

文件系统实习报告

姓名： 学号：

日期： 2020/12/18

目录

内容一：任务完成情况.....	4
任务完成列表（Y/N）	4
具体 Exercise 完成情况.....	4
Exercise1	4
Exercise2	24
Exercise3	33
Exercise4	45
Exercise5	56
Exercise6	64
Exercise7	69
Challenge	78
内容二：遇到的困难以及收获.....	81
内容三：对课程或 Lab 的意见和建议.....	82
内容四：参考文献.....	83

内容一：任务完成情况

任务完成列表 (Y/N)

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise7	Challenge
第一部分	Y	Y	Y	Y	Y	Y	Y	Y

具体 Exercise 完成情况

Exercise1

这一部分要求阅读十份源代码，下面我将逐一分析各部分的具体内容。总体上来说，这些源代码包含了目前 Nachos 与文件系统相关的所有类和方法

(一) filesys.h 和 filesys.cc

这两份代码中包含了 Nachos 中文件系统的实现和一些管理函数。filesys.h 中提供了两个版本的文件系统，目前默认使用的是加入 FILESYS_STUB 宏定义时的文件系统，这个文件系统的实现类似于 UNIX 的文件系统实现。另外一个文件系统的声明需要开启 FILESYS 的宏定义，并关闭 FILESYS_STUB 的宏定义。这个文件系统的方法实现位于 filesys.c 中。下面我们来分析一下这两个文件系统。

一、FILESYS_STUB 下的文件系统

这个文件系统中主要包含了四个方法：构造函数、创建文件、打开文件、移除文件。

1.构造函数

这个文件系统的构造函数为空，但是需要传入 format 参数，以保证使用方法一致。

2.Create

这个函数判断能否创建一个文件。Create 通过调用 OpenForWrite 进行文件创建，如果描述符为-1，返回 FALSE，表明创建失败，否则关闭描述符，并返回 TRUE，表示创建成功。

OpenForWrite 在 sysdep.cc 中定义，是对 open 系统调用的一个封装。这里调用的 open 指定了 O_CREAT 宏，这样保证了如果文件不存在时可以创建新文件，而存在时则不用。

```

bool create(char *name, int initialSize) {
    int fileDescriptor = OpenForWrite(name);

    if (fileDescriptor == -1) return FALSE;
    Close(fileDescriptor);
    return TRUE;
}

int
OpenForWrite(char *name)
{
    int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);

    ASSERT(fd >= 0);
    return fd;
}

```

3.Open

这个函数用于打开一个供读写的文件。函数首先调用一个以 O_RDWR 权限打开的文件（这里要求文件要已经存在了）并获取其文件描述符。如果获取成功，函数根据文件描述符创建一个 Nachos 中的打开文件类 Openfile 的对象，并返回其指针；否则函数返回空指针。

```

OpenFile* Open(char *name) {
    int fileDescriptor = OpenForReadWrite(name, FALSE);

    if (fileDescriptor == -1) return NULL;
    return new OpenFile(fileDescriptor);
}

int
OpenForReadWrite(char *name, bool crashOnError)
{
    int fd = open(name, O_RDWR, 0);

    ASSERT(!crashOnError || fd >= 0);
    return fd;
}

```

4.Remove

这个函数调用 Ulink 函数（C 语言中 unlink 函数的一个包装），来删除参数传递的文件路径所对应的文件，并根据返回值判断删除成功还是失败。

```

bool Remove(char *name) { return Unlink(name) == 0; }

bool
Unlink(char *name)
{
    return unlink(name);
}

```

二、FILESYS 下的文件系统

这个宏下的类除了包含上述方法外，还包含了列出文件和打印文件信息（内容）的两个

方法，以及一个目录文件的打开文件指针，和一个记录空闲块映射的位图的打开文件（把位图封装到打开文件中）指针。

```
class FileSystem {
public:
    FileSystem(bool format);           // Initialize the file system.
                                         // Must be called *after* "synchDisk"
                                         // has been initialized.
                                         // If "format", there is nothing on
                                         // the disk, so initialize the directory
                                         // and the bitmap of free blocks.

    bool Create(char *name, int initialSize);
                                         // Create a file (UNIX creat)

    OpenFile* Open(char *name);        // Open a file (UNIX open)

    bool Remove(char *name);          // Delete a file (UNIX unlink)

    void List();                     // List all the files in the file system

    void Print();                    // List all the files and their contents

private:
    OpenFile* freeMapFile;           // Bit map of free disk blocks,
                                         // represented as a file
    OpenFile* directoryFile;         // "Root" directory -- list of
                                         // file names, represented as a file
};
```

下面让我们来看一下这个文件系统中各个方法是如何实现的，它们都存放于 filesys.cc 中。

1.构造函数

这个函数用于初始化文件系统。如果传入的参数 format 为 TRUE，那么函数会格式化文件系统。格式化主要完成以下几项工作：

(1) 函数首先创建新的空闲块位图、目录文件，以及它们的头部。

```
BitMap *freeMap = new BitMap(NumSectors);
Directory *directory = new Directory(NumDirEntries);
FileHeader *mapHdr = new FileHeader;
FileHeader *dirHdr = new FileHeader;

DEBUG('f', "Formatting the file system.\n");
```

(2) 接下来，函数为位图和目录文件的文件头和文件内容分配空间，并在位图中进行占用空间的标记。为位图和目录文件的内容分配空间时，要求必须分配成功，否则 ASSERT 失败直接退出。文件的占用空间也会在头部中记录。

```
freeMap->Mark(FreeMapSector);
freeMap->Mark(DirectorySector);

// Second, allocate space for the data blocks containing the contents
// of the directory and bitmap files. There better be enough space!

ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
```

两个文件的大小和头所在的扇区在 filesys.cc 中的宏中给出定义。可以看到，现在只支持一级目录，且目录项不能超过 10 个。

```
#define FreeMapSector          0
#define DirectorySector         1

#define FreeMapFileSize          (NumSectors / BitsInByte)
#define NumDirEntries            10
#define DirectoryFileSize        (sizeof(DirectoryEntry) * NumDirEntries)
```

接下来，函数把两个文件的头部先写回到磁盘，然后利用文件头来创建两个文件的打开文件对象，再把上面初始化好的位图和目录文件写回磁盘。

```
DEBUG('f', "Writing headers back to disk.\n");
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);

freeMapFile = new OpenFile(FreeMapSector);
directoryFile = new OpenFile(DirectorySector);

DEBUG('f', "Writing bitmap and directory back to disk.\n");
freeMap->WriteBack(freeMapFile);           // flush changes to
directory->WriteBack(directoryFile);
```

上述过程已经完成了对文件系统的初始化。如果开启了-f 的调试选项，函数还会打印位图和目录文件的内容，并把刚才分配的位图文件、目录文件、二者的文件头全部释放。

```
if (Debug.IsEnabled('f')) {
    freeMap->Print();
    directory->Print();

    delete freeMap;
    delete directory;
    delete mapHdr;
    delete dirHdr;
}
```

如果没有开启格式化选项，那么函数不会重新建立目录文件和位图文件，而是直接利用预定义好的 sector 来打开对应的文件。

```
freeMapFile = new OpenFile(FreeMapSector);
directoryFile = new OpenFile(DirectorySector);
```

在 Nachos 运行过程中，目录文件和位图文件始终保持打开的状态，用于管理文件。

2.Create

这个函数用于在 Nachos 文件系统中创建一个新文件。目前的 Nachos 文件系统不支持动态改变文件大小，因此在参数中指定了创建时的初始化文件大小 initialSize。

创建一个文件要经过以下几个步骤：

Step1：函数读取目录文件，并检查这个文件是否已经存在。如果存在，函数判断创建失败，并返回 FALSE。

```

directory = new Directory(NumDirEntries);
directory->FetchFrom(directoryFile);

if (directory->Find(name) != -1)
    success = FALSE;           // file is already in directory
}

```

Step2: 函数读取位图文件，寻找是否还有可以容纳文件头的 sector。如果没有，函数判断创建失败，并返回 FALSE。

```

freeMap = new BitMap(NumSectors);
freeMap->FetchFrom(freeMapFile);
sector = freeMap->Find();      /
if (sector == -1)
    success = FALSE;          /

```

Step3: 函数试图在目录文件中为新文件添加目录项。如果没有空余的目录项，函数判断创建失败，并返回 FALSE。

```

else if (!directory->Add(name, sector))
    success = FALSE;        // no space in directory
}

```

Step4: 函数创建文件头对象，并根据传入的 initialSize 在位图中寻找空间。如果没有足够的空间，函数判断创建失败，并返回 FALSE。

```

hdr = new FileHeader;
if (!hdr->Allocate(freeMap, initialSize))
    success = FALSE;        // no space on disk for data
}

```

以上步骤如果都没有失败，那么函数判断创建文件成功，并把文件头、目录文件、位图文件相应地进行更新。在释放所有函数打开的文件后，返回 TRUE。

```

success = TRUE;
// everthing worked, flush all changes back to disk
hdr->WriteBack(sector);
directory->WriteBack(directoryFile);
freeMap->WriteBack(freeMapFile);

    |     delete hdr;
}
    delete freeMap;
}
delete directory;
return success;
}

```

3.Open

这个函数用于打开一个可供读写的文件。函数根据文件路径名在目录文件中查找是否存在这个文件。如果存在，函数打开这个文件，释放自己创建的目录对象，并返回打开文件指针；否则返回 NULL。

```

OpenFile *
FileSystem::Open(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
    int sector;

    DEBUG('f', "Opening file %s\n", name);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
        openFile = new OpenFile(sector);           // name was found
    delete directory;                           // return
    return openFile;                           // return
}

```

4.Remove

这个函数用于从文件系统中删除一个文件，分以下几个步骤来实现：

Step1: 函数打开目录文件，并查找是否存在文件路径名所对应的目录项。如果不存在，立即返回 FALSE。

```

directory = new Directory(NumDirEntries);
directory->FetchFrom(directoryFile);
sector = directory->Find(name);
if (sector == -1) {
    delete directory;
    return FALSE;                         // file not found
}

```

Step2: 函数从目录文件中读取相应的文件头，同时打开位图文件，并在位图文件中释放文件内容和文件头所占用的空间。此外，函数还要从目录文件中移除对应的目录项。

```

fileHdr = new FileHeader;
fileHdr->FetchFrom(sector);

freeMap = new BitMap(NumSectors);
freeMap->FetchFrom(freeMapFile);

fileHdr->Deallocate(freeMap);
freeMap->Clear(sector);
directory->Remove(name);

```

Step3: 函数把更新了的位图文件和目录文件写回磁盘，释放自己打开的文件后，返回 TRUE。

```

freeMap->WriteBack(freeMapFile);
directory->WriteBack(directoryFile);
delete fileHdr;
delete directory;
delete freeMap;
return TRUE;

```

5.List

这个函数用于列出所有记录在 Nachos 目录文件中的目录项。函数打开目录文件后，调用目录文件的 List 方法，来实现上述功能。

```

void
FileSystem::List()
{
    Directory *directory = new Directory(NumDirEntries);

    directory->FetchFrom(directoryFile);
    directory->List();
    delete directory;
}

```

6.Print

这个函数用于打印整个文件系统的所有信息，包括：位图和目录文件头的信息、位图中记录的空闲块信息、目录文件中记录的所有目录项信息。

```

void
FileSystem::Print()
{
    FileHeader *bitHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;
    BitMap *freeMap = new BitMap(NumSectors);
    Directory *directory = new Directory(NumDirEntries);

    printf("Bit map file header:\n");
    bitHdr->FetchFrom(FreeMapSector);
    bitHdr->Print();

    printf("Directory file header:\n");
    dirHdr->FetchFrom(DirectorySector);
    dirHdr->Print();

    freeMap->FetchFrom(freeMapFile);
    freeMap->Print();

    directory->FetchFrom(directoryFile);
    directory->Print();

    delete bitHdr;
    delete dirHdr;
    delete freeMap;
    delete directory;
}

```

以上是 Nachos 对整个文件系统的构建。下面让我们来看一下 Nachos 文件系统中各部分的构成。

(二) filehdr.h 和 filehdr.cc

这两份代码定义了 Nachos 文件系统中的文件头（类似于 UNIX 中的 i-node）类 FileHeader，并给出了 FileHeader 类中成员方法的实现。

一、FileHeader 类

这个类用于描述 Nachos 文件系统中储存的文件的信息，比较关键的对象有：文件的字节数、文件在磁盘中占据的段（扇区）数、文件占据的每一个磁盘扇区的扇区号（被组织

为一个数组)。

```
private:  
    int numBytes;      // Number of bytes  
    int numSectors;   // Number of sectors  
    int dataSectors[NumDirect];
```

注意到，如下图所示，文件扇区号数组有最大值限制。可以计算出，目前的文件系统中，所支持的最大文件大小为 3840B。而且在（一）的分析中，我们也知道，每个文件都要在创建时指定好文件大小。

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))  
#define MaxFileSize (NumDirect * SectorSize)
```

FileHeader 没有定义自己的构造函数，使用的是默认构造函数。下面让我们来逐个分析一下这个类中所包含的方法。

```
public:  
    bool Allocate(BitMap *bitMap, int fileSize);  
        // including allocating space  
        // on disk for the file data  
    void Deallocate(BitMap *bitMap);           // De-  
        // allocate  
    void FetchFrom(int sectorNumber);         // Initialize  
    void WriteBack(int sectorNumber);          // Write  
        // back to disk  
  
    int ByteToSector(int offset); // Convert a byte  
        // to the disk sector containing  
        // the byte  
  
    int FileLength();      // Return the length of the file  
        // in bytes  
  
    void Print();          // Print the contents of the file
```

1. Allocate

这个函数用于初始化刚创建的文件头。函数根据传入的位图指针和文件大小，计算文件所需要占据的扇区个数，并在位图中进行分配和标记。如果位图中保存的空闲扇区数量不够，函数会返回 FALSE；否则，函数在位图中寻找空闲扇区，分配完毕后返回 TRUE。

```
bool  
FileHeader::Allocate(BitMap *freeMap, int fileSize)  
{  
    numBytes = fileSize;  
    numSectors = divRoundUp(fileSize, SectorSize);  
    if (freeMap->NumClear() < numSectors)  
        return FALSE;           // not enough space  
  
    for (int i = 0; i < numSectors; i++)  
        dataSectors[i] = freeMap->Find();  
    return TRUE;  
}
```

2. Deallocate

这个函数用于释放文件索取的所有磁盘扇区。函数遍历磁盘扇区数组的每一个已分配项，首先确认这个扇区是否被分配了（未分配会报错），然后在位图中释放这个扇区。

```
void
FileHeader::Deallocate(BitMap *freeMap)
{
    for (int i = 0; i < numSectors; i++) {
        ASSERT(freeMap->Test((int) dataSectors[i]));
        freeMap->Clear((int) dataSectors[i]);
    }
}
```

3. FetchFrom

这个函数根据传入的磁盘扇区号，从 system.cc 中的全局磁盘对象中读取扇区内容。

```
void
FileHeader::FetchFrom(int sector)
{
    synchDisk->ReadSector(sector, (char *)this);
}
```

4. WriteBack

这个函数根据传入的扇区号，向 system.cc 中的全局磁盘对象写入被修改的内容。

```
void
FileHeader::WriteBack(int sector)
{
    synchDisk->WriteSector(sector, (char *)this);
}
```

5. ByteToSector

这个函数根据传入的文件字节偏移量，返回这个字节所在的磁盘扇区的扇区号

```
int
FileHeader::ByteToSector(int offset)
{
    return(dataSectors[offset / SectorSize]);
}
```

6. FileLength

这个函数返回文件的字节长度，也就是私有成员 numBytes 的大小。

```
int
FileHeader::FileLength()
{
    return numBytes;
}
```

7. Print

这个函数打印文件头的内容，包括：文件的字节大小、文件每个扇区所在的磁盘扇区

号、文件每个扇区中的具体内容。

```
void
FileHeader::Print()
{
    int i, j, k;
    char *data = new char[SectorSize];

    printf("FileHeader contents. File size: %d. File blocks:\n", numBytes);
    for (i = 0; i < numSectors; i++)
        printf("%d ", dataSectors[i]);
    printf("\nFile contents:\n");
    for (i = k = 0; i < numSectors; i++) {
        synchDisk->ReadSector(dataSectors[i], data);
        for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++) {
            if ('\040' <= data[j] && data[j] <='\176') // isprint(data[j])
                printf("%c", data[j]);
            else
                printf("\\%x", (unsigned char) data[j]);
        }
        printf("\n");
    }
    delete [] data;
}
```

(三) directory.h 和 directory.cc

这两份代码中包含了 Nachos 操作系统中的目录和目录项的定义。目录类中包含了一些成员方法，它们在 directory.cc 中实现。下面让我们来分析一下这两个类。

一、DirectoryEntry

这个类是目录项类，它路面的内容很简单，包含三个成员变量： **inUse** 指明这个目录项是否正在被使用； **sector** 指明这个目录项所对应的文件的文件头所在的磁盘扇区号； **name** 这个字符数组存放文本形式的文件名。Nachos 目前的实现限制了文件名最大长度为 9 个字符， name 中+1 是为了提供‘\0’的结尾字符。

```
class DirectoryEntry {
public:
    bool inUse;          // Is this directory entry in use?
    int sector;          // Location on disk to find the
                         // FileHeader for this file
    char name[FileNameMaxLen + 1]; // Text name for file, with +1 for
                                 // the trailing '\0'
};

#define FileNameMaxLen    9 // for simplicity, we assume
                         // file names are <= 9 characters long
```

二、Directory

这个类是目录类，其中包含两个私有成员，分别是页表条目数量和页表条目数组的指针；包含一个私有方法和九个公有方法。下面让我们来分析一下这些方法都提供了哪些功能。

```

public:
    Directory(int size);    // Initialize an empty directory
    // with space for "size" files
    ~Directory();           // De-allocate the directory

    void FetchFrom(OpenFile *file); // Init directory contents from disk
    void WriteBack(OpenFile *file); // Write modifications to
    // directory contents back to disk

    int Find(char *name); // Find the sector number of the
    // FileHeader for file: "name"

    bool Add(char *name, int newSector); // Add a file name into the directory

    bool Remove(char *name); // Remove a file from the directory

    void List(); // Print the names of all the files
    // in the directory
    void Print(); // Verbose print of the contents
    // of the directory -- all the file
    // names and their contents.

private:
    int tableSize; // Number of directory entries
    DirectoryEntry *table; // Table of pairs:
    // <file name, file header location>

    int FindIndex(char *name); // Find the index into the directory
    // table corresponding to "name"

```

1.构造函数和析构函数

目录类的构造函数根据传入的 `size` 参数，初始化一个大小为 `size` 的目录项数组，并记录这个尺寸。初始化时，每个目录项的 `inUse` 属性都是 `FALSE`，表示它们都没有被使用。
Nachos 的文件系统中支持的目录项个数目前为 10 个，定义在 `filesys.cc` 中。

```

Directory::Directory(int size)
{
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 0; i < tableSize; i++)
        table[i].inUse = FALSE;
}

```

目录类的析构函数就是把目录项数组释放掉。

```

Directory::~Directory()
{
    delete [] table;
}

```

2.FetchFrom

这个函数利用传入的目录文件，从磁盘中向当前的目录类读取目录项数组的内容。

```

void
Directory::FetchFrom(OpenFile *file)
{
    (void) file->ReadAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}

```

3.WriteBack

这个函数向传入的目录文件中写入目录类中所记录的目录项数组的内容，以更新磁盘中的目录文件。

```
void
Directory::WriteBack(OpenFile *file)
{
    (void) file->WriteAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}
```

4.FindIndex

这个函数用来寻找文件所在的目录项条目。函数遍历目录类中的目录项数组，如果某个目录项记录的文件名称与传入的文件名称相匹配，返回对应目录项的下标，否则函数返回-1，代表文件不存在。这个函数是目录类中的私有方法。

```
int
Directory::FindIndex(char *name)
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse && !strcmp(table[i].name, name, FileNameMaxLen))
            return i;
    return -1;           // name not in directory
}
```

5.Find

这个函数根据文件路径，返回这个文件路径相对应的文件的文件头所在的磁盘扇区号。函数首先调用 FindIndex 函数，寻找这个文件所在的目录项下标。如果非负，函数从目录项表中读取对应目录项中的扇区号并返回；否则，函数返回-1。

```
int
Directory::Find(char *name)
{
    int i = FindIndex(name);

    if (i != -1)
        return table[i].sector;
    return -1;
}
```

6.Add

这个函数用于向目录类中添加新的文件。函数首先判断这个文件是否存在，如果存在，返回 FALSE。然后函数寻找是否还有未使用的表项。如果没有，返回 FALSE；否则函数把这个表项的 inUse 设置为 TRUE，修改表项的文件名和扇区号，最后返回 TRUE。可以看到，现在 Nachos 的文件系统同时可存在的文件被限制为最多 10 个。

```
bool
Directory::Add(char *name, int newSector)
{
    if (FindIndex(name) != -1)
        return FALSE;

    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse) {
            table[i].inUse = TRUE;
            strcpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
            return TRUE;
        }
    return FALSE; // no space. Fix when we have extensible files.
}
```

7.Remove

这个函数用于从目录项表中移除文件。函数首先根据传入的路径名搜索目录项表，如果存在表项，就把它 inUse 设置为 FALSE，并返回 TRUE，指明移除成功；如果这个文件不存在，那么就返回 FALSE。

```
bool
Directory::Remove(char *name)
{
    int i = FindIndex(name);

    if (i == -1)
        return FALSE; // name not in directory
    table[i].inUse = FALSE;
    return TRUE;
}
```

8.List

这个函数输出目前正在使用的目录项。函数遍历整个目录项表，把正在使用的目录项中的文件名输出出来。

```
void
Directory::List()
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse)
            printf("%s\n", table[i].name);
}
```

9.Print

这个函数输出整个目录类的全部信息。函数遍历目录项表，把目前正被使用的目录项中的文件名和文件头所在磁盘扇区号输出出来，然后加载文件头并输出文件头中的信息。

```
void
Directory::Print()
{
    FileHeader *hdr = new FileHeader;

    printf("Directory contents:\n");
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse) {
            printf("Name: %s, Sector: %d\n", table[i].name, table[i].sector);
            hdr->FetchFrom(table[i].sector);
            hdr->Print();
        }
    printf("\n");
    delete hdr;
}
```

(四) openfile.h 和 openfile.cc

这两份代码中包含了打开文件类。Nachos 提供了两个打开文件类，当开启 FILESYS_STUB 选项时，使用基于 UNIX 实现的类；当开启 FILESYS 宏时，使用另外一

个打开文件类。下面让我们分别来分析一下。

一、FILESYS_STUB 下的打开文件类

这个文件类中包含两个私有成员，分别是：文件描述符、当前的文件位置（偏移）

```
private:  
    int file;  
    int currentOffset;
```

除此之外，打开文件类还包含了一些方法：

1.构造函数和析构函数

打开文件类的构造函数就是初始化文件描述符和文件位置；析构函数则是关闭已经打开的文件。

```
OpenFile(int f) { file = f; currentOffset = 0; } // open the file  
~OpenFile() { Close(file); } // close the file
```

2.ReadAt

这个函数首先根据传入的位置设置要读取的文件位置，然后调用 ReadPartial 函数，从 file 中的 position 处开始，读取 numBytes 字节数据，拷贝到 into 指针指向的位置。执行完毕后，函数返回读取的字节数。Lseek 是对 lseek 的 UNIX 系统调用的封装，ReadPartial 是对 read 这一 UNIX 系统调用的封装。

```
int ReadAt(char *into, int numBytes, int position) {  
    Lseek(file, position, 0);  
    return ReadPartial(file, into, numBytes);  
}  
  
void  
Lseek(int fd, int offset, int whence)  
{  
    int retVal = lseek(fd, offset, whence);  
    ASSERT(retVal >= 0);  
}  
  
int  
ReadPartial(int fd, char *buffer, int nBytes)  
{  
    return read(fd, buffer, nBytes);  
}
```

3.WriteAt

这个函数首先根据传入的位置设置要读取的文件位置，然后调用 WriteFile 函数，从 file 中的 position 处开始，向文件中写入 numBytes 字节数据，数据由 from 指针指向的位置提供。Lseek 是对 lseek 的 UNIX 系统调用的封装，WriteFile 是对 write 这一 UNIX 系统调用的封装。执行完毕后，函数返回写入的字节数。

```

int WriteAt(char *from, int numBytes, int position) {
    lseek(file, position, 0);
    WriteFile(file, from, numBytes);
    return numBytes;
}

void
WriteFile(int fd, char *buffer, int nBytes)
{
    int retVal = write(fd, buffer, nBytes);
    ASSERT(retVal == nBytes);
}

```

4.Read

这个函数从当前 **OpenFile** 类中记录的文件位置开始，读取 **numBytes** 字节的数据，储存到 **into** 指针指向的位置，并更新当前的文件位置。最后，函数返回读取的字节数。

```

int Read(char *into, int numBytes) {
    int numRead = ReadAt(into, numBytes, currentOffset);
    currentOffset += numRead;
    return numRead;
}

```

5.Write

这个函数从当前 **OpenFile** 类中记录的文件位置开始，写入 **numBytes** 字节的数据，数据来源为 **from** 指针指向的位置，写入后，函数更新当前的文件位置。最后，函数返回写入的字节数。

```

int Write(char *from, int numBytes) {
    int numWritten = WriteAt(from, numBytes, currentOffset);
    currentOffset += numWritten;
    return numWritten;
}

```

6.Length

这个函数返回文件的大小。Tell 函数是对 tell 系统调用的封装。对于 i386 机器，由于它没有 tell 这个系统调用，因此 Tell 使用 lseek 来完成上述功能。

```

int Length() { lseek(file, 0, 2); return Tell(file); }

int
Tell(int fd)
{
#ifdef HOST_i386
    return lseek(fd, 0, SEEK_CUR);
#else
    return tell(fd);
#endif
}

```

二、FILESYS 下的打开文件类

这个打开文件类中包含两个私有成员变量：当前文件的文件头、文件的当前文件位

置。此外，与（一）类似，这个类中也实现了一些方法。

```
private:  
    FileHeader *hdr;  
    int seekPosition;
```

1.构造函数和析构函数

构造函数从传入的扇区号处读取文件头的信息，并初始化类中的私有文件头成员。文件位置被初始化为 0。析构函数则是把文件头释放掉。

```
OpenFile::OpenFile(int sector)  
{  
    hdr = new FileHeader;  
    hdr->FetchFrom(sector);  
    seekPosition = 0;  
}  
  
OpenFile::~OpenFile()  
{  
    delete hdr;  
}
```

2.Seek

这个函数把文件位置修改为传入的 position 值。

```
void  
OpenFile::Seek(int position)  
{  
    seekPosition = position;  
}
```

3.ReadAt

这个函数从 position 处读取 numBytes 字节数据，储存在 into 指向的内存位置，并返回读取的字节数。由于这里没有使用系统调用，所以我们需要处理一些极端情况和特殊情况：

(1) 如果 numBytes≤0，或者 position 超出文件长度，读取失败，返回 0 字节。

```
int fileLength = hdr->FileLength();  
int i, firstSector, lastSector, numSectors;  
char *buf;  
  
if ((numBytes <= 0) || (position >= fileLength))  
    return 0; // check request
```

(2) 如果读取的字节数加上文件位置大于文件长度，那么把读取的长度修改为剩余的字节数。

```
if ((position + numBytes) > fileLength)  
    numBytes = fileLength - position;
```

完成以上检查后，函数正式开始读取。函数先计算数据的起始扇区和结束扇区，然后

构造一个缓冲区 buf（因为读取的时候以扇区为单位读取，但是要获取的数据可能没有对齐，所以要先构造一个缓冲区），再逐扇区向 buf 中写入读取的数据。最后，函数把从 position 开始的长为 numBytes 的数据拷贝到 into 处，释放 buf，返回真正读取的字节数。

```

DEBUG('f', "Reading %d bytes at %d, from file of length %d.\n",
      numBytes, position, fileLength);

firstSector = divRoundDown(position, SectorSize);
lastSector = divRoundDown(position + numBytes - 1, SectorSize);
numSectors = 1 + lastSector - firstSector;

// read in all the full and partial sectors that we need
buf = new char[numSectors * SectorSize];
for (i = firstSector; i <= lastSector; i++)
    synchDisk->ReadSector(hdr->ByteToSector(i * SectorSize),
                           &buf[(i - firstSector) * SectorSize]);

// copy the part we want
bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
delete [] buf;
return numBytes;

```

4. WriteAt

这个函数从 position 开始，向文件中写入 numBytes 字节的数据，数据来源为 from 指向的位置，最后返回真正写入的字节数。

与 ReadAt 类似，WriteAt 也要进行两方面参数的检查。如果 $numBytes \leq 0$ ，或者文件位置超出了文件长度，那么返回 0；如果文件位置+写入长度大于文件最大的长度，那么只能写入一部分数据。

```

int fileLength = hdr->FileLength();
int i, firstSector, lastSector, numSectors;
bool firstAligned, lastAligned;
char *buf;

if ((numBytes <= 0) || (position >= fileLength))
    return 0; // check request
if ((position + numBytes) > fileLength)
    numBytes = fileLength - position;

```

检查完毕后，函数还是先计算起始扇区和结束扇区，然后构建缓冲区 buf。

```

DEBUG('f', "Writing %d bytes at %d, from file of length %d.\n",
      numBytes, position, fileLength);

firstSector = divRoundDown(position, SectorSize);
lastSector = divRoundDown(position + numBytes - 1, SectorSize);
numSectors = 1 + lastSector - firstSector;

buf = new char[numSectors * SectorSize];

```

如果数据不满足起始和结尾的扇区对齐，那么函数先把这两个扇区读到 buf 中。

```

firstAligned = (position == (firstSector * SectorSize));
lastAligned = ((position + numBytes) == ((lastSector + 1) * SectorSize));

read in first and last sector, if they are to be partially modified
if (!firstAligned)
    ReadAt(buf, SectorSize, firstSector * SectorSize);
if (!lastAligned && ((firstSector != lastSector) || firstAligned))
    ReadAt(&buf[(lastSector - firstSector) * SectorSize],
           SectorSize, lastSector * SectorSize);

```

接下来，函数从 from 开始拷贝 numBytes 字节的数据到 position 处。最后，函数再把 buf 写回磁盘对应的位置，释放 buf 后，返回真正写入的大小。

```
bcopy(from, &buf[position - (firstSector * SectorSize)], numBytes);

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

5.Read

这个函数从 OpenFile 类中记录的 seekPosition 开始，调用 ReadAt 读取 numBytes 字节数据，写入 into 中，更新 seekPosition 后，返回读取到的字节数。

```
int
OpenFile::Read(char *into, int numBytes)
{
    int result = ReadAt(into, numBytes, seekPosition);
    seekPosition += result;
    return result;
}
```

6.Write

这个函数从 OpenFile 类中记录的 seekPosition 开始，调用 WriteAt 写入 numBytes 字节数据，数据来源为 into。在更新 seekPosition 后，返回读取到的字节数。

```
int
openFile::Write(char *into, int numBytes)
{
    int result = WriteAt(into, numBytes, seekPosition);
    seekPosition += result;
    return result;
}
```

7.Length

这个函数返回从文件头获取的文件大小。

```
int
OpenFile::Length()
{
    return hdr->FileLength();
```

(五) bitmap.h 和 bitmap.cc

这两份代码中实现了位图类，用于管理磁盘的空闲块（扇区）。

一、BitMap 类

位图类中包含了三个成员变量：位的总数、字（4 字节）的总数（向上取整）、位图的存储单元（用一个无符号整数的数组表示）。此外，这个类中还实现了一些方法。

```

class BitMap {
public:
    BitMap(int nitems); // Initialize a bitmap, with "nitems" bits
    // initially, all bits are cleared.
    ~BitMap(); // De-allocate bitmap

    void Mark(int which); // Set the "nth" bit
    void Clear(int which); // Clear the "nth" bit
    bool Test(int which); // Is the "nth" bit set?
    int Find(); // Return the # of a clear bit, and as a side
    // effect, set the bit.
    // If no bits are clear, return -1.
    int NumClear(); // Return the number of clear bits

    void Print(); // Print contents of bitmap

    // These aren't needed until FILESYS, when we will need to read and
    // write the bitmap to a file
    void FetchFrom(OpenFile *file); // fetch contents from disk
    void WriteBack(OpenFile *file); // write contents to disk

private:
    int numBits; // number of bits in the bitmap
    int numWords; // number of words of bitmap storage
    // (rounded up if numBits is not a
    // multiple of the number of bits in
    // a word)
    unsigned int *map; // bit storage
};

```

1. 构造函数和析构函数

构造函数根据传入的 `nitems` 参数初始化所需要的位数，然后计算所需要的字数，再根据 `numWords`，建立一个有 `numWords` 个元素的无符号整数数组，作为位图的数据存储，并用 `map` 指向它。最后，位图中的每个位都会被初始化为 0。

```

BitMap::BitMap(int nitems)
{
    numBits = nitems;
    numWords = divRoundUp(numBits, BitsInWord);
    map = new unsigned int[numWords];
    for (int i = 0; i < numBits; i++)
        Clear(i);
}

```

析构函数只需要把位图数组释放。

```

BitMap::~BitMap()
{
    delete map;
}

```

2. Mark

这个函数用于把传入的 `which` 所对应的位图位进行标记。

```

void
BitMap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);
    map[which / BitsInWord] |= 1 << (which % BitsInWord);
}

```

3.Clear

这个函数用于把传入的 which 所对应的位图位进行清除。

```
void
BitMap::Clear(int which)
{
    ASSERT(which >= 0 && which < numBits);
    map[which / BitsInWord] &= ~(1 << (which % BitsInWord));
}
```

4.Test

这个函数用于检查传入的 which 所对应的那一位是否被标记。如果被标记，返回 TRUE，未被标记，返回 FALSE。

```
bool
BitMap::Test(int which)
{
    ASSERT(which >= 0 && which < numBits);

    if (map[which / BitsInWord] & (1 << (which % BitsInWord)))
        return TRUE;
    else
        return FALSE;
}
```

5.Find

这个函数用于寻找位图中是否存在空闲位。函数遍历 0~numBits-1 的值，对每一个位调用 Test 进行检测。如果存在空闲位，那么函数标记这个位，并返回下标；否则函数返回 -1，表示不存在空闲位了。

```
int
BitMap::Find()
{
    for (int i = 0; i < numBits; i++)
        if (!Test(i)) {
            Mark(i);
            return i;
        }
    return -1;
}
```

6.NumClear

这个函数用于统计空闲位的个数。函数遍历每一位，调用 Test 进行检测，若为空闲位，则增加计数变量。最后返回空闲位总数。

```
int
BitMap::NumClear()
{
    int count = 0;

    for (int i = 0; i < numBits; i++)
        if (!Test(i)) count++;
    return count;
}
```

7.Print

这个函数用于输出位图的标记情况。函数输出所有已经被标记了的位图位号。

```
void
BitMap::Print()
{
    printf("Bitmap set:\n");
    for (int i = 0; i < numBits; i++)
        if (Test(i))
            printf("%d, ", i);
    printf("\n");
}
```

8.FetchFrom

这个函数用于从文件中读取位图信息，在加载位图文件到某个函数中的位图类时可能会用到。

```
void
BitMap::FetchFrom(OpenFile *file)
{
    file->ReadAt((char *)map, numWords * sizeof(unsigned), 0);
}
```

9.WriteBack

这个函数用于把位图类中的位图数组写回传入的文件中，在同步磁盘的位图文件时会用到。

```
void
BitMap::WriteBack(OpenFile *file)
{
    file->WriteAt((char *)map, numWords * sizeof(unsigned), 0);
}
```

Exercise2

这一部分要求我们增加文件描述信息，并突破文件名的长度限制。我们逐个来解决。

首先来考虑添加更多的文件属性。

我们需要添加时间戳、文件类型、路径等信息。我们知道，文件头是用来储存文件信息的，所以我们应该考虑修改文件头类 FileHeader。

注意到，文件头是储存在磁盘上，按需读入内存的，而且目前的实现中限制了文件头大小为磁盘的 1 个扇区（128 字节）。**目前的文件头中，128 字节里有 8 字节（两个 int）用来储存文件的字节大小和字大小，余下 120 字节全部分配给了磁盘扇区索引。**因此，为了保存更多的信息，一个直观的想法是，减少磁盘扇区索引数组的元素个数，给其他信息

让出一部分空间。

为了获取文件类型（文件扩展名），我们需要添加相应的工具函数。这个工具函数我在 filehdr.cc 中进行实现。主要思路就是利用 strrchr 函数找到文件名中最后一次出现“.”的位置。

```
char*
GetFileExtension(char* filename)
{
    char* last = strrchr(filename, '.');
    if(last == NULL || last == filename)
        return "";
    return last+1;
}
```

为了获取文件时间，我们可以考虑使用 time.h 中提供的一些函数，把时间转换为字符串，大致方法如下。由于 **asctime** 生成的原始字符串在结尾有一个换行符，会占据空间，因此这里进一步进行了字符串处理：

```
char*
GetTime()
{
    time_t RawTime = time(NULL);
    struct tm* CurrentTime = localtime(&RawTime);

    char* t = asctime(CurrentTime);
    char* result = new char[strlen(t)];

    strncpy(result, t, strlen(t)-1);
    result[strlen(t)-1] = '\0';

    return result;
}
```

根据 Lab 的要求，我们需要添加文件拓展名、创建时间、上次访问时间、上次修改时间（路径在后面实现）。再加上原来已经拥有的属性，文件头中现在有 2 个 int 型变量，有 3 个时间串。asctime 的时间串为一个包含 26 个字符的字符串（包含结尾的‘\0’）。此处为了简便，我们不妨规定文件扩展名最长 4 个字符。这样，我们可以先通过宏定义来固定上述长度，以简化后续的操作。

```
// #define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define NumTimeString 3
#define NumIntProperty 2
#define TimeStringLength 25 // 24 chars + '\0'
#define ExtensionLength 5 // 4 chars + '\0'

#define AllStringLength NumTimeString*TimeStringLength+ExtensionLength
#define NumDirect ((SectorSize - NumIntProperty*sizeof(int) - AllStringLength*sizeof(char)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)
```

可以计算出，现在文件最多可以有 10 个扇区来存放内容。

对应地，**我们需要在类中添加相应的成员变量。这里还添加了一个文件头的扇区标**

号，是为了简化寻找文件头的操作，这个属性不写回磁盘（前面的数据占满了 128 字节，所以不会把 headerSector 写入磁盘扇区）。

```
/* Lab5 Exercise2: additional file attributes */
char fileExtension[ExtensionLength];
char createTime[TimeStringLength];
char modifyTime[TimeStringLength];
char lastVisitTime[TimeStringLength];

// Don't save to disk, in order to simplify file header location.
int headerSector;
```

同时，我们还需要在类中实现操作这些新增成员变量的方法。其中的初始化函数 CreateInit 在 filehdr.cc 中实现，在第一次创建这个类时使用。

```
// Lab5: methods for operating additional file attributes
void CreateInit(char* ext); // used for initialization

void SetFileExtension(char* ext){ strcpy(fileExtension, ext); } // "" if no extension
void SetCreateTime(char* t){ strcpy(createTime, t); }
void SetModifyTime(char* t){ strcpy(modifyTime, t); }
void SetLastVisitTime(char* t){ strcpy(lastVisitTime, t); }

void SetHeaderSector(int s){ headerSector = s; }
int GetHeaderSector(){ return headerSector; }

void
FileHeader::CreateInit(char* ext)
{
    SetFileExtension(ext);

    char* curTime = GetTime();
    SetCreateTime(curTime);
    SetModifyTime(curTime);
    SetLastVisitTime(curTime);
}
```

下一步便是把上面实现的方法添加到文件系统的实现中。我们需要修改以下几处位置：

(1) 创建文件时 (FileSystem::Create) 中，要添加对文件头初始化的 CreateInit 的调用。同时还要修改目录文件和位图文件的修改时间和访问时间。

```
success = TRUE;
// everthing worked, flush all changes back to disk
hdr->CreateInit(GetFileExtension(name));
hdr->WriteBack(sector);

directory->WriteBack(directoryFile);
freeMap->WriteBack(freeMapFile);

char* curTime = GetTime();
FileHeader *mapHdr = new FileHeader;
FileHeader *dirHdr = new FileHeader;
mapHdr->FetchFrom(FreeMapSector);
dirHdr->FetchFrom(DirectorySector);
mapHdr->SetLastVisitTime(curTime);
mapHdr->SetModifyTime(curTime);
dirHdr->SetLastVisitTime(curTime);
dirHdr->SetModifyTime(curTime);
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);
delete mapHdr;
delete dirHdr;
```

(2) 打开文件时，为了在文件头中维护时间戳，我们需要在 OpenFile 的文件头属性中记录文件头所在的磁盘扇区。

```
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    hdr->SetHeaderSector(sector);
    seekPosition = 0;
}
```

(3) 在读取文件内容时，应该更新我们维护的“访问时间”这一时间戳。

```
// copy the part we want
bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);

// Lab5: additional file attributes
hdr->SetLastVisitTime(GetTime());

delete [] buf;
return numBytes;
```

(4) 在写入文件内容时，应该更新“访问时间”和“修改时间”这两个时间戳。

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

// Lab5: additional file attributes
// keep time consistency
char* curTime = GetTime();
hdr->SetLastVisitTime(curTime);
hdr->SetModifyTime(curTime);

delete [] buf;
return numBytes;
```

(5) 关于上述文件属性修改后写回磁盘的时机，对于普通文件有两种选择：一种是修改后即刻写回，这样可以相对提高文件系统的可靠性，但是访盘次数增加会导致效率变低。另一种方案是，在析构这个打开文件类的时候（等价于关闭文件时）再写入所有的更新。这一方案提升了效率，但由于同步不够频繁，会让文件系统可靠性变低。考虑到 Nachos 本身只是一个进程，异步问题相对来说并没有真实的文件系统那么严重，所以出于效率的考虑，我选择在关闭文件时（析构函数中）写回所有的修改。

```
OpenFile::~OpenFile()
{
    hdr->WriteBack(hdr->GetHeaderSector());
    delete hdr;
}
```

(6) 除了处理普通文件的文件头，我们还应处理位图文件和目录文件的文件头，因此还要相应地修改文件系统的构造函数。为了检验正确性，我为两个文件定义了它们的扩展

名，分别为：BHdr、DHdr。

```
BitMap *freeMap = new BitMap(NumSectors);
Directory *directory = new Directory(NumDirEntries);
FileHeader *mapHdr = new FileHeader;
mapHdr->CreateInit("BHdr");
FileHeader *dirHdr = new FileHeader;
dirHdr->CreateInit("DHdr");
```

(7) 删除一个文件时，也要修改目录文件和位图文件的修改时间和访问时间。

```
freeMap->WriteBack(freeMapFile);           // flush to disk
directory->WriteBack(directoryFile);         // flush to disk

char* curTime = GetTime();
FileHeader *mapHdr = new FileHeader;
FileHeader *dirHdr = new FileHeader;
mapHdr->FetchFrom(FreeMapSector);
dirHdr->FetchFrom(DirectorySector);
mapHdr->SetLastVisitTime(curTime);
mapHdr->SetModifyTime(curTime);
dirHdr->SetLastVisitTime(curTime);
dirHdr->SetModifyTime(curTime);
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);
delete mapHdr;
delete dirHdr;
```

(8) 为了检查文件内容的维护是否成功，我还修改了 FileHeader 的 Print 方法。

```
printf("----- %s ----- \n", "FileHeader contents");
printf("File Extension: %s\n", fileExtension);
printf("Create Time: %s\n", createTime);
printf("Last Visit Time: %s\n", lastVisitTime);
printf("Last Modify Time: %s\n", modifyTime);

printf("File size: %d. File blocks:\n", numBytes);
for (i = 0; i < numSectors; i++)
    printf("%d ", dataSectors[i]);
printf("\nFile contents:\n");
for (i = k = 0; i < numSectors; i++) {
    synchDisk->ReadSector(dataSectors[i], data);
    for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++) {
        if ('\040' <= data[j] && data[j] <= '\176') // isprint(data[j])
            printf("%c", data[j]);
        else
            printf("\\\\%x", (unsigned char) data[j]);
    }
    printf("\n");
}
printf("-----\n\n\n");
```

现在，我们已经完成了对上面添加的文件扩展的维护。进一步地，为了简化输出，我也添加了一个命令行参数-V，以选择是否省略停机时输出的机器信息，VERBOSE 在 system.h 中定义为外部变量，在 interrupt.cc 中赋值。被屏蔽的语句如下所示。

```
if (!strcmp(*argv, "-V"))
    VERBOSE = FALSE; // silent machine output
```

```

if(VERBOSE)
{
    printf("No threads ready or runnable, and no pending interrupts.\n");
    printf("Assuming the program completed.\n");
}

void
Interrupt::Halt()
{
    if(VERBOSE)
    {
        printf("Machine halting!\n\n");
        stats->Print();
    }
    Cleanup();      // Never returns.
}

```

为了测试上述修改的正确性，我编写了一个简单的 shell 脚本，通过连续运行命令行来
进行文件添加、读取和删除的操作。为了体现出时间的差别，我在创建文件、读取文件删
除文件的命令行语句之间添加了 sleep。

```

#!/bin/sh

# this file saves in filesys/test/
cd ../

echo "==== copies file \"small\" from filesys/test/ to Nachos DISK (and add extension) ==="
./nachos -V -cp test/small small.abc

sleep 2 # to observe the modification time change

echo ""
echo "==== Read content of the file in Nachos DISK ==="
./nachos -V -p small.abc

echo ""
echo "==== prints the contents of the entire file system ==="
./nachos -V -D

sleep 2

echo ""
echo "==== remove previous file ==="
./nachos -V -r small.abc

echo ""
echo "==== prints the contents of the entire file system ==="
./nachos -V -D

```

上述脚本运行结果如下：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/filesys/test$ bash test2_1.sh
==== copies file "small" from filesys/test/ to Nachos DISK (and add extension) ===
Cleaning up...
==== Read content of the file in Nachos DISK ===
This is the spring of our discontent.

Cleaning up...

```


可以看到，位图文件和目录文件可以正确地显示它们的文件头，且每次因为创建文件或者删除文件而修改位图和目录文件内容时，它们的文件头中相应的时间戳也会被修改。而对于普通文件，它的文件头的时间戳也会随着文件操作而相应地更新。可以认为上述修改能够达到我们的目标。

接下来，我们来考虑如何突破文件名的长度限制。文件名这一属性被储存在这个文件的目录项中，现在最长只有 9 字节，由定义在 `directory.h` 中的 `FileNameMaxLen` 决定。

```
#define FileNameMaxLen 9 // for simplicity, we assume  
// file names are <= 9 characters long
```

因此，一个突破文件名长度限制的最直观的办法是修改这个宏参数，把值调大。我们不妨假设每个目录项最长为 99 字节，那么我们可以修改宏的数值如下：

```
#define FileNameMaxLen    99
```

这样，每个目录项最大支持 99 字节的文件名了，我们来简单测试一下：

脚本的运行结果如下：

可以看到，文件名确实可以比原来加长了，说明这个方法可以解决文件名的限制。

但实际上，我们知道，这个方法仍然存在一些问题。每个目录项被强行拉长了，这样增加了目录文件的空间。参考《现代操作系统》的相关介绍，一种解决方案是，把文件名都组织到一个文件中，便可以压缩目录文件的空间。但这种方法要求文件支持变长，这个实现留待之后进行。

实际上，如下图所示，很多操作系统也都设计了文件名称的上限。对于实验来说，最大 99 字节的文件名已经足够了。

Common file system maximum filename length

System	Maximum length (characters)
8-bit FAT	9
FAT12, FAT16, FAT32	11
POSIX "Fully portable filenames"	14
exFAT	255
NTFS	255
Mac OS HFS	255
most UNIX file systems	255

Exercise3

这一部分要求我们扩展文件的长度（大小），实现简介索引，以突破直接索引对空间的限制。

注意到，文件系统创建文件时，调用的是 FileHeader 的 Allocate 方法为文件分配空间。的 Allocate 中，只是简单地通过比对文件所需的扇区数和文件头中的扇区数组能装载的最大扇区数来判断能否分配这个文件。因此，现在既要求文件定长，文件的长度又不能特别长。

除此之外，ByteToSector 和 Deallocate 也都是基于直接索引来设计的，我们也需要对它们进行修改。

在考察了 Nachos 的磁盘情况后，我决定让文件头存储的的最后一个扇区作为二级索引项，倒数第二个扇区作为一级索引项。此外，我决定采取 USE_INDIRECT 来对这个模式进行条件编译。因此，修改过程如下：

(1) 为了下面的修改更方便，首先为上面的设计添加一些宏选项，如下图所示：

```

// #define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define NumTimeString 3
#define NumIntProperty 2
#define TimeStringLength 25 // 24 chars + '\0'
#define ExtensionLength 5 // 4 chars + '\0'

#define AllStringLength (NumTimeString*TimeStringLength+ExtensionLength)

#ifndef USE_INDIRECT

#define NumDirect ((SectorSize - NumIntProperty*sizeof(int) - AllStringLength*sizeof(char)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)

#else

#define NumDataSectors ((SectorSize - NumIntProperty*sizeof(int) - AllStringLength*sizeof(char)) / sizeof(int))
#define NumDirect (NumDataSectors - 2)
#define OneLevelIdx (NumDataSectors - 2) // index = number+1
#define TwoLevelIdx (NumDataSectors - 1)
#define LevelNum (SectorSize / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize) + (LevelNum * SectorSize) + (LevelNum * LevelNum * SectorSize)

#endif

```

(2) 接下来修改分配方案。当空闲位置足够多时，函数首先检查直接索引不能容纳整个文件。如果能，则采用直接索引；否则，函数进一步检查是需要一级索引，还是二级索引。每个扇区最多可以容纳 32 个扇区号，因此二级索引就可以覆盖整个磁盘空间。修改结果如下图所示。注意到，**由于索引也要占据额外的扇区，因此只比较数据扇区是不够的，还要相应加上索引可能占用的扇区数来和空闲扇区进行比较。**

```

#ifndef USE_INDIRECT
    DEBUG('f', "Now use direct mapping.\n");
    for (int i = 0; i < numSectors; i++)
        dataSectors[i] = freeMap->Find();
#else

    DEBUG('f', "Now use indirect mapping.\n");
    if(numSectors <= NumDirect)
    {
        DEBUG('f', "Just need to use direct mapping.\n");
        for (int i = 0; i < numSectors; i++)
            dataSectors[i] = freeMap->Find();
    }

    else
    {
        //printf("233\n");
        if(numSectors <= LevelNum + NumDirect)
        {
            DEBUG('f', "Need to use one-level indirect mapping.\n");
            if(numClear < numSectors + 1) // cannot hold single index and data
                return FALSE;

            // direct
            for(int i=0; i<NumDirect; ++i)
            {
                dataSectors[i] = freeMap->Find();
            }

            // one-level
            dataSectors[OneLevelIdx] = freeMap->Find();
            int OneLevelIndexes[LevelNum];
            for(int i=0; i<numSectors - NumDirect; ++i)
            {
                OneLevelIndexes[i] = freeMap->Find();
            }
            synchDisk->WriteSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
        }
    }
}

```

```

        else if(numSectors <= LevelNum*LevelNum + LevelNum + NumDirect)
        {
            DEBUG('f', "Need to use two-level indirect mapping.\n");

            int LeftSectors = numSectors - NumDirect - LevelNum;
            int SecondLevelNum = divRoundUp(LeftSectors, LevelNum);
            if(numClear < numSectors + 1 + 1 + SecondLevelNum) // cannot hold double index and data
                return FALSE;

            // direct
            for(int i=0; i<NumDirect; ++i)
            {
                dataSectors[i] = freeMap->Find();
            }

            // one-level
            dataSectors[OneLevelIdx] = freeMap->Find();
            int OneLevelIndexes[LevelNum];
            for(int i=0; i<LevelNum; ++i)
            {
                OneLevelIndexes[i] = freeMap->Find();
            }
            synchDisk->WriteSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);

            // two-level
            dataSectors[TwoLevelIdx] = freeMap->Find();
            int TwoLevelIndexes[LevelNum];
            for(int i=0; i<SecondLevelNum; ++i)
            {
                TwoLevelIndexes[i] = freeMap->Find();
                int SingleLevelIndexes[LevelNum];
                for(int j=0; j<LevelNum && (j+i*LevelNum)<LeftSectors; ++j)
                {
                    SingleLevelIndexes[j] = freeMap->Find();
                }
                synchDisk->WriteSector(TwoLevelIndexes[i], (char*)SingleLevelIndexes);
            }
            synchDisk->WriteSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);
        }
        else
        {
            printf("Allocation error occurs!\n");
            ASSERT(FALSE);
        }
    }
}

```

(3) 然后修改为偏移量寻找扇区的函数，本质上也是分段寻找。

```

int
FileHeader::ByteToSector(int offset)
{
#ifndef USE INDIRECT
    return(dataSectors[offset / SectorSize]);
#else

    int DirectSize = NumDirect*SectorSize;
    int SingleSize = DirectSize + LevelNum*SectorSize;
    int DoubleSize = SingleSize + LevelNum*LevelNum*SectorSize;

    if(offset < DirectSize)
        return(dataSectors[offset/SectorSize]);

    if(offset < SingleSize)
    {
        int idx = (offset - DirectSize) / SectorSize;
        int OneLevelIndexes[LevelNum];
        synchDisk->ReadSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
        //printf("idx %d, data %d\n", idx, OneLevelIndexes[idx]);
        return OneLevelIndexes[idx];
    }

    int TwoLevelNum = (offset - SingleSize) / (LevelNum*SectorSize);
    int TwoLevelOneNum = ((offset - SingleSize) % (LevelNum*SectorSize)) / SectorSize;
    int TwoLevelIndexes[LevelNum];
    synchDisk->ReadSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);
    int OneLevelIndexes[LevelNum];
    synchDisk->ReadSector(TwoLevelIndexes[TwoLevelNum], (char*) OneLevelIndexes);
    return OneLevelIndexes[TwoLevelOneNum];

#endif
}

```

(4) 最后修改释放扇区的函数，也是进行文件大小的分段处理

```
#else

    int direct_idx, single_idx, double_idx;
    DEBUG("f", "Deallocating index table.\n");

    // direct mapping
    for(direct_idx=0; direct_idx<numSectors && direct_idx<NumDirect; ++direct_idx)
    {
        ASSERT(freeMap->Test((int) dataSectors[direct_idx])); // ought to be marked!
        freeMap->Clear((int) dataSectors[direct_idx]);
    }

    // one level
    if(numSectors > NumDirect)
    {
        DEBUG("f", "Deallocating one-level index table.\n");
        int OneLevelIndexes[LevelNum];
        synchDisk->ReadSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
        for(direct_idx=NumDirect; direct_idx<numSectors && single_idx<LevelNum; ++direct_idx, ++single_idx)
        {
            ASSERT(freeMap->Test((int) OneLevelIndexes[single_idx])); // ought to be marked!
            freeMap->Clear((int) OneLevelIndexes[single_idx]);
        }
        ASSERT(freeMap->Test((int) dataSectors[OneLevelIdx])); // ought to be marked!
        freeMap->Clear((int) dataSectors[OneLevelIdx]);

        if(numSectors > NumDirect + LevelNum)
        {
            DEBUG("f", "Deallocating two-level index table.\n");
            int TwoLevelIndexes[LevelNUM];
            synchDisk->ReadSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);
            for(direct_idx = NumDirect+LevelNum, single_idx=0; direct_idx<numSectors && single_idx<LevelNum; ++single_idx)
            {
                synchDisk->ReadSector(TwoLevelIndexes[single_idx], (char*)OneLevelIndexes);
                for(double_idx=0; direct_idx<numSectors && double_idx<LevelNum; ++direct_idx, ++double_idx)
                {
                    ASSERT(freeMap->Test((int)OneLevelIndexes[double_idx]));
                    freeMap->Clear((int)OneLevelIndexes[double_idx]);
                }

                ASSERT(freeMap->Test((int)TwoLevelIndexes[single_idx]));
                freeMap->Clear((int)TwoLevelIndexes[single_idx]);
            }

            ASSERT(freeMap->Test((int)dataSectors[TwoLevelIdx]));
            freeMap->Clear((int)dataSectors[TwoLevelIdx]);
        }
    }
}

#endif
```

(5) 此外，为了直观地展示上面的修改带来的效果，我也对 FileHeader 的 Print 函数进行了些修改。

```
void
FileHeader::Print()
{
    int i, j, k;
    char *data = new char[SectorSize];

    printf("----- %s ----- \n", "FileHeader contents");
    printf("File Extension: %s\n", fileExtension);
    printf("Create Time: %s\n", createTime);
    printf("Last Visit Time: %s\n", lastVisitTime);
    printf("Last Modify Time: %s\n", modifyTime);

#ifndef USE_INDIRECT
    printf("File size: %d. File blocks:\n", numBytes);
    for (i = 0; i < numSectors; i++)
        printf("%d ", dataSectors[i]);
    printf("\nFile contents:\n");
    for (i = k = 0; i < numSectors; i++) {
        synchDisk->ReadSector(dataSectors[i], data);
        for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++) {
            printChar(data[j]);
        }
        printf("\n");
    }
#endif
```

```

#ifndef _FILE_H_
#define _FILE_H_

#include "disk.h"
#include "sector.h"

// File structure
// +-----+
// | 0x00: file size (4 bytes) |
// | 0x04: file blocks (4 bytes) |
// | 0x08: numSectors (4 bytes) |
// | 0x0C: NumDirect (4 bytes) |
// | 0x10: LevelNum (4 bytes) |
// +-----+
// Data sectors
// +-----+
// | 0x00: OneLevelIndexes[0] (4 bytes) |
// | 0x04: TwoLevelIndexes[0] (4 bytes) |
// | 0x08: OneLevelIndexes[1] (4 bytes) |
// | 0x0C: TwoLevelIndexes[1] (4 bytes) |
// | ... |
// | 0x00: OneLevelIndexes[LevelNum] (4 bytes) |
// | 0x04: TwoLevelIndexes[LevelNum] (4 bytes) |
// +-----+
// Data
// +-----+
// | 0x00: dataSectors[0] (4 bytes) |
// | 0x04: dataSectors[1] (4 bytes) |
// | ... |
// | 0x00: dataSectors[numSectors] (4 bytes) |
// +-----+



// Function prototypes
void printFileHeader();
void printFileContents();
void printChar(char oriChar);

#endif // _FILE_H_

```

这里，我把输出文件内容的语句封装为了 printChar 这个函数。

```

char*
printChar(char oriChar)
{
    if ('\040' <= oriChar && oriChar <= '\176') // isprint(oriChar)
        printf("%c", oriChar); // Character content
    else
        printf("\\\%x", (unsigned char)oriChar); // Unreadable binary content
}

```

为了简化结构，我把文件名的最大长度缩短到了 85 字节，这样目录文件不需要使用多级索引。同时调整了输出顺序，先输出目录，再输出位图的占用情况

现在我们已经基本完成了多级索引机制的实现。我编写了三个脚本进行测试，分别测试不需要多级索引，需要一级索引，需要二级索引的情况

不需要多级索引的脚本为：

```
#!/bin/sh

# this file saves in filesystem/test/
cd ../

echo "==== initialize Nachos file system ==="
./nachos -V -f

echo "==== copies file \"big\" from filesystem/test/ to Nachos DISK (and add extension) ==="
./nachos -V -cp test/big big.abc

echo ""
echo "==== prints the contents of the entire file system ==="
./nachos -V -D

sleep 2

echo ""
echo "==== remove previous file ==="
./nachos -V -r big.abc

echo ""
echo "==== prints the contents of the entire file system ==="
./nachos -V -D
```

测试结果如下：

```
Directory file header:  
----- FileHeader contents -----  
File Extension: DHdr  
Create Time: Sun Dec 20 21:05:34 2020  
Last Visit Time: Sun Dec 20 21:05:36 2020  
Last Modify Time: Sun Dec 20 21:05:36 2020  
File size: 960. File blocks:  
  Direct indexing:  
    3 4 5 6 7 8 9 10  
File contents:
```

需要一级索引的脚本如下所示。这里通过查阅资料，使用了 dd 命令来生成特定大小的二进制文件：

```
#!/bin/sh

echo "===" Generate middle-sized file for single indirect indexing ==="
dd if=/dev/urandom of=midFile count=3 bs=1024 # 3KB

# this file saves in filesys/test/
cd ../

echo "===" initialize Nachos file system ==="
./nachos -V -f

echo "===" copies file \"big\" from filesys/test/ to Nachos DISK (and add extension) ==="
./nachos -V -cp test/midFile midFile.efg

echo ""
echo "===" prints the contents of the entire file system ==="
./nachos -V -D

sleep 2

echo ""
echo "===" remove previous file ==="
./nachos -V -r midFile.efg

echo ""
echo "===" prints the contents of the entire file system ==="
./nachos -V -D
```

测试结果如下（省略了二进制文件的内容，因为是随机生成的值）：

需要二级索引的脚本如下,和上面类似,也是使用 dd 命令生成特定大小的二进制文件:

```
#!/bin/sh

echo "===" Generate large file to test double-level indirect directory"
dd if=/dev/zero of=largeFile count=20 bs=1024 # 20KB

cd ../
make

echo "===" initialize Nachos system ==="
./nachos -f -V

echo "===" copy largeFile to Nachos's file system ==="
./nachos -V -cp test/largeFile largeFile

echo ""
echo "===" prints the contents of the entire file system ==="
./nachos -V -D

sleep 2

echo ""
echo "===" remove previous file ==="
./nachos -V -r largeFile

echo ""
echo "===" prints the contents of the entire file system ==="
./nachos -V -D
```

测试结果如下（省略了 20KB 的二进制文件的值，生成的是全 0 文件）：

可以看到，系统可以正确地为不同大小的文件分配合适数量的磁盘扇区，并且在移除文件时，系统也可以正确地释放所有扇区。因此，可以认为上述两级索引的实现是正确的。

Exercise4

这一部分要求我们实现多级目录。在上面的分析中，我们已经看到了，Nachos 的原始目录只支持一级结构，我们需要对它进行扩展。

为了实现方便，这个 Exercise 中的所有实现全部需要 MULTI_LEVEL_DIR 这个宏来进行条件编译，并且使用 D 的调试选项来输出提示信息（d、f 已经被占用了）。

为了能够方便地组织实现文件路径，我定义了一个结构体 **FilePath**，用于储存目录的层级结构，它在 filehdr 中定义，内容如下：

```
#define MAX_DIR_DEPTH 5

typedef struct
{
    /* data */
    char* DirArray[MAX_DIR_DEPTH];
    int DirDep; // dep of root dir = 0
    char* Base;
} FilePath;
```

接下来，我们要考虑怎样能够把字符串转换为路径名。在查阅资料后，我发现，我们可以使用 libgen.h 中的 **dirname**、**basename** 等函数来实现这个目标。因此，我在 filehdr.h 中声明了 PathParser 这个函数，并在 filehdr.cc 中实现了它，大致方法如下：

```
FilePath
PathParser(char* path)
{
    FilePath filepath;

    if(path[0] == '/')
        path = &path[1]; // skip root dir
    char* ts1 = strdup(path);
    char* ts2 = strdup(path);

    // note that these two functions will directly change the string
    char* curDir = dirname(ts1);
    filepath.Base = strdup(basename(ts2));

    int depth = 0;
    for(depth=0; path[depth]; path[depth]== '/' ? depth++: *path++); // calculate depth
    filepath.DirDep = depth;
    ASSERT(depth <= MAX_DIR_DEPTH);

    while(strcmp(curDir, "."))
    {
        filepath.DirArray[--depth] = strdup(basename(curDir));
        curDir = dirname(curDir);
    }

    return filepath;
}
```

下一步我们要考虑如何维护目录文件，主要包含以下几方面问题：

(1) 如何根据路径名寻找是否存在对应的最后一级目录文件。

要想寻找目录文件，首先应该找到对应的磁盘扇区。我们可以利用上面实现的工具函数来达成这一目标。大致思路是：首先把传入的路径名（带路径的文件名）分解为

FilePath 结构。如果得到的结构中 **DirDep** 为 0，说明就在根目录下，返回根目录描述符。否则，函数逐在目录文件中，利用每一级的子目录名称，逐级寻找子目录对应的目录项。找到后，返回扇区号；没有找到，则返回-1。

```
int
FileSystem::FindDirSector(char* path)
{
    FilePath filepath = PathParser(path);

//    printf("---> depth is %d\n", filepath.DirDep);
//    for(int i=0; i<filepath.DirDep; ++i)
//        printf("----> %s\n", filepath.DirArray[i]);

    int sector = DirectorySector;
    if(filepath.DirDep != 0) // not root dirFindDirSector
    {
        OpenFile* dirFile;
        Directory* tmpDir;
        for(int i=0; i<filepath.DirDep; ++i)
        {
            DEBUG('D', "====> Finding directory %s in sector %d.\n", filepath.DirArray[i], sector);
            dirFile = new OpenFile(sector);
            tmpDir = new Directory(NumDirEntries);
            tmpDir->FetchFrom(dirFile);
            //tmpDir->Print();
            sector = tmpDir->Find(filepath.DirArray[i]);
            if(sector == -1)
            {
                DEBUG('D', "====> Fail to find directory %s in sector %d.\n", filepath.DirArray[i], sector);
                break;
            }
            delete dirFile;
            delete tmpDir;
        }
    }
    return sector;
}
```

更进一步地，我们可以把上述函数包装，返回一个指向目录类的指针，以达成我们最初的目标。具体实现如下。这里返回 **void*** 是因为在试验时直接返回 **Directory*** 会出错，暂时未找到原因。

```
void*
FileSystem::FindDir(char* path)
{
    Directory* returnDir = new Directory(NumDirEntries);
    int sector = FindDirSector(path);

    if(sector == DirectorySector) // in root
    {
        returnDir->FetchFrom(directoryFile);
    }
    else if(sector != -1)
    {
        OpenFile* dirFile = new OpenFile(sector);
        returnDir->FetchFrom(dirFile);
        delete dirFile;
    }
    else
    {
        DEBUG('D', "====> No such directory. Maybe it has been deleted.\n");
    }

    // Directly return Directory* will raise error, I don't know why.
    return (void*)returnDir;
}
```

以上两个方法作为 **FileSystem** 的成员函数，在开启 **MULTI_LEVEL_DIR** 时条件编译。

(2) 目录文件的创建。为了区分创建目录文件还是普通文件，我在创建目录文件时设置传入 Create 的参数 initialSize 为 -1。此外，为了统一实现，我规定每个目录文件中最多可以存放 10 个目录项，目录项大小即为目录文件的大小。对于 Create 函数，我们也应该进行一些修改。

首先，根据 initialSize 判断是否为目录文件。

```
#ifndef MULTI_LEVEL_DIR
    DEBUG('f', "Creating file %s, size %d\n", name, initialSize);
#else
    bool isDir = FALSE;
    if(initialSize == -1) // this means that we will create directory file
    {
        isDir = TRUE;
        initialSize = DirectoryFileSize;
        DEBUG('D', "====> Creating directory %s, size is %d.\n", name, initialSize);
    }
    else
        DEBUG('f', "Creating file %s, size %d\n", name, initialSize);
#endif
```

如果是目录文件，使用上面实现的函数来寻找最后一级的目录文件

```
#ifndef MULTI_LEVEL_DIR
    directory->FetchFrom(directoryFile);
#else
    int DirSector = FindDirSector(name);
    // printf("---> dir sector %d\n", DirSector);
    ASSERT(DirSector != -1); // if we want to create a file, the directory must exist
    OpenFile* DirFile = new OpenFile(DirSector);
    directory->FetchFrom(DirFile);
    FilePath filepath = PathParser(name);
    if(filepath.DirDep > 0)
    {
        name = filepath.Base; // delete directory
    }
#endif
```

然后修改初始化方法。这里统一为目录文件设置后缀为 DIR，用 Dir_Ext 宏表示。

```
#ifdef MULTI_LEVEL_DIR
    if(isDir)
        hdr->CreateInit(Dir_Ext);
    else
        hdr->CreateInit(GetFileExtension(name));
        hdr->WriteBack(sector);
        //hdr->Print();

        if(isDir) // initialize directory
        {
            Directory* dir = new Directory(NumDirEntries);
            OpenFile* subDir = new OpenFile(sector);
            dir->WriteBack(subDir);
            //dir->Print();
            delete dir;
            delete subDir;
        }

        directory->WriteBack(DirFile);
        freeMap->WriteBack(freeMapFile);
        delete DirFile;

#else ...
```

(3) 打开文件时的操作。最原始的 Open 操作中，只支持在根目录下查找，因此我们应该根据上面实现的方法对 Open 中获取目录文件的部分进行相应的修改。

```
#ifndef MULTI_LEVEL_DIR
    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
#else
    directory = (Directory*)FindDir(name);
    FilePath filepath = PathParser(name);
    if(filepath.DirDep > 0)
    {
        name = filepath.Base;
    }
#endif
```

(4) 删除文件时的操作。与 Open 的道理类似，我们也要修改 Remove，以适应多级目录机制。在目前的实现中，由于时间关系，暂时没有实现递归地删除目录及其中的内容的方法。此外，**我们不允许直接删除一个里面仍然有内容的目录（直接删除会导致部分扇区无法回收）**。为了进行目录是否为空的判断，我在 Directory 类中添加了一个 IsEmpty 的判断方法。

```
bool
Directory::IsEmpty()
{
    bool empty = TRUE;
    for(int i=0; i<tableSize; ++i)
    {
        if(table[i].inUse)
        {
            empty = FALSE;
            break;
        }
    }
    return empty;
}
```

对 Remove 的修改如下所示。由于原来的 Remove 框架中只支持对根目录下文件和目录的删除，这里为了保证代码的一致性，当传入的内容为多级目录时，不允许使用这个方法（-r 选项）删除。

```
#ifndef MULTI_LEVEL_DIR
    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
#else
    directory = (Directory*)FindDir(name);
    FilePath filepath = PathParser(name);
    if(filepath.DirDep>0)
    {
        DEBUG('D', "====> '-r' can only be used to delete file/dir in root directory!\n");
        delete directory;
        return FALSE;
    }
#endif
```

```

#ifndef MULTI_LEVEL_DIR
if(!strcmp(fileHdr->GetFileExtension(), Dir_Ext))
{
    DEBUG('D', "====> You are trying to delete a directory in root directory!\n");
    OpenFile* SubDirFile = new OpenFile(sector);
    Directory* SubDirectory = new Directory(NumDirEntries);
    SubDirectory->FetchFrom(SubDirFile);
    SubDirectory->Print();
}

if(!SubDirectory->IsEmpty())
{
    DEBUG('D', "====> This directory isn't empty. Operation failed!\n");
    delete fileHdr;
    delete directory;
    delete SubDirFile;
    delete SubDirectory;
    return FALSE;
}
#endif

```

为了删除多级目录下的文件或目录，我额外实现了一个方法。由于时间有限，这里也是暂未完成递归删除的实现。这个函数的执行流程如下。

函数首先试图获取最后一级的目录文件，如果不存在，认为删除失败，返回 FALSE。

```

int DirSector;
OpenFile* DirFile;
Directory* DirDirectory = new Directory(NumDirEntries);

BitMap *FreeMap;

FileHeader *FileHdr;
int FileSector;

// find directory first
DirSector = FindDirSector(name);
// printf("----> dirsector %d\n", DirSector);
if(DirSector == -1) // not exists
{
    DEBUG('D', "====> Dir of %s doesn't exists!\n", name);
    return FALSE;
}
DirFile = new OpenFile(DirSector);
DirDirectory->FetchFrom(DirFile); // read directory content
// DirDirectory->Print();

```

然后函数检查待删文件/目录是否存在，如果不存在，认为删除失败，返回 FALSE。

```

// find the sector of object file
FilePath filepath = PathParser(name);
if (filepath.DirDep > 0)
{
    name = filepath.Base;
}
FileSector = DirDirectory->Find(name);
// printf("----> %s\n", name);
// printf("----> sector %d\n", FileSector);
if (FileSector == -1) // this file/directory doesn't exist
{
    delete DirDirectory;
    DEBUG('D', "====> Object File/Dir %s doesn't exists!\n", name);
    return FALSE;
}

```

接下来函数检查这个文件是否为目录文件。如果为目录文件，**函数检查里面的目录中是否为空，如果不为空，不允许删除（否则在不使用递归的情况下，无法彻底回收待删目录下内容的位图，那部分扇区在格式化之前始终无法使用了）。**

```
// read file header
FileHdr = new FileHeader;
FileHdr->FetchFrom(FileSector);

// we must ensure that there are no file in subdirectory
if(!strcmp(FileHdr->GetFileExtension(), Dir_Ext))
{
    DEBUG('D', "====> You are Trying to delete a directory!\n");
    OpenFile* SubDirFile = new OpenFile(FileSector);
    Directory* SubDirectory = new Directory(NumDirEntries);
    SubDirectory->FetchFrom(SubDirFile);
    SubDirectory->Print();

    if(!SubDirectory->IsEmpty())
    {
        DEBUG('D', "====> This directory isn't empty. Operation failed!\n");
        delete SubDirFile;
        delete SubDirectory;
        delete FileHdr;
        delete DirDirectory;
        delete DirFile;
        return FALSE;
    }
}
```

如果通过了上述检查，函数便在获取到的目录文件和位图文件中进行相应的修改，并把修改写回磁盘中，最后返回 TRUE（删除成功）。

```
// free the file and its sector
FreeMap = new BitMap(NumSectors);
FreeMap->FetchFrom(freeMapFile);
FileHdr->Deallocate(FreeMap);           // delete data
FreeMap->Clear(FileSector);            // delete header
DirDirectory->Remove(name);

// flush change to disk
FreeMap->WriteBack(freeMapFile);
// *** note that we need to flush to correct directory
DirDirectory->WriteBack(DirFile);

DEBUG('D', "====> Successfully remove a multi-level file!\n");
delete FileHdr;
delete DirDirectory;
delete DirFile;
delete FreeMap;
return TRUE;
```

(5) 除了以上这些必要的修改外，为了能够方便地使用目录文件，我在命令行中添加了三个命令行选项，分别是：-mkdir [目录路径]，创建目录、-ls [目录路径]，显示指定目录下的内容（-l 只能显示根目录下的内容）、-rm [文件/目录路径]，删除多级文件/目录。

```

#ifndef MULTI_LEVEL_DIR
    // Lab5: multi-level Directory Operations
    else if (!strcmp(*argv, "-mkdir")) { // make directory
        ASSERT(argc > 1);
        MakeDir(*(argv + 1));
        argCount = 2;
    } else if (!strcmp(*argv, "-rm")) { // remove Nachos file or directory
        ASSERT(argc > 1);
        bool success = fileSystem->RemoveDir(*(argv + 1));
        argCount = 2;
    } else if (!strcmp(*argv, "-ls")) { // list Nachos directory
        ASSERT(argc > 1);
        fileSystem->ListDir(*(argv + 1));
        argCount = 2;
    }
#endif // MULTI_LEVEL_DIR

```

MakeDir 函数在 ftest.cc 中实现，是对 Create 创建目录的一个封装。

```

void
MakeDir(char *dirname)
{
    DEBUG('D', "Making directory.\n");
    fileSystem->Create(dirname, -1);
}

```

(6) 由于文件系统中现有的 List 也是只支持打印根目录中的内容，为了能够列出多级目录下的文件系统结构，我添加了一个名为 ListDir 的函数。

```

void
FileSystem::ListDir(char* directory)
{
    printf("Now List Directory: %s.\n", directory);
    Directory* dir = (Directory*)FindDir(strcat(directory, "/anything"));
    dir->List();
    delete dir;
}

```

现在我们已经在文件系统中基本实现了多级目录机制，让我们来对上述实现进行测试。为了简化测试，我编写了两个 Shell 脚本。

第一个脚本用于生成如下的文件结构。

- / (根目录)
 - folder/
 - test1/
 - dir/
 - medium (文件)
 - test2/
 - small (文件)
 - big (文件)
 - try/
 - small (文件)

脚本在拷贝文件的同时也会测试多级目录下的文件读取。脚本的具体内容如下。

```

#!/bin/sh

# this file saves in filesys/test/
cd ../

# use -V to disable verbose machine messages
echo "===" format the DISK ==="
./nachos -V -f

echo ""
echo ""
echo "===" create a directory called \"folder\" ==="
./nachos -V -d D -mkdir folder

echo ""
echo ""
echo "===" create a directory called \"tryr\" ==="
./nachos -V -d D -mkdir tryr

echo ""
echo ""
echo "===" create additional two directories called \"test1\" \"test2\" in \"folder\" ==="
./nachos -V -d D -mkdir folder/test1
./nachos -V -d D -mkdir folder/test2

echo ""
echo ""
echo "===" create additional directoriy called \"dir\" in \"folder/test1\" ==="
./nachos -V -d D -mkdir folder/test1/dir

echo ""
echo ""
echo "===" copy file \"big\" to \"folder\" ==="
./nachos -V -d D -cp test/big folder/big

```

```

./nachos -V -d D -p folder/big

echo ""
echo ""
echo "===" copy file \"medium\" to \"folder/test1/dir/medium\" ==="
./nachos -V -d D -cp test/medium folder/test1/dir/medium
./nachos -V -d D -p folder/test1/dir/medium

echo ""
echo ""
echo "===" copy file \"small\" to \"folder/test2/small\" ==="
./nachos -V -d D -cp test/small folder/test2/small
./nachos -V -d D -p folder/test2/small

echo ""
echo ""
echo "===" copy file \"small\" to \"small\" ==="
./nachos -V -d D -cp test/small small
./nachos -V -d D -p small

echo ""
echo ""
echo "===" list all files and directories in root directory ==="
./nachos -V -l

echo ""
echo ""
echo "===" list folder ==="
./nachos -V -ls folder

echo ""
echo ""
echo "===" list folder/test1 ==="
./nachos -V -ls folder/test1

```

```

echo ""
echo ""
echo "===" list folder/test2 ==="
./nachos -V -ls folder/test2

echo ""
echo ""
echo "===" list folder/try ==="
./nachos -V -ls folder/try

echo ""
echo ""
echo "===" list folder/test1/dir ==="
./nachos -V -ls folder/test1/dir

```

实际运行结果如下：

```
bash test4.sh
== format the DISK ==

== create a directory called "folder" ==
Making directory.
==== Creating directory folder, size is 960.

== create a directory called "tryr" ==
Making directory.
==== Creating directory try, size is 960.

== create additional two directories called "test1" "test2" in "folder" ==
Making directory.
==== Creating directory folder/test1, size is 960.
==== Finding directory folder in sector 1.
Making directory.
==== Creating directory folder/test2, size is 960.
==== Finding directory folder in sector 1.

== create additional directoriy called "dir" in "folder/test1" ==
Making directory.
==== Creating directory folder/test1/dir, size is 960.
==== Finding directory folder in sector 1.
==== Finding directory test1 in sector 11.

== copy file "big" to "folder" ==
==== Finding directory folder in sector 1.
==== Finding directory folder in sector 1.
==== Finding directory folder in sector 1.
This is the spring of our discontent.

This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.
This is the spring of our discontent.

== copy file "medium" to "folder/test1/dir/medium" ==
==== Finding directory folder in sector 1.
==== Finding directory test1 in sector 11.
==== Finding directory dir in sector 29.
==== Finding directory folder in sector 1.
==== Finding directory test1 in sector 11.
==== Finding directory dir in sector 29.
==== Finding directory folder in sector 1.
==== Finding directory test1 in sector 11.
==== Finding directory dir in sector 29.
This is the spring of our discontent.

== copy file "small" to "folder/test2/small" ==
==== Finding directory folder in sector 1.
==== Finding directory test2 in sector 11.
==== Finding directory folder in sector 1.
==== Finding directory test2 in sector 11.
==== Finding directory folder in sector 1.
==== Finding directory test2 in sector 11.
This is the spring of our discontent.

== copy file "small" to "small" ==
This is the spring of our discontent.

== list all files and directories in root directory ==
folder
try
small
```

```

==== list folder ====
Now List Directory: folder.
test1
test2
big

==== list folder/test1 ====
Now List Directory: folder/test1.
dir

==== list folder/test2 ====
Now List Directory: folder/test2.
small

==== list folder/try ====
Now List Directory: folder/try.

==== list folder/test1/dir ====
Now List Directory: folder/test1/dir.
medium

```

第二个脚本用于测试删除文件操作。函数会尝试用-r 删除多级目录下的内容、非空目录、普通文件、空目录，用-rm 删除文件、空目录、非空目录。脚本内容如下：

```

#!/bin/sh

cd ../
# this script must run after test4.sh

echo ""
echo ""
echo "==== try to directly remove a unempty folder in root dir, which should fail ==="
./nachos -V -d D -r folder

echo ""
echo ""
echo "==== try to directly remove an empty folder in root dir, named with try ==="
./nachos -V -d D -r try

echo ""
echo ""
echo "==== try to remove small in root ==="
./nachos -V -d D -r small

echo ""
echo ""
echo "==== try to remove the file \"folder/test2/small\" ==="
./nachos -V -d D -rm folder/test2/small

echo ""
echo ""
echo "==== try to remove the directory \"folder/test2\" ==="
./nachos -V -d D -r folder/test2

echo ""
echo ""
echo "==== try to remove the directory \"folder\" ==="
./nachos -V -d D -rm folder

echo ""
echo ""
echo "==== try to remove the directory \"folder/test1/dir\" ==="
echo "==== remove the file in \"folder/test1/dir\" first ==="
./nachos -V -d D -rm folder/test1/dir/medium
echo ""
./nachos -V -d D -rm folder/test1/dir

echo ""
echo ""
echo "==== list all files and directories in root directory ==="
./nachos -V -l

echo ""
echo ""
echo "==== list folder ===="
./nachos -V -ls folder

```

```

echo ""
echo ""
echo "===" list folder/test1 ==="
./nachos -V -ls folder/test1

echo ""
echo ""
echo "===" list folder/test2 ==="
./nachos -V -ls folder/test2

echo ""
echo ""
echo "===" list folder/test1/dir ==="
./nachos -V -ls folder/test1/dir

```

运行结果如下：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/filesys/test$ bash test4_remove.sh

==== try to directly remove a unempty folder in root dir, which should fail ====
====> You are trying to delete a directory in root directory!
====> This directory isn't empty. Operation failed!

==== try to directly remove an empty folder in root dir, named with try ====
====> You are trying to delete a directory in root directory!

==== try to remove small in root ===

==== try to remove the file "folder/test2/small" ===
====> Finding directory folder in sector 1.
====> Finding directory test2 in sector 11.
====> Successfully remove a multi-level file!

==== try to remove the directory "folder/test2" ===
====> Finding directory folder in sector 1.
====> '-r' can only be used to delete file/dir in root directory!

==== try to remove the directory "folder" ===
====> You are Trying to delete a directory!
====> This directory isn't empty. Operation failed!

==== try to remove the directory "folder/test1/dir" ===
==== remove the file in "folder/test1/dir" first ===
====> Finding directory folder in sector 1.
====> Finding directory test1 in sector 11.
====> Finding directory dir in sector 29.
====> Successfully remove a multi-level file!

====> Finding directory folder in sector 1.
====> Finding directory test1 in sector 11.
====> You are Trying to delete a directory!
====> Successfully remove a multi-level file!

==== list all files and directories in root directory ===
folder

==== list folder ===
Now List Directory: folder.
test1
test2
big
[

==== list folder/test1 ===
Now List Directory: folder/test1.

==== list folder/test2 ===
Now List Directory: folder/test2.

==== list folder/test1/dir ===
Now List Directory: folder/test1/dir.

```

可以看到，现在的文件系统可以正确地生成上述文件层级结构，并且可以按照设想的功能给出删除结果。因此可以认为上述的实现是正确的。

Exercise5

这一部分要求我们实现文件长度的动态变化。为了实现这个功能，我们可以设计为：初始为文件分配一定长度，当遇到文件长度不足的情况时，再进行扩展。这样，我们便不需要修改 Create 函数。

因此，我们主要需要完成的任务为，如果写入时遇到文件空间不足的情况，文件可以自动地加长，所以我们需要实现一个扩张文件长度的方法。具体实现如下图所示。因为文件伸长不仅需要分配扇区，还需要改变文件头中的属性内容，而且注意到文件头中维护了 numBytes、numSectors 两个涉及文件长度的属性，所以我把这个方法设置为了 FileHeader 的一个成员方法。实现思路如下。

- 首先，函数对传入的扩张参数进行必要的检查。注意到，我们规定，这个函数只在扩张时才调用，所以扩张的执行条件为：（1）传入的尺寸大于 0。（2）扩张后确实需要新的磁盘扇区。（3）扩张后的大小不超过目前磁盘空闲空间的大小。因此，检查部分的代码如下：

```
// ensure extension
if(ExpandBytes < 0)
{
    DEBUG('f', "====> Expand size should be a non-negative number!\n");
    return FALSE;
}

int afterBytes = numBytes+ExpandBytes;
int afterSectors = divRoundUp(afterBytes, SectorSize);
int clearSectors = freeMap->NumClear();

// don't need to get new sector
if(afterSectors == numSectors)
{
    DEBUG('f', "====> Previous file size is large enough!\n");
    numBytes = afterBytes;
    return TRUE;
}

// too large to hold the whole file
int deltaSectors = afterSectors-numSectors;
if(deltaSectors > clearSectors)
{
    DEBUG('f', "====> Empty size is not large enough to hold the file!\n");
    return FALSE;
}
```

- 检查完毕后，函数对不同大小的扩张进行讨论，应该覆盖以下几种情况：
 - 扩张后仍能被直接索引覆盖，这时直接分配直接索引区域即可。

```

DEBUG('f', "====> Start to expand file, expand size is %d, need %d sectors.\n", ExpandBytes, deltaSectors);
if(afterSectors <= NumDirect) // just need direct index
{
    for(int i=numSectors; i<afterSectors; ++i)
    {
        dataSectors[i] = freeMap->Find();
    }
}

```

如果扩张后不能被直接索引覆盖，当我们没开启多级索引时，我们可以直接返回扩张失败的提示（return FALSE）。

```

    else
    {
#ifndef USE_INDIRECT
        DEBUG('E', "====> There is no enough space in Sector Table.\n");
        return FALSE;
#else

```

否则，我们应该考虑以下几种情况：

(2) 从不使用索引扩张到只使用一级索引。首先把直接索引填满，然后添加多出的需要使用一级索引的部分。此时要注意判断能否装下索引扇区。

```

if(numSectors <= NumDirect && afterSectors <= NumDirect + LevelNum)
{
    DEBUG('E', "====> Need to use one-level indirect index **from 0 to 1**.\n");
    if(deltaSectors+1 > clearSectors)
    {
        DEBUG('f', "====> There is no enough space for additional file space and space table.\n");
        return FALSE;
    }

    // fill direct index
    for(int i=numSectors; i<NumDirect; ++i)
    {
        dataSectors[i] = freeMap->Find();
    }

    // one-level
    dataSectors[OneLevelIdx] = freeMap->Find();
    int OneLevelIndexes[LevelNum];
    for(int i=0; i<afterSectors - NumDirect; ++i)
    {
        OneLevelIndexes[i] = freeMap->Find();
    }
    synchDisk->WriteSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
}

```

(3) 从使用一级索引扩张到使用更多的二级索引，这时只需要利用循环来添加新的扇区项。

```

else if(afterSectors <= NumDirect + LevelNum)
{
    DEBUG('f', "====> Need to use one-level indirect index **from 1 to 1**.\n");

    int OneLevelIndexes[LevelNum];
    synchDisk->ReadSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
    for(int i=numSectors - NumDirect; i<afterSectors - NumDirect; ++i)
    {
        OneLevelIndexes[i] = freeMap->Find();
    }
    synchDisk->WriteSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
}

```

(4) 从不使用多级索引扩张到需要二级索引。这里的实现与 Allocate 类似，先填满直接索引和间接索引，再填充所需的二级索引。

```
else if(numSectors <= NumDirect && afterSectors <= NumDirect + LevelNum + LevelNum*LevelNum)
{
    DEBUG('E', "====> Need to use two-level indirect index ***from 0 to 2**.\n");
    int LeftSectors = afterSectors - NumDirect - LevelNum;
    int SecondLevelNum = divRoundUp(LeftSectors, LevelNum);
    if(clearSectors < numSectors + 1 + 1 + SecondLevelNum) // cannot hold double index and data
    {
        DEBUG('f', "====> There is no enough space for additional file space and space table.\n");
        return FALSE;
    }

    // direct
    for(int i=numSectors; i<NumDirect; ++i)
    {
        dataSectors[i] = freeMap->Find();
    }

    // one-level
    dataSectors[OneLevelIdx] = freeMap->Find();
    int OneLevelIndexes[LevelNum];
    for(int i=0; i<LevelNum; ++i)
    {
        OneLevelIndexes[i] = freeMap->Find();
    }
    synchDisk->WriteSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);

    // two-level
    dataSectors[TwoLevelIdx] = freeMap->Find();
    int TwoLevelIndexes[LevelNum];
    for(int i=0; i<SecondLevelNum; ++i)
    {
        TwoLevelIndexes[i] = freeMap->Find();
        int SingleLevelIndexes[LevelNum];
        for(int j=0; j<LevelNum && (j+1*LevelNum)<LeftSectors; ++j)
        {
            SingleLevelIndexes[j] = freeMap->Find();
        }
        synchDisk->WriteSector(TwoLevelIndexes[i], (char*)SingleLevelIndexes);
    }
    synchDisk->WriteSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);
}

else if(numSectors > NumDirect + LevelNum || afterSectors > NumDirect + LevelNum + LevelNum*LevelNum)
```

(5) 从使用一级索引扩张到需要二级索引。先填满一级索引，再填充需要的二级索引即可。此时要注意判断能否装下索引扇区。

```

else if(numSectors <= NumDirect + LevelNum && afterSectors <= NumDirect + LevelNum + LevelNum*LevelNum)
{
    DEBUG('E', "====> Need to use two-level indirect index **from 1 to 2**.\n");

    int LeftSectors = afterSectors - NumDirect - LevelNum;
    int SecondLevelNum = divRoundUp(LeftSectors, LevelNum);
    if(clearSectors < numSectors + 1 + SecondLevelNum) // cannot hold double index and data
    {
        DEBUG('f', "====> There is no enough space for additional file space and space table.\n");
        return FALSE;
    }

    // fill one level
    int OneLevelIndexes[LevelNum];
    synchDisk->ReadSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);
    for(int i=numSectors - NumDirect; i<LevelNum; ++i)
    {
        OneLevelIndexes[i] = freeMap->Find();
    }
    synchDisk->WriteSector(dataSectors[OneLevelIdx], (char*)OneLevelIndexes);

    // two-level
    dataSectors[TwoLevelIdx] = freeMap->Find();
    int TwoLevelIndexes[LevelNum];
    for(int i=0; i<SecondLevelNum; ++i)
    {
        TwoLevelIndexes[i] = freeMap->Find();
        int SingleLevelIndexes[LevelNum];
        for(int j=0; j<LevelNum && (j+i*LevelNum)<LeftSectors; ++j)
        {
            SingleLevelIndexes[j] = freeMap->Find();
        }
        synchDisk->WriteSector(TwoLevelIndexes[i], (char*)SingleLevelIndexes);
    }
    synchDisk->WriteSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);
}

else if(afterSectors <= NumDirect + LevelNum + LevelNum*LevelNum) ...

```

(6) 二级索引内部扩张。这时要讨论两种情况，即是否需要一个新的二级索引项。如果需要新的索引项，还要判断能否装下新加入的扇区。

这里要注意下标的计算，非常容易出错。

```

else if(afterSectors <= NumDirect + LevelNum + LevelNum*LevelNum)
{
    DEBUG('f', "====> Need to use two-level indirect index **from 2 to 2**.\n");

    int LeftSectors = afterSectors - NumDirect - LevelNum;
    int SecondLevelNum = divRoundUp(LeftSectors, LevelNum);

    int prevLeftSectors = numSectors - NumDirect - LevelNum;
    int prevSecondLevelNum = divRoundUp(prevLeftSectors, LevelNum);

    int TwoLevelIndexes[LevelNum];
    synchDisk->ReadSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);

    if(SecondLevelNum == prevSecondLevelNum)
    {
        int SingleLevelIndexes[LevelNum];
        synchDisk->ReadSector(TwoLevelIndexes[prevSecondLevelNum-1], (char*)SingleLevelIndexes);
        //printf("----%d %d %d----\n", SecondLevelNum, LeftSectors, prevSecondLevelNum , prevLeftSectors);
        for(int j=(prevLeftSectors%LevelNum); (j+(SecondLevelNum-1)*LevelNum)<LeftSectors; ++j)
        {
            SingleLevelIndexes[j] = freeMap->Find();
        }
        //printf("---- %d %d ----\n", SingleLevelIndexes[(prevLeftSectors%LevelNum)-1], SingleLevelIndexes[prevLeftSectors%LevelNum]);
        synchDisk->WriteSector(TwoLevelIndexes[prevSecondLevelNum-1], (char*)SingleLevelIndexes);
    }
}

```

```

else
{
    if(deltaSectors+1 > clearSectors)
    {
        DEBUG('f', "====> There is no enough space for additional file space and space table.\n");
        return FALSE;
    }
    // fill prev second-level index first
    int SingleLevelIndexes[LevelNum];
    synchDisk->ReadSector(TwoLevelIndexes[prevSecondLevelNum-1], (char*)SingleLevelIndexes);
    for(int j=(prevLeftSectors%LevelNum); j>0 && j<LevelNum; ++j)
    {
        SingleLevelIndexes[j] = freeMap->Find();
    }
    synchDisk->WriteSector(TwoLevelIndexes[prevSecondLevelNum-1], (char*)SingleLevelIndexes);

    // fill new second-level indexes
    for(int i=prevSecondLevelNum; i<SecondLevelNum; ++i)
    {
        TwoLevelIndexes[i] = freeMap->Find();
        int SingleLevelIndexes[LevelNum];
        for(int j=0; j<LevelNum && (j+i*LevelNum)<LeftSectors; ++j)
        {
            SingleLevelIndexes[j] = freeMap->Find();
        }
        synchDisk->WriteSector(TwoLevelIndexes[i], (char*)SingleLevelIndexes);
    }
    synchDisk->WriteSector(dataSectors[TwoLevelIdx], (char*)TwoLevelIndexes);
}

```

最后，为了实现的严谨性，当仍然出现不够使用的情况时，使用 ASSERT 语句报错。

```

else
{
    // should not occur, since we have judged whether all empty sectors can hold this extension.
    DEBUG('f', "====> There is no enough space in Sector Table.\n");
    ASSERT(FALSE);
}

```

实现了动态伸长的方法后，我们应该把它应用到文件的写操作中。我们可以把它添加到 OpenFile 类中的 WriteAt 函数。添加内容如下

```

if(position+numBytes > fileLength)
{
    DEBUG('f', "====> need to expand file length!\n");

    // get bitmap
    BitMap* freeMap = new BitMap(NumSectors);
    OpenFile* freeMapFile = new OpenFile(FreeMapSector);
    freeMap->FetchFrom(freeMapFile);

    // try to extend file, should success
    ASSERT(hdr->ExpandFileSize(freeMap, numBytes));

    // flush change to disk
    hdr->WriteBack(hdr->GetHeaderSector());
    freeMap->WriteBack(freeMapFile);

    // update
    delete freeMapFile;
    delete freeMap;
    fileLength = hdr->FileLength();

}

```

这样我们就实现了文件动态伸长的机制。为了进行测试，我们可以使用 Nachos 自带的 -t 选项，这个选项会触发 ftest.cc 中的测试函数。由于 Nachos 原来的函数为 u 发覆盖从 0 级索引直接扩展到 2 级索引的情况，所以我对函数进行了一些修改。

```
void
PerformanceTest()
{
    stats->Print();
    printf("\n\n\n");

    printf("----- Starting file system performance test of sequential increasing -----\\n");
    FileWrite();
    FileRead();
    if (!fileSystem->Remove(FileName)) {
        printf("Perf test: unable to remove %s\\n", FileName);
        return;
    }

    printf("\n\n\n");

    printf("----- Starting file system performance test of jumpin increasing -----\\n");
    LongWrite();
    LongRead();
    if (!fileSystem->Remove(FileNameLong)) {
        printf("Perf test: unable to remove %s\\n", FileNameLong);
        return;
    }

    printf("\n\n\n");
    stats->Print();
}

#define FileName      "TestFile"
#define FileNameLong   "TestFileLong"
#define Contents      "1234567890"
#define ContentSize    strlen(Contents)
#define FileSize       ((int)(ContentSize * 5000))

static void
FileWrite()
{
    OpenFile *openFile;
    int i, numBytes;

    printf("Sequential write of %d byte file, in %d byte chunks\\n",
           FileSize, ContentSize);
    if (!fileSystem->Create(FileName, 0))
    {
        printf("Perf test: can't create %s\\n", FileName);
        return;
    }
    openFile = fileSystem->Open(FileName);
    if (openFile == NULL)
    {
        printf("Perf test: unable to open %s\\n", FileName);
        return;
    }
    for (i = 0; i < FileSize; i += ContentSize)
    {
        numBytes = openFile->Write(Contents, ContentSize);
        if (numBytes < ContentSize)
        {
            printf("Perf test: unable to write %s\\n", FileName);
            delete openFile;
            return;
        }
    }
    delete openFile; // close file
}
```

```

static void
FileRead()
{
    OpenFile *openFile;
    char *buffer = new char[ContentSize];
    int i, numBytes;

    printf("Sequential read of %d byte file, in %d byte chunks\n",
        FileSize, ContentSize);

    if ((openFile = fileSystem->Open(FileName)) == NULL)
    {
        printf("Perf test: unable to open file %s\n", FileName);
        delete [] buffer;
        return;
    }
    for (i = 0; i < FileSize; i += ContentSize)
    {
        numBytes = openFile->Read(buffer, ContentSize);
        // printf("%s\n", buffer);
        if ((numBytes < ContentSize) || strncmp(buffer, Contents, ContentSize))
        {
            printf("Perf test: unable to read %s\n", FileName);
            delete openFile;
            delete [] buffer;
            return;
        }
    }
    delete [] buffer;
    delete openFile;    // close file
}

```

```

char* str = "1234567890";
char str1[10001] = "";

void
LongWrite()
{
    int numBytes;
    OpenFile *openFile;

    if (!fileSystem->Create(FileNameLong, 0))
    {
        printf("Perf test: can't create %s\n", FileNameLong);
        return;
    }
    openFile = fileSystem->Open(FileNameLong);
    if (openFile == NULL)
    {
        printf("Perf test: unable to open %s\n", FileNameLong);
        return;
    }

    for(int i=0; i<(1000); ++i)
        strcat(str1, str);

    numBytes = openFile->Write(str1, 10000);
    if (numBytes < 10000)
    {
        printf("Perf test: unable to write %s\n", FileNameLong);
        delete openFile;
        return;
    }
    delete openFile;
}

```

```

void
LongRead()
{
    int numBytes;
    OpenFile *openFile;
    char *buffer = new char[10001];

    if ((openFile = fileSystem->Open(FileNameLong)) == NULL)
    {
        printf("Perf test: unable to open file %s\n", FileNameLong);
        delete [] buffer;
        return;
    }

    numBytes = openFile->Read(buffer, 10000);
    if ((numBytes < 10000) || strncmp(buffer, str1, 10000))
    {
        printf("%d %d\n", strlen(buffer), strlen(str1));
        //printf("%s\n\n%s", buffer, str1);
        printf("Perf test: unable to read %s\n", FileNameLong);
        delete openFile;
        delete [] buffer;
        return;
    }
    delete [] buffer;
    delete openFile;
}

```

可以看到，它会把一段长度为 10 的字符串重复写入 5000 次，然后读取这个文件，并把它从文件系统中删除。接下来，函数会直接创建一个 Nachos 文件，并写入 Nahcos 磁盘中。

我们知道，测试的第一部分中，只要能够成功完成 0-1、1-2 级索引的伸展，并成功匹配，即可证明 0-0、0-1、1-1、1-2、2-2 扩展的正确性。第二部分中，只要能够成功完成 0-2 扩展，并成功匹配，即可证明 0-2 扩展的正确性。

实际运行结果如下。

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/filesys$ ./nachos -f -V
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/filesys$ ./nachos -t -V -d E
Ticks: total 1110, idle 1000, system 110, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

----- Starting file system performance test of sequential increasing -----
Sequential write of 50000 byte file, in 10 byte chunks
==> Need to use one-level indirect index **from 0 to 1**.
==> Need to use two-level indirect index **from 1 to 2**.
Sequential read of 50000 byte file, in 10 byte chunks

----- Starting file system performance test of jumpin increasing -----
==> Need to use two-level indirect index **from 0 to 2**.

Ticks: total 778565520, idle 775789500, system 2776020, user 0
Disk I/O: reads 51672, writes 20847
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

```

可以看到，函数可以成功执行上述所有情况的扩展（没有不匹配或者无法写入的提醒输出），并且在读取文件的匹配时没有出现错误，因此，可以认为，上述在多级索引机制下的文件扩展实现是正确的。

Exercise6

这一部分要求我们借鉴异步访问磁盘的工作原理，实现一个异步的控制台。我们可以分为两步来完成

(一) 源代码阅读

首先，让我们来分析一下 synchdisk.h 和 synchdisk.cc 中实现的异步磁盘类。

这个类中包含了三个私有成员：一个磁盘类对象，即我们需要维护的原始磁盘文件；一个信号量，用于控制磁盘的读写操作；一个互斥锁，用于控制线程对磁盘的互斥访问。

```
private:  
    Disk *disk;           // Raw disk device  
    Semaphore *semaphore; // To synchronize requesting thread  
                           // with the interrupt handler  
    Lock *lock;          // Only one read/write request  
                           // can be sent to the disk at a time
```

除此之外，这个类中还包含一些方法。让我们来逐个进行分析。

1. 构造函数和析构函数

Synchdisk 类的构造函数中对三个私有成员变量进行了初始化。信号量的初值为 0。

```
SynchDisk::SynchDisk(char* name)  
{  
    semaphore = new Semaphore("synch disk", 0);  
    lock = new Lock("synch disk lock");  
    disk = new Disk(name, DiskRequestDone, (int) this);  
}
```

初始化的磁盘以 DiskRequestDone 作为同步处理函数，它的参数就是这个指向互斥磁盘对象自身的指针。

```
static void  
DiskRequestDone (int arg)  
{  
    SynchDisk* disk = (SynchDisk *)arg;  
  

```

DiskRequestDone 这个函数调用传入的异步磁盘指针所指向的对象的 RequestDone 函数，用于释放磁盘的读写权限。

```
void  
SynchDisk::RequestDone()  
{  
    semaphore->V();  
}
```

这个类的析构函数就是释放它的成员变量。

```
SynchDisk::~SynchDisk()
{
    delete disk;
    delete lock;
    delete semaphore;
}
```

2. ReadSector

当线程调用这个方法读取磁盘时，线程首先试图获取互斥锁，以保证对磁盘的互斥访问，然后线程向磁盘申请一个读请求，并等待中断处理程序提示它读请求已经完成。最后，线程释放互斥锁，放行其他线程。

```
void
SynchDisk::ReadSector(int sectorNumber, char* data)
{
    lock->Acquire();           // only one disk I/O at a time
    disk->ReadRequest(sectorNumber, data);
    semaphore->P();            // wait for interrupt
    lock->Release();
}
```

3. WriteSector

这个函数与 ReadSector 思路相同，只是把读请求换成了写请求。

```
void
SynchDisk::WriteSector(int sectorNumber, char* data)
{
    lock->Acquire();           // only one disk I/O at a time
    disk->WriteRequest(sectorNumber, data);
    semaphore->P();            // wait for interrupt
    lock->Release();
}
```

4. RequestDone

已在前面分析过，用于在完成磁盘的读/写操作时，释放磁盘信号量。这个函数应该由 Nachos 的时钟中断触发。

(二) SynchConsole 的实现

原始的 console 类位于 machine/console.h 中。借鉴上面的实现思路，我们可以尝试用类似的手段对 Console 类进行封装。

Console 类的内容如下。核心的操作函数为 PutChar 和 GetChar，这两个函数分别为输入命令行和读取命令行。可以预想到，这两个是 SynchConsole 类必须要封装以避免竞争的函数。

然而，Disk 中只有一个 HandleInterrupt 用于响应中断并处理完成的读/写请求，因此 SynchDisk 也实现了 DiskRequestDone 的外部函数和 RequestDone 的内部函数，作为中断的

Handler。与 Disk 不同，Console 中存在着 readHandler 和 writeHandler 两个处理函数等待传入，因此可以预见我们需要分别实现读/写结束后的处理函数。

```
class Console {
public:
    Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail,
            VoidFunctionPtr writeDone, int callArg);
    // initialize the hardware console device
    ~Console(); // clean up console emulation

    // external interface -- Nachos kernel code can call these
    void PutChar(char ch); // Write "ch" to the console display,
    // and return immediately. "writeHandler"
    // is called when the I/O completes.

    char GetChar(); // Poll the console input. If a char is
    // available, return it. Otherwise, return EOF.
    // "readHandler" is called whenever there is
    // a char to be gotten

    // internal emulation routines -- DO NOT call these.
    void WriteDone(); // internal routines to signal I/O completion
    void CheckCharAvail();

private:
    int readFileNo; // UNIX file emulating the keyboard
    int writeFileNo; // UNIX file emulating the display
    VoidFunctionPtr writeHandler; // Interrupt handler to call when
    // the PutChar I/O completes
    VoidFunctionPtr readHandler; // Interrupt handler to call when
    // a character arrives from the keyboard
    int handlerArg; // argument to be passed to the
    // interrupt handlers
    bool putBusy; // Is a PutChar operation in progress?
    // If so, you can't do another one!
    char incoming; // Contains the character to be read,
    // if there is one available.
    // Otherwise contains EOF.
};
```

进一步阅读 console 的代码后，可以发现，Console 在构造以后会首先在中断处理函数中“预定”一个“可读检查”操作（最终调用 CheckCharAvail）。每次进行读检查时，又会预定下一次读检查。当检查时发现有可以读的内容，CheckCharAvail 会调用 readHandler，允许 Console 对象进行 GetChar 操作。

当获取到读内容之后，目前的控制台会调用 PutChar 把读取到的字符直接写入输出文件。PutChar 中“预定”了一次写完毕检查操作（最终调用 WriteDone），WriteDone 中会调用 writeHandler，通知控制台线程读取完毕。

通过以上分析，我们可以这样来设计 SynchConsole。

首先，SynchConsole 中应该包含一个 Console 对象和一个用于不同线程之间互斥的锁。此外，为了实现上述的可读等待和写完毕等待，我们应该设计两个信号量。

```
private:
    Console* console; // raw console object
    Lock* lock; // mutex among different threads
    Semaphore* semReadAvail;
    Semaphore* semWriteDone;
```

类似于 SynchDisk，我们可以这样来设计 SynchConsole 的成员方法。

在实现成员方法之前，我们应该设计两个全局的 Handler 函数，作为 console 的

readHandler 和 **writeHandler**。这两个方法把传入的参数作为 **SynchConsole** 对象的指针，再调用这个 **SynchConsole** 对象内部的处理函数。

```
// helper functions for two handlers
static void
SynchReadAvail(int arg)
{
    SynchConsole *ptr = (SynchConsole *)arg;
    ptr->ReadAvail();
}

static void
SynchWriteDone(int arg)
{
    SynchConsole *ptr = (SynchConsole *)arg;
    ptr->WriteDone();
}
```

类比于 **SynchDisk**，处理函数的功能就是释放 **SynchConsole** 类中的两个信号。

```
void
SynchConsole::WriteDone()
{
    semWriteDone->V();
}

void
SynchConsole::ReadAvail()
{
    semReadAvail->V();
}
```

构造函数的功能就是初始化四个成员变量。初始化 **console** 时，需要传入上面实现的两个处理函数的指针，使用的参数为当前的 **SynchDisk** 对象的指针，传入 **this** 即可。

```
SynchConsole::SynchConsole(char* ReadFile, char* WriteFile)
{
    lock = new Lock("synch console lock");
    semReadAvail = new Semaphore("synch console read avail", 0);
    semWriteDone = new Semaphore("synch console write done", 0);
    console = new Console(ReadFile, WriteFile, SynchReadAvail, SynchWriteDone, (int)this);
}
```

析构函数只需释放上面分配的对象。

```
SynchConsole::~SynchConsole()
{
    delete console;
    delete lock;
    delete semReadAvail;
    delete semWriteDone;
}
```

为了实现原有控制台的功能，我们需要实现两个封装函数。封装的方法借鉴了 **progtest.cc** 中的 **ConsoleTest** 的语句顺序和 **SynchDisk** 中的两个异步请求的实现。

对于读请求，在获取了线程间的互斥锁后，**首先要获取可读的信号，确保有可读的内容以后再调用 GetChar**，最后释放线程互斥锁。

```

char
SynchConsole::SynchGetChar()
{
    lock->Acquire();
    semReadAvail->P(); // wait for available chars
    char ch = console->GetChar();
    lock->Release();
    return ch;
}

```

对于写请求，在获取了线程间的互斥锁后，**首先调用 PutChar 试图向输出文件写入内容，再试图获取写完毕信号**。当确保写入完毕以后，再释放线程互斥锁。

```

void
SynchConsole::SynchPutChar(char ch)
{
    lock->Acquire();
    console->PutChar(ch);
    semWriteDone->P(); // wait for write done
    lock->Release();
}

```

最后，我们可以仿照 progtest.cc 中的 ConsoleTest，来实现 SynchConsoleTest 函数，以测试我们的实现。

```

void
SynchConsoleTest(char* in, char* out)
{
    char ch;
    synchConsole = new SynchConsole(in, out);

    for(;;)
    {
        ch = synchConsole->SynchGetChar();

        if(ch == '!')
        {
            synchConsole->SynchPutChar(']');
            synchConsole->SynchPutChar('\n');
            return;
        }
        if(ch == '~')
        {
            synchConsole->SynchPutChar('[');
            synchConsole->SynchPutChar('\n');
            break;
        }
        else
        {
            synchConsole->SynchPutChar(ch);
        }
    }
    printf("You've put '~' in synch console!\n");
}

```

为了方便触发，我在 main.cc 中加入了对-sc 命令行参数的解析，用于调用 SynchConsoleTest。

```

else if(!strcmp(*argv, "-sc"))
{
    if(argc==1)
        SynchConsoleTest(NULL, NULL);
    else{
        ASSERT(argc > 2);
        SynchConsoleTest(*(argv+1), *(argv+2));
        argCount = 3;
    }
    interrupt->Halt();
}

```

运行之后，测试控制台的功能，结果如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/filesys$ ./nachos -V -sc
hello world
hello world
this is nachos synch console test
this is nachos synch console test
now test quit with output
now test quit with output
~
[
You've put '~' in synch console!
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/filesys$ ./nachos -V -sc
hello world
hello world
this is nachos synch console test
this is nachos synch console test
now test quit without test
now test quit without test
!
]
```

可以看到，SynchConsole 可以像原有的控制台那样正确处理输入，可以认为上面的实现是正确的。

Exercise7

这一部分要求我们实现文件访问的同步互斥，主要要求为：多线程访问时文件位置互不干扰、读写都是原子操作、序列化的（读写之间的互斥）、删除前保证没有线程打开这个文件。

对于第一个要求，回顾 OpenFile 的构造函数，我们可以发现，**每次根据文件头创建打开文件对象时，对象内部都会维护一个文件位置**，不同对象间的文件位置互不干扰。因此，**原有的机制已经支持了文件位置互不干扰的要求**。

```
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    hdr->SetHeaderSector(sector);
    seekPosition = 0;
}
```

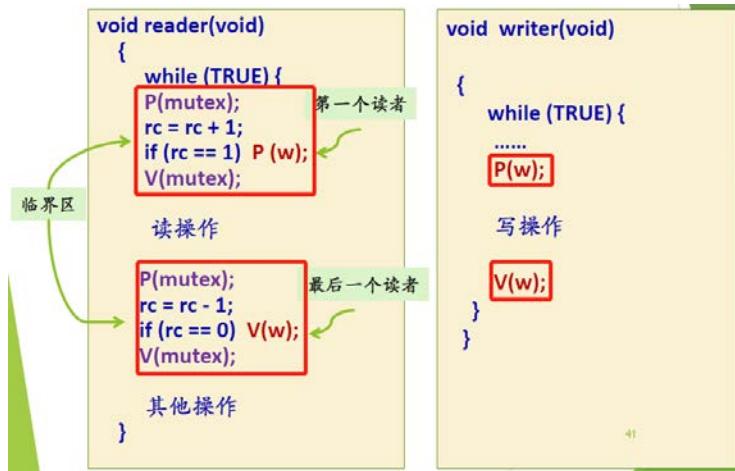
对于第二个要求，实际上就是要求对文件的访问实现读者-写者问题的解决方案。我们不妨采用第一类读者-写者问题的思路：当存在一个写者时，其他所有读者写者全部等待；当读者数量不为 0 时，读者会优先进行读操作，直到最后一个读者结束文件操作后，写者才能进行写操作。

注意到**我们先前的实现中，限制了文件头大小为 1 个扇区**。因此，**为了实现对文件的互斥请求，我们可以通过对文件头所在的扇区添加互斥锁的方式来解决这个问题**。大致思路是：为每个扇区（每个扇区都可能被用作文件头）维护一个互斥信号量和读者的计数变

量。当第一个读者线程到达时，获取对文件头扇区的互斥锁；当最后一个读者退出时，释放对这个文件头的互斥锁。第一类读者-写者问题中，需要在第一个读者进入时获取临界区的互斥锁，在最后一个读者退出时释放这个互斥锁，所以**我们还需要为每个扇区维护一个读者个数的计数变量。这个变量也需要一个互斥锁来保护。**

注意到，**打开文件时，每个线程都会维护自己的打开文件类，所以 zhi 在 OpenFile 类添加互斥是行不通的。**

根据第一类读者写者问题的解法，我们可以把互斥部分封装为函数。



根据上面的构思，我们可以首先在 synchDisk 类中添加一些用于互斥的成员变量和函数，具体实现如下

```

Semaphore* mutex[NumSectors]; // protect file
int readerCnt[NumSectors]; // reader number
Lock* readerLock; // protect readercnt

void PlusReader(int sector);
void MinusReader(int sector);
void PlusWriter(int sector);
void MinusWriter(int sector);

void
SynchDisk::PlusWriter(int sector)
{
    DEBUG('C', "-> writer try to get lock\n");
    // printf("====> writer try to get lock %d\n", sector);
    mutex[sector]->P();
    // printf("====> Now the writer start to write %d.\n", sector);
}

void
SynchDisk::MinusWriter(int sector)
{
    // printf("====> Now the writer finish writing %d.\n", sector);
    DEBUG('C', "-> writer release lock\n");
    mutex[sector]->V();
}

```

```

void
SynchDisk::PlusReader(int sector)
{
    DEBUG('C', "-> reader try to get lock\n");
    // printf("====> reader try to get lock %d\n", sector);
    readerLock->Acquire();
    readerCnt[sector]++;
    // printf("====> reader cnt is %d\n", readerCnt[sector]);
    if(readerCnt[sector]==1)
    {
        DEBUG('C', "-> the first reader need to acquire mutex semaphore.\n");
        //printf("====> the first reader need to acquire mutex semaphore %d.\n", sector);
        mutex[sector]->P();
    }
    readerLock->Release();
}

void
SynchDisk::MinusReader(int sector)
{
    DEBUG('C', "-> reader release lock\n");
    readerLock->Acquire();
    readerCnt[sector]--;
    // printf("====> reader cnt is %d\n", readerCnt[sector]);
    if(readerCnt[sector]==0)
    {
        DEBUG('C', "-> the last reader need to release mutex semaphore.\n");
        //printf("====> the last reader need to release mutex semaphore %d.\n", sector);
        mutex[sector]->V();
    }
    readerLock->Release();
}

```

接下来，我们需要把这些互斥操作添加到读写操作中。读写是在 OpenFile 类中实现的，所以我们需要修改 Read 和 Write 函数。为了更好地测试并发性，我在两个操作中还添加了主动让出 CPU 的语句，需要开启 CONCURRENT_TEST 宏。

```

int
openFile::Read(char *into, int numBytes)
{
    synchDisk->PlusReader(hdr->GetHeaderSector());

    int result = ReadAt(into, numBytes, seekPosition);

#ifdef CONCURRENT_TEST
    // if we want to test concurrency,
    // we can force the thread to yield before leaving critical area
    DEBUG('C', "====> Reader yield CPU.\n");
    currentThread->Yield();
#endif

    seekPosition += result;

    synchDisk->MinusReader(hdr->GetHeaderSector());
    return result;
}

int
openFile::Write(char *into, int numBytes)
{
    synchDisk->PlusWriter(hdr->GetHeaderSector());

    int result = WriteAt(into, numBytes, seekPosition);

#ifdef CONCURRENT_TEST
    // if we want to test concurrency,
    // we can force the thread to yield before leaving critical area
    DEBUG('C', "====> Writer yield CPU.\n");
    currentThread->Yield();
#endif

    seekPosition += result;

    synchDisk->MinusWriter(hdr->GetHeaderSector());
    return result;
}

```

为了实现第三个要求，我们需要再维护一个访问者（读者+写者）的计数变量。在创建一个 OpenFile 对象时，对这个变量加 1；关闭 OpenFile 时，把这个变量-1。同样，为了保证对计数变量操作的原子性，我们需要加一个锁。

```

Lock* numberLock; // protect numCnt
int visitorCnt[NumSectors]; // reader & writer numbers

OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    hdr->SetHeaderSector(sector);
    seekPosition = 0;

    synchDisk->numberLock->Acquire();
    synchDisk->visitorCnt[sector]++;
    // printf("----> sector %d number is %d\n", sector, synchDisk->visitorCnt[sector]);
    synchDisk->numberLock->Release();
}

OpenFile::~OpenFile()
{
    hdr->WriteBack(hdr->GetHeaderSector());

    synchDisk->numberLock->Acquire();
    synchDisk->visitorCnt[hdr->GetHeaderSector()]--;
    // printf("----> sector %d number is %d\n", hdr->GetHeaderSector(),
    //        synchDisk->visitorCnt[hdr->GetHeaderSector()]);
    synchDisk->numberLock->Release();

    delete hdr;
}

```

对于上述添加的所有变量，我们还要在构造函数中进行相应的初始化，在析构函数中进行相应的释放。

```

SynchDisk::SynchDisk(char* name)
{
    semaphore = new Semaphore("synch disk", 0);
    lock = new Lock("synch disk lock");
    disk = new Disk(name, DiskRequestDone, (int) this);

    readerLock = new Lock("reader cnt lock");
    numberLock = new Lock("visitor cnt lock");
    for(int i=0; i<NumSectors; ++i)
    {
        readerCnt[i] = 0;
        visitorCnt[i] = 0;
        mutex[i] = new Semaphore("sector lock", 1);
    }
}

SynchDisk::~SynchDisk()
{
    delete disk;
    delete lock;
    delete semaphore;

    delete readerLock;
    for(int i=0; i<NumSectors; ++i)
    {
        delete mutex[i];
    }
    delete numberLock;
}

```

同时，在执行删除操作之前，我们要对相应的计数变量进行判断，如果不是 0，那么删除失败。

```
synchDisk->numberLock->Acquire();
if(synchDisk->visitorCnt[sector]!=0)
{
    DEBUG('C', "==== This file is still being used by %d visitors!\n", synchDisk->visitorCnt[sector]);
    synchDisk->numberLock->Release();
    delete fileHdr;
    return FALSE;
}
synchDisk->numberLock->Release();

freeMap = new BitMap(NumSectors);
freeMap->FetchFrom(freeMapFile);

fileHdr->Deallocate(freeMap);      // remove data blocks
freeMap->Clear(sector);           // remove header block
directory->Remove(name);

freeMap->WriteBack(freeMapFile);    // flush to disk
directory->WriteBack(directoryFile); // flush to disk
```

现在我们已经基本上完成了这个 Exercise 的要求，下面我们来编写程序对上述的功能进行测试。为了能够更方便地测试，我仿照 Exercise5 的-t 选项，在 ftest.cc 中编写了测试程序，并在 main.cc 中添加了相应的命令行选项。

我编写了两段测试程序。

第一段测试程序用于检验当同时存在读写线程时的情况。此外，这个程序还可以用来检验多写者同时读的效果，和当仍然存在访问者时，试图删除文件的效果。大致思路为：主线程 Fork 几个用于读文件的子线程，与此同时，主线程不断向文件里写内容（这里设置的是写 3 次）。由于开启了上面提到的调试选项，主线程每写一次就会让出一次 CPU。写完以后，主线程会不断尝试删除文件，删除失败则让出 CPU，否则正常退出。

```
void
ConcurrencyTest()
{
    printf("Multi-reading test starts!\n");

    Thread* t1 = new Thread("reader1");
    t1->Fork(read, (void*)1);
    Thread* t2 = new Thread("reader2");
    t2->Fork(read, (void*)2);
    Thread* t3 = new Thread("reader3");
    t3->Fork(read, (void*)3);

    printf("Write file\n");
    FileWrite();

    currentThread->Yield();

    printf("try to remove file\n");
    while(!fileSystem->Remove(fileName))
    {
        printf("Fail to delete.\n");
        currentThread->Yield();
    }
    printf("Finish deleting.\n");
}

void
read(int num)
{
    for(int i=0; i<3; ++i)
    {
        printf("Thread %d is reading. Cnt: %d\n", num, i+1);
        FileRead();
    }
}
```

以上代码中使用的 `FileWrite` 和 `FileRead` 与 Exercise5 中使用的相同，把写次数缩短到了 3 次。

```
#define FileName      "TestFile"
#define FileNameLong   "TestFileLong"
#define Contents       "1234567890"
#define ContentSize    strlen(Contents)
#define FileSize       ((int)(ContentSize * 3))
```

对于我们的要求，应该出现的结果是：主线程在写的过程中，虽然让出了 CPU，但是读线程因为写未完成而阻塞。当主线程写完后，尝试删除时，发现仍然有访问者（未读完的读线程），则主线程会不断地尝试删除，直到所有的读线程结束操作，删除成功，所有线程退出。（**注意，这里由于在 Write 和 Read 中开启了 Yield，所以应该出现上述结果，如果未开启，由于线程调度算法的问题，可能产生不同的结果**）

实际测试效果如下：

```
./nachos -mt -d C -V
Multi-reading test starts!
Write file
Sequential write of 30 byte file, in 10 byte chunks
Thread 1 is reading. Cnt: 1
Sequential read of 30 byte file, in 10 byte chunks
Thread 2 is reading. Cnt: 1
Sequential read of 30 byte file, in 10 byte chunks
Thread 3 is reading. Cnt: 1
Sequential read of 30 byte file, in 10 byte chunks
-> writer try to get lock
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
-> reader try to get lock
-> the first reader need to acquire mutex semaphore.
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
-> reader try to get lock
-> reader try to get lock
====> Writer yield CPU.
-> writer release lock
try to remove file
====> Reader yield CPU.
-> reader release lock
-> reader try to get lock
====> Reader yield CPU.
-> reader release lock
-> reader try to get lock
====> Reader yield CPU.
-> reader release lock
-> reader try to get lock
====> Reader yield CPU.
Fail to delete.
====> Reader yield CPU.
-> reader release lock
-> reader try to get lock
====> Reader yield CPU.
-> reader release lock
-> reader try to get lock
====> Reader yield CPU.
```

```
--> reader release lock
-> reader try to get lock
==== This file is still being used by 3 visitors!
Fail to delete.
==== Reader yield CPU.
-> reader release lock
==== Reader yield CPU.
-> reader release lock
==== Reader yield CPU.
-> reader release lock
-> the last reader need to release mutex semaphore.
==== This file is still being used by 3 visitors!
Fail to delete.
Thread 2 is reading. Cnt: 2
Sequential read of 30 byte file, in 10 byte chunks
-> reader try to get lock
-> the first reader need to acquire mutex semaphore.
==== Reader yield CPU.
-> reader release lock
-> the last reader need to release mutex semaphore.
-> reader try to get lock
-> the first reader need to acquire mutex semaphore.
Thread 3 is reading. Cnt: 2
Sequential read of 30 byte file, in 10 byte chunks
-> reader try to get lock
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
Thread 1 is reading. Cnt: 2
Sequential read of 30 byte file, in 10 byte chunks
-> reader try to get lock
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
-> reader try to get lock
==== This file is still being used by 3 visitors!
Fail to delete.
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
==== This file is still being used by 3 visitors!
Fail to delete.
==== Reader yield CPU.
-> reader release lock
==== Reader yield CPU.
-> reader release lock
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
```

```
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
==== This file is still being used by 3 visitors!
Fail to delete.
==== Reader yield CPU.
-> reader release lock
==== Reader yield CPU.
-> reader release lock
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
-> the last reader need to release mutex semaphore.
==== This file is still being used by 3 visitors!
Fail to delete.
Thread 3 is reading. Cnt: 3
Sequential read of 30 byte file, in 10 byte chunks
-> reader try to get lock
-> the first reader need to acquire mutex semaphore.
==== Reader yield CPU.
-> reader release lock
-> the last reader need to release mutex semaphore.
-> reader try to get lock
-> the first reader need to acquire mutex semaphore.
Thread 1 is reading. Cnt: 3
Sequential read of 30 byte file, in 10 byte chunks
-> reader try to get lock
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
Thread 2 is reading. Cnt: 3
Sequential read of 30 byte file, in 10 byte chunks
-> reader try to get lock
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
==== This file is still being used by 3 visitors!
Fail to delete.
==== Reader yield CPU.
-> reader release lock
-> reader try to get lock
==== Reader yield CPU.
-> reader release lock
```

```

-> reader try to get lock
====> Reader yield CPU.
-> reader release lock
-> reader try to get lock
====> This file is still being used by 3 visitors!
Fail to delete.
====> Reader yield CPU.
-> reader release lock
====> Reader yield CPU.
-> reader release lock
====> Reader yield CPU.
-> reader release lock
-> the last reader need to release mutex semaphore.
====> This file is still being used by 3 visitors!
Fail to delete.
Finish deleting.

```

可以看到，整个流程与我们预想的相同。

但是上面的线程无法测试多线程同时写的效果。因此我编写了第二段代码。

这段代码的流程为：主线程首先创建并写入一个文件。然后，主线程 Fork 两个对这个文件进行写操作的线程，并像第一段程序那样试图删除文件。

```

void
ConcurrencyTest1()
{
    FileWrite();

    printf("----- rewrite ----- \n");
    Thread* t1 = new Thread("reader1");
    t1->Fork(write, (void*)1);
    Thread* t2 = new Thread("reader2");
    t2->Fork(write, (void*)2);

    currentThread->Yield();

    printf("try to remove file\n");
    while(!fileSystem->Remove(FileName))
    {
        printf("Fail to delete.\n");
        currentThread->Yield();
    }
    printf("Finish deleting.\n");
}

void
write(int num)
{
    printf("Thread %d is reading.\n", num);
    OpenFile *openFile;
    int i, numBytes;

    printf("Sequential write of %d byte file, in %d byte chunks\n",
    FileSize, ContentSize);
    openFile = fileSystem->Open(FileName);
    if (openFile == NULL)
    {
        printf("Perf test: unable to open %s\n", FileName);
        return;
    }
    for (i = 0; i < FileSize; i += ContentSize)
    {
        numBytes = openFile->Write(Contents, ContentSize);
        if (numBytes < ContentSize)
        {
            printf("Perf test: unable to write %s\n", FileName);
            delete openFile;
            return;
        }
    }
    delete openFile; // close file
}

```

按照要求，我们应该得到的结果是，线程依次写，且写的时候不会退出。（**注意，这里由于在 Write 和 Read 中开启了 Yield，所以应该出现上述结果，如果未开启，由于线程调度算法的问题，可能产生不同的结果**）

实际运行效果如下：

```
./nachos -mt1 -d C -V
Sequential write of 30 byte file, in 10 byte chunks
-> writer try to get lock
seek position 0
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
seek position 10
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
seek position 20
====> Writer yield CPU.
-> writer release lock
----- rewrite -----
Thread 1 is reading.
Sequential write of 30 byte file, in 10 byte chunks
Thread 2 is reading.
Sequential write of 30 byte file, in 10 byte chunks
try to remove file
-> writer try to get lock
seek position 0
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
-> writer try to get lock
====> This file is still being used by 2 visitors!
Fail to delete.
seek position 10
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
====> This file is still being used by 2 visitors!
Fail to delete.
seek position 20
====> Writer yield CPU.
-> writer release lock
====> This file is still being used by 2 visitors!
Fail to delete.
seek position 0
====> Writer yield CPU.
-> writer release lock

-> writer try to get lock
====> This file is still being used by 1 visitors!
Fail to delete.
seek position 10
====> Writer yield CPU.
-> writer release lock
-> writer try to get lock
====> This file is still being used by 1 visitors!
Fail to delete.
seek position 20
====> Writer yield CPU.
-> writer release lock
====> This file is still being used by 1 visitors!
Fail to delete.
Finish deleting.
```

可以看到，整个流程与我们预想的相同。（**为了区分不同文件，我额外输出了每个打开文件中的文件位置，可以看到文件位置输出依次从 0 连续变到 20，证明是单个文件连续写的**）。

通过以上测试，我们可以看到，Nachos 可以按照我们预设的要求来处理并发读写操作，可以认为上述的实现是正确的。

Challenge

这一部分我完成了 Challenge2

Challenge2

这一部分要求我们实现管道。我们可以使用一个特殊的管道文件来模拟这一机制。管道文件作为一个与位图文件和根目录文件相似的文件，首先在文件系统的构造时指定它的文件头位置（这里我规定，管道文件的文件头占用标号为 2 的扇区），然后在文件系统中添加读管道和写管道的方法。

具体实现如下。

首先，添加一个管道文件头扇区的宏，固定管道文件头再文件系统的位置，便于查找，并在文件系统的构造函数中初始化管道文件头。

```
#define PipeSector 2

BitMap *freeMap = new BitMap(NumSectors);
Directory *directory = new Directory(NumDirEntries);
FileHeader *mapHdr = new FileHeader;
mapHdr->CreateInit("BHdr");
FileHeader *dirHdr = new FileHeader;
dirHdr->CreateInit("DHdr");

// pipe
FileHeader* pipHdr = new FileHeader;
pipHdr->CreateInit("PHdr");

// First, allocate space for FileHeaders for the directory and bitmap
// (make sure no one else grabs these!)
freeMap->Mark(FreeMapSector);
freeMap->Mark(DirectorySector);

// pipe
freeMap->Mark(PipeSector);

// Second, allocate space for the data blocks containing the contents
// of the directory and bitmap files. There better be enough space!

ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

ASSERT(pipHdr->Allocate(freeMap, 0));

DEBUG('f', "Writing headers back to disk.\n");
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);

// pipe
pipHdr->WriteBack(PipeSector);
```

接下来，我们需要添加读管道内容的方法。大致思路就是，利用固定的文件头打开管道文件，读取一定长度的内容，长度不允许超过目前管道文件中内容的长度。

```

int
FileSystem::ReadPipe(char* dst)
{
    FileHeader* pipHdr = new FileHeader;
    pipHdr->FetchFrom(PipeSector);

    OpenFile* pipe = new OpenFile(PipeSector);
    pipe->Read(dst, pipHdr->FileLength());
    dst[pipHdr->FileLength()] = '\0';

    // printf("len is %d\n", pipHdr->FileLength());
    DEBUG('P', "Read from pipe file\n");
    delete pipHdr;
    delete pipe;
    return strlen(dst);
}

```

然后，添加向管道写的方法，流程大致与读类似。

```

void
FileSystem::WritePipe(char* data, int length)
{
    data[length] = '\0';

    OpenFile* pipe = new OpenFile(PipeSector);
    pipe->Write(data, length+1);

    DEBUG('P', "Write into pipe file\n");
    delete pipe;
}

```

注意，这里由于之前实现了文件的信息维护，所以不需要在上面两个函数中维护管道文件的文件头中的 numBytes。

此外，我们每次都从文件位置为 0 的地方进行管道的读写，这样可以节省文件空间。对于超出已写入长度的部分，我们不对内容进行任何假设（基于与访问未初始化内存相似的思想）。

接下来，对我们实现的管道进行简单的测试。我们首先写入一些内容，然后读取它们，并写入 Nachos 的文件中，最后在 main 中添加 “-pt” 选项以触发这个测试。

```

void
PipeTest()
{
    fileSystem->Create(NAME, MAXLEN);
    OpenFile* file = fileSystem->Open(NAME);

    // fileSystem->Print();

    for(int i=0; i<3; ++i)
    {
        WritePipeTest();
        ReadPipeTest(file);
    }

    fileSystem->Print();
}

```

```
void
ReadPipeTest(OpenFile* file)
{
    printf("Read from Pipe, save into file %s\n", NAME);

    char data[MAXLEN+1];
    int length = fileSystem->ReadPipe(data);
    data[length] = '\0';
    printf("Pipe content is: %s, len is %d\n", data, length);

    file->Write(data, length);
}

void
WritePipeTest()
{
    char data[MAXLEN+1];
    scanf("%s", data);

    printf("Write to Pipe, data is %s\n", data);
    fileSystem->WritePipe(data, strlen(data));
}

} else if(!strcmp(*argv, "-pt")) {
    PipeTest();
}
```

可以看到，我们可以向管道中写入内容，同时也可以从管道中正常读取内容并写入 Nachos 文件系统，并且也可以把从管道读到的字符串写回 Nachos 的某个文件。可以认为上述实现是正确的。

内容二：遇到的困难以及收获

1. 在实验的最开始，我无法正常显示文件系统的输出。在阅读了大量代码后，我发现问题出现在 Nachos 的命令行解析中。**Nachos 在 main 中的命令行解析分为两部分，如果开启了 THREADS 宏，那么第一部分的命令行解析会把 argc 的值改变，造成无法进行第二部分的命令行解析。**所以我在 filesystem 的 Makefile 文件中删除了-DTHREADS 选项。

2. 变量过多带来的失误。在实现二级索引时，由于命名的变量过多，加之命名方式过于相近，因此在写回/赋值时出现了变量混用的情况，且为调试带来了

很大的困难。这一点以后我应该注意。

3. 时间获取的实现。在维护文件时间时，一开始我并不是很了解 C 中自动生成时间的函数，在查阅资料后才了解到 ctime 这个库。此外，**注意到 ctime 中生成时间的原始结果多了一个\n，但这是不必要的，且会占用本就宝贵的文件头空间，所以储存时，我删去了这个换行符，从而增加了 1 个扇区标号的位置。**

4. 对文件路径分割的实现。刚开始入手这一 Exercise 时，我并不知道 C 中也有路径解析的库函数，在查阅 stackoverflow 等网站后才了解到了这一功能。

5. 多级索引及多级索引中的自由扩展的实现。在完成 Exercise5 时，一开始我简单地认为扩展的实现与 Allocate 完全一致。但是，**由于我的实现涉及了一级索引和二级索引，因此扩展可能有 6 种情况。在完成了分类讨论后，我的扩展才真正成功。**

6. 多级目录实现时对删除功能的实现。在一开始，我简单地认为，递归删除就是直接在文件上级目录中删除当前目录的目录项，但是这样做会导致目录中内容占用的扇区无法释放，造成磁盘空间的浪费。真正的递归删除应该确实需要涉及到递归的实现（个人认为），由于时间关系，我并没能很好地完成这一功能，因此我对文件夹删除进行了特判，不允许删除时目录文件中有有效项，这样规避了上面的问题。

内容三：对课程或 Lab 的意见和建议

希望可以把引导手册进一步完善，现在每次阅读代码的时候需要我们递归阅读和寻找其他额外的代码，但我们往往不知道应该去哪里能找到，这样浪费了许多不必要的时间。我觉得可以把某些常用函数（或者重要辅助函数）的位置标注出来，这样可以节省遍历搜索的时间。

也希望可以为调研类的作业提供更多的提示和说明。现在我们只能漫无目的地去遍历搜索百度、google 的结果，我们无法确认博客等网站表述内容的正确性和严谨性，而且有的调研题目很难找到相关博客。让我们在短时间内查询所有体系结构/操作系统的手册，对于还有许多其他课程的本科生来说在时间上不是很现实。所以希望课程可以提供更多相关资源，或者提示我们应该去看手册的哪些章节，去找什么样的权威资料。这样既可以让我们学到准确的知识，也可以提升

我们的学习效率。

同时希望可以整理一遍整个实验的文档还有代码。首先，现在 Lab 的要求和题目说明就类似“已知 $1+1=2$ ，请求解下列积分”这类漫画中出现的情境。简单的题干背后经常包含巨大的代码实现量和额外的代码阅读量，以及为了完成单个功能所需要的额外查阅资料的任务量。题目描述简单也让人并不知道应该如何下手，这让一名正在学习操作系统的学生成为完成 Lab 的开始经常感到十分茫然。我觉得教学用的实验系统即使不能与时俱进地更新换代，但教学团队内部实现一遍以后再给出更多的提示语和引导，应该是可以做到的吧？就像大二上的 ICS 那样，每个实验都有专门的助教来维护。虽然贵教学团队可能没有 ICS 助教团队那样庞大的团体，但是至少代代相传一份“参考实现”，并给出适当的提示，对于一个沿用多年的实验框架应该不困难吧？

其次，现在的代码中的缩进是 Tab 和空格混用，而且命令行解析的实现十分混乱，甚至出现了不同功能的命令行解析之间会互相干扰的问题。我觉得这些与操作系统实习并没有很大的关联的问题不应该让学生来承担修改的任务。这门课程已经使用了相当长时间的 Nachos，经过这么长时间的教学积累，却还是下发给我们放入 VSCode 中甚至连缩进都没办法对齐的代码，真的让人十分失望。

内容四：参考文献

1. Wikipedia—— inode:

<https://en.wikipedia.org/wiki/Inode>

2. Getting file extension in C:

<https://stackoverflow.com/questions/5309471/getting-file-extension-in-c>

3. time() 获取系统时间

<https://www.cnblogs.com/zkz1742/p/5641759.html>

4. 《现代操作系统》 文件系统部分

5. How To Quickly Generate A Large File On The Command Line (With Linux)

<https://skorks.com/2010/03/how-to-quickly-generate-a-large-file-on-the-command-line-with-linux/>

6. How to get the directory path and file name from a absolute path in C on Linux

<https://www.systutorials.com/how-to-get-the-directory-path-and-file-name-from-a-absolute-path-in-c-on-linux/>