

# 操作系统实习lab5文件系统

## Nachos文件系统的工作原理

在Nachos系统中，文件系统主要由如下几种结构组成

1. 磁盘被分为磁盘块，每个磁盘块按照地址开始依次编号，每个磁盘块必须整块使用，不可分割
2. 每个文件分成文件头和数据部分，文件头存放了文件的管理相关数据，在nachos中，为 `int numBytes; int numSectors; int dataSectors[NumIndex];` 三者分别代表文件大小，所包含的磁盘块数量，以及下面的数组索引到对应的包含数据的磁盘块。需要注意的是，文件头本身也分配为一个磁盘块的大小，因此索引数组的大小是固定的
3. 目录文件也是一种文件，在磁盘上和普通文件没有本质的不同，也分为文件头和数据块唯一的不同点是目录文件作为一个格式化文件，整个文件是多个目录项组成的，起文件内容是高度格式化的，因此为了便于目录的便利，操作系统会给目录文件单独设计结构体，以便目录文件在加载进入内存时能进行有效的格式化，而不是像普通文件一样是一个长的字符串。在nachos中。目录文件由 `class Directory` 类标识，其内部包含了一个定长的 `DirectoryEntry table[]` 数组，代表每个目录文件的最大大小是严格限制的
4. 文件在加载进入内存之后，需要一种适用于内存和代码访问的结构体形式的结构，在nachos中，这个结构体是 `openFile`，结构体中保存了文件头的信息，这样就可以根据文件头找到对应的文件数据块。同时对磁盘的读写操作进行封装，将对文件相对位置的读写变成对磁盘块的读写。

## 修改文件系统

需要达到的目的：

1. 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。
2. 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。
3. 实现多级目录
4. 动态调整文件长度，对文件的创建操作和写入操作进行适当修改，以使其符合实习要求。

需要作出的修改

1. 在nachos中，每个文件头刚好是一个磁盘块的大小，其内容较为精炼，因此这里增加文件属性不在文件头 `FileHeader` 中增加，以免减少文件索引数组的长度，因此在目录项中 `DirectoryEntry` 增加文件相关属性信息，因为目录项是目录文件的数据部分，因此可以保存较长的数据，在目录项中增加如下属性

```
bool isDirectory;    // 是否是目录
bool nameOnDisk;     // 名称是否在磁盘块
int nameDiskSector;  // 名称在磁盘块的位置
long createDate;     // 标准时间， 创建时间
long lastChangeDate; // 标准时间， 上次修改
int sector;          // Location on disk to find the
char name[FileNameMaxLen + 1] = {};
```

上面的属性 `nameOnDisk` 变量标识对于长文件名，可以单独分配一个磁盘块用于存储其文件名，也就是文件名长度最多可以达到128字节，如果想要扩展更大的文件名，还可以直接创建一个文件存储文件名。但是128字节文件名已经比较长，因此只扩展依次。同时增加了文件的创建，修改时间，设置为1900-1-1至今的时间差，便于转换为各种格式显示

2. 文件的数据块索引在文件头 `FileHeader::dataSector[]` 中保存，因为文件头是一个格式统一的数据结构，因此其结构体大小不可变，为了增加可以保存的文件长度，增加为二级索引，将最后一个磁盘块设置为二级索引磁盘块。为此，对于文件的磁盘块查找也需要进行变更，主要集中在函数 `FileHeader::ByteToSector()` 中，用于将文件相对位置转换为磁盘块号的函数，同时，文件长度增长也在这里发生。如果查找的文件相对位置已经超过了当前文件长度，则表明是写操作增加了文件长度（对于读写操作最初会检查文件长度，因此不会出现读操作越界的行为），需要申请磁盘块并增加索引，将其导出到磁盘保存。

```
//-----  
// FileHeader::ByteToSector  
// Return which disk sector is storing a particular byte within the file.  
// This is essentially a translation from a virtual address (the  
// offset in the file) to a physical address (the sector where the  
// data at the offset is stored).  
//  
// "offset" is the location within the file of the byte in question  
// 如果查找的offset相对应的sector还没有申请，则申请新的sector并返回申请结果，  
// 如果超过了一级索引的范围，则需要申请二级索引并进行对应磁盘块的返回  
// 传入虚拟文件系统的指针是为了通过文件系统申请新的磁盘块  
//-----  
  
int FileHeader::ByteToSector(int offset, FileSystem* filesys)  
{  
    int index = offset / SectorSize;  
    if(index >= numSectors){ // 长度变化，写操作会发生  
        numSectors ++;  
        int secNum = filesys->findEmptySector();  
        if(secNum == -1){  
            ASSERT(FALSE);  
        }  
        if(index == NumDirectIndex){ //需要新建二级映射  
            dataSectors[index] = secNum;  
            secNum = filesys->findEmptySector();  
            if(secNum == -1)  
                ASSERT(FALSE);  
            int* buf = new int[SectorSize / sizeof(int)];  
            buf[0] = secNum;  
            synchDisk->writeSector(dataSectors[index], (char*)buf);  
            return secNum;  
        }else if(index > NumDirectIndex){ //已经存在二级映射  
            int* buf = new int[SectorSize / sizeof(int)];  
            synchDisk->readSector(dataSectors[NumDirectIndex], (char*)buf);  
            buf[index - NumDirectIndex] = secNum;  
            synchDisk->writeSector(dataSectors[NumDirectIndex], (char*)buf);  
            return secNum;  
        }else{ // 一级映射  
            dataSectors[index] = secNum;  
            return secNum;  
        }  
    }else{ // 长度不变化，不需要新分配磁盘块  
        if(index < NumDirectIndex){ // 一级映射  
            return dataSectors[index];  
        }else{ // 二级映射  
            int* buf = new int[SectorSize / sizeof(4)];  
            return buf[index - NumDirectIndex];  
        }  
    }  
}
```

```

    }

}

```

3. 实现多级目录较为简单，因为目录文件本质也是一个文件，需要作出改变的就是两点，第一是在目录项中标识该文件是否为目录文件，第二是需要调整文件系统的遍历算法，能够拆分字符串，对多级目录进行索引。因此在 `FileSystem.cc` 中增加如下方法，用于索引多级目录

```

/*
    分割文件名称， 获得其所属文件夹
*/
int FileSystem::findFatherDirectory(char *name, int pwdSec) {
    char *tmp = name;
    if (*tmp == '/') {
        tmp++;
        name++;
    }
    int len = 0;
    while (*tmp != '\0' && tmp != 0 && *tmp != '/') {
        len++;
        tmp++;
    }
    char *fileName = new char[len + 1];
    memcpy(fileName, name, len);
    fileName[len] = '\0';
    if (tmp == 0 || *tmp == '\0') { // 文件
        return pwdSec;
    } else { // 文件夹
        Directory *pwd = new Directory(pwdSec);
        int childSec = pwd->Find(fileName);
        if (childSec == -1)
            return -1;
        return findFatherDirectory(tmp, childSec);
    }
}

```

4. 由于动态调整文件长度的操作本质就是动态给文件分配磁盘块的操作，封装在 `FileHeader::ByteToSector()` 函数中。

## 文件系统同步

### 同步磁盘的实现原理

在nachos系统中使用锁封装了一个 `synchDisk`，使其读写操作达到同步效果，实现原理入戏：以 `SynchDisk::ReadSector(int sectorNumber, char* data)` 函数为例，该函数首先申请磁盘锁，然后执行 `disk->ReadRequest(sectorNumber, data);` 操作，该操作会立即反回，并在磁盘操作结束会调用初始化 `Disk` 类 `disk = new Disk(name, DiskRequestDone, (int) this);` 时传入的回调函数 `DiskRequsetsDone()`，然后反回后执行 `P()` 操作阻塞，直到回调函数执行 `V()` 操作，磁盘读取操作才完成并释放锁，以这样的封装达到对应的同步机制

### SyschConsole

根据上述的原理，实现SynchConsole如下，对于读操作个写操作分别有一个回调函数，在执行完真正的 `Console` 的读写操作之后，阻塞当前进程，直到回调函数执行 `V()` 操作,这里以写操作为例

```

static void

```

```

ConsolewriteDone(int arg)
{
    SynchConsole *synchConsole = (SynchConsole *)arg;

    synchConsole->writeDoneHandler();
}

SynchConsole::SynchConsole(char *readFile, char *writeFile)
{
    writeSemaphore = new Semaphore("synch write console", 0);
    readSemaphore = new Semaphore("synch read console", 0);
    writeLock = new Lock("synch console write lock");
    readLock = new Lock("synch console read lock");
    console = new Console(readFile, writeFile, ConsoleReadAvail,
        ConsolewriteDone, (int)this);
}
SynchConsole::void PutChar(char ch)
{
    writeLock->Acquire();           // only one disk I/O at a time
    console->PutChar(ch);
    writeSemaphore->P();           // wait for interrupt
    writeLock->Release();
}
void SynchConsole::writeDoneHandler()
{
    writeSemaphore->V();
}

```

## 文件的同步互斥访问需求

1. 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。
2. 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。
3. 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。

## 同步互斥的实现

1. 文件的真正数据是以文件头 `FileHeader` 来标识的，在linux系统中，是操作系统返回一个文件描述符 `int fd`，在nachos中，我实现为返回一个 `OpenFile`，其中保存着线程自己的文件读取位置，而文件系统中保存一个列表，其中包含了所有已经打开的文件头，如果线程申请打开的文件已经在列表中，则只新建一个 `OpenFile`，其中包含的文件头指针使用已经打开的文件头指针，以此来标识对于同一个文件的操作。

实现如下：

```

class OpenFileTable
{ // 用于管理所有的打开文件
public:
    int headSec;
    FileHeader *fileHdr;
    int openCount;
    bool toRemove;
    Directory* father;
}

```

```

ReadWriteLock *lock;
openFileTable()
{
    headSec = -1;
    fileHdr = 0;
    openCount = 0;
    toRemove = FALSE;
    father = 0;
    lock = new ReadWriteLock();
}
};
/* 线程请求打开一个文件，先查看是否已经打开，如果没有则加入打开文件表，如果有，则复用文件头指针，并增加引用计数*/
OpenFile *
FileSystem::Open(char *name) {

    Directory *directory = new Directory(
        findFatherDirectory(name, RootDirectorySector));
    OpenFile *openFile = 0;
    int sector;
    DEBUG('f', "Opening file %s\n", name);
    sector = directory->Find(getFileName(name));
    if (sector >= 0) { // 文件存在
        int index = openFileIndex(sector); //在文件打开表中的位置
        if (index == -1) { //第一次打开
            FileHeader *hdr = addFile2OpenTable(sector, directory);
            ASSERT(hdr != 0); // 最多打开1024文件
            openFile = new OpenFile(hdr);
        } else { // 文件已经被打开过
            fileTable[index].openCount++;
            openFile = new OpenFile(fileTable[index].fileHdr);
            delete directory;
        }
    }
    openFile->filesys = this;
    return openFile;
}

```

2. 文件的操作是原子化的，为此在文件打开表中针对每个打开的文件设置读写锁，也就是同步机制的lab中实现的读者写者问题，进行封装，同时文件系统提供同步之后的读写操作如下，由于对于同一个文件的读写会追踪到同一个FileHeader上，因此能够保证对于同一个文件的操作是同步的：

```

int FileSystem::fread(OpenFile *file, char *into, int numBytes) {
    int index = openFileIndex(file);
    if (index == -1)
        return -1;
    fileTable[index].lock->prepareRead();
    int count = file->Read(into, numBytes);
    fileTable[index].lock->doneRead();
    return count;
}
int FileSystem::fwrite(OpenFile *file, char *into, int numBytes) {
    int index = openFileIndex(file);
    if (index == -1)
        return -1;
    fileTable[index].lock->preparewrite();
    int count = file->Write(into, numBytes);
}

```

```

        fileTable[index].lock->donewrite();
        return count;
    }

```

3. 在文件打开表中设置 toRemove 标志位，在文件被删除时只设置该标志位，并在 Close() 函数中检查当前关闭的文件是否还有引用，如果没有，则执行删除操作

```

bool FileSystem::Remove(char *name) {
    Directory *directory;
    BitMap *freeMap;
    FileHeader *fileHdr;
    int sector;

    directory = new Directory(findFatherDirectory(name, RootDirectorySector));
    sector = directory->Find(name);
    if (sector == -1) {
        delete directory;
        return FALSE; // file not found
    }
    int index = openFileIndex(sector);
    if (index != -1) { // 文件正在打开
        fileTable[index].toRemove = TRUE;
        return TRUE;
    }
    return deleteFile(sector, fileTable[index].father);
}

```

## 实现pipe机制

在linux中，pipe管道符设计为一种特殊的文件，其读写操作和常规文件的读写操作相同。在本实验中，为了不再OpenFile中增加太多的判定逻辑，新建一种文件 PipeFile，提供读写操作，内部维护一个缓冲区和读写位置，不会真正的导入磁盘，对于读取过的数据会进行删除。

```

class PipeFile{
private:
    char* buffer;
    int readPossion;
    int writePossion;
    int bufLen;
public:
    PipeFile();
    int write(char* buf, int numBytes);
    int read(char* buf, int numBytes);
};

int PipeFile::read(char* buf, int numBytes){
    int count = 0;
    if(numBytes + readPossion > bufLen){
        count = bufLen - readPossion;
    }else{
        count = numBytes;
    }
    strncpy(buf, buffer + readPossion, count);
    readPossion += count;
    bufLen -= count;
}

```

```

char* newBuf = new char[4096];
strncpy(newBuf, buffer + readPosion, bufLen);
delete buffer;
buffer = newBuf;

return count;
}

```

## 测试结果

### 1. 文件系统基本操作

1. 测试创建，多级目录等操作执行如下代码

```

filesystem->Create("/home", 0, FALSE);
filesystem->Create("/tmp", 0, FALSE);
filesystem->Create("/home/li", 0, FALSE);
Copy("/home/lihaiyang/test", "/home/li/test");

```

执行完毕后文件 DISK 内容如下:

```

Windows PowerShell
lihaiyang@DESKTOP-P5HBSU9: /mnt/c/Users/lihaiyang/Desktop/NachosLab/nachos/nachos_dianti/nachos-3.4

00000180 00 00 00 00 02 00 00 00 01 00 df b7 ff ff ff ff |.....|
00000190 1e a7 e8 5d 00 00 00 00 04 00 00 00 68 6f 6d 65 |...].home|
000001a0 00 00 00 00 00 00 00 00 01 00 00 00 ff ff ff ff |.....|
000001b0 1e a7 e8 5d 00 00 00 00 06 00 00 00 74 6d 70 00 |...].tmp.|
000001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000200 00 00 00 00 24 00 00 00 01 00 00 00 05 00 00 00 |...$.|
00000210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000280 00 00 00 00 01 00 00 00 01 00 df b7 ff ff ff ff |.....|
00000290 1e a7 e8 5d 00 00 00 00 08 00 00 00 6c 69 00 00 |...].li..|
000002a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000300 00 00 00 00 04 00 00 00 01 00 00 00 07 00 00 00 |.....|
00000310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000400 00 00 00 00 24 00 00 00 01 00 00 00 09 00 00 00 |...$.|
00000410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000480 00 00 00 00 01 00 00 00 00 00 df b7 ff ff ff ff |.....|
00000490 1e a7 e8 5d 00 00 00 00 0a 00 00 00 74 65 73 74 |...].test|
000004a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000500 00 00 00 00 04 00 00 00 01 00 00 00 0b 00 00 00 |.....|
00000510 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000580 00 00 00 00 31 32 33 34 35 36 37 38 39 30 31 32 |...123456789012|
00000590 33 34 35 36 37 38 39 30 0a 00 00 00 74 65 73 74 |34567890....test|

```

可以看到文件和目录已经创建成功，文件内容也成功拷贝

### 2. 测试读写同步

```

void testSynchRead() {
    OpenFile* file = filesystem->Open("/home/li/test");
    for (int i = 0; i < 5; i++) {
        char buf[10] = { };
        filesystem->fread(file, buf, 1);
        printf("thread %s read 1 char from file , get char : %s\n",
            currentThread->getName(), buf);
        currentThread->Yield();
    }
}

void testSynchWrite() {
    OpenFile* file = filesystem->Open("/home/li/test");
    for (int i = 0; i < 5; i++) {

```



```

char* buf = "abcdefghijklmn";
fileSystem->fwrite(file, buf, 1);
printf("thread %s write 1 char from file , get char : %c\n",
        currentThread->getName(), buf[i]);
currentThread->Yield();
    }
}

```

执行以下测试代码

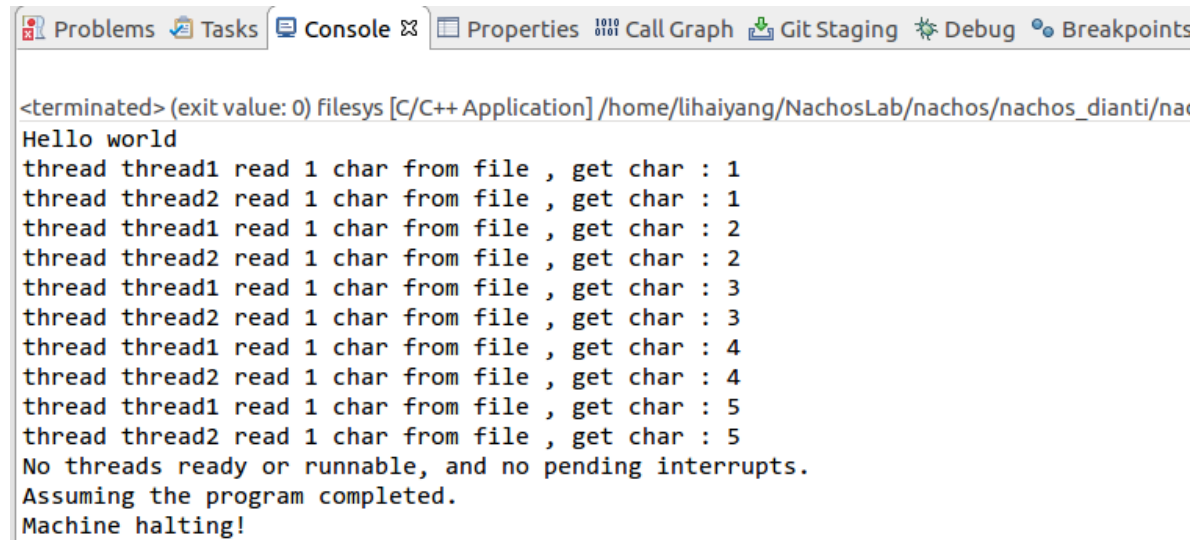
```

fileSystem->Create("/home", 0, FALSE);
fileSystem->Create("/tmp", 0, FALSE);
fileSystem->Create("/home/li", 0, FALSE);
Copy("/home/lihaiyang/test", "/home/li/test");

Thread* t1 = new Thread("thread1");
t1->Fork(testSynchRead, 0);
Thread* t2 = new Thread("thread2");
t2->Fork(testSynchRead, 0);

```

获得以下输出



The screenshot shows a debugger interface with a console window. The console output is as follows:

```

<terminated> (exit value: 0) filesystem [C/C++ Application] /home/lihaiyang/NachosLab/nachos/nachos_dianti/nachos
Hello world
thread thread1 read 1 char from file , get char : 1
thread thread2 read 1 char from file , get char : 1
thread thread1 read 1 char from file , get char : 2
thread thread2 read 1 char from file , get char : 2
thread thread1 read 1 char from file , get char : 3
thread thread2 read 1 char from file , get char : 3
thread thread1 read 1 char from file , get char : 4
thread thread2 read 1 char from file , get char : 4
thread thread1 read 1 char from file , get char : 5
thread thread2 read 1 char from file , get char : 5
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

```

执行以下测试

```

Thread* t1 = new Thread("thread1");
t1->Fork(testSynchRead, 0);
Thread* t3 = new Thread("thread3");
t3->Fork(testSynchWrite, 0);

```

获得如下输出



Hello world

```
thread thread1 read 1 char from file , get char : 1
thread thread1 read 1 char from file , get char : 2
thread thrad3 write 1 char from file , get char : a
thread thread1 read 1 char from file , get char : 3
thread thrad3 write 1 char from file , get char : b
thread thread1 read 1 char from file , get char : 4
thread thrad3 write 1 char from file , get char : c
thread thread1 read 1 char from file , get char : 5
thread thrad3 write 1 char from file , get char : d
thread thrad3 write 1 char from file , get char : e
```

```
00000410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
* Thread* write = new Thread("write");
00000480 00 00 00 00 01 00 00 00 00 00 de b7 ff ff ff ff |.....|
00000490 58 d7 e9 5d 00 00 00 00 0a 00 00 00 74 65 73 74 |X..].Breakpool.test| Men
000004a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000500 00 00 00 00 04 00 00 00 01 00 00 00 0b 00 00 00 |.....|
00000510 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
* read1 read 1 char from file , get char : 1
00000580 00 00 00 00 61 61 61 61 61 36 37 38 39 30 31 32 |....aaaaa6789012|
00000590 33 34 35 36 37 38 39 30 0a 00 00 00 00 00 00 00 |34567890.....|
000005a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
* read1 read 1 char from file , get char : 3
```