、

文件系统实习报告

杨东升 1400012898

2017.5.5

目录

内容一：总体概述

本实验主要是针对 Nachos 的文件系统的理解和修改。包括了文件系统的构建，创建、打开、删除文件，对文件头、目录的理解，对文件进行读写操作，以及多级目录、文件长度的扩展等。其次还有针对文件系统的同步互斥机制的理解和修改。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Challenge1	Challenge2
第一部分	Y	Y	Y	Y	Y		
第二部分	Y	Y					
第三部分						Y	Y

（总之就是都完成了）

具体 Exercise 的完成情况

第一部分：

Exercise1: 源代码阅读

阅读 Nachos 源代码中与文件系统相关的代码，理解 Nachos 文件系统的工作原理。

code/filesys/filesys.h 和 code/filesys/filesys.cc

code/filesys/filehdr.h 和 code/filesys/filehdr.cc

code/filesys/directory.h 和 code/filesys/directory.cc

code /filesys/openfile.h 和 code /filesys/openfile.cc

code/userprog/bitmap.h 和 code/userprog/bitmap.cc

1）关于 filesys.h 和 filesys.cc 文件：这两个代码文件定义了 nachos 文件系统的体系结构，以及在整个文件尺度上的操作。文件系统主要的实现原理是，通过一个位图和一个目录的管理，可以获取目录下的每一个文件的文件头的磁盘位置，也可以获取磁盘中那些扇区被分配，哪些是空闲的。这在 FileSystem 类中对应的字段是两个 OpenFile 对象，freeMapFile 和 directoryFile。

Nachos 文件系统主要实现的功能有创建文件，打开文件，删除文件，以及打印出文件信息等等。具体的函数实现方式如下：

构造函数用来构建文件系统，需要格式化，还需要建立一个位图，初始化为总扇区数，用来管理磁盘的扇区；再建立一个目录，初始化为最大文件容量（默认为 10）；然后建立对应的文件头，给两个文件（位图文件和目录文件）分配磁盘空间，然后将对应信息写入磁盘对应扇区；两个文件头默认放在 0 号和 1 号扇区，这样就将文件系统初始化了。

Create 函数用来创建文件，首先读取位图和目录（文件系统类中的两个 OpenFile 对象已经打开了这两个文件，因此可以直接读写信息），首先看目录中是不是有这个文件，没有继续；然后看位图中有没有一个空余扇区存放文件头，有则继续，没有则失败；然后看目录下文件是否满了，没有继续；接着看磁盘中是否有足够扇区存放文件中的内容，有则继续；最后文件头分配磁盘扇区，文件头写回，目录修改后也写回，位图文件也修改后写回。

Open 函数用来打开文件，返回一个 OpenFile 对象，实现方式很简单，就是从目录下找到对应文件的文件头所在扇区，然后根据这个扇区来建立一个 OpenFile 对象，并且返回。

Remove 函数基本就是 Create 函数的逆过程。

2）关于文件头定义文件 filehdr.h 和 filehdr.cc。这里定义了文件头的类，文件头包含了文件存储在磁盘上的有用信息，包括文件大小，文件占用的扇区数，对应的扇区号。函数则主要提供了分配和回收两个接口。

Allocate 函数表示分配磁盘扇区，首先根据文件大小确定扇区数，然后再从扇区的位图中找到对应的空闲扇区。Deallocate 表示回收，就是将对应的扇区从位图中划去。

3）directory.h 和 directory.cc 文件。这里定义了目录类和目录条目类。每一个目录条目可以存放一个文件，默认的目录下有 10 个条目，说明最多可以容纳 10 个文件。条目中记录了是否被使用、文件头的扇区号以及文件名。目录类主要就是一个条目数组，提供了查找、添加、删除等文件操作。

Find 和 FindIndex 就是通过文件名来查找文件，如果找到则返回文件头所在扇区号；Add 就是添加文件，首先找到一个可用的文件条目，然后根据给定的 newSector 参数记录文件头的扇区号；Remove 是删除文件，只要将对应的条目设置为未使用即可。

4）openfile.h 和 openfile.cc 文件，这里定义了打开文件的类。该类中主要包含了一个文件头，然后根据这个文件头就可以进行文件的读写操作了。

ReadAt 和 WriteAt 函数是读写的实现函数，其原理是根据文件指针的位置，找到文件在磁盘上的对应的扇区，然后再对应的扇区处，通过 synchDisk 的 ReadSector 和 WriteSector 函数进行读写操作。synchDisk 在文件 synchdisk.cc 中实现，它包装了磁盘读写操作，通过锁来实现互斥访问。磁盘操作实现在 disk.cc 中

5）bitmap.cc 和 bitmap.h 文件提供了位图这个工具。通过 find 函数可以找到一个未分配的位，并且把这个位标记为已分配。

Exercise2: 扩展文件属性

增加文件描述信息，如“类型”、“创建时间”、“上次访问时间”、“上次修改时间”、“路径”等等。尝试突破文件名长度的限制。

突破文件名长度只需要把宏定义 FileNameMaxLen 改得足够大。

我认为，路径应该加在 directoryEntry 类中，因为它是用来找到这个文件的途径；类型，各种时间应该放在 fileheader 类中，因为这些是文件本身的属性，与文件位置无关。

因此，我在 FileHeader 类中添加了以下一些字段：createTime（创建时间）、lastUseTime（上次访问时间）、lastModifyTime（上次修改时间）、fileType（文件类型）。DirectoryEntry 类中添加了以下一些字段：path（路径）。文件类型 0 表示空、1 表示字符文件、2 表示二进制文件。

加入这些字段后，注意把 filehdr.h 中的最大可表示扇区数修正，由于我加入了 4 个 longint 整型变量，所以要把系数从 2 改为 6：

```
#define NumDirect   ((SectorSize - 6 * sizeof(int)) / sizeof(int))
#define MaxFileSize     (NumDirect * SectorSize)
```

时间相关的变量的类型是 time_t 类型，include <time.h>头文件后，使用 time()函数可以获取当前时间，使用 asctime()函数可以以字符串方式显示时间。

在 openfile.cc 的构造函数中，我修改 hdr->lastUseTime 为当前时间。

```
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
    hdrSector = sector ;
    time (&hdr->lastUseTime) ;
    printf("useTime: %s\n", asctime(gmtime(&hdr->lastUseTime)));
}
```

在 filesys.cc 的 create 函数中，我修改 hdr->createTime 为当前时间；同时赋予文件类型。

```
else {
success = TRUE;
everthing worked, flush all changes back to disk
    time (&hdr->createTime) ;
    printf("createTime: %s\n", asctime(gmtime(&hdr->createTime)));
    hdr->fileType = 0 ;
    hdr->WriteBack(sector);
    directory->WriteBack(directoryFile);
    freeMap->WriteBack(freeMapFile);
```

在 openfile.cc 的 Write 函数中，我修改 hdr->lastModifyTime 为当前时间；

```
int
OpenFile::Write(char *into, int numBytes)
{
   int result = WriteAt(into, numBytes, seekPosition);
   seekPosition += result;
   time (&hdr->lastModifyTime) ;
   printf("modifyTime: %s\n", asctime(gmtime(&hdr->lastModifyTime)));
   return result;
}
```

测试运行时，我选择使用 nachos 自带的 fstest.cc 中的 PerformanceTest 函数，测试方法是使用-t 选项运行 nachos。需要特别注意的是，在编译时必须在 filesys 的 makefile 文件中第十一行去掉对 THREADS 的的定义，原因是 THREADS 部分在 main 中检查参数时，会去掉前两个参数，因此后面的模块的参数就丢失了。PerformanceTest 函数的主要功能是创建一个文件，并对之重复读写多次，直到写满文件。运行效果如下：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -t
useTime: Sun May  7 07:55:18 2017

useTime: Sun May  7 07:55:18 2017

Starting file system performance test:
Sequential write of 50000 byte file, in 10 byte chunks
TestFile: 20
name: TestFile
path: /
createTime: Sun May  7 07:55:18 2017

useTime: Sun May  7 07:55:18 2017

modifyTime: Sun May  7 07:55:18 2017
```

可以看到，各项时间都被正确地设置了。路径也是正确的。但是暂时读写还有一些问题，需要在后面的 lab 中处理。

Exercise3: 扩展文件长度

改直接索引为间接索引，以突破文件长度不能超过 4KB 的限制。

在这里简单地使用二级索引来解决问题，这样文件长度就可以突破 4KB 的限制。文件头中有 32-6=26 个扇区索引，这 26 个扇区索引又指向了 26 个子索引扇区，每个子索引扇区有 32 个索引。这样就一共有将近 1K 个索引，文件长度可以超过 100KB。

改变索引类型主要需要修改 Filehdr 类，即文件头。文件头中 dataSectors 的意义由指向存放文件数据的扇区号变成了指向子索引的扇区号。这样一来，在 Allocate 函数中，首先根据总的扇区数计算出需要的子索引扇区数 numSemiHeaders，然后判断总共需要的扇区数是否足够（包括子索引消耗的扇区数）。然后首先给子索引分配扇区，再在每一个子索引

中，给文件数据分配扇区。子索引所在的扇区就是代码中的 semiDirect，可以存放 NumSemiDirect=32 个子索引。每当 32 个子索引填满了这个扇区，就需要把这个扇区写回到磁盘中，然后再把后面的子索引写入下一个扇区。需要注意的是，最后如果没有填满一个扇区，也需要把这个记录着子索引的扇区写回到磁盘中。代码如下：

```
#define NumDirect   ((SectorSize - 6 * sizeof(int)) / sizeof(int))
#define NumSemiDirect 32
#define MaxFileSize     (NumDirect * SectorSize)

bool
FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors  = divRoundUp(fileSize, SectorSize);
    int numSemiHeaders = divRoundUp(numSectors, SectorSize);
    if (freeMap->NumClear() < numSectors + numSemiHeaders)
    return FALSE;       // not enough space

    for (int i = 0; i < numSemiHeaders; i++)
        dataSectors[i] = freeMap->Find();
    int idx, idy = 0, offset ;
    int semiDirect[NumSemiDirect] ;
    while( idy < numSectors )
    {
        idx = idy / NumSemiDirect ;
        offset = idy % NumSemiDirect ;
        semiDirect[offset] = freeMap->Find();
        idy ++ ;
        if( offset == NumSemiDirect - 1 )
            synchDisk->WriteSector(dataSectors[idx], (char*) semiDirect);
    }
    if( offset != NumSemiDirect - 1 )
        synchDisk->WriteSector(dataSectors[idx], (char*) semiDirect);
    return TRUE;
}
```

同样，在 Deallocate 函数中，首先要读取所有的子索引，对所有的子索引对应的扇区进行释放。然后再释放记录着子索引的扇区。特别需要注意的是，最后剩下的子索引可能不是一整个扇区的子索引，所以需要特殊对待一下。

```
void
FileHeader::Deallocate(BitMap *freeMap)
{
    int semiDirect[NumSemiDirect] ;
    int numSemiHeaders = divRoundUp(numSectors, SectorSize);
    for (int i = 0; i < numSectors; i++) {
        ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
        if( i != numSectors-1 )
        {
            synchDisk->ReadSector(dataSectors[i], (char*) semiDirect);
            for( int j = 0 ; j < NumSemiDirect; j ++ )
            {
                ASSERT(freeMap->Test(semiDirect[j])) ;
                freeMap->Clear(semiDirect[j]) ;
            }
        }
        else
        {
            int last = numSectors % NumSemiDirect ;
            synchDisk->ReadSector(dataSectors[i], (char*) semiDirect);
            for( int j = 0 ; j < last ; j ++ )
            {
                ASSERT(freeMap->Test(semiDirect[j])) ;
                freeMap->Clear(semiDirect[j]) ;
            }
        }
        freeMap->Clear((int) dataSectors[i]);
    }
}
```

在 ByteToSector 函数中，需要重新计算索引扇区，和字节偏移。先读出记录子索引的扇区，再根据新的偏移从其中找到对应的扇区返回。

```
int
FileHeader::ByteToSector(int offset)
{
    int idx = offset / SectorSize / NumSemiDirect ;
    int newoffset = (offset / SectorSize) % NumSemiDirect ;
    int semiDirect[NumSemiDirect] ;
    synchDisk->ReadSector(dataSectors[idx], (char*) semiDirect);
    return(semiDirect[newoffset]);
}
```

为了测试对大文件的支持，我在 test 文件夹中新建了一个 huge 文件，大小为 7kb，超过了原来支持的 4kb，然后我通过 nachos 的-cp 命令，调用 copy 函数把这个文件从 linux 拷贝到 nachos 中。实验结果如下页图所示。

从实验结果中可以看出 huge 文件使用了 51 个 sector，超过了原来 nachos 最大的 32 个 sector，程序运行正常，说明对大文件的支持成功了。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -cp test/huge huge_in_nachos
useTime: Sun May  7 13:14:10 2017

useTime: Sun May  7 13:14:10 2017

name: huge_in_nachos
path: /
numSectors: 51
createTime: Sun May  7 13:14:10 2017

useTime: Sun May  7 13:14:10 2017

Finish Thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 91945020, idle 91857830, system 87190, user 0
Disk I/O: reads 2187, writes 719
```

Exercise 4 实现多级目录

首先我们应该想清楚几个问题，那就是多级目录文件是什么样的文件，应该存在哪里，以及应该如何使用。我认为，多级目录中的每一个目录都是一个独立的文件，里面可以存 10 个普通文件或其他目录文件。根目录文件指针有一个特殊的名字叫 diretoryFile，并处在常开状态。其他目录文件与普通文件没有本质区别。这些目录文件和普通文件一起存放在磁盘扇区中。使用这些目录文件时，需要一级一级，通过当前目录的名字从父目录中找到当前目录的文件头扇区，再把文件内容导入到临时目录变量中，并且从这个临时目录再找下一级目录的头文件扇区，直到最后一级为止。

首先，我们需要对 Directory 类进行修改。我们添加一个 char* path 字段，在 diractory 构造函数中加入 path 参数，并且把 path 赋给 table 中每一项的 path。尽管此时 table 中还没有文件头，但是提前对 path 赋值是比较方便的。其实这个功能不会影响程序的正确性，因为目录的寻址等操作都不依赖这里的 path。但是这样从结构上来看会更加完整。代码如下：

```
Directory::Directory(int size, char * mpath)
{
    table = new DirectoryEntry[size];
    tableSize = size;
    this->path = new char[100] ;
    strcpy(this->path, mpath) ;
    for (int i = 0; i < tableSize; i++)
    {
        table[i].inUse = FALSE;
        table[i].sector = -1 ;
        strcpy(table[i].name, "NULL") ;
        strcpy(table[i].path, mpath) ;
    }
}
```

在 filesys.cc 中，要修改各个函数寻址文件头的方法。 需要在创建文件、打开文件和删除文件中给出路径。然后根据路径找到相应的目录进行文件操作。

先要添加一个工具函数 GetDirectoryFromPath()，功能是根据路径来找到相应的目录文件指针。这里假定一定能找到。该函数的实现如下：

```
void FileSystem::GetDirectoryFromPath(char* path, OpenFile *&open)
{
    int len = strlen(path) ;
    int idx = 0 ;
    Directory * directory = new Directory(NumDirEntries, "") ;
    directory -> FetchFrom(directoryFile) ;
    OpenFile * openfile = directoryFile ;
    while(idx < len)
    {
        int nowIdx = idx ;
        while( path[nowIdx] != '/' )
        {
            nowIdx ++ ;
        }
        char * name = new char[nowIdx-idx+1] ;
        for( int i = 0 ; i < nowIdx-idx ; i ++ )
        {
            name[i] = path[idx+i] ;
        }
        name[nowIdx-idx] = '\0' ;
        int sector = directory->Find(name) ;
        if( openfile != directoryFile ) delete openfile ; //注意
        if( sector >= 0 ) openfile = new OpenFile(sector) ;
        //printf("hdr: %d\n", openfile->hdrSector) ;
        directory -> FetchFrom(openfile) ;
        idx = nowIdx + 1 ;  //注意
        delete[] name ;
    }
    open = openfile ;
    delete directory ;
}
```

首先根据根目录的文件指针打开根目录文件，导入临时目录变量。然后对每一级目录重复下面操作：根据'/'字符分隔出下一级目录的名字；在临时目录变量中下找到下一级目录的文件头分区；打开下一级目录文件，关闭上一级目录文件，这里需要注意如果是根目录则不能关闭；从文件中导出目录到临时目录变量中，重复上述操作。还需要注意的一点是，需要把每一级目录之间的'/'跳过去。

最后，我们把找到的最后一级目录的文件指针赋给参数中文件指针的引用。需要注意的是，在程序中其他地方不能随意修改这个文件指针的内容，因为它是一个引用变量。

有了工具函数之后，我们修改现有的文件操作函数。Create 函数前面部分修改如下。可以看到，我们定义了一个 OpenFIle 指针，传入 GetDirectoryFromPath() 作为实参，另一个参数是文件想要创建在的路径。然后我们从 openfile 指针导入目录到临时目录变量。

```
bool
FileSystem::Create(char *name, int initialSize, char *path)
{
    Directory *directory;
    BitMap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;

    if( strcmp(path, "") ) printf("Creating file %s in %s\n\n", name, path);
    else printf("Creating directory %s in root\n\n", name);
    DEBUG('f', "Creating file %s, size %d\n", name, initialSize);

    directory = new Directory(NumDirEntries, path);
    OpenFile * openfile = NULL;
    GetDirectoryFromPath(path, openfile) ;
    directory->FetchFrom(openfile) ;
```

需要格外注意的是，添加了目录信息后，需要把目录写回到 openfile 文件。

```
        directory->WriteBack(openfile);
        freeMap->WriteBack(freeMapFile);
}
```

Open 函数修改如下，也是先找所在目录，再从这个目录打开文件。由于目录临时变量的 table 使我们唯一想要的，所以初始化目录临时变量时路径不重要。

```
OpenFile *
FileSystem::Open(char *name, char *path)
{
    Directory *directory = new Directory(NumDirEntries, path); //随意写
    OpenFile *openfile = NULL ;
    OpenFile *openFile = NULL ;
    int sector;
    printf("open path: %s\n", path) ;
    GetDirectoryFromPath(path, openfile) ;
    DEBUG('f', "Opening file %s\n", name);
    directory->FetchFrom(openfile); //
    sector = directory->Find(name);
    if (sector >= 0)
    openFile = new OpenFile(sector);    // name was found in directory
    delete directory;
    return openFile;                // return NULL if not found
}
```

remove 函数做和 create 类似的修改，这里不再赘述。还是需要注意最后目录文件要写回 openfile。

我们还需要添加一个创建目录的函数。我认为目录和普通文件没有区别，所以创建目录就是创建一个空文件。创建目录的代码与创建文件的代码基本相同，不同之处是目录文件的大小固定为 128，文件类型为 3。这里贴出完整代码：

```
bool FileSystem::CreateDirectory(char * name, char * path)
{
    Directory *directory;
    BitMap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;

    if( strcmp(path, "") ) printf("Creating directory %s in %s\n\n", name, path);
    else printf("Creating directory %s in root\n\n", name);

    directory = new Directory(NumDirEntries, name);
    OpenFile * openfile = NULL ;
    GetDirectoryFromPath(path, openfile) ;
    directory->FetchFrom(openfile) ;

    if (directory->Find(name) != -1)
      success = FALSE;          // file is already in directory
    else {
        freeMap = new BitMap(NumSectors);
        freeMap->FetchFrom(freeMapFile);
        sector = freeMap->Find();   // find a sector to hold the file header
        if (sector == -1)
            success = FALSE;        // no free block for file header
        else if (!directory->Add(name, sector))
            success = FALSE;    // no space in directory
        else {
            hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, 128))
                success = FALSE;    // no space on disk for data
            else {
                success = TRUE;
                time (&hdr->createTime) ;
        //fortime printf("createTime: %s\n", asctime(gmtime(&hdr->createTime)));
                hdr->fileType = 3 ;
                hdr->WriteBack(sector);
                //printf("hdr: %d\n", openfile->hdrSector) ;
                directory->WriteBack(openfile);
                freeMap->WriteBack(freeMapFile);
            }
            delete hdr;
        }
        delete freeMap;
        }
        delete directory;
        return success;
}
```

为了测试时打印目录信息，我们需要修改 List 函数，让它先根据路径找到目录，再打印目录中有什么文件，代码如下：

```
void
FileSystem::List(char * path)
{
    printf("List path %s:\n", path) ;
    Directory * directory = new Directory(NumDirEntries, "");
    OpenFile * openfile = NULL ;
    GetDirectoryFromPath(path, openfile) ;
    directory->FetchFrom(openfile) ;
    directory->List();
    delete directory;
}
```

在 filesys.h 中，我们需要为上述函数的定义加入参数 char * path，默认值为空字符串。并且要加入新函数的定义。

下面我们测试程序的正确性。我在 fstest.cc 中添加了新函数 DirectoryTest()，功能是创建多个目录 dir1，dir2，在目录 dir2 中再创建二级目录 dir3。并且在 dir1 和 dir3 中都新建文件，分别是 File1 和 File3。代码如下：

```
void DirectoryTest()
{
    fileSystem->CreateDirectory("dir1", "") ;
    fileSystem->CreateDirectory("dir2", "") ;
    fileSystem->List("") ;
    fileSystem->Create("File1", 100, "dir1/") ;
    fileSystem->CreateDirectory("dir3", "dir2/") ;
    fileSystem->List("dir1") ;
    fileSystem->List("dir2") ;
    fileSystem->Create("File3", 100, "dir2/dir3/") ;
    fileSystem->List("dir2/dir3") ;
}
```

需要注意的是，创建目录是后面不需要加’/’，但是把目录当成路径时需要在目录名字后面加上’/’，这是因为我的目录寻址函数以’/’为一级目录的结尾标志。

我在 main.cc 中加入命令-m，用于调用 DirectoryTest() 。另外要在 main.cc 中声明函数 DirectoryTest()

运行时，首先使用-f 命令对磁盘格式化，然后使用-m 命令执行测试函数，运行结果如下所示。可以看到，各个目录和文件都被成功创建，List 的结果都是正确的。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -m
Creating directory dir1 in root

Creating directory dir2 in root

List path :
dir1
dir2

Creating file File1 in dir1/

Creating directory dir3 in dir2/

List path dir1:
File1

List path dir2:
dir3

Creating file File3 in dir2/dir3/

List path dir2/dir3:
File3

Finish Thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Exercise5 动态调整文件长度

对文件的创建操作和写入操作进行适当修改，以使其符合实习要求。

由于当前文件系统的 OpenFile 只支持读写操作，因此目前只需要动态增长文件大小，不需要减小文件大小。

首先在文件创建时，也就是在 FileSystem 类中的 Create 函数中，将 initiaSize 变量置为 0。这样一来就可以简单地让刚创建的文件不进行磁盘分配了。

然后要修改的是 FileHeader 类，添加一个 Expand 函数，用来扩充文件大小。函数的结构和 Allocate 函数差不多，参数也是两个，一个是扇区空闲位图，另一个是新的文件大小。

在这个函数中，首先要计算出新的子索引扇区数目，以及新的文件扇区数目。然后基于之前已经有的扇区，给新的子索引扇区和新的文件信息扇区分配磁盘扇区。然后做法跟 Allocate 函数一样，循环地将子索引扇区写回到磁盘中。这里还要注意边界条件。

```
FileHeader::Expand(BitMap *freeMap, int expandSize)
{
    int newNumBytes = expandSize ;
    int newNumSectors  = divRoundUp(expandSize, SectorSize);
    //printf("numSectors: %d\n", numSectors) ;
    int numSemiHeaders = divRoundUp(numSectors, NumSemiDirect);
    int newNumSemiHeaders = divRoundUp(newNumSectors, NumSemiDirect);
    printf("Expand: %d, %d\n", expandSize, newNumSectors) ;
    if (freeMap->NumClear() < newNumSectors - numSectors
                        + newNumSemiHeaders - numSemiHeaders)
    return FALSE;       // not enough space

    for (int i = numSemiHeaders; i < newNumSemiHeaders; i++)
        dataSectors[i] = freeMap->Find();
    int idx, idy = numSectors, offset ;
    int semiDirect[NumSemiDirect] ;
    if( idy % NumSemiDirect != 0 )
    {
        idx = idy / NumSemiDirect ;
        synchDisk->ReadSector(dataSectors[idx], (char*) semiDirect);
    }
    while( idy < newNumSectors )
    {
        idx = idy / NumSemiDirect ;
        offset = idy % NumSemiDirect ;
        semiDirect[offset] = freeMap->Find();
        idy ++ ;
        if( offset == NumSemiDirect - 1 )
            synchDisk->WriteSector(dataSectors[idx], (char*) semiDirect);
    }

    if( offset != NumSemiDirect - 1 )
        synchDisk->WriteSector(dataSectors[idx], (char*) semiDirect);

    numBytes = newNumBytes ;
    numSectors = newNumSectors ;
    return TRUE;
}
```

我在 filesys.cc 的 Create 函数中限制一个文件的初始大小不超过 128 Byte。之所以不是直接设置为 0 Byte，是因为那样做必然会执行扩大空间操作，造成额外的开销。

```
bool
FileSystem::Create(char *name, int initialSize, char *path)
{
    if( initialSize > 128 ) initialSize = 128 ;
```

在 openfile.cc 的 WriteAt 函数中，需要判断所需空间是否大于文件大小，如果是的话，读取出 freeMap，调用 Expand 函数扩大空间，再写回 hdr 和 freeMap。注意屏蔽掉函数开头对越界的检查。

```
//if ((numBytes <= 0) || (position >= fileLength))
if (numBytes <= 0)
    return 0;                   // check request
if ((position + numBytes) > fileLength)
//numBytes = fileLength - position;
{
    BitMap * freeMap = new BitMap(NumSectors) ;
    OpenFile * openfile = new OpenFile(0) ;
    freeMap->FetchFrom(openfile) ;
    //printf("%d, %d, %d, position: %d\n", hdrSector
    hdr->Expand(freeMap, position + numBytes) ;
    hdr->WriteBack(hdrSector) ;
    freeMap->WriteBack(openfile) ;
    fileLength = hdr->FileLength() ;
    delete freeMap ;
    delete openfile ;
}
```

为了验证代码的正确性，我调用 Copy 函数，仍然拷贝 huge 文件。从运行结果可以看出，首先分配了一个扇区也就是 128 字节给 huge，然后每次扩大 1000 字节（为了显示的行数少一些，把粒度从 10 改为了 1000），也就是一个扇区的大小。这说明程序运行正确。

```
Creating directory huge_in_nachos in root

Allocate: 1
open path:
Expand: 1000, 8
pos: 1000
Expand: 2000, 16
pos: 2000
Expand: 3000, 24
pos: 3000
Expand: 4000, 32
pos: 4000
Expand: 5000, 40
pos: 5000
Expand: 6000, 47
pos: 6000
Expand: 6508, 51
pos: 6508
Finish Thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

第二部分 文件访问的同步与互斥

Exercise1 源代码阅读

a) 阅读 Nachos 源代码中与异步磁盘相关的代码，理解 Nachos 系统中异步访问模拟磁盘的工作原理。

filesys/synchdisk.h 和 filesys/synchdisk.cc

这两个文件中描述的是一个经过抽象的磁盘，在这个磁盘通过软件的方式实现了同步互斥机制。即不能对这个磁盘同时进行读写操作。

同步磁盘类中主要提供 ReadSector 和 WriteSector 函数，分别表示读写磁盘扇区。还有一个函数是 RequestDone，这个函数用来处理磁盘中断。

对于同步磁盘实现的机制，该类中声明了一个 Disk 指针对象，一个信号量和一个锁。在实现读写扇区的时候，首先要获得锁，表示获得了磁盘的唯一控制权，然后进行调用磁盘的读写函数，硬盘读写函数会根据涉及的扇区数计算一个延时，并会规划一个磁盘中断，等待一定的时钟周期后会有中断产生。随后调用信号量的 P 操作，等待中断处理函数调用 V 操作。被唤醒之后释放锁。

中断处理函数是以函数指针的形式传给 Disk 的构造函数的。Disk 的相关函数在规划中断时会使用这个中断处理函数，在中断到来时，中断处理函数进行 V 操作。

b) 利用异步访问模拟磁盘的工作原理，在 Class Console 的基础上，实现 Class SynchConsole。

我在 console.cc 和 console.h 中加入了 SynchConsole 类，具体实现和 synchdisk 文件基本上差不多，除了将 disk 换成 console，对应的读写操作换成 putchar 和 getchar。然后对于 console 中断，分为两种，一个是读中断，一个是写中断，这样就要写两个处理函数，也要有两个信号量，一个为等待读的字符到来，一个为等待写完成。

类定义如下：

```
class SynchConsole {
    public:
        SynchConsole(char *readFile, char *writeFile);
        ~SynchConsole() ;
        char GetChar() ;
        void PutChar(char ch) ;
    private:
        Console *console ;
        Lock *readLock ;
        Lock *writeLock ;
};
```

构造与析构函数：

```
SynchConsole::SynchConsole(char *readFile, char *writeFile)
{
    readLock = new Lock("readLock") ;
    writeLock = new Lock("writeLock") ;
    readAvail = new Semaphore("readavail", 0) ;
    writeDone = new Semaphore("writedone", 0) ;
    console = new Console(readFile, writeFile, ReadAvail, WriteDone, 0) ;
}

SynchConsole::~SynchConsole()
{
    delete console ;
    delete readLock ;
    delete writeLock ;
}
```

中断处理函数：

```
static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvail(int arg) { readAvail->V(); }
static void WriteDone(int arg) { writeDone->V(); }
```

封装的读写函数：

```
char SynchConsole::GetChar()
{
    char ch ;
    readLock->Acquire() ;
    readAvail->P() ;
    ch = console->GetChar() ;
    readLock->Release() ;
    return ch ;
}
void SynchConsole::PutChar(char ch)
{
    writeLock->Acquire() ;
    console->PutChar(ch) ;
    writeDone->P() ;
    writeLock->Release() ;
}
```

修改 progtest.cc 中的 ConsoleTest 函数如下：

```
void
ConsoleTest (char *in, char *out)
{
    char ch;
    synchConsole = new SynchConsole(in, out) ;
    for (;;) {
        ch = synchConsole->GetChar() ;
        synchConsole->PutChar(ch) ;
        if(ch == 'q') return ;
    }
```

执行结果如下，程序可以正常运行：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -c
hello world
hello world
q
qMachine halting!
```

Exercise2 实现文件系统的同步互斥访问机制，达到如下效果：

a) 一个文件可以同时被多个线程访问。且每个线程独自打开文件，独自拥有一个当前文件访问位置，彼此间不会互相干扰。

b) 所有对文件系统的操作必须是原子操作和序列化的。例如，当一个线程正在修改一个文件，而另一个线程正在读取该文件的内容时，读线程要么读出修改过的文件，要么读出原来的文件，不存在不可预计的中间状态。

c) 当某一线程欲删除一个文件，而另外一些线程正在访问该文件时，需保证所有线程关闭了这个文件，该文件才被删除。也就是说，只要还有一个线程打开了这个文件，该文件就不能真正地被删除。

对于 a，这个要求在 Nachos 已经实现的系统中以及完成了，因为每一个打开的文件都会返回一个 OpenFile 对象，而这些对象之间互相不干扰，每个对象都有自己唯一的访问位置。

对于 b，由于每一次 OpenFile 的读写操作都分割为多次对多个扇区分别进行读写操作，因此需要将这些操作变成一个原子操作。在这里可以在 SynchDisk 类中，对每一个扇区分配一个锁，既然每一个文件头对应一个扇区，那就可以通过这样的方式，对文件加锁。首先在 SynchDisk 类中添加这些锁的字段，然后构造函数中初始化。同时提供两个函数，AcquireLock 和 ReleaseLock，通过传递扇区号参数来决定获取和释放哪些锁。

对于 c，只需要在 filehdr.h 中加入 openNum 变量，打开文件时对它加 1，关闭文件时对它减一，删除时判断它是否是 0，是 0 才能删除。

在 synchDisk.h 中添加锁指针数组和操作锁的函数：

```
void AcquireLock(int sector)
{
    lockset[sector]->Acquire() ;
}
void ReleaseLock(int sector)
{
    lockset[sector]->Release() ;
}
Lock *lockset[NumSectors] ;
```

在 synchDisk.cc 中初始化锁指针数组：

```
SynchDisk::SynchDisk(char* name)
{
    semaphore = new Semaphore("synch disk", 0);
    lock = new Lock("synch disk lock");
    for( int i = 0 ; i < NumSectors ; i ++)
    {
        lockset[i] = new Lock("sector lock") ;
    }
    disk = new Disk(name, DiskRequestDone, (int) this);
}
```

Filesys.cc 的 Remove 函数只需简单判断一下 openNum 即可，这里不展示代码。

在 openfile.cc 中要对 WriteAt，ReadAt，构造函数，析构函数加锁：

```
OpenFile::OpenFile(int sector)
{
    //if( synchDisk->lockset[25]->owner != NULL )
    //else printf("open OK\n") ;
    synchDisk->AcquireLock(sector) ;
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
    hdrSector = sector ;
    time (&hdr->lastUseTime) ;
    hdr->openNum ++ ;
    printf("create openfile by thread %d, ", curr
    printf("openNum: %d\n", hdr->openNum) ;
    hdr->WriteBack(sector) ;
    synchDisk->ReleaseLock(sector) ;
    //fortime printf("sector: %d, useTime: %s\n",
}
```

注意要写回文件头。

```
OpenFile::~OpenFile()
{
    synchDisk->AcquireLock(hdrSector) ;
    hdr->FetchFrom(hdrSector);
    hdr->openNum --;
    hdr->WriteBack(hdrSector) ;
    printf("delete openfile by thread %d, ", cur
    printf("openNum: %d\n", hdr->openNum) ;
    delete hdr;
    synchDisk->ReleaseLock(hdrSector) ;
}
```

注意先读文件头，再写文件头，防止有其他线程修改过文件头。

下面的代码对 ReadAt 和 WriteAt 都是一样的，所以只展示一处：

```
// write modified sectors back
    synchDisk->AcquireLock(hdrSector) ;
    for (i = firstSector; i <= lastSector; i++)
        synchDisk->WriteSector(hdr->ByteToSector(i * SectorSize),
                               &buf[(i - firstSector) * SectorSize]);
    synchDisk->ReleaseLock(hdrSector) ;
```

在 fstest.cc 中，编写两个线程的测试函数：

```
void FileThreadTest()
{
    for(int i = 0 ; i < 2 ; i ++)
    {
        Thread * thread ;
        if( i == 0 ) thread = new Thread("Test2", 1) ;
        else thread = new Thread("Test3", 1) ;
        thread->Fork(ForkFunction, 0) ;
    }
}
```

两个线程中的一个创建文件 TestFile1，两个线程都是先读后写，最后都试图删除文件：

```
void ForkFunction(int arg)
{
    OpenFile *openFile = NULL ;
    int i, numBytes;
    if( currentThread->getTid() == 2 ){

        if (!fileSystem->Create(FileName, 128, "")) {
            printf("Perf test: can't create %s\n", FileName);
            //return;
        }
    }
    while (openFile == NULL) {
        openFile = fileSystem->Open(FileName, "");
        currentThread->Yield() ;
    }
    for (i = 0; i < FileSize; i += ContentSize) {
        numBytes = openFile->Write(Contents, ContentSize);
        if (numBytes < 10) {
            printf("Perf test: unable to write %s\n", FileName);
            delete openFile;
            return;
        }
    }
    delete openFile ;
    openFile = NULL ;
```

读文件和写文件要打开和关闭两次文件。下面的代码与上面的代码相连。

```
    while (openFile == NULL) {
        openFile = fileSystem->Open(FileName, "");
        currentThread->Yield() ;
    }
    currentThread->Yield() ;
    char *buffer = new char[ContentSize];
    for (i = 0; i < FileSize; i += ContentSize) {
        numBytes = openFile->Read(buffer, ContentSize);
        if ((numBytes < 10) || strncmp(buffer, Contents, ContentSize)) {
            printf("Perf test: unable to read %s, %s\n", FileName, buffer);
            delete openFile;
            delete [] buffer;
            return;
        }
    }
    delete [] buffer;
    delete openFile;
    currentThread->Yield() ;
    if (!fileSystem->Remove(FileName, "")) {
      printf("Perf test: unable to remove %s\n", FileName);
      return;
    }
}
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -mt
create openfile by thread 1, openNum: 2
create openfile by thread 1, openNum: 2
Finish Thread: main
Creating file TestFile1 in root
open TestFile1 by thread 3
Allocate: 1
open TestFile1 by thread 3
open TestFile1 by thread 2
create openfile by thread 3, openNum: 1
delete openfile by thread 3, openNum: 0
open TestFile1 by thread 3
create openfile by thread 2, openNum: 1
create openfile by thread 3, openNum: 2
delete openfile by thread 2, openNum: 1
open TestFile1 by thread 2
delete openfile by thread 3, openNum: 0
create openfile by thread 2, openNum: 1
Perf test: unable to remove TestFile1
Finish Thread: Test3
delete openfile by thread 2, openNum: 0
remove TestFile1 by thread 2
Finish Thread: Test2
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1083030, idle 1061780, system 21250, user 0
Disk I/O: reads 621, writes 52
```

从上图测试结果可以看出，文件相互交叉运行。Create openfile 是创建打开文件指针，delete openfile 是删除文件指针，openNum 代表文件被打开的次数。可以看到，Test3 线程先结束，试图删除文件，但此时 openNum 还是 1，所以删除失败。最后 Test2 结束时删除文件才成功。

第三部分 Challenges

Challenge 1 性能优化

a) 例如，为了优化寻道时间和旋转延迟时间，可以将同一文件的数据块放在同一磁道上。

b) 使用 cache 机制减少磁盘访问次数，例如延迟写和预读取。

优化寻道时间等，只需要修改 BitMap 寻找空位的策略就行，因此我在 BitMap 类中添加了一个辅助函数 FindGroup，表示给定一个一定大小的空位需求，尽量在同一个磁道上寻找这些空位。函数的参数包含要求空位个数和一个指针，用于存放结果。

函数的策略就是最佳适配，找到一个拥有足够多且空位数最小的磁道：

```
void BitMap::FindGroup(int num, int * group)
{
    int NumTracks = 32 ;
    int SectorsPerTrack = 32 ;
    int freeSectors[32] = {} ;
    int mins = SectorsPerTrack + 1 ;
    int best = - 1 ;
    for( int i = 0 ; i < NumTracks ; i ++)
    {
        for( int j = 0 ; j < SectorsPerTrack; j ++ )
        {
            if( !Test(i*SectorsPerTrack + j) )
                freeSectors[i] ++ ;
        }
        if( freeSectors[i] >= num && freeSectors[i] < mins )
        {
            best = i ;
            mins = freeSectors[i] ;
        }
    }
```

接下来分配这些扇区。如果找不到同区的位置，就按照原来的办法分配扇区：

```
    if(best != -1)
    {
        printf("track: %d\n", best) ;
        printf("sectors: \n") ;
        int offset = 0 ;
        for( int j = 0 ; j < num ; j ++ )
        {
            while(Test(best*SectorsPerTrack + offset))
            {
                offset ++ ;
            }
            printf("%d, ", offset) ;
            Mark(best*SectorsPerTrack + offset) ;
            group[j] = best*SectorsPerTrack + offset ;
            offset ++ ;
        }
        printf("\n") ;
        return ;
    }
    for( int i = 0 ; i < num ; i ++ )
    {
        group[i] = Find() ;
    }
}
```

然后，把 allocate 和 expand 中调用 Find 的语句修改为 FindGroup 语句，以 expand 为例，修改的代码如下：

```
if( newNumSemiHeaders-numSemiHeaders > 0 )
    freeMap->FindGroup(newNumSemiHeaders-numSemiHeaders, &dataSectors[numSemiHeaders]);

if( newNumSectors - idy + offset > NumSemiDirect )
{
    freeMap->FindGroup(NumSemiDirect-offset, &semiDirect[offset]);
    synchDisk->WriteSector(dataSectors[idx], (char*) semiDirect);
    idy += NumSemiDirect-offset ;
}
else
{
    freeMap->FindGroup(newNumSectors - idy, &semiDirect[offset]);
    synchDisk->WriteSector(dataSectors[idx], (char*) semiDirect);
    idy += newNumSectors - idy ;
}
```

然后我采取了缓存和预取策略进行磁盘的优化，这里修改 SynchDisk 类，增加一个大小为 16 个扇区的缓存，然后预取策略是每一次读连续 4 个扇区的内容。通过轮转算法来进行替换。主要需要修改的地方是 ReadSector 函数，首先要检查该扇区的内容在不在缓存里，这里使用一个 tag 数组来标识缓存对应的扇区，找到的话直接复制返回。否则进行连续 4 次读磁盘操作，转移到缓存中。

首先在类定义中声明缓存数组 cache，tag 数组，轮转标志 turn。然后在构造函数中初始化这些变量

```
SynchDisk::~SynchDisk()
{
    for( int i = 0 ; i < 16 ; i ++ )
    {
        if(tag[i] != -1) {
            disk -> WriteRequest(tag[i] , cache[i]) ;
            semaphore->P();
        }
    }
    delete disk;
    delete lock;
    delete semaphore;
}
```

在将要读取硬盘时，先查找缓存中是否有需要的扇区，如果有，返回缓存中的数据；如果没有，到磁盘中一次性拷贝 4 个连续扇区进入缓存。使用轮转策略替换缓存中的扇区。被替换掉的扇区如果不是空扇区，需要写回到硬盘。写回时注意要一个一个写，即每写一个调用一次 P 操作。

```
SynchDisk::ReadSector(int sectorNumber, char* data)
{
    lock->Acquire();            // only one disk I/O at a time
    for( int i = 0 ; i < 16 ; i ++ )
    {
        if(tag[i] == sectorNumber)
        {
            bcopy(cache[i], data, SectorSize) ;
            lock->Release() ;
            return ;
        }
    }
    for( int i = 0 ; i < 4 ; i ++ )
    {
        if( sectorNumber + i < NumSectors )
        {
            if(tag[turn] != -1) {
                disk -> WriteRequest(tag[turn] , cache[turn]) ;
                semaphore->P();
            }
            disk -> ReadRequest(sectorNumber + i , cache[turn]) ;
            semaphore->P();
            if( i == 0 ) bcopy(cache[turn], data, SectorSize) ;
            tag[turn] = sectorNumber + i ;
            turn = (turn + 1) % 16 ;
        }
    }
    lock->Release() ;
    return ;
}
```

在写回磁盘的函数中，需要判断扇区是否在缓存中，如果在缓存中，就把扇区更新到缓存中。如果不在缓存中，直接写回硬盘。

```
void
SynchDisk::WriteSector(int sectorNumber, char* data)
{
    lock->Acquire();            // only one disk I/O at a time
    for( int i = 0 ; i < 16 ; i ++ )
    {
        if(tag[i] == sectorNumber)
        {
            bcopy(data, cache[i], SectorSize) ;
            lock->Release() ;
            return ;
        }
    }
    disk->WriteRequest(sectorNumber, data);
    semaphore->P();          // wait for interrupt

    lock->Release();
}
```

在析构函数中，要把所有缓存中的扇区写回硬盘。

```
SynchDisk::~SynchDisk()
{
    for( int i = 0 ; i < 16 ; i ++ )
    {
        if(tag[i] != -1) {
                disk -> WriteRequest(tag[i] , cache[i]) ;
                semaphore->P();
            }
    }
    delete disk;
    delete lock;
    delete semaphore;
}
```

为了测试程序的有效性，我调用拷贝函数，拷贝一个大文件。截取结果的一部分如下所示。可以看到，在 expend 4000 的时候，占用了第一磁道；在 expand 5000 的时候，首先为一级目录分配了一个扇区，然后又需要分配 8 个扇区。为了能分配在同一磁道中，程序分配了第二个磁道给这 8 个扇区；在 expand 6000 的时候，只需要分配 7 个扇区，因此程序把第一磁道的剩下 7 个扇区分配给这个文件了。

```
Expand: 4000, 32
track: 1
sectors:
16, 17, 18, 19, 20, 21, 22, 23,
0, openNum: 4
delete openfile by thread 1, openNum: 3
0, openNum: 3
create openfile by thread 1, openNum: 4
Expand: 5000, 40
track: 1
sectors:
24,
track: 2
sectors:
0, 1, 2, 3, 4, 5, 6, 7,
0, openNum: 4
delete openfile by thread 1, openNum: 3
0, openNum: 3
create openfile by thread 1, openNum: 4
Expand: 6000, 47
track: 1
sectors:
25, 26, 27, 28, 29, 30, 31,
0, openNum: 4
delete openfile by thread 1, openNum: 3
0, openNum: 3
create openfile by thread 1, openNum: 4
```

Challenge 2 实现 pipe 机制

重定向 openfile 的输入输出方式，使得前一进程从控制台读入数据并输出至管道，后一进程从管道读入数据并输出至控制台。

这里我使用两个线程来模拟前端线程和后端线程。构建了一个 synchConsole，然后使用两个信号量来模拟 pipe 机制的生产者消费者问题，用于 pipe 管道文件。简便起见，在这里 pipe 文件只能容纳一个字符。

PipeThread1 表示前端读字符的线程，首先读取一个字符，然后根据生产者消费者问题调用空槽的信号量 out，写进 pipe 文件，释放一个产品信号量 in，如此循环。PipeThread2 表示后端线程，首先调用产品信号量，读 pipe 文件，再释放空槽信号量，输出到控制台，如此循环。

代码修改在 fstest.cc 中

```
SynchConsole *synchConsole ;
Semaphore *in, *out ;
OpenFile * pipe ;

void PipeThread1(int arg)
{
    while(1)
    {
        char ch = synchConsole->GetChar() ;
        out->P() ;
        pipe->WriteAt(&ch, 1, 0) ;
        in->V() ;
        if(ch == 'q') break ;
    }
}
void PipeThread2(int arg)
{
    while(1)
    {
        char ch ;
        in->P() ;
        pipe->ReadAt(&ch, 1, 0) ;
        out->V() ;
        synchConsole->PutChar(ch) ;
        if(ch == 'q') break ;
    }
}
```

主线程负责初始化各个变量，包括创建和打开管道文件，创建控制台，创建读写线程等等。

```
    synchConsole = new SynchConsole(NULL, NULL) ;
    in = new Semaphore("in", 0) ;
    out = new Semaphore("out", 1) ;
    fileSystem->Create("pipe", 128, "") ;
    pipe = fileSystem->Open("pipe", "") ;
    Thread *t1 = new Thread("Test2", 1) ;
    Thread *t2 = new Thread("Test3", 1) ;
    t1->Fork(PipeThread1, 0) ;
    t2->Fork(PipeThread2, 0) ;
}
```

程序运行结果如下所示，可以看到控制台的功能是正确的。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -mt
0, openNum: 3
create openfile by thread 1, openNum: 4
1, openNum: 3
create openfile by thread 1, openNum: 4
Creating file pipe in root
open pipe by thread 1
22, openNum: 1
create openfile by thread 1, openNum: 2
pipe, 22
Finish Thread: main
hello world
hello world
how are you
how are you
q
Finish Thread: Test2
qFinish Thread: Test3
```

内容三：遇到的困难以及解决方法

困难 1：编译，运行有陷阱

编译时需要注释掉主程序中的线程部分，防止吞参数；运行时需要先进行-f 格式化，否则结果会混乱。

困难 2：需要考虑的情况很多

多级目录需要循环找下一级目录，每次释放指针时要注意不能释放根目录指针；每次修改目录和文件头都要注意写回；多线程时，修改文件头要先重新读取，再立即写回；缓存在最后析构的时候要写回硬盘，否则会出现第一次运行正确，以后运行错误的神奇情况。使用整块磁道分配时，还需要重写 expand 函数。有好多问题在第一次写的时候是很难注意到的，需要在调试时耐心改正。

困难 3：测试程序比较难构造

比如多级目录测试要能体现出多级树形结构；多线程文件操作要体现打开的文件不能被删除，这就需要有一个正在打开的文件。磁盘性能优化要体现出磁道分配的整块策略等等。

困难 4：编译链接的依赖关系比较复杂

在添加代码时，经常会遇到需要改写 makefile 的情况，比如编写控制台同步的代码文件时。我没有找到好的解决方案，只好通过把相关的函数写到原来的控制台代码中来解决这个问题。

困难 5：需要进行版本控制

后面的 exercise 有时会与前面的 exercise 冲突。这就导致完成了后面的 exercise 后前面的 exercise 就不能跑了。因此，我需要把所有版本加以保存。使用 Git 是一个好的选择。

内容四：收获及感想

通过这次 lab，我对文件系统的结构和操作的了解更加深入了。我在调试代码，维护大的工程方面的能力也有了很大提高。文件是非常复杂的，而现代操作系统又能让这一切井井有条地高效运行，使我对操作系统也有了由衷的赞叹之情。

内容五：对课程的意见和建议

我认为这次 lab 布置的时间比较短，任务比较重，应该给予我们更多的时间。

内容六：参考文献

【1】现代操作系统　陈向群