名的FAT文件系统，要求能够正常显示fat.img中的文件名。 2、支持写操作的FAT 16文件系统 在已有的只读文件基础上，实现可支持创建文件，写文件，删除文件的功能，同时能够支持文件属性的修改。 要求:能够在文件系统中支持touch，mkdir，cp，rm等命令，支持的功能越多越好。 3、实现线程安全的FAT 16文件系统 要求:在多线程访问文件系统时，需要保证读写请求被正确处理，能够使用fio工具测试文件系统，并且有不错的性能结果。

实验主要步骤

1、设计出支持读写的文件系统 2、根据设计方案实现simple_fat16.c的代码 3、挂载fuse，根据测试方案进行调试修改

实验内容

实现了

1、支持：cp、mkdir、rm、touch等命令 2、长文件名文件的显示、创建、搜索 3、线程安全

没有完成

1、根目录目录项创建数量达到上限时的错误处理 2、簇分配达到上限时的错误处理 3、对删除过的目录项空间的再利用 4、长目录项的间隔放置（实现这一点其实需要在实现了3的基础上才能完成）

所构想出来的函数

```
void sector_read(FILE *fd, unsigned int secnum, void *buffer)
向buffer返回扇区secnum的数据
void sector_write(FILE *fd, unsigned int secnum, void *buffer)
向扇区secnum返回buffer
char **path_split(char *path, int *pathDepth_ret)
输入带'/'的path，返回分解后pathDepth_ret层的paths
char *path_format(char *path,char flag)
根据flag的值将path转化成短文件目录名
WORD fat_entry_by_cluster(FAT16 *fat16_ins, WORD ClusterN)
返回fat表项
void first_sector_by_cluster(FAT16 *fat16_ins, WORD ClusterN, WORD
*FatClusEntryVal, WORD *FirstSectorofCluster, BYTE *buffer)
返回了buffer以及该簇的第一个扇区号
```

```
char *get_long_filename(int j, BYTE* buffer, BYTE* buffer_cache,char
*shortname)
```
根据shortname得到chknum然后从当前扇区以及逻辑上连续的前一个扇区中找到长文件名
```
int find_root(FAT16 *fat16_ins, DIR_ENTRY *Dir, const char *path)
```
找到返回0，没找到返回1，填充Dir
```
int find_subdir(FAT16 *fat16_ins, DIR_ENTRY *Dir, char **paths, int pathDepth,
int curDepth)
```
找到返回0，没找到返回1，填充Dir
```
int fat16_readdir(const char *path, void *buffer, fuse_fill_dir_t filler,
                  off_t offset, struct fuse_file_info *fi)
```
根据路径显示文件目录项
```
int fat16_read(const char *path, char *buffer, size_t size, off_t offset,
               struct fuse_file_info *fi)
```
根据路径寻找文件并读取
```
int fat16_mkdir(const char *path, mode_t mode)
```
根据路径寻找文件应该所在位置然后创建
```
int fat16_mknod(const char *path, mode_t mode,dev_t dev)
```
根据路径寻找文件应该所在位置然后创建
```
void format_date_time(WORD *date_format,WORD *time_format)
```
生成符合规格的时间和日期
```
int fat16_write(const char *path, const char *buffer, size_t size, off_t
offset, struct fuse_file_info *fi)
```
根据路径寻找文件并写入
```
int fat16_create(const char *path, mode_t mode, struct fuse_file_info *fi)
```
根据路径寻找文件应该所在位置然后进入
```
int fat16_rmdir(const char* path)
```
删除文件夹
```
void Create_shortname_Dir(char *shortname, BYTE mode,DIR_ENTRY* Dir)
```
根据输入的符合规格的短文件名以及mode填充短目录项Dir
```
void Create_longname_Dir(char *path,char *shortname,LDIR_ENTRY* LDir,int*
LDirCnt)
```
根据输入的path以及短文件名填充长文件名目录项Dirs，并返回长文件名目录项个数
```
WORD find_empty_fat()//同时在这将该簇的四个扇区都初始化了（注：这一步可以改，因为write函
数只需要找新簇号，不需要初始化，而其他对目录文件进行写操作的需要初始化）
```
寻找空闲的fat表项并返回簇号
```
void change_fat()
```
根据簇号和需要填充的簇号改变fat
```
void clear_dir(WORD Cluster)
```
根据簇号进入扇区内将所有的簇释放
```
void clear_fat(WORD ClusterN)
```
根据簇号进入fat表改变
```
int fat16_unlink(const char* path)
```
删除文件
```
int fat16_utimens(const char* path,const struct timespec tv[2])
```
修改文件访问时间

实现3个目标分别使用的方法

1、实现长文件名 首先是在fat16.h中多定义了一个新的结构体

```
typedef struct
{
    BYTE LDIR_Ord;
    BYTE LDIR_Name1[10];
    BYTE LDIR_Attr;
    BYTE LDIR_Type;
    BYTE LDIR_Chksum;
    BYTE LDIR_Name2[12];
    WORD LDIR_FstClusLO;
    BYTE LDIR_Name3[4];
}__attribute__ ((packed))LDIR_ENTRY;
```

搜索：根据chknum判断该文件是否与短文件名目录项匹配，同时根据偏移量为0处的值判断该长目录项是否是最后一个长目录项 创建：

```
void Create_longname_Dir(char *path,char *shortname,LDIR_ENTRY* LDir,int*
LDirCnt)
```

输入文件名、短文件名、往LDir填充长目录项，并在LDirCnt中写入长目录项个数，当找到一个目录项以0x00开头后，说明目录已经搜索到了结尾，此时就将后面的LDirCnt+1（包括短目录项）的区域填充，考虑了跨扇区跨簇，未考虑上限

```
if(sector_buffer[32*j]==0x00)//发现了一个空的目录项，插进去
            {
                free(pathc);
                pathc = path_format(paths[0], flag);
                Create_shortname_Dir(pathc,ATTR_DIRECTORY , &Dir);
                Create_longname_Dir(paths[0], pathc, LDir, &LongDirCnt);
                //todo:进入目录内设置.和..两个目录
                //这里没有考虑根目录项数不够的情况
                //查看是否跨扇区
                if((j+LongDirCnt)<BYTES_PER_SECTOR/32)
                {
                    memcpy(sector_buffer+32*(j+LongDirCnt),&Dir,32);
                    LongDirCnt--;
                    flag=0;
                }
                else
                {
                    sector_read(fat16_ins-
>FirstRootDirSecNum+RootDirCnt,sector_buffer_temp);//取下一个扇区
                    i = (j+LongDirCnt)%BYTES_PER_SECTOR/32;
                    LongDirCnt = LongDirCnt-i-1;
                    memcpy(sector_buffer_temp+32*i,&Dir,32);
                    i--;
                    flag=0;
                    while(i>=0)
                    {
```

```
                    memcpy(sector_buffer_temp+32*i,&LDir[flag],32);
                    flag++;
                    i--;
                }
                sector_write(fat16_ins->FirstRootDirSecNum+RootDirCnt,
sector_buffer_temp);
            }
            while(LongDirCnt>=0)
            {
                memcpy(sector_buffer+32*(j+LongDirCnt),&LDir[flag],32);
                LongDirCnt--;
                flag++;
            }
            sector_write(fat16_ins->FirstRootDirSecNum+RootDirCnt-1,
sector_buffer);
            first_sector_by_cluster(fat16_ins, Dir.DIR_FstClusLO,
&FatClusEntryVal, &FirstSectorofCluster, sector_buffer);
            //设置. ..
            Create_shortname_Dir(s[0],ATTR_DIRECTORY , &Dir);
            memcpy(sector_buffer,&Dir,32);
            Create_shortname_Dir(s[1],ATTR_DIRECTORY , &Dir);
            Dir.DIR_FstClusLO = 0x0000;
            memcpy(sector_buffer+32,&Dir,32);
            sector_write(FirstSectorofCluster, sector_buffer);
            //free掉
            free(pathc);
            for(i = 0;i<pathDepth;i++)
            {
                free(paths[i]);
            }
            free(paths);
            return 0;
        }
```

删除： 删除操作做的不够好，由于本次没有实现长目录项的间隔放置，我做的删除操作就是判断出应该有多少个长目录项，然后将短目录项的前n个目录项进行删除

```
if(strcmp(name,paths[0])==0)
        {
            temp = strlen(name)+1;
            temp = (temp+13-1)/13+1;//求出所有的长目录项

            while(j>=0&&temp>0)
            {
                sector_buffer[32*j] = 0xe5;
                temp--;
                j--;
            }
```

```
                sector_write(fat16_ins->FirstRootDirSecNum+RootDirCnt-1,
sector_buffer);
                if(temp>0)
                {
                    j=1;
                    while(temp>0)
                    {
                        sector_buffer_temp[BYTES_PER_SECTOR-32*j] = 0xe5;
                        temp--;
                        j++;
                    }
                    sector_write(fat16_ins->FirstRootDirSecNum+RootDirCnt-2,
sector_buffer_temp);
                }
                //进入文件夹内搜索所有的文件，将所有的分配的簇号清空
                clear_dir(ClusterN);
                clear_fat(ClusterN);
                for(i = 0;i<pathDepth;i++)
                {
                    free(paths[i]);
                }
                free(paths);
                return 0;
            }
```

2、文件的写 写方面与read函数无太大区别，就是先找出所需要写的第一个扇区（其应该是该文件的第几个簇（可能非连续），在过程中如果超过了文件所分配的簇号就分配新的簇，然后进行写，使用一个total变量来判断是否写结束

```
if(find_root(fat16_ins, &Dir, path,&DirSecNum,&DirNum)==0)
    {
        ClusterN = Dir.DIR_FstClusLO;
        ClusterNum = (offset / fat16_ins->Bpb.BPB_BytsPerSec)/fat16_ins-
>Bpb.BPB_SecPerClus;
        SecNum = (offset / fat16_ins->Bpb.BPB_BytsPerSec)%fat16_ins-
>Bpb.BPB_SecPerClus;
        SecOffset = offset % fat16_ins->Bpb.BPB_BytsPerSec;
        temp = 0;
        FatClusEntryVal =  fat_entry_by_cluster(fat16_ins, ClusterN);
        while(temp<ClusterNum)
        {
            if(FatClusEntryVal == 0xffff)//需要再跳一个簇然而已经没有分配新的簇了
            {
                FatClusEntryVal = find_empty_fat();//找个新的簇
                change_fat(fat16_ins,ClusterN,FatClusEntryVal);
            }
            ClusterN = FatClusEntryVal;
            FatClusEntryVal =  fat_entry_by_cluster(fat16_ins, ClusterN);
            temp++;
```

```
        }
        FirstSectorofCluster = ((ClusterN - 2) * fat16_ins-
>Bpb.BPB_SecPerClus) + fat16_ins->FirstDataSector ;
        sector_read(FirstSectorofCluster+SecNum, sector_buffer);
        temp = BYTES_PER_SECTOR - SecOffset;
        if(temp > size)
        {
            memcpy(sector_buffer+SecOffset,buffer,size);
            sector_write(FirstSectorofCluster+SecNum, sector_buffer);
        }
        else
        {
            memcpy(sector_buffer+SecOffset,buffer,temp);
            total = temp;
            sector_write(FirstSectorofCluster+SecNum, sector_buffer);
            SecNum++;
            while(total < size)
            {
                if(SecNum == fat16_ins->Bpb.BPB_SecPerClus)//整个簇都读完了
                {
                    ClusterNum = FatClusEntryVal;
                    if(ClusterNum == 0xffff)//空的
                    {
                        ClusterNum = find_empty_fat();//找个新的簇
                        change_fat(fat16_ins,ClusterN,ClusterNum);
                    }
                    ClusterN = ClusterNum;

 first_sector_by_cluster(fat16_ins,ClusterN,&FatClusEntryVal,&FirstSectorofClu
ster,sector_buffer);
                    SecNum = 1;
                }
                else
                {
                    sector_read(FirstSectorofCluster+SecNum, sector_buffer);
                    SecNum++;
                }
                temp = (size - total > BYTES_PER_SECTOR)?BYTES_PER_SECTOR:
(size - total);
                memcpy(sector_buffer,buffer+total,temp);
                sector_write(FirstSectorofCluster+SecNum-1, sector_buffer);
                total +=temp;
            }
        }
        //改变目录项
        Dir.DIR_FileSize +=size;
        sector_read(DirSecNum, sector_buffer);
        memcpy(sector_buffer+32*DirNum,&Dir,32);
        sector_write(DirSecNum, sector_buffer);
```

```
        return size;
    }
```

3、线程安全读写 本次实验所做的是填写fuse的operation这一个结构体所要求的函数，而这一个实现其实原本目的就是为了给不了解内核知识的人所用的，所以在其外部已经实现了多线程安全的读写锁，因此我们所需要做的就是使我们内部的函数都满足可重入要求，或者给无法实现可重入的函数加锁 在fuse.c文件中的try_get_path函数中，存在一段实现了读写锁的代码（除此之外上下还有queue_element_unlock、queue_element_wakeup等函数）

```
static int try_get_path(struct fuse *f, fuse_ino_t nodeid, const char *name,
      char **path, struct node **wnodep, bool need_lock)
{
  unsigned bufsize = 256;
  char *buf;
  char *s;
  struct node *node;
  struct node *wnode = NULL;
  int err;

  *path = NULL;

  err = -ENOMEM;
  buf = malloc(bufsize);
  if (buf == NULL)
    goto out_err;

  s = buf + bufsize - 1;
  *s = '\0';

  if (name != NULL) {
    s = add_name(&buf, &bufsize, s, name);
    err = -ENOMEM;
    if (s == NULL)
      goto out_free;
  }
//重点在这
  if (wnodep) {
    assert(need_lock);
    wnode = lookup_node(f, nodeid, name);
    if (wnode) {
      if (wnode->treelock != 0) {
        if (wnode->treelock > 0)
          wnode->treelock += TREELOCK_WAIT_OFFSET;
        err = -EAGAIN;
        goto out_free;
      }
      wnode->treelock = TREELOCK_WRITE;
    }
  }
```

```
  for (node = get_node(f, nodeid); node->nodeid != FUSE_ROOT_ID;
       node = node->parent) {
    err = -ENOENT;
    if (node->name == NULL || node->parent == NULL)
      goto out_unlock;

    err = -ENOMEM;
    s = add_name(&buf, &bufsize, s, node->name);
    if (s == NULL)
      goto out_unlock;

    if (need_lock) {
      err = -EAGAIN;
      if (node->treelock < 0)
        goto out_unlock;

      node->treelock++;
    }
  }

  if (s[0])
    memmove(buf, s, bufsize - (s - buf));
  else
    strcpy(buf, "/");

  *path = buf;
  if (wnodep)
    *wnodep = wnode;

  return 0;

 out_unlock:
  if (need_lock)
    unlock_path(f, nodeid, wnode, node);
 out_free:
  free(buf);

 out_err:
  return err;
}
```

而在所有会改变文件的函数中，都存在着使用这一段的代码，这里举rmdir为例子

```
static void fuse_lib_rmdir(fuse_req_t req, fuse_ino_t parent, const char
*name)
{
  struct fuse *f = req_fuse_prepare(req);
  struct node *wnode;
```

```
  char *path;
  int err;
//这里
  err = get_path_wrlock(f, parent, name, &path,
&wnode);
//
  if (!err) {
    struct fuse_intr_data d;

    fuse_prepare_interrupt(f, req, &d);
    err = fuse_fs_rmdir(f->fs, path);
    fuse_finish_interrupt(f, req, &d);
    if (!err)
      remove_node(f, parent, name);
    free_path_wrlock(f, parent, wnode, path);
  }
  reply_err(req, err);
}

static int get_path_wrlock(struct fuse *f, fuse_ino_t nodeid, const char
*name,
         char **path, struct node **wnode)
{
  return get_path_common(f, nodeid, name, path, wnode);
}


static int get_path_common(struct fuse *f, fuse_ino_t nodeid, const char
*name,
         char **path, struct node **wnode)
{
  int err;

  pthread_mutex_lock(&f->lock);
  //这里
  err = try_get_path(f, nodeid, name, path, wnode, true);
  //
  if (err == -EAGAIN) {
    struct lock_queue_element qe = {
      .nodeid1 = nodeid,
      .name1 = name,
      .path1 = path,
      .wnode1 = wnode,
    };
    debug_path(f, "QUEUE PATH", nodeid, name, !!wnode);
    err = wait_path(f, &qe);
    debug_path(f, "DEQUEUE PATH", nodeid, name, !!wnode);
  }
  pthread_mutex_unlock(&f->lock);
```

```
  return err;
}
```

因为助教给的代码中使用了一个文件指针作为全局变量（文件指针无法进行复制），因此我们如果为了避免大量的加锁开销，我选择了在删除所有的文件指针，然后在sector_read和sector_write这两个函数里使用即时打开即时关闭的文件指针，这样就不会出现多线程会出现的一些bug，但是这里同时会增加一个开文件指针的开销

```
void sector_read(unsigned int secnum, void *buffer)
{
    FILE *fd;
    fd = fopen(FAT_FILE_NAME, "rb");
    fseek(fd, BYTES_PER_SECTOR * secnum, SEEK_SET);
    fread(buffer, BYTES_PER_SECTOR, 1, fd);
    fclose(fd);
}
void sector_write(unsigned int secnum, void *buffer)
{
    FILE *fd;
    fd = fopen(FAT_FILE_NAME, "rb+");
    fseek(fd, BYTES_PER_SECTOR * secnum, SEEK_SET);
    fwrite(buffer, 1, BYTES_PER_SECTOR, fd);
    fclose(fd);
}
```

实现的逻辑

mkdir

1、查找上级目录：即/xxx/xxxx/dir的话就先查找/xxxx/xxxx的位置 2、进入上级目录，遍历上级目录先创建出短文件名，检查短文件名前6个是否相同，若相同： （1）：如果长文件名小于6个字节则返回创建失败，原因是同名 （2）：如果长文件名大于6个字节则记录"～N"的N，用于之后建立新的～N的依据 3、找到空白区域后，通过Create_longname_Dir和Create_shortname_Dir（该函数分配新的簇号）进行生成所需要的长目录和短目录 4、放置目录项： （4.1）该扇区放置不下所有的目录，则将缓冲扇区设置为下一个扇区，若该扇区已经为簇的最后一个扇区，则新开一个簇，并将该簇全部清零 按倒序放置下所有扇区 5、设置目录内内容 进入目录项，将该簇全部清零 将目录开头两项设置为"."和".." 6、释放所有资源 测试： 根目录： 1、扇区边界创建目录 ok 子目录： 1、扇区边界创建目录 ok 2、簇边界创建目录 ok

mknod

1、与mkdir一致 2、与mkdir一致 3、与mkdir一致 4、与mkdir一致 5、释放所有资源 测试： 根目录： 1、扇区边界创建目录 ok 子目录： 1、扇区边界创建目录 ok 2、簇边界创建目录 ok

rmdir

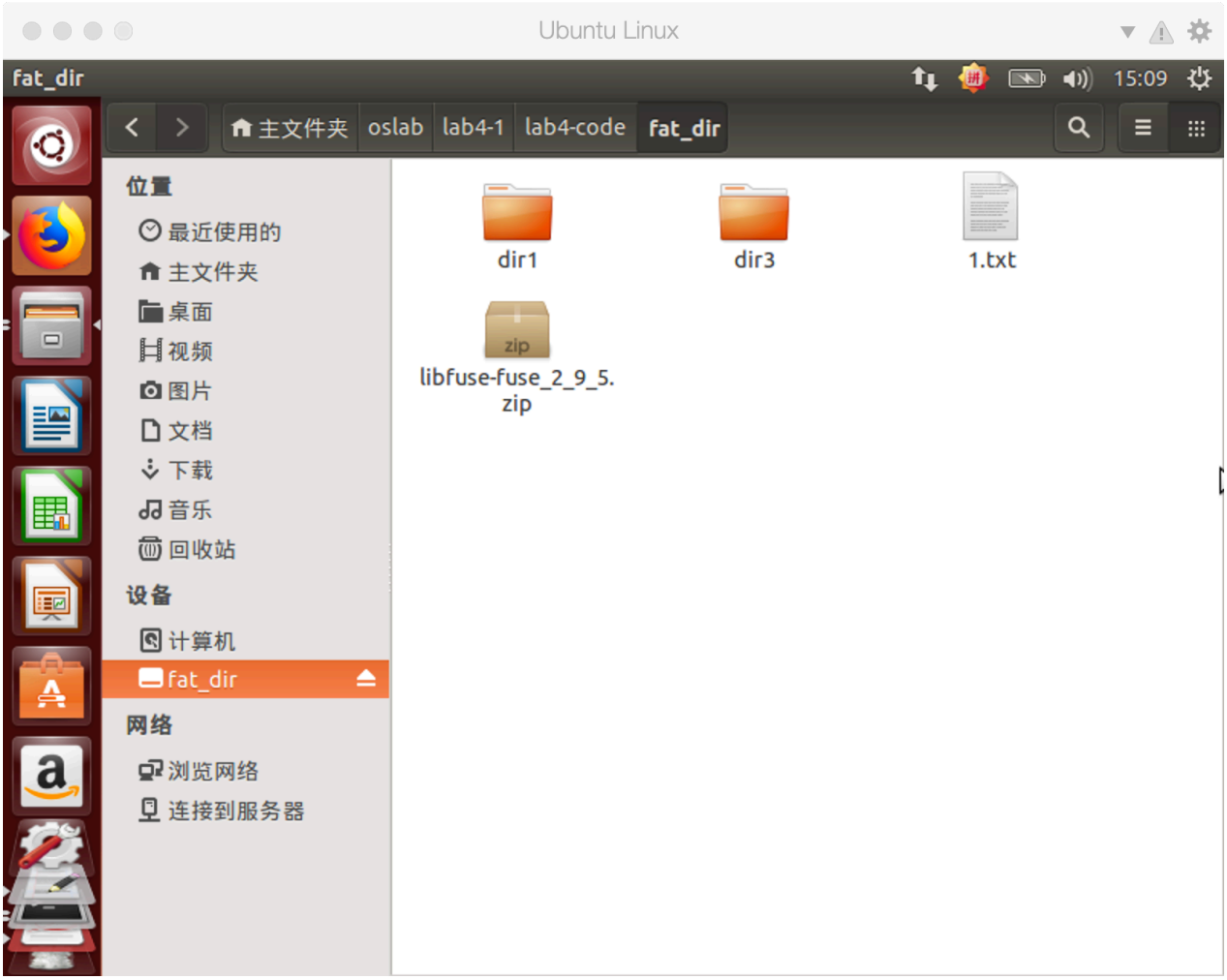
1、查找上级目录 2、根据路径搜索目录项 每找到一个短文件名目录项，就计算出他的chknum，并根据长文件目录项偏移量0处的信息来获取所有的长目录项以及记录他们所在的位置，之后放进get_longfile_name里翻译出长文件名，然后再与路径进行比对 3、找到目录项之后将所有的长目录项、短目录项的偏移量0处设为0xe5，然后记录下目录的首簇号后将首簇号清零 4、处理目录中的FAT表项 根据目录的首簇号进入目录，遍历目录 每碰到一个短文件名目录项，先判断其是目录还是文件 是文件就记录下其簇号然后将簇号清零，然后修改FAT表项 是目录就递归调用该函数 测试： 根目录： 1、删除空目录 ok 2、删除带文件带子目录的目录 ok 子目录： 1、删除空目录 ok 2、删除带文件带子目录的目录 ok

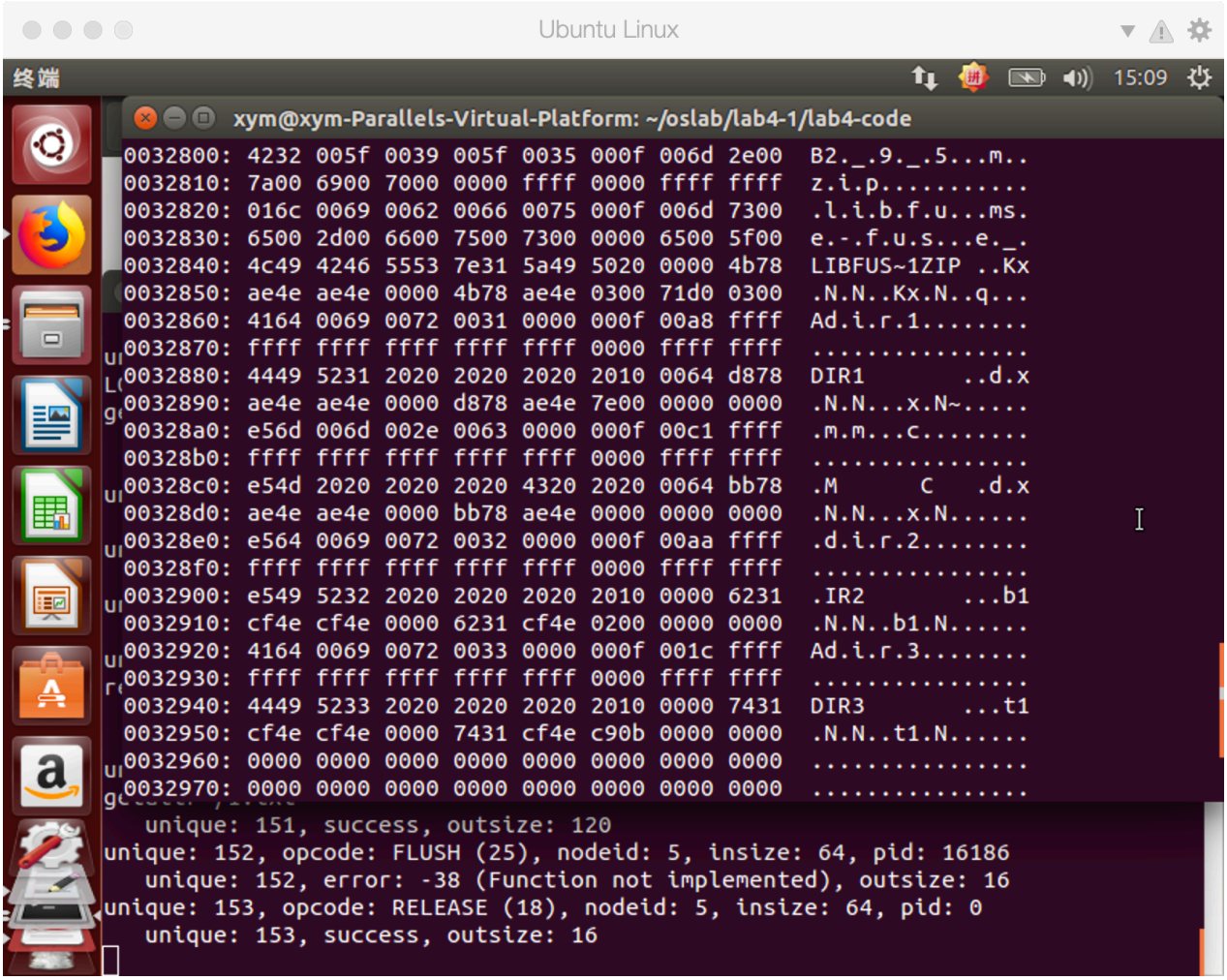
unlink

1、与rmdir一致 2、与rmdir一致 3、与rmdir一致 4、将其首簇号清零，处理文件的fat表项 测试 1、根目录删除文件 ok 2、子目录删除文件 ok

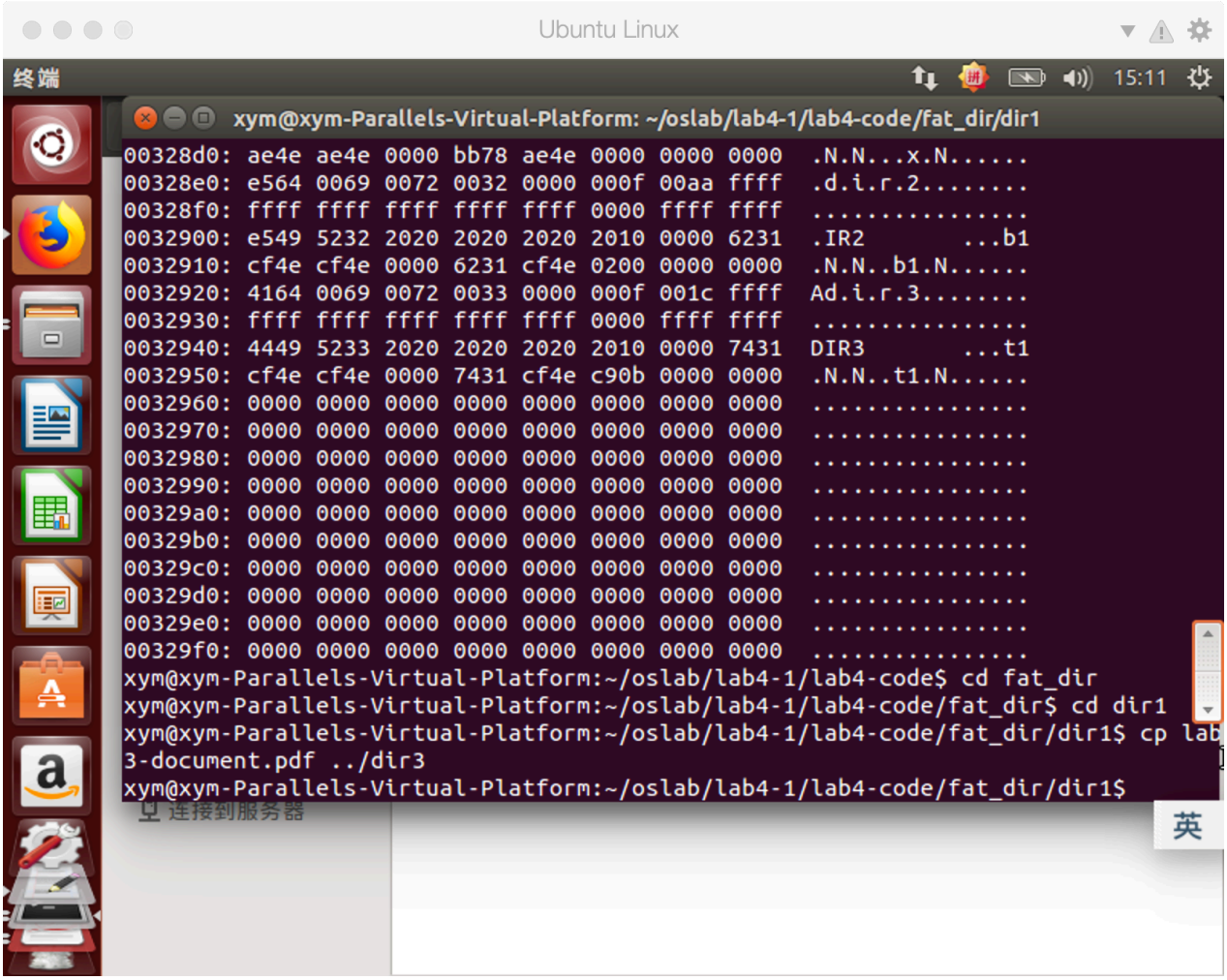
实验截图

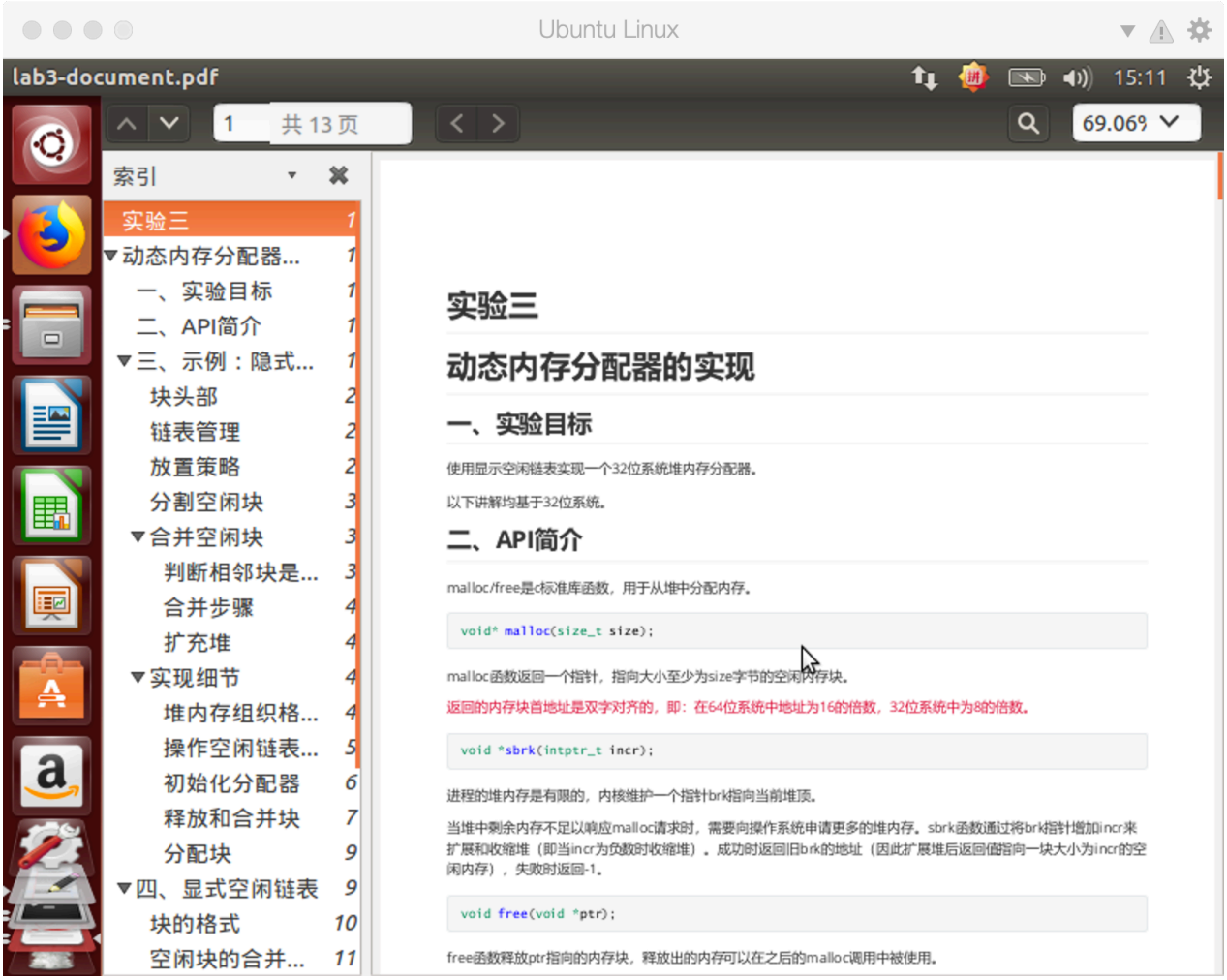
修改后的文件夹以及根目录的编码





执行了cp操作后的文件夹，以及可读的pdf


Ubuntu Linux
终端
15:11
xym@xym-Parallels-Virtual-Platform: ~/oslab/lab4-1/lab4-code/fat_dir/dir1
00328d0: ae4e ae4e 0000 bb78 ae4e 0000 0000 0000 .N.N...x.N......
00328e0: e564 0069 0072 0032 0000 000f 00aa ffff .d.i.r.2........
00328f0: ffff ffff ffff ffff ffff 0000 ffff ffff ....
0032900: e549 5232 2020 2020 2020 2010 0000 6231 .IR2 ...b1
0032910: cf4e cf4e 0000 6231 cf4e 0200 0000 0000 .N.N..b1.N......
0032920: 4164 0069 0072 0033 0000 000f 001c ffff Ad.i.r.3........
0032930: ffff ffff ffff ffff ffff 0000 ffff ffff ....
0032940: 4449 5233 2020 2020 2020 2010 0000 7431 DIR3 ...t1
0032950: cf4e cf4e 0000 7431 cf4e c90b 0000 0000 .N.N..t1.N......
0032960: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0032970: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0032980: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0032990: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00329a0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00329b0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00329c0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00329d0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00329e0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00329f0: 0000 0000 0000 0000 0000 0000 0000 0000 ....
xym@xym-Parallels-Virtual-Platform:~/oslab/lab4-1/lab4-code$ cd fat_dir
xym@xym-Parallels-Virtual-Platform:~/oslab/lab4-1/lab4-code/fat_dir$ cd dir1
xym@xym-Parallels-Virtual-Platform:~/oslab/lab4-1/lab4-code/fat_dir/dir1$ cp lab
3-document.pdf ../dir3
xym@xym-Parallels-Virtual-Platform:~/oslab/lab4-1/lab4-code/fat_dir/dir1$
连接到服务器
英



实验总结

不足

除了前面没有实现的几个功能以外，我想还有一些可以进行优化修改的内容 1、前面因为要读取连续的几个长文件名，由于实现的原因，需要考虑可能会存在跨扇区的几个长文件目录，因此我们需要使用一个缓存扇区来存储前一个扇区，在这里做的只是简单地在读取下一个扇区前将前一个扇区的内容读入一个tempbuffer中，而因此开销里会多一项memcpy，而我想到了一个比较好的方法，就是开始开辟两个buffer，奇数扇区时读到一个buffer，偶数扇区时读到另一个buffer，这样在检查扇区的时候前一个扇区一定在另一个buffer里，这样可以减少一些时间开销 2、在读取目录文件的时候，与其使用buffer去读取512个字节，不如开辟16个目录项，然后将信息全部读到目录项里，这样处理起来更加好懂（在时间上似乎不会有太多影响） 3、还有就是函数实现上的问题，因为本次实验是在lab4的基础之上修改的，有一些函数的功能其实在这一次实验上不是特别的好用，但是我在这里有的部分没有选择写新的函数，而是在原先基础上修改了一些函数，如果可以的话应该需要将模块化写的更好一点

感想

本次实验其实难度并不大，因为fuse的operation这组操作已经很明确地告诉你你应该完成的事情，而且其的包装函数方面也已经实现了多线程的读写安全（但是自己如果没有写好还是容易出现问题），但是从中我还是了解到了很多fat的实现方法，以及考虑了一些在文件系统实现中会遇到的问题，有些问题我的确是图轻松而跳过了，有些解决方法我不知道我自己是否是现在最常用的解决方法，总而言之实验整体还是很愉快的。