Project 4 FileSystem

作者：西安电子科技大学

2014.2.7

文件系统是操作系统的五大功能模块之一，主要实现操作系统对程序、数据、设备等的管理。

一、 当前 pintos 文件系统的功能

当前 pintos 文件系统已经实现了基本的文件创建删除功能。文件是固定大小，连续存放的。

(1) 文件在磁盘的存储方式：

每个文件都有一个 disk_inode 存放在磁盘的一个扇区上，其结构如下：

```
struct inode_disk
{
    block_sector_t start; //文件数据每一个起始块
    off_t length;         //文件长度。
    unsigned magic;       //
    uint32_t unused[125];
};
```

现在 pintos 文件是连续的，而且在创建时指定其大小后，再不能改变大小。由起始扇区和文件大小就能找到所有文件数据。

目录也是一个文件，只是其内容中都存放如下结构：

```
struct dir_entry
{
    Block_sector_t inode_sector;   //文件 inode_disk 所在扇区。
    Char name[NAME_MAX+1];
    Bool in_use;
};
```

(2) 磁盘空闲空间管理方式。

空闲空间是用位图来表示的。其中位图也是一个文件，其 disk inode 存放在 Sector 0.

位图文件大小显然与磁盘大小有关，用一个位表示一个扇区是否被分配，一个扇区 512 字节，一个扇区作为一个物理块，创建一个 2M 的磁盘，有 1024*1024*2/512bit=4096bit=4096/8=512Byte。

(3) 文件系统初始化过程：

在 init.c:main()函数中调用了 filesys_init();

1. 在 filesys_init()中，初始化了 bitmap,而且对磁盘进行了格式化。其中格式化就是创建了两个文件，一个用来管理空闲块的位置图文件，一个是根目录文件。
2. Filesys_init()中调用了 free_map_init(),在 free_map_init()中调用 bitmap_create()创建了位图，大小依据磁盘大小。而且标记了 0 1 两个扇区为已经分配，作为 free_map_file 的 disk_inode 空间和根目录文件的 disk_inode 空间。此时 free_map 只是在内存中。
3. Filesys_init()中又调用了 do_format()

在 do_format()中：调用 了 free_map_create()，创建了 free_map_file,即一个文件，其 disk_inode 已经在上面分配了，再分配文件大小即可，这是由调用 inode_create 函数来实现的。创建这个文件时显然需要分配磁盘空闲块，就从在内存中的 free_map 位图中分配就可以。文件创建好了之后，打开文件，把内存中的位图 free_map 写到磁盘上，这是调用 bitmap_write()实现的，本质还是用 file_write()实现的。File_write()是通过 inode_write_at()写磁盘的。

在 do_format()中还创建根目录文件，ROOT_DIR_SECTOR 已经标记了，分配文件所需要的空间就行了。

4. Filesys_init()调用了 free_map_open()把 free_map 读入内存。其实就是打开上面创建的 free_map 文件，读数据入内存。直到系统关闭时才将 free_map 写回到磁盘上。

   首先在内存中建立了磁盘的位图，标记了根目录和 free_map 本身的 disk_inode 结构扇区。

上面的内存 free_map 一建立就保证创建文件时可以分配磁盘空间了。而 inode_create()中就需要调用 free_map_allocate()来获取空闲物理块。如果不格式化，则以前必然格式化过，直接读入 free_map 就行。

(4) 目录创建过程。

Pintos 原来已经实现了创建目录的功能，但是只创建根目录，而且根目录只能包含 16 个文件，即 16 个 dir_entry 结构。

目录本质也是一个文件，根目录是特殊的文件，在 filesys.h 中有宏定义：

#define ROOT_DIR_SECTOR 1.

在 ROOT_DIR_SECTOR 中，也就是第 1 个扇区中存放了根目录的 disk_inode.上面已经提到了，在格式化时创建了根目录。

(5) 文件打开过程。

调用 filesys_open().

每个打开的文件在内在都维护了一个唯一的数据结构 inode(为了与 disk_inode 区别，这个叫 memory inode).以下是 memory inode 结构：

```
Struct inode
{
    Struct list_elem elem;
    Block_sector_t sector;
    Int open_cnt;
    Bool removed;
     Int deny_write_cnt;
    Struct inode_disk data;
};
```

打开一个文件显然就是在内存中创建一个 inode 结构，其中 struct inode_disk data 保存了该文件对应的磁盘中的 inode. 如果此文件已经打开了，则只需要增加 open_cnt 的值，不用再创建一个 inode. 这些 inode 通过一个链表链结到一起的，在 filesys_init()中初始化了这个链表。

具体过程：

调用 filesys_open(const char filename); 首先打开根目录，然后在根目录中依据 filename 查找文件。如果找到了，也就是找到了此文件对应的 struct dir_entry 结构，里面记录了文件 disk_inode, disk_inode 中记录文件数据起始位置和文件大小，这就可以读写文件了。当然，还需要建立文件对应的 memory inode.其中 data 就是此文件的

disk_inode.

通过 filesys_open()只能得到文件的 inode. 如果此文件是普通文件，则调用 fileopen(inode)把 inode 包装成 struct file 结构;如果是目录就调用 dir_open 把 inode 包装成 struct dir.

Struct file 结构如下：

```
Struct file
{
    Struct inode    *inode;
    Off_t    pos;
     Bool deny_write;
};
```

(6) 文件的创建过程.

创建一个文件要有文件名和文件大小，每一个文件在磁盘中都有一个 struct disk_inode 结构，每个文件和目录都要在一个目录下，在目录文件中，struct dir_entry 中记录了每个该目录下的文件的文件名，以及每个文件 disk_inode 所在扇区号。

具体是通过调用 filesys_create(const char *name,off_t initial_size)实现的.

首先调用 struct dir *dir =dir_open_root()打开根目录。以下 struct dir 结构。

```
Struct dir
{
   Struct inode *inode;
    Off_t pos
}；本质就是一个文件，读写目录与读写普通文件没有区别。
```

然后调用 free_map_allocate 分配一个扇区 sector 作为新文件的 disk_inode.

再调用 inode_create(sector,initial_size)分配文件所需要磁盘空间，分配的空间是连续的扇区。

最后创建一个 struct dir_entry 结构，把文件名和其 sector 填入其中，在目录中找一个位置放入就可以了。

(7) 文件的读写。

调用 filesys_open()打开文件，这就得到了 struct file 结构，如下：

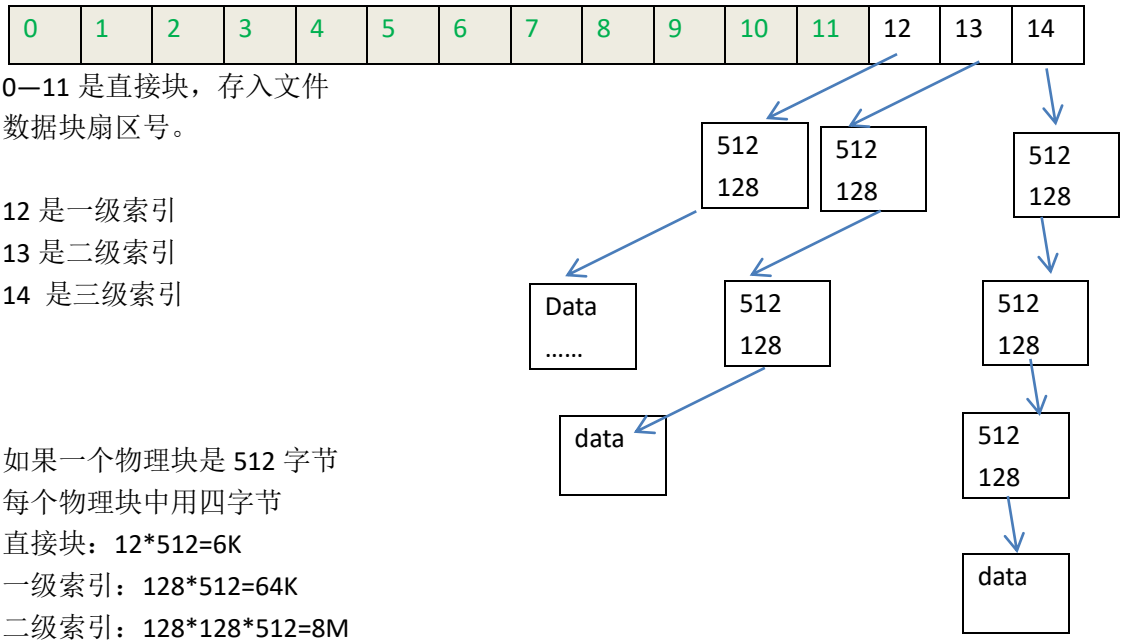
些结构记录了当前文件指针位置 pos。

File_read() file_write()分配是通过 inode_read_at()和 inode_write_at()实现的。

二、 对 disk_inode 的改进

当前 pintos 文件系统限制很大。文件需要连续存储，这会导致大量磁盘碎片。文件大小固定，不能动态增长文件，只有一个目录。这里主要把连续存储方式改为了 linux 中三级索引结构，而且可以动态增长文件，可以创建子目录。

三级索引结构：



0—11 是直接块，存入文件
数据块扇区号。

12 是一级索引
13 是二级索引
14 是三级索引

如果一个物理块是 512 字节
每个物理块中用四字节
直接块：12*512=6K
一级索引：128*512=64K
二级索引：128*128*512=8M
三级索引：128*128*128*512=1GB
总大小 1GB8M72K

通过修改 struct disk_inode 结构来实现三级索引结构。修改过的 disk_inode

```
Struct inode_disk
{
    Off_t length;                    //文件长度
    Uint32_t blocks[BLOCK_NUM];      //三级索引区
    Unsigned isdir;                  //此文件是不是目录
    Unsigned magic;
    Uint32_t unused[110];
};
```

修改 inode_create()
Inode_read_at()
Inode_write_at()
三个函数即可。

空闲磁盘块依然用位图管理。可以用 free_map_allocate()获得一个空闲物理块。
用 free_map_release()来释放物理块。只用写文件才有可能增长一个文件的大小。
读写文件一般是给出.

1. 改变了 disk_inode，自然要从 inode_create 开始入手。代码见附录。
   因为要区分目录和普通文件，所以对原来的 inode_create 进行了扩展，改为了 bool inode_create_ex (block_sector_t sector, off_t length,uint32_t isdir);增加了 isdir 参数。重新定义了一个函数：

```
bool inode_create(block_sector_t sector,off_t length)
{
    return inode_create_ex(sector,length,0);
```

}

用这个来代替原来的 inode_create。

Inode_create 执行时文件的 disk_inode 已经分配了，只需要分配文件数据空间，这里并不从 free_map 中分配空间,只是简单的把索引初始化为 0。新创建的文件内容应为全 0，所以调用了 inode_write_at()来把文件内容写为全 0;在写的过程中如果发现空间未分配，则会分配。

2. 读写文件

读写文件时会给出文件指针偏移 offset 和要读写的大小。需要根据 offset 来确定要读写的扇区，通过上面的公式可以计算出扇区号。计算代码见附录。

定义如下结构：

```
struct PosInfo
{
    uint16_t lev;   //索引级别
    uint32_t off;   //数据扇区内部偏移
    uint32_t sn[4]; //sn[i]表示 i 级索引块的扇区号
    uint32_t np[4];//np[i]表示 i 级索引块的偏移.
};
```

根据 offset 可以计算出 offset 在几级索引中----lev

最后的数据块中的偏移 -------off

举例：如果 offset=215364 size=865 buff=…

要从文件 215364 字节处读取 865 字节到 buff 中。

经计算：

Lev=2

Sn[0]=0 blocks[15]

Np[0]=13

0	…	11	12	13	14

Sn[1]=0

Np[1]=2

Sn[2]=0

Np[2]=24

由 np[0]=13 可以通过读取扇区 blocks[np[0]]到缓冲区 arr[512]中去

此时 arr[]中就是一级索引块

Arr[np[1]]就是二级索引块的扇区号，由此又可以把二级索引块读入到 arr[]中。

此时 arr[]就是二级索引块，

Arr[np[2]]就是文件数据块的扇区号了，把数据块读入到 arr[]中就可以对数据进行读写了。

这个扇区读写完毕之后，offset 也向后移动了 x 字节，这时又需要重新确定 struct PosInfo 结构。

注意：写的时候，如果最后 offset 已经大小了文件长度，则认为扩展了文件长度，修改 disk_inode 中的文件长度。这就实现了文件的动态增长。

3. inode 删除

需要收回文件空间，显然有了 struct PosInfo 结构，收回空间不是难事儿。代码见附

录。

三、 多级目录的建立

现有目录只是根目录，而如今要实现多级目录，而且像 linux 系统一样，有一个环境变量 pwd 来存放当前目录。新目录支持如： . ./ .. ../ ../../ /
等操作，而且允许用户程序切换当前目录、打开目录、建立目录、读目录，都是通过系统调用 system call 实现的，以下列举了要实现的系统调用：

(1) Bool chdir(const char *dir)
切换当前目录。其中 dir 中可能有 /a/b/c/ ../../c/d 这样的字符串。
需要字符串处理，最后打开目录，如果失败则切换失败，否则切换成功。

(2) Mkdir(const char *dir)
创建目录，如：/a/b/c 在/a/b/目录下建立 c 目录。
调用附录中的 DirCreate()创建目录。

(3) readdir(int fd,char *name)
将目录当作普通文件来读，但是不能写。这使得 file_write 中判断一下，如果是目录则写入失败。

(4) isdir(int fd)
由文件句柄判断此文件是不是目录文件。可以根据 disk_inode 中的 isdir 变量来判断。

(5) inumber
返回一个数来区分是目录还是文件，通过在文件 disk_inode 变量区分，所以目录和文件可以统一分配空间。

无论是切换目录还是创建目录，以及在目录下打开文件，创建文件，删除文件，都需要打开目录，要有一个目录字符串处理函数，于是 char * MakePath(const char *from)应用而生。这个函数把 pwd 和 from 相结合，去掉其中../之类的符号，转化为标准的纯绝对路径，如:/a/b/c/d /a/b/c/d/wyg.cpp。

目录确定就能打开目录，struct dir *OpenDir(char *path,int *pos)函数来实现此功能。这又是一个字符串处理函数，可以用‘/’作为区分。

无论是切换还是创建目录，都可以用打开目录来确定是否给出的目录合法。创建目录时，要指定 disk_inode 中的 isdir 为 true. 在 struct thread 中加入指针 char *pwd;来保存当前目录，使用时，如果 pwd 为 NULL,则认为是根目录‘/’;如果需要改变当前目录，则需要申请空间，线程退出时会释放掉空间。

删除目录时要注意，目录是否为空，可以逐个验证每个 struct dir_entry 中的 bool use; 变量来判断该目录下是否有目录或文件。

目录是在磁盘上的，磁盘是多个进程所共享的，对目录的操作也需要同步。主要在打开目录，删除目录时需要获得相应的锁。最简间的办法是给整个磁盘加锁，事实上允许一个进程操作某个文件或目录时允许其他进程也操作相同或者是不同的目录。所以打开目录后就释放锁。

四、 buffer cache 系统设计

为了加快对磁盘的读写，在内存开辟出一定大小的空间作为磁盘缓冲区。于是需要设计一套缓冲系统。有的 VM 中的缓冲区设计的基础，设计这个就简单多了。

分析 pintos 对块设备的读写可以发现，把有的外部块设备的读写都是通过 block.c 中的 block_read()函数和 block_write()函数来实现的。在这两个函数中，在实际操作物理磁盘前，先在 cache 中查找要读写的块，如果在 cache 中，则直接从 cache 中读写，否则从磁盘调入 cache 再读写。这就有可能要从 cache 中淘汰出一个扇区。采用 LRU 算法可以有效的进行淘汰。

具体实现：

在内存开辟了 64 个扇区的空间作为 cache.

```
struct BlockCache
{
  size_t SecNo;       //扇区号
  bool Use;           //该空间是否被占用
  unsigned Num; //LRU 中使用
  bool Dirty;       //该扇区有没有被写过
  struct lock Lock;
};
struct Sec         //一个扇区的空间
{
  unsigned char data[512];
};
unsigned int PassTime=0;
bool Inited=false;
struct Sec SecArr[CacheSize]; //这里是 64 个扇区的空间
struct BlockCache ManArr[CacheSize]; //每个扇区对应一个 BlockCache 结构，用于管理些扇区。
```

(1) cache 空间分配与回收

在 ManArr[]数组中，初始化为全未使用，当需要调入一个磁盘扇区时就可以依据 Used 变量找一个未使用的内存扇区。

(2) 使用情况统计

每次读写磁盘时就把每个块的 Num 变量加 1，当前读写的扇区的 num 清 0.

(3) 需要淘汰扇区时，选择 num 最大的，即最久未使用的扇区淘汰出去。

(4) 读写时就在 ManArr[]中顺序查找要读写的块，找到就可以直接读写，否则需要从磁盘调入。

(5) 同步问题。磁盘缓冲系统出多线程共享，所以在调入调出时都需要同步。共享的是每个内存扇区，所以可以给每个扇区加一把锁，这就是 BlockCache 中的 lock 的作用。

(6) 每隔一段时间需要把缓冲中的数据写回到磁盘上去，所以另创建了一个线程，这个线程每隔一定时间把 cache 中的块都写回到磁盘中去。

附录中有 cache 的实现代码。

五、 Pintos 文件系统与外界的交换方法

pintos 如果把虚拟机外的文件复制到文件系统中的，又如何把把文件系统中的东西复制到虚拟机外的？

这是由 filesys/ fsutil.c 中的函数完成的。

文件 1 的文件头	文件 1 的数据…	…..	…	文件 2 的文件头	数据…..	….	文件 3

文件在 pintos 系统之外时被复制到了一个虚拟硬盘中，其格式是顺序结构。文件名等文件信息在一个扇区中，紧接着就是文件数据部分。其存储结构如下：

上面一个方格代表一个扇区。

Pintos 启动前由外部程序把文件一个一个的按如上格式装入一个磁盘。Pintos 启动后再从这个磁盘中读取文件，并在 filesystem 中建立相关的文件。

附录：

1. Inode create 代码：

```
bool inode_create(block_sector_t sector,off_t length)
{
  return inode_create_ex(sector,length,0);
}
bool
inode_create_ex (block_sector_t sector, off_t length,uint32_t isdir)
{
   struct inode_disk *disk_inode = NULL;
   bool success = false;

   ASSERT (length >= 0);

   /* If this assertion fails, the inode structure is not exactly
        one sector in size, and you should fix that. */
   ASSERT (sizeof *disk_inode == BLOCK_SECTOR_SIZE);

   disk_inode = calloc (1, sizeof *disk_inode);
   if (disk_inode != NULL)
     {
       size_t sectors = bytes_to_sectors (length);
       disk_inode->length = length;
    disk_inode->isdir=isdir;
       disk_inode->magic = INODE_MAGIC;
    int i;
    for(i=0;i<BLOCK_NUM;i++)       //15 个索引初始化为 0，表示未分配空间
         disk_inode->blocks[i]=0;
    success=true;
       static char zeros[BLOCK_SECTOR_SIZE];
   struct inode ie;
   ie.deny_write_cnt=0;
   ie.data=*disk_inode;
    for (i = 0; i < sectors; i++)
    {
            size_t    minb=length<512?length:512;
            inode_write_at(&ie,zeros,minb,i*512);   //新文件，内容全 0,写的
            length-=512;                            //过程中会分配空间给文件。
    }
   *disk_inode=ie.data;
       block_write (fs_device, sector, disk_inode); //保存 disk_inode
       free (disk_inode);
```

```
            }
          return success;
    }
```

2. 由文件偏移计算出所在扇区号：

```
    #define index0 (12*512)
    #define index1 (index0+128*512)
    #define index2 (index1+128*128*512)
    #define index3 (index2+128*128*128*512)
    struct PosInfo
    {
      uint16_t lev;
      uint32_t off;

      uint32_t sn[4];
      uint32_t np[4];
    };
    int    GetPos(struct PosInfo *pi,off_t off)
    {
      if(off<0||off>=index3) return 1;
      if(off>=index2)
      {
            off-=index2;
            pi->lev=3;

            pi->sn[0]=0;
            pi->np[0]=14;

            pi->sn[1]=0;
            pi->np[1]=off/(128*128*512);
            off-=pi->np[1]*(128*128*512);

            pi->sn[2]=0;
            pi->np[2]=off/(128*512);
            off-=pi->np[2]*(128*512);

            pi->sn[3]=0;
            pi->np[3]=off/512;
            pi->off=off%512;
      }
      else if(off>=index1)
      {
            off-=index1;
            pi->lev=2;
```

```
            pi->sn[0]=0;
            pi->np[0]=13;

            pi->sn[1]=0;
            pi->np[1]=off/(512*128);

            off-=pi->np[1]*(512*128);

            pi->sn[2]=0;
            pi->np[2]=off/512;

            pi->off=off%512;
        }
        else if(off>=index0)
        {
            off-=index0;
            pi->lev=1;

            pi->sn[0]=0;
            pi->np[0]=12;

            pi->sn[1]=0;
            pi->np[1]=off/512;

            pi->off=off%512;
        }
        else
        {
            pi->lev=0;
            pi->sn[0]=0;
            pi->np[0]=off/512;
            pi->off=off%512;
        }
        return 0;
    }
```

3. 文件读写：

```
    /* Reads SIZE bytes from INODE into BUFFER, starting at position OFFSET.
       Returns the number of bytes actually read, which may be less
       than SIZE if an error occurs or end of file is reached. */
    off_t
    inode_read_at (struct inode *inode, void *buffer_, off_t size, off_t offset)
    {
      off_t begin_offset=offset;
```

```
  uint8_t *buffer = buffer_;
  off_t bytes_read = 0;
//   ==================wyg add====================================
  struct PosInfo pi;
  uint32_t *arr=NULL;
          arr=(uint32_t *)malloc(BLOCK_SECTOR_SIZE);
  off_t cur_read;
  while(size>0)
  {

  if(GetPos(&pi,offset)!=0)          goto des1;
  cur_read= 512-pi.off<size?512-pi.off:size;
  if(pi.lev>=0)
  {
    uint32_t nsec=inode->data.blocks[pi.np[0]];
    if(nsec!=0)
    {
        pi.sn[0]=nsec;
        block_read(fs_device,nsec,(void *)arr);
    }
    else
        goto des1;
  }
  if(pi.lev>=1)
  {
    uint32_t nsec=arr[pi.np[1]];
    if(nsec!=0)
    {
          pi.sn[1]=nsec;
          block_read(fs_device,nsec,(void *)arr);
    }
    else
        goto des1;

  }
  if(pi.lev>=2)
  {
    uint32_t nsec=arr[pi.np[2]];
    if(nsec!=0)
    {
        pi.sn[2]=nsec;
        block_read(fs_device,nsec,(void *)arr);
    }
    else goto des1;
```

```
    }
    if(pi.lev>=3)
    {
      uint32_t nsec=arr[pi.np[3]];
      if(nsec!=0)
      {
            pi.sn[3]=nsec;
            block_read(fs_device,nsec,(void*)arr);
      }
      else
            goto des1;
    }
ASSERT(pi.lev<4);
  memcpy(buffer+bytes_read,(void *)arr+pi.off,cur_read);
  size-=cur_read;
  offset+=cur_read;
  bytes_read+=cur_read;
  continue;
des1:
  memset(buffer+bytes_read,0,cur_read);
  bytes_read+=cur_read;
  size-=cur_read;
  offset+=cur_read;
}
free(arr);
off_t ShouldRead=inode->data.length-begin_offset;
if(ShouldRead<0)
  ShouldRead=0;
if(ShouldRead<bytes_read)
  bytes_read=ShouldRead;
return bytes_read;
//==================end==================================
}

/* Writes SIZE bytes from BUFFER into INODE, starting at OFFSET.
     Returns the number of bytes actually written, which may be
     less than SIZE if end of file is reached or an error occurs.
     (Normally a write at end of file would extend the inode, but
     growth is not yet implemented.) */
off_t
inode_write_at (struct inode *inode, const void *buffer_, off_t size,
                off_t offset)
{
   const uint8_t *buffer = buffer_;
```

```
  off_t bytes_written = 0;
  uint8_t *bounce = NULL;

  if (inode->deny_write_cnt)
    return 0;
  struct PosInfo pi;
  uint32_t *arr=NULL;
  //while(arr==NULL)
arr=(uint32_t *)malloc(BLOCK_SECTOR_SIZE);
  uint32_t cur_write;
  while(size>0)
  {
    GetPos(&pi,offset);
    cur_write=512-pi.off<size?512-pi.off:size;
    if(pi.lev>=0)
    {
        uint32_t nsec=inode->data.blocks[pi.np[0]];
        if(nsec==0)
        {
            if (!free_map_allocate (1, &nsec))
            goto des1;
          inode->data.blocks[pi.np[0]]=nsec;
          memset(arr,0,512);
        }
        else
            block_read(fs_device,nsec,arr);
        pi.sn[0]=nsec;
    }
    if(pi.lev>=1)
    {
        uint32_t nsec=arr[pi.np[1]];
        if(nsec==0)
        {
            if(!free_map_allocate(1,&nsec))
                goto des1;
            arr[pi.np[1]]=nsec;
            block_write(fs_device,pi.sn[0],arr);
            memset(arr,0,512);
        }
        else
            block_read(fs_device,nsec,arr);
        pi.sn[1]=nsec;
    }
    if(pi.lev>=2)
```

```
    {
        uint32_t nsec=arr[pi.np[2]];
        if(nsec==0)
        {
            if(!free_map_allocate(1,&nsec))
                goto des1;
            arr[pi.np[2]]=nsec;
            block_write(fs_device,pi.sn[1],arr);
            memset(arr,0,512);
        }
        else
            block_read(fs_device,nsec,arr);
        pi.sn[2]=nsec;
    }
    if(pi.lev>=3)
    {
        uint32_t nsec=arr[pi.np[3]];
        if(nsec==0)
        {
            if(!free_map_allocate(1,&nsec))
                goto des1;
            arr[pi.np[3]]=nsec;
            block_write(fs_device,pi.sn[2],arr);
            memset(arr,0,512);
        }
        else
            block_read(fs_device,nsec,arr);
        pi.sn[3]=nsec;
    }
    if(pi.lev>=4)
        goto des1;
    memcpy((void *)arr+pi.off,buffer+bytes_written,cur_write);
    block_write(fs_device,pi.sn[pi.lev],arr);
    bytes_written+=cur_write;
    offset+=cur_write;
    size-=cur_write;
  }
des1:
  if(offset > inode->data.length)
  {
      inode->data.length=offset;
block_write(fs_device,inode->sector,&inode->data);
  }
  //lock_release(&inode->xlock);
```

```
//    lock_release(&inode->slock);
    free(arr);
    return bytes_written;
}
```

4. inode 删除代码：为里用 arr brr crr 来增加删除速度，不过效果不佳。

```
void
inode_close (struct inode *inode)
{
  /* Ignore null pointer. */
  if (inode == NULL)
    return;

  /* Release resources if this was the last opener. */
  if (--inode->open_cnt == 0)
    {
      /* Remove from inode list and release lock. */
      list_remove (&inode->elem);

      /* Deallocate blocks if removed. */
      if (inode->removed)
        {
//-----------------------------------------------------------------
          int i;
          for(i=0;i<12;i++)
                if(inode->data.blocks[i]!=0)
                      free_map_release(inode->data.blocks[i],1);
          if(inode->data.blocks[12]!=0)
          {
                unsigned *arr=(unsigned *)malloc(512);
                ASSERT(arr!=NULL);
                block_read(fs_device,inode->data.blocks[12],arr);
                int j;
                for(j=0;j<128&&arr[j]!=0;j++)
                            free_map_release(arr[j],1);
                free(arr);
                free_map_release(inode->data.blocks[12],1);
          }
          if(inode->data.blocks[13]!=0)
          {
                unsigned *arr=(unsigned *)malloc(512);
                unsigned *brr=(unsigned *)malloc(512);
                ASSERT(arr!=NULL&&brr!=NULL)
                block_read(fs_device,inode->data.blocks[13],arr);
                int j,k;
```

```
                for(j=0;j<128&&arr[j]!=0;j++)
                {
                    block_read(fs_device,arr[j],brr);
                    for(k=0;k<128&&brr[k]!=0;k++)
                            free_map_release(brr[k],1);
                    free_map_release(arr[j],1);
                }
                free_map_release(inode->data.blocks[13],1);
                free(arr);
                free(brr);
            }
            if(inode->data.blocks[14]!=0)
            {
                unsigned *arr=(unsigned *)malloc(512);
                unsigned *brr=(unsigned *)malloc(512);
                unsigned *crr=(unsigned *)malloc(512);
                ASSERT(arr!=0&&brr!=0&&crr!=0);
                block_read(fs_device,inode->data.blocks[14],arr);
                int j,k,l;
                for(j=0;j<128&&arr[j]!=0;j++)
                {
                    block_read(fs_device,arr[j],brr);

                        for(k=0;k<128&&brr[k]!=0;k++)
                        {
                            block_read(fs_device,brr[k],crr);
                            for(l=0;l<128&&crr[l]!=0;l++)
                                free_map_release(crr[l],1);
                            free_map_release(brr[k],1);
                        }

                    free_map_release(arr[j],1);
                }
                free_map_release(inode->data.blocks[14],1);
                free(arr);
                free(brr);
                free(crr);
            }
        free_map_release(inode->sector,1);
          }
           free (inode);
         }
}
```

5. 目录字符串处理函数：

```
void xstrcpy(char *to,const char *from)
{
  while(*to++=*from++);
}
char * MakePath(const char *from)
{
  //ASSERT(to!=NULL);
  const char *pwd=GetPwd();
  char *to;
  int lf=strlen(from);
  int len=lf;
  if(pwd!=NULL)
        len+=strlen(pwd);
  to=(char *)malloc(len+5);//！注意这里，没有释放内存
  if(to==NULL)
  {
        printf("error\n");
        return 0;
  }
  if(pwd==NULL)
        xstrcpy(to,"/");
  else
  xstrcpy(to,pwd);
  int lp=strlen(to);
  int pos=lp-1;
  if(to[lp-1]!='/')
  {
        to[lp]='/';
        to[lp+1]=0;
  }
  if(from[0]=='/')
  {
        //xstrcpy(to,from);
        //return to;
        pos=0;
  }

  char pre=0;
  int i;
  for(i=0;i<lf;i++)
  {

        if(from[i]=='.'&&pre=='.')
```

```
        {
            if(pos>0)pos--;
            while(pos>0&&to[pos]!='/')
                pos--;
        }
        else if(from[i]=='/')
        {
            if(to[pos]!='/')
                to[++pos]=from[i];
        }
        else if(from[i]=='.'&&pre==0);

        else if(pre=='/'&&from[i]=='.');
        else
        to[++pos]=from[i];
        pre=from[i];
    }
    to[++pos]=0;
    return to;

}
```

6. 目录打开函数 打开一个目录，返回 struct dir 结构

```
struct dir *OpenDir(char *path,int *pos)
{
  lock_acquire(&DirOpenLock);
  struct dir *dir=dir_open_root();
  struct inode *inode=NULL;
  int cur=1,i,n;
  n=strlen(path);
  for(i=1;i<n;i++)
        if(path[i]=='/')
        {
            path[i]=0;
            dir_lookup(dir,path+cur,&inode);
            dir_close(dir);
            if(inode==NULL)
                goto end;
            if(inode->data.isdir!=1)
                goto end;
            dir=dir_open(inode);
            inode=NULL;
            cur=i+1;
        }
```

```
    *pos=cur;
    lock_release(&DirOpenLock);
    return dir;

  end:
    lock_release(&DirOpenLock);
    return NULL;
}
```

7. 几个系统调用的实现：

```
void IMkDir(struct intr_frame *f)
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    char *DirName=*((int *)f->esp+1);
  int n=strlen(DirName);
  if(DirName[n-1]=='/')
  {
      DirName[n-1]=0;
      n--;
  }
  if(n<=0)
    {
    f->eax=0;
    return;
    }
  f->eax=DirCreate(DirName);
}
void IChDir(struct intr_frame *f)
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    char *DirName=*((int *)f->esp+1);
  char *path=MakePath(DirName);
  int n=strlen(path);
  if(path[n-1]!='/')
  {
      path[n++]='/';
      path[n++]='a';
      path[n]=0;
  }
  char *tpath=malloc(n+1);
  ASSERT(tpath!=NULL);
  xstrcpy(tpath,path);
  int cur=0;
```

```
  struct dir *dir=OpenDir(path,&cur);
  if(dir==NULL)
  {
        f->eax=0;
        free(path);
        return;
  }
  dir_close(dir);
  while(tpath[n]!='/')
        tpath[n--]=0;
  struct thread *curthr=thread_current();
  if(curthr->pwd!=NULL)
  free(curthr->pwd);
  n=strlen(tpath);
  curthr->pwd=malloc(n+1);
  xstrcpy(thread_current()->pwd,tpath);
  f->eax=1;
  free(path);
  free(tpath);
}
void IReadDir(struct intr_frame *f)
{
      if(!is_user_vaddr(((int *)f->esp)+6))
         ExitStatus(-1);
      if((char *)*((unsigned int *)f->esp+5)==NULL)
            {
                  f->eax=-1;
                  ExitStatus(-1);
            }
      int fd=*((unsigned int *)f->esp+4);
  char *name=*((unsigned int *)f->esp+5);
      struct file_node *fp=GetFile(thread_current(),fd);
  if(fp->f->inode->removed)
  {
        f->eax=0;
        return ;
  }
  struct dir dirx;
  dirx.pos=fp->f->pos;
  dirx.inode=fp->f->inode;
  if(dir_readdir(&dirx,name))
        f->eax=1;
  else
        f->eax=0;
```

```
  fp->f->pos=dirx.pos;
}
void IInumber(struct intr_frame *f)
{
      if(!is_user_vaddr(((int *)f->esp)+2))
         ExitStatus(-1);
      int fd=*((int *)f->esp+1);
  struct file_node *fn=GetFile(thread_current(),fd);
  f->eax= fn->f->inode->sector;
}
void IIsDir(struct intr_frame *f)
{
      if(!is_user_vaddr(((int *)f->esp)+2))
         ExitStatus(-1);
      int fd=*((int *)f->esp+1);
  struct file_node *fn=GetFile(thread_current(),fd);
  f->eax= fn->f->inode->data.isdir;
}
```

8. 目录创建函数，系统调用 mkdir()就是调用此函数实现的。

```
bool
DirCreate(const char *name)
{
   block_sector_t inode_sector = 0;
//    struct dir *dir = dir_open_root ();
  char *path=MakePath(name);
  ASSERT(path!=0);
  int cur;
  struct dir *dir=OpenDir(path,&cur);
   bool success = (dir != NULL
                   && free_map_allocate (1, &inode_sector)
                   &&  inode_create_ex  (inode_sector,  20*sizeof(struct
dir_entry),1)
                   && dir_add(dir,path+cur, inode_sector));
   if (!success && inode_sector != 0)
      free_map_release (inode_sector, 1);
   dir_close (dir);
  free(path);
   return success;
}
```

9. Cache 磁盘缓冲实现代码

Cache.h:

```
#ifndef __CACHE_H__
#define __CACHE_H__
#include<stdio.h>
```

```
#include<string.h>
#include"threads/malloc.h"
#include"filesys/filesys.h"
#define CacheSize 64 //64 sectors used 32KB 8Pages
#include"devices/block.h"
extern unsigned int PassTime;
extern bool Inited;
void InitCacheMan(void);
void CacheRead(block_sector_t sector,void *buffer);
void CacheWrite(block_sector_t,const void *buffer);
int Fetch(block_sector_t sector);
int InCache(block_sector_t sector);
int Evict(void);
void WriteBack(int n);
void DestroyCacheMan(void);
void CountSec(int n);
void WriteAllBack(void);
#endif
```

Cache.c:

```
#include"cache.h"
#include<string.h>
#include"devices/block.h"
#include"threads/synch.h"
struct lock CacheLock;
struct BlockCache
{
  size_t SecNo;
  bool Use;
  unsigned Num;
  bool Dirty;
  //bool Locked;
  struct lock Lock;
};
struct Sec
{
  unsigned char data[512];
};
unsigned int PassTime=0;
bool Inited=false;
struct Sec SecArr[CacheSize];
struct BlockCache ManArr[CacheSize];
void InitCacheMan(void)
{
  int i;
```

```
  for(i=0;i<CacheSize;i++)
  {
        ManArr[i].SecNo=0;
        ManArr[i].Use=false;
        ManArr[i].Num=0;
        ManArr[i].Dirty=false;
        lock_init(&ManArr[i].Lock);
  }
  PassTime=0;
  lock_init(&CacheLock);
  Inited=true;
}

void CacheRead(block_sector_t sector,void *buffer)
{
  int n=InCache(sector);
  if(n==-1)
        n=Fetch(sector);
  lock_acquire(&ManArr[n].Lock);
  ASSERT(n!=-1);
  memcpy(buffer,SecArr[n].data,BLOCK_SECTOR_SIZE);
  CountSec(n);
  lock_release(&ManArr[n].Lock);
//      Fetch(sector+1);
}
void CacheWrite(block_sector_t sector,const void *buffer)
{
  int n=InCache(sector);
  if(n==-1)
        n=Fetch(sector);
  lock_acquire(&ManArr[n].Lock);
  memcpy(SecArr[n].data,buffer,BLOCK_SECTOR_SIZE);
  ManArr[n].Dirty=true;
  CountSec(n);
  lock_release(&ManArr[n].Lock);
//      Fetch(sector+1);
  //WriteBack(n);
}
int Fetch(block_sector_t sector)
{
  lock_acquire(&CacheLock);
  int i,n=-1;
  for(i=0;i<CacheSize;i++)
        if(ManArr[i].Use==false)
```

```
        {
                n=i;
                break;
        }
  if(n==-1)
        n=Evict();
  fs_device->ops->read(fs_device->aux,sector,SecArr[n].data);
  fs_device->read_cnt++;
  ManArr[n].Use=true;
  ManArr[n].SecNo=sector;
  ManArr[n].Num=0;
  ManArr[n].Dirty=false;
  lock_release(&CacheLock);
//      printf("Fetch run\n");
  return n;
}
int InCache(block_sector_t sector)
{
  int i;
  for(i=0;i<CacheSize;i++)
        if(ManArr[i].Use==true && ManArr[i].SecNo==sector)
                return i;
  return -1;
}



int Evict()
{
  int i,n=-1;
  unsigned int maxn=0;
  for(i=0;i<CacheSize;i++)
  {
  if(ManArr[i].Use==true&&ManArr[i].Num>=maxn)
        {
                maxn=ManArr[i].Num;
                n=i;
        }
  }
  lock_acquire(&ManArr[n].Lock);
  ASSERT(n!=-1);
  WriteBack(n);
  ManArr[n].Use=false;
  lock_release(&ManArr[n].Lock);
```

```
//      printf("Evict run\n");
  return n;
}
void WriteBack(int n)
{
  fs_device->ops->write(fs_device->aux,ManArr[n].SecNo,SecArr[n].data);
  fs_device->write_cnt++;
  ManArr[n].Dirty=false;
}
void DestroyCacheMan(void)
{
  WriteAllBack();
}
void CountSec(int n)
{
  int i;
  for(i=0;i<CacheSize;i++)
        if(ManArr[i].Use==true)
              ManArr[i].Num++;
  ManArr[n].Num=0;
}
void WriteAllBack(void)
{
  int i;
  for(i=0;i<CacheSize;i++)
        if(ManArr[i].Use==true&&ManArr[i].Dirty==true)
              WriteBack(i);
}
```