Lab4 文件系统 实验报告

内容一：总体概述

本次Lab的内容是在阅读源代码的基础上，通过对Nachos文件系统代码的修改完善其文件系统。主要实现功能：扩展文件属性、突破文件名长度限制、扩展文件长度、实现多级目录、动态调整文件长度、实现文件系统的同步互斥访问机制、文件打开计数、文件读写锁、多线程pipe通信机制。

内容二：任务完成情况

任务完成列表

Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise7	Challenge2
Y	Y	Y	Y	Y	Y	Y	Y

Exercise 1 调研

- `code/filesys/filesys.h`和`code/filesys/filesys.cc`实现了Nachos系统中的文件系统`FileSystem`，代码中定义了两个文件系统，`FILESYS_STUB`宏定义的是基于宿主机Linux上的文件系统，调用LInux文件系统的功能暂时实现Nachos文件系统的功能，等到Nachos本身的文件系统可用。另一个是Nachos本身的文件系统，成员变量有记录空闲磁盘块的文件`freeMapFile`和目录文件`directoryFile`。成员方法包括：
  - `FileSystem(bool format);`构造函数，接受一个bool类型的参数format，含义为磁盘是否需要初始化，若为真，则需要初始化磁盘，new出一个记录空闲磁盘块的`freemap`和目录`directory`,将它们分别写入文件；若为假，则只需要打开`FreeMap`和`Directory`对应的文件。
  - `bool Create(char *name, int initialSize);`创建文件，参数为文件名和字节数(初始版本中文件大小固定)。创建文件的步骤是：确定文件是否已经存在→为文件头分配一个磁盘扇区→为数据区分配磁盘扇区→将文件名加入目录中→将新的文件头写入磁盘→更新Bitmap和目录，写回磁盘。创建成功后返回True,如文件已存在或磁盘空间不足，则返回`False`.
  - `OpenFile* Open(char *name);`根据文件名返回打开文件指针，步骤是通过目录查找文件头所在的扇区，再读取文件头，返回打开文件的指针。
  - `bool Remove(char *name);`根据文件名删除文件，在目录中查找文件名，如果文件名存在，查找到存放文件头的扇区，读取文件头后删除，根据文件头释放数据区占据的磁盘块，从目录中删去文件名，更新目录和freeMap，写回磁盘文件。
  - `void List();`列出根目录下所有文件名。
  - `void Print();`打印文件系统信息(空闲区表bitmao、目录内容、每个文件的文件头和数据)。

- `code/filehdr/filehdr.h` 和 `code/filesys/filehdr.cc`:定义了Nachos系统中的文件头`FileHeader`类。该类成员变量有文件的字节数`numBytes`、文件数据占用的磁盘块的个数以及`numSectors`、记录每个磁盘块的数组`dataSectors`,成员方法是对文件头的一系列操作函数，包括:
  - `Allocate(BitMap *bitMap, int fileSize);`为文件分配空闲磁盘块，参数包括文件大小(字节数`fileSize`)和空闲磁盘块的位图指针`bitmap`,其操作是根据文件大小计算需要的磁盘块数目，若剩余的空闲磁盘块数目不足则返回`False`,否则调用bitmap的find方法逐一分配空闲磁盘块，返回true;
  - `Deallocate(BitMap *freeMap)`释放文件占据的磁盘块，更新维护空闲磁盘块的位图;
  - `FetchFrom`:从磁盘中读出本文件头;
  - `WriteBack`:将本文件头内容写回磁盘
  - `ByteToSector(int offset)`:返回距开头offset字节数据所在的磁盘块指针;
  - `FileLength`:返回文件大小(字节数); `Print`: 打印文件信息
- `directory.cc`和`directory.h`:定义了Nachos文件系统中目录类`Directory`，其形式是由`DirectoryEntry`（包含文件名、文件头所在磁盘块）组成的数组，成员方法有:
  - `Directory(int size)`构造函数，初始化目录表
  - `FetchFrom(OpenFile *file)`:从磁盘中读取目录
  - `WriteBack(OpenFile *file)`: 将自身写回磁盘文件
  - `FindIndex(char *name)`:根据文件名寻找相应的页表项位置(数组下标)，如果没有对应返回-1
  - `Find(char *name)`: 通过调用`FindIndex`根据文件名寻找相应的文件头所在扇区，如果没有对应返回-1
  - `Add(char *name, int newSector)`:向目录中添加文件，接受文件名和文件头所在磁盘块两个参数，若添加成功返回TRUE,若添加失败（目录已满或已有此文件名），则返回FALSE;
  - `Remove(char *name)`:从目录中删除文件，成功返回TRUE，失败(没有此文件)则返回FALSE;
  - `List()`打印本目录包含的所有文件名,`Print()`打印本目录包含的所有文件详细信息。
- `openfile.h` 和 `openfile.cc`:定义Nachos系统用于读写文件的数据结构`OpenFile`,其私有成员包括指向文件头的指针`hdr`和当前偏移量`seekPosition`(距离文件开头的字节数)，公有成员是一些读写文件的方法，包括:
  - `OpenFile(int sector)`构造函数，根据给定的磁盘块从中读取文件头，将`seekPosition`设为0;
  - `~OpenFile()`析构函数
  - `SeekPosition(int position)`:设定读写指针位置
  - `ReadAt(char *into, int numBytes, int position)`:从position位置开始读取numBytes个字节，写入into所指的位置。过程中要检查读取字节数是否不为正数或超过文件大小:

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

接着计算起始和结束的磁盘块，调用磁盘的ReadSector函数将磁盘内容读入缓冲区中，再复制到目标位置，返回成功读取的字节数。

- `Read(char *into, int numBytes)`:封装ReadAt方法，从当前读写指针位置读指定字节数到目标位置，更新读写指针；
- `WriteAt(char *from, int numBytes, int position)`:以from指向的位置为源，向position位置写入`numBytes`个字节的数据。过程中要检查写入的字节数是否不为正数或超过文件大小：

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

接着计算起始和结束的磁盘块，设立缓冲区，检查要写的第一个和最后一个磁盘块是否对齐，如果未对齐则读入缓冲区；再将from指向的内容复制到缓冲区，再调用磁盘的`WriteSector`方法逐个将缓冲区内容写入磁盘块，删除缓冲区，返回成功写入的字节数。

- `Length()`：返回打开的文件长度(字节数)。

- `code/userprog/bitmap.h`和`code/userprog/bitmap.cc`:位图数据结构，在文件系统中用于记录磁盘块的使用情况。

Exercise 2 扩展文件属性

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

修改directory.h，增加文件类型、突破文件名长度限制

```
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'
    bool isDirectory;
    char* name;
};
```

为此需要修改directory.cc中的Add函数

```
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;
            table[i].isDirectory = FALSE;    //新增 默认是新建一个文件
            // strncpy(table[i].name, name, FileNameMaxLen);
            table[i].name = name;    //修改name变量赋值方法
            table[i].sector = newSector;
        return TRUE;
  }
    return FALSE; // no space.  Fix when we have extensible files.
}
```

修改filehdr.h，在文件头类中新增“创建时间”、“上次访问时间”、“上次修改时间”。

```
class FileHeader{
    ...
    ...
    public:
    time_t createTime;  // Create time of the file
    time_t lastAccessTime;  // Last access time of the file
    time_t lastWriteTime; // Last write time of the file
}
```

为此需要修改filehdr.cc，在给文件分配sector的时候初始化“创建时间”。

```
bool
FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    time_t currentTime = time(NULL);
    createTime = currentTime;
    ...
}
```

并且修改openfile.cc，在文件读写的时候修改“上次访问时间”、“上次修改时间”。

```
int
OpenFile::Read(char *into, int numBytes)
{
    int result = ReadAt(into, numBytes, seekPosition);
    time_t currentTime = time(NULL);
    hdr->lastAccessTime = currentTime;
    seekPosition += result;
    hdr->WriteBack(hdrSector);
    return result;
}

int
OpenFile::Write(char *into, int numBytes)
{
```

```
    int result = WriteAt(into, numBytes, seekPosition);
    seekPosition += result;
    time_t currentTime = time(NULL);
    hdr->lastAccessTime = currentTime;
    hdr->lastWriteTime = currentTime;
    hdr->WriteBack(hdrSector);
    // printf("Writed %d\n", result);
    return result;
}
```

测试：

修改main.cc，将#ifdef THREADS 到 #endif注释掉，让命令行参数能被后面的文件系统模块识别。

修改fstest.cc

```
#define FileSize  ((int)(ContentSize * 50))    //原本是5000，暂时还没实现单文件大于4kb，所以改成50
static void
FileWrite()
{
    ...
    if (!fileSystem->Create(FileName, FileSize))  //这里给文件分配它需要的空间，不知为何原代码是0
    ...
}
```

运行可见正常进行了读写操作

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
Ticks: total 1070, idle 1000, system 70, 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
Sequential write of 500 byte file, in 10 byte chunks
createTime: 1607842268, lastAccessTime: 128, lastWriteTime: 16
createTime: 1607842268, lastAccessTime: 1607842268, lastWriteTime: 1607842268
Sequential read of 500 byte file, in 10 byte chunks
createTime: 1607842268, lastAccessTime: 1607842268, lastWriteTime: 1607842268
createTime: 1607842268, lastAccessTime: 1607842268, lastWriteTime: 1607842268
Ticks: total 2194520, idle 2186020, system 8500, user 0
Disk I/O: reads 123, writes 160
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

```
Machine halting!

Ticks: total 2194520, idle 2186020, system 8500, user 0
Disk I/O: reads 123, writes 160
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

Exercise 3 扩展文件长度

> 改直接索引为间接索引，以突破文件长度不能超过4KB的限制。

修改filehdr.h，在文件头中增加一个表示“下一文件头所在扇区”的变量，为-1时表示已经是最后一个文件头。

```
private:
    ...
    int dataSectors[NumDirect];   // Disk sector numbers for each data
          // block in the file
    int nextHdrSector;  //下一文件头所在扇区
```

为此需要修改filehdr.cc中的`Allocate`，`Deallocate`，`ByteToSector`，`Print`等函数。需要注意的是第一个文件头、中间的文件头、最后的文件头写回操作的不同。

```
bool
FileHeader::Allocate(BitMap *freeMap, int fileSize)
{

    time_t currentTime = time(NULL);
    createTime = currentTime;
    lastAccessTime = currentTime;
    lastWriteTime = currentTime;
    numBytes = fileSize;
    numSectors  = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < numSectors + numSectors / NumDirect) //nextHdr也需要sector!
  return FALSE;   // not enough space

    FileHeader *hdr = this;
    FileHeader *nextHdr;
    int curSec;     //用于帮助第一个之后的文件头writeback
    int i;
    for (i = 0; i < numSectors; i++){
```

```
        if(i % NumDirect == 0 && i != 0){   //每个文件头里放k * (NumDirect - 1)个索引
            nextHdr = new FileHeader;
            hdr->nextHdrSector = freeMap->Find();
            // printf("allocate %d\n", hdr->nextHdrSector);
            if(i != NumDirect)  hdr->WriteBack(curSec);     //第一个文件头不需要也无法在这里writeback
            curSec = hdr->nextHdrSector;
            hdr = nextHdr;
        }
        hdr->dataSectors[i % NumDirect] = freeMap->Find();
        // printf("allocate %d\n", hdr->dataSectors[i % NumDirect]);
    }
    hdr->nextHdrSector = -1;    //最后一个文件头
  if(i >= NumDirect)  hdr->WriteBack(curSec);     //最后一个文件头writeBack回磁盘
    return TRUE;
}
```

测试：将fstest.cc中文件长度设置为

```
#define FileSize  ((int)(ContentSize * 5000))
```

可见现在也能正常运行，disk i/o数是之前的一千倍

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
Ticks: total 1070, idle 1000, system 70, 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
Sequential write of 50000 byte file, in 10 byte chunks
sector: 5
createTime: 1607912100, lastAccessTime: 1607912103, lastWriteTime: 1607912103
Sequential read of 50000 byte file, in 10 byte chunks
sector: 5
createTime: 1607912100, lastAccessTime: 1607912105, lastWriteTime: 1607912103
Ticks: total 1388466520, idle 1384334490, system 4132030, user 0
Disk I/O: reads 122400, writes 15334
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1388466520, idle 1384334490, system 4132030, user 0
Disk I/O: reads 122400, writes 15334
```

```
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

Exercise 4 实现多级目录

在FileSystem和Directory类创建文件相关的函数中加入判断：文件是否是目录文件

```
bool Create(char *name, int initialSize, bool isDirectory);

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

在fileSystem类中实现两个方法`findDirectory`，`splitFileName`，在输入一个文件的绝对路径后取得它所在的目录、文件名。

```
//假如name == "pku/ss/5430"且存在目录pku/ss/，则得到这个目录的头文件所在的sector
int FileSystem::findDirectory(char *name, int curDirSec){
    char *tmp = name;
    if(*tmp == '/'){    //如果文件名最开始是'/'，去掉它
        tmp++;
        name++;
    }
    int len = 0;
    while(*tmp!='/' && *tmp!='\0' &&tmp!=0){
        tmp++;
        len++;
    }
    char *chdDirName = new char[len + 1];
    memcpy(chdDirName, name, len);  //三次递归分别是pku, ss, 5430 两次目录一次文件
    chdDirName[len] = '\0';
    // printf("curDirSec: %d\n", curDirSec);
    // printf("chdDirName: %s tmp: %s\n", chdDirName, tmp);
    if(tmp == 0 || *tmp == '\0'){   //说明name是文件名，chdDirName也就是文件名
        return curDirSec;
    }
    else{   //chdDirName是一个目录名
        Directory *curDir = new Directory(DirectoryFileSize);
        OpenFile *curDirFile = new OpenFile(curDirSec);
        curDir->FetchFrom(curDirFile);
        int chdDirSec = curDir->Find(chdDirName);
        if(chdDirSec == -1)     //当前目录下没有这个子目录
            return -1;
```

```
        return findDirectory(tmp, chdDirSec);
    }
}
//假如name == "pku/ss/5430"且存在目录pku/ss/，则得到字符串“5430”
char* FileSystem::splitFileName(char *name){
    char *tmp = name;
    if(*tmp == '/'){
        tmp++;
        name++;
    }
    int len = 0, last = 0;
    while(*tmp!='\0' &&tmp!=0){
        tmp++;
        len++;
        if(*tmp == '/')
            last = len+1;
    }
    char *chdDirName = new char[len + 1 - last];
    memcpy(chdDirName, name + last, len - last);
    chdDirName[len - last] = '\0';
    // printf("True Name: %s\n", chdDirName);
    return chdDirName;
}
```

然后对`FileSystem::Create`进行修改，利用上面两个函数使得创建文件时直接在子目录中添加。

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

    DEBUG('f', "Creating file %s, size %d\n", name, initialSize);
    int fileDirSec = findDirectory(name, DirectorySector);  //
    name = splitFileName(name);   //

    if(fileDirSec == -1){
        printf("***** %s's dir not exist*****\n", name);
        ASSERT(false);
    }
    OpenFile *fileDirFile = new OpenFile(fileDirSec);
    directory = new Directory(NumDirEntries);
    directory->FetchFrom(fileDirFile);  //
    ...
}
```

修改Directory类的List、Print方法

```
void
Directory::List()
{
   for (int i = 0; i < tableSize; i++)
  if (table[i].inUse){
        char *type = "File";
        if(table[i].isDirectory)
            type = "Directory";
        printf("%s\t %s\n", table[i].name, type);
        if(table[i].isDirectory){
            printf("***in %s***\n", table[i].name);
            OpenFile *chdDirFile = new OpenFile(table[i].sector);
            Directory *chdDir = new Directory((sizeof(DirectoryEntry) * 10));
            chdDir->FetchFrom(chdDirFile);
            chdDir->List();
            printf("***in %s***\n", table[i].name);
        }
    }

}

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("Name: %s is a directory ? %d\n",table[i].name,
table[i].isDirectory);
        if(table[i].isDirectory){
            printf("Name: %s is a directory\n");
            OpenFile *chdDirFile = new OpenFile(table[i].sector);
            Directory *chdDir = new Directory((sizeof(DirectoryEntry) * 10));
            chdDir->FetchFrom(chdDirFile);
            chdDir->Print();
        }
  }
    printf("\n");
    delete hdr;
}
```

测试：

在fstest.cc中新增一个测试函数

```
void multDirTest(){
    printf("Starting file system mult Directory  test:\n");
    OpenFile *rootDirFile = new OpenFile(1);
    Directory *rootDir = new Directory((sizeof(DirectoryEntry) * 10));
    rootDir->FetchFrom(rootDirFile);
    rootDir->List();
    printf("*******creating*******:\n");
    // return;
    fileSystem->Create("pkuHere", SectorSize, false);
    fileSystem->Create("pkuss", SectorSize, true);
    fileSystem->Create("pkueecs", SectorSize, true);
    fileSystem->Create("pkuss/yj3", SectorSize, true);
    fileSystem->Create("pkuss/yj3/yzh", SectorSize, false);
    rootDir->FetchFrom(rootDirFile);
    rootDir->List();
}
```

测试可见运行成功，并且扇区分配符合期望

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -f -tmd
Starting file system mult Directory  test:
*******creating*******:
pkuHere  File on sector: 5
pkuss    Directory on sector: 7
***in pkuss***
yj3      Directory on sector: 11
***in yj3***
yzh      File on sector: 13
***in yj3***
***in pkuss***
pkueecs  Directory on sector: 9
***in pkueecs***
***in pkueecs***
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Exercise 5 文件动态增长

注意到固定文件长度读/写溢出检测是在openfile.cc的`OpenFile::WriteAt` `OpenFile::ReadAt`发生的，因此只需在`OpenFile::WriteAt`修改：

```
int
OpenFile::WriteAt(char *from, int numBytes, int position)
{
    ...
    if ((numBytes <= 0) || (position >= fileLength)){ //如果文件长度不够用
        // ASSERT(false);
        fileSystem->AddSector(hdr, hdrSector);    //需要把这个函数写在FileSystem
类，因为要传递freeMap、freeMapFile进去
        return WriteAt(from, numBytes, position);   //增加好了之后重新写一次
    }
  ...
}
```

需要把`AddSector`函数写在FileSystem类，因为要传递freeMap、freeMapFile进去

```
void FileSystem::AddSector(FileHeader *hdr, int hdrSector){
    BitMap *freeMap = new BitMap(NumSectors);
    freeMap->FetchFrom(freeMapFile);
    hdr->AddSector(freeMap, freeMapFile, hdrSector);
}
```

下面是AddSector具体过程

```
void FileHeader::AddSector(BitMap *freeMap, OpenFile* freeMapFile, int
FirstHdrSec){
    time_t currentTime = time(NULL);
    createTime = currentTime;
    lastAccessTime = currentTime;
    lastWriteTime = currentTime;

    if (freeMap->NumClear() < 2) //nextHdr需要一个sector
  return ;    // not enough space

    FileHeader *hdr = this;
    FileHeader *nextHdr;
    int curSec = FirstHdrSec;     //用于帮助最后一个文件头writeback，因为它有改动
    int i;
    for (i = 0; i < numSectors; i++){   //先找到最后一个文件头
        if(i % NumDirect == 0 && i != 0){
            nextHdr = new FileHeader;
            nextHdr->FetchFrom(hdr->nextHdrSector);
            curSec = hdr->nextHdrSector;
```

```
            hdr = nextHdr;
        }
    }
    if(i % NumDirect == 0 && i != 0){      //如果当前最后一个文件头刚好满了，需要新增一个文件头
        int nextHdrSec = freeMap->Find();
        hdr->nextHdrSector = nextHdrSec;     //先把当前最后文件头的下一文件头sec从-1改成新值
        hdr->WriteBack(curSec);      //把当前最后文件头写回
        nextHdr = new FileHeader;
        hdr = nextHdr;
        hdr->nextHdrSector = -1;     //新的最后文件头的下一文件头sec为-1
        curSec = nextHdrSec;     //新的最后文件头所在sec就是刚刚分配的
        printf("Added a new fileHeader on sector: %d\n", curSec);
    }
    hdr->dataSectors[i % NumDirect] = freeMap->Find();
    printf("Added a new Sector: %d\n", hdr->dataSectors[i % NumDirect]);
    hdr->WriteBack(curSec);      //最后一个文件头writeBack回磁盘
    freeMap->WriteBack(freeMapFile);
    numBytes += SectorSize;
    numSectors  += 1;
    return;

}
```

测试：

测试函数写在fstest.cc，创建一个原长度为0的文件，然后向里面写入超过一个SectorSize的内容：

```
void DynamicTest(){
    fileSystem->Create("dynamic.txt", 0, false);
        OpenFile *openFile;
    int i, numBytes;

    openFile = fileSystem->Open("dynamic.txt");
    if (openFile == NULL) {
        printf("Dynamic test: unable to open %s\n", FileName);
        return;
    }
    for (i = 0; i < 15; i++) {
        numBytes = openFile->Write(Contents, ContentSize);
        printf("Write %d bytes:%s\n",10,Contents);
    }
    delete openFile;  // close file
}
```

运行结果可见成功进行

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -f -td
Added a new Sector: 6
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Write 10 bytes:1234567890
Added a new Sector: 7
Write 10 bytes:1234567890
Write 10 bytes:1234567890
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Exercise 6 源代码阅读

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

> The Disk simulation of Nachos is asynchronous, that is when we sending the "request" to Disk the Disk will return immediately. When the job is complete (`DiskDone`), Disk then trigger the interrupt.
>
> But in the multi-thread scenaro this is hard to maintain. So here comes the `SynchDisk`. It is based on the `Disk` and guarantees the following condition:
>
> - Only one thread can access Disk at a time
> - The thread will wait until the request is finished

synchdisk类将磁盘类与信号量、锁封装在一起，互斥锁lock保证对磁盘的读写操作是互斥进行的。当一次读/写请求后产生中断，通过调用中断处理函数`RequestDone`将信号量semaphore释放，从而使在等待信号量的其他线程得以访问磁盘。

```
class SynchDisk {
  public:
    SynchDisk(char* name);        // Initialize a synchronous disk,
          // by initializing the raw Disk.
    ~SynchDisk();     // De-allocate the synch disk data
```

```
    void ReadSector(int sectorNumber, char* data);
              // Read/write a disk sector, returning
              // only once the data is actually read
          // or written.  These call
              // Disk::ReadRequest/WriteRequest and
          // then wait until the request is done.
    void WriteSector(int sectorNumber, char* data);

    void RequestDone();     // Called by the disk device interrupt
          // handler, to signal that the
          // current disk operation is complete.

  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
};
```

以readSector为例，首先申请锁，然后调用磁盘类的readRequest执行读操作，并在其中预安排中断来执行信号量的V操作。然后P操作等待中断发生，最后释放锁。

```
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();
}
```

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

创建一个SynchConsole类，将Console类与互斥锁、读/写信号量封装在一起

```
class SynchConsole {
  public:
    SynchConsole(char *readFile, char *writeFile);
    ~SynchConsole();
    void PutChar(char ch);
    char GetChar();
    void WriteDone();
    void CheckCharAvail();

  private:
    Console *console;
```

```
    Lock *lock;
    Semaphore *readSemaphore;
    Semaphore *writeSemaphore;
};
```

因为c++指针不能指向成员函数，于是利用两个static函数帮助调用成员函数（两个中断处理函数）。

```
static void ConsoleReadAvail(int arg){
    SynchConsole *console = (SynchConsole *)arg;
    console->CheckCharAvail();
}

static void ConsoleWriteDone(int arg){
    SynchConsole *console = (SynchConsole *)arg;
    console->WriteDone();
}


SynchConsole::SynchConsole(char *readFile, char *writeFile){
    readSemaphore = new Semaphore("readSemaphore", 0);
    writeSemaphore = new Semaphore("writeSemaphore", 0);
    lock = new Lock("synch console lock");
    console = new Console(readFile, writeFile, ConsoleReadAvail,
ConsoleWriteDone, (int)this);
}
```

在控制台读写的过程写法与synchdisk一致。

```
void SynchConsole::PutChar(char ch){
    lock->Acquire();
    console->PutChar(ch);
    writeSemaphore->P();
    lock->Release();
}

//注意这里调用信号量的时间在Getchar之前
char SynchConsole::GetChar(){
    lock->Acquire();
    readSemaphore->P();
    char ch = console->GetChar();
    lock->Release();
    return ch;
}
```

测试：将输入到控制台的打印出来

```
static SynchConsole *synchConsole;
void SynchConsoleTest(char *in, char *out){
    printf("Enter SynchConsoleTest\n");
    char ch;
    synchConsole = new SynchConsole(in, out);
    for(;;){
        ch = synchConsole->GetChar();
        synchConsole->PutChar(ch);
        if(ch == 'q')   return;
    }
}
```

验证成功

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -sc
Enter SynchConsoleTest
12345
12345
yzhshiki
yzhshiki
^C
Cleaning up...
```

Exercise 7 实现文件系统的同步互斥访问机制

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

每个openfile都有自己的seekPosition，不需要修改。

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

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

修改fileheader

```
class FileHeader {
  ...
  public:
    time_t createTime;  // Create time of the file
    time_t lastAccessTime;  // Last access time of the file
    time_t lastWriteTime; // Last write time of the file
    int readerCount;  //读者数量，即正在openfile的read函数数量
    int userCount;    //用户数量，即openfile数
    Lock *rwlock;   //读写锁
    Lock *rclock;   //读者数量锁
    void AddSector(BitMap *freeMap, OpenFile* freeMapFile, int FirstHdrSec);

};
```

修改NumDirect

```
#define NumDirect   ((SectorSize - 5 * sizeof(int) - 3 * sizeof(time_t) - 2 *
sizeof(Lock*)) / sizeof(int))
```

在给文件头分配扇区时初始化

```
bool
FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    readerCount = 0;
    userCount = 0;
    rclock = new Lock("readCount Lock");
    rwlock = new Lock("reader-writer Lock");
    ...
}
```

如果只在Allocate初始化，则会导致之后从Disk读文件时，锁指针指向的内存并没有锁，因为是重新./nachos。所以在FetchFrom中加入判断：如果此文件没有线程已经打开，则初始化锁

```
void
FileHeader::FetchFrom(int sector)
{
    synchDisk->ReadSector(sector, (char *)this);
    if(userCount == 0){
        rwlock = new Lock("read-write lock");
        rclock = new Lock("reader count lock");
    }
}
```

对openfile的构造函数、析构函数、read、write进行修改，在操作前后加锁/释放锁

```
OpenFile::OpenFile(int sector)
```

```
{
    hdr = new FileHeader;
    hdrSector = sector;
    hdr->FetchFrom(hdrSector);
    hdr->rwlock->Acquire();
    if(hdrSector > 1)   //必须要加这个判断，因为bitmap和directory在0、1扇区。而文件系统没有delete它们，不会有--所以不能++
        hdr->userCount ++;  //打开这一文件的用户加一
    hdr->WriteBack(hdrSector);
    hdr->rwlock->Release();
    seekPosition = 0;
}

OpenFile::~OpenFile()
{
    if(hdrSector> 1){   //再保护一下bitmap和directory
        hdr->rwlock->Acquire();
        hdr->FetchFrom(hdrSector);

        hdr->userCount --;
        hdr->WriteBack(hdrSector);
        hdr->rwlock->Release();
        delete hdr;
    }
}

int OpenFile::Read(char *into, int numBytes)
{
    hdr->rclock->Acquire();
    hdr->FetchFrom(hdrSector);
    hdr->readerCount ++;    //读者加一
    hdr->WriteBack(hdrSector);
    if(hdr->readerCount == 1)   //第一个读者获得读写锁
        hdr->rwlock->Acquire();
    hdr->rclock->Release();

    ...

    hdr->rclock->Acquire();
    hdr->FetchFrom(hdrSector);
    hdr->readerCount --;
    hdr->WriteBack(hdrSector);
    if(hdr->readerCount == 0)
        hdr->rwlock->Release(); //最后一个读者释放读写锁
    hdr->rclock->Release();

    return result;
}
```

```
int OpenFile::Write(char *into, int numBytes)
{
    hdr->rwlock->Acquire();

    ...

    hdr->rwlock->Release();
    return result;
}
```

在删除文件出增加判断：

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

    if (i == -1)
      return FALSE;     // name not in directory
    FileHeader *hdr = new FileHeader;
    hdr->FetchFrom(table[i].sector);
    if(hdr->userCount > 0)      //如果该文件还有用户打开着
        return FALSE;
    table[i].inUse = FALSE;
    return TRUE;
}
```

为了保证每一个文件的userCount的正确性，每一次new OpenFile后（除了bitmap和directory），在用完后要delete。为此修改了Directory::List()、FileSystem::Create、FileSystem::findDirectory，因为之前都没注意delete openfile的问题。

值得注意的是，openfile的构造函数和write、read都需要rwlock，那么在测试的时候，如果在write过程中yield来让另一个线程read是不可行的，不论是新线程要打开文件还是读文件都会被卡在rwlock上。

Challenge 2 实现pipe机制

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

使用文件模拟pipe机制，即用一个文件作为缓冲区实现进程间通信。在文件系统初始化时规定第2扇区为管道文件的文件头，再分配制指定的大小给管道文件。需要向管道写或从管道读数据时指明数据源/目的和字节数即可。

```
/* filesys.cc  FileSystem::FileSystem */
  FileHeader *pipeHdr = new FileHeader;
```

```
  freeMap->Mark(PipeSector);
  ASSERT(pipeHdr->Allocate(freeMap, PipeFileSize));
  pipeHdr->WriteBack(PipeSector);

int FileSystem::writePipe(char *data, int numbytes){
    OpenFile *pipeFile = new OpenFile(PipeSector);
    int size = pipeFile->Write(data, numbytes);
    delete pipeFile;
    return size;
}

int FileSystem::readPipe(char* data,int numBytes){
    OpenFile * pipeFile = new OpenFile(PipeSector);
    int size = pipeFile->Read(data,numBytes);
    delete pipeFile;
    return size;
}
```

编写测试函数，创建两个线程分别写入和读出管道内容：

```
void readPipeTest(int which){
    int len = 100, readlen = 0;
    char *data = new char[len];
    readlen = fileSystem->readPipe(data, len);
    if(readlen == len)
        printf("%s Read bytes : %s from pipe\n", currentThread->getName(),
data);
    else
        printf("read failed\n");
    return;
}

void writePipeTest(int which){
    int len = 0, writelen = 0;
    char *data = new char[100];
    len = 100;
    printf("input:");
    scanf("%s", data);
    writelen = fileSystem->writePipe(data, len);
    if(writelen == len)
        printf("%s write bytes : %s to pipe\n", currentThread->getName(),
data);
    else
        printf("write failed\n");
    return;
}

void PipeTest(){
```

```
    Thread *thread1 = new Thread("Thread 1");
    thread1->Fork(writePipeTest, 1);
    Thread *thread2 = new Thread("Thread 2");
    thread2->Fork(readPipeTest, 2);
}
```

测试：可见模拟的Pipe机制可以正确实现进程间通信。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos  -tp
no -f create freemapfile
input:123456789101112
Thread 1 write bytes : 123456789101112 to pipe
Thread 2 Read bytes : 123456789101112 from pipe
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

内容三 遇到的困难及解决方法

- #define FreeMapFileSize　(NumSectors / BitsInByte) 为什么要除以每个字节的比特数
- 在filesys文件夹下的makefile去掉-DTHREADS为什么没有效果（在main.cc中还是执行了线程那部分读取参数
- #define ConsoleTime　100 为什么把100增大到400以上控制台测试就会失败

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

      disk->RequestDone();
  }

  //调用时：
  disk = new Disk(name, DiskRequestDone, (int) this);
  ```

  这是什么初始化手段，传入的arg如何代表一个类？就是把this指针转为int
- 把文件读写锁的指针放在fileheader里作为变量，然而这样在不-f的情况下，从disk读到指针位置但内存中并没有那个文件读写锁（只有-f后 在fileheader的allocated函数中new了。
- 一定要及时释放空间！比如delete openfile。Exercise7中，因为FileSystem类没有析构函数，其中的bitmap和directory的openfile也就没有被delete，没有调用openfile的析构函数，影响了我对文件userCount的计算

内容四 收获及感想

熟悉了文件系统以及一些c++特性。

内容五 对课程的意见或建议

无。

参考文献