|  |
| --- |
|  |
| 文件系统实习报告 |
|  |

|  |
| --- |
| **姓名 方明 学号 S1748007**  **日期 4/24/2018** |

目录

[内容一：总体概述 3](#_Toc380430085)

[内容二：任务完成情况 3](#_Toc380430086)

[任务完成列表（Y/N） 3](#_Toc380430087)

[具体Exercise的完成情况 3](#_Toc380430088)

[内容三：遇到的困难以及解决方法 7](#_Toc380430089)

[内容四：收获及感想 7](#_Toc380430090)

[内容五：对课程的意见和建议 8](#_Toc380430091)

[内容六：参考文献 8](#_Toc380430092)

## 内容一：总体概述

本实习希望阅读Nachos文件系统的相关代码，来了解文件系统的基本实现， 并通过修改Nachos系统平台的底层源代码，来扩展w文件系统的相关功能。从而加深对文件系统的理解。

## 内容二：任务完成情况

### 任务完成列表（Y/N）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Exercise1 | Exercise2 | Exercise3 | Exercise4 | Exercise5 | Exercise 6 |
| 完成情况 | Y | Y | Y | Y | Y |  |

### 具体Exercise的完成情况

**Exercise1**

filesys.h：Nachos文件系统的数据结构。 文件系统是一组存放在磁盘上的按照目录组织的文件。 文件的操作有，创建，打开，删除。 对于一个已打开的文件可以读取，写入，关闭，这部分操作在openfile.h中。

 Filesys.cc: 为文件操作的具体实现。在系统中每一个文件包括一个文件头，它存放在磁盘的一个扇区（sector）中。 Nachos规定每个文件头的大小恰好是一个扇区。文件还包含一个记录该文件有多少个数据块的数字和文件目录系统的一个目录项。

Filehdr.h：管理文件头的数据结构。 文件头用来描述一个文件在磁盘哪里可以找到数据。通常会包括文件的长度等。

filehdr.cc：管理文件头的实现。文件头可以理解为Unix系统中i-node节点。一个文件头可以通过两种方式来初始化：对于一个新文件， 我们可以修改在内存中的数据结构，让它指向新分配的数据块；对于一个已经存在磁盘上的文件，我们只需要从磁盘上读取相关信息。

Directory.h：按照Unix的方式来管理文件和目录。一个目录其实是一组数据对：<file name: sector #>。 如果我们知道了文件名，我们就可以找到目录下该文件在磁盘的哪个地方。

Directory.cc：实现了文件的目录管理。一个目录由一组固定大小的目录项组成。每个目录项代表一个文件，包括文件名和文件头的磁盘位置。由于目录项的个数固定（Nachos为10），所以一个目录只有10个文件或目录。

Openfile.h 打开，关闭，读取，写入具体文件的数据结构。有两种实现， 一种是调用Unix系统的操作来完成；另一种是真实的实现，将这些操作转化为读取磁盘扇区的请求，对于这种实现，我们并不用担心多线程的并发访问。

Openfile.cc 管理已打开的Nachos文件的具体实现。在Nachos中，我们需要打开文件，然后才能进行读写的操作，一旦我们完成读写操作后，就要通过删除数据结构OpenFile来关闭文件。为了方便，打开文件后，我们会将文件头放在内存中。

Bitmap.h/Bitmap.cc 定义一个位图的数据结构。其实就是个位数组。它作为空闲磁盘块分配表，以文件的形式存放。每一个位标志了一个磁盘块的状态。

**Exercise2**

扩展文件信息，我在DirectoryEntry添加文件信息,来扩展文件类型，创建时间和修改时间等信息:

class DirectoryEntry {

public:

…

int type;

unsigned long create\_time;

unsigned long last\_modify\_time;

…

char name[FileNameMaxLen + 1];

};

文件类型是int type, 当为0是代表文件夹， 1代表文件， 2代表其它文件类型。

创建时间和上次修改时间的类型都为unsigned long类型。是一个绝对时间，以秒来计算。当前时间是从一个基准时间（1970-1-1 00:00:00 +0:00），到现在的秒数，用一个整数表示。我们可以方便的通过这个时间来计算出当前时间的用户可读的形式。

对于文件名字的长度，Nachos的设计是一个以空字符结尾的字符串，所以文件名的长度是FileNameMaxLen=9， 我们只需要将这个宏改大一点比如120，就可以将文件名的最大长度扩展到120个字符了。

对于Path，我们并没有，有一个字符串数组来存储一个路径。那样做有两个不好的地方：第一个就是不够灵活，一旦将该文件移动到别的地方，我们都需要更新路径，而且还有大小限制；第一个就是浪费内存。但是我的实现只有有一个指针指向父目录项，向上搜索知道父为NULL位置，这样这条搜索路径的文件名组合起来就是文件的路径了。代码大致如下：

class Directory {

public:

…

void GetPath(char \*path);

private:

Directory \* parent;

DirectoryEntry\* table;

…

}

class DirectoryEntry {

public:

…

Directory \*host;

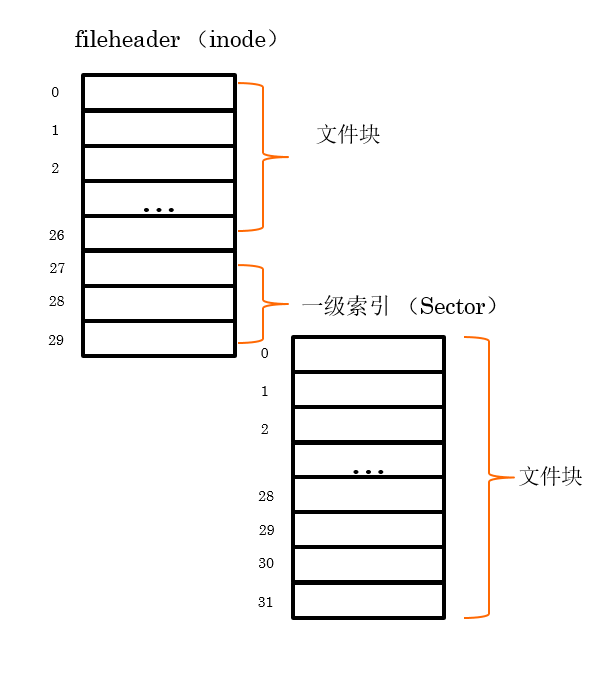
…

};

**Exercise3 扩展文件长度**

我们已经知道，Nachos文件大小为约4KB，真时的大小为 NumDirect \* SectorSize, 而SectorSize=128, NumDirect = (SectorSize-2\*sizeof(int))/sizeof(int) = 30, 3840B。 要想突破这个限制， 我们需要修改直接引用为间接引用。

由于NumDirect为30， 我们只要将其中的部分0-26作为直接引用，指向物理扇区，剩下的指向另一个二级索引. 由于一个SectorSize为128， int的大小4， 所以二级索引最多指向32个扇区。其示意图为：



这样设计后，我们的大小扩展到了26\*128 + 3 \* 32 \* 128 = 15616B,几乎是以前的4倍。

**Exercise4 实现多级目录**

我们需要文件进行分级管理,基本思路就是将class Directory扩展为一个树形结构。

class Directory {

public:

…

void GetPath(char \*path);

private:

Directory \* parent;

Directory \* child;

DirectoryEntry\* table;

…

}

首先需要扩展一下构造函数(在exercise2中我们已经为DirectoryEntry扩展了一个个Directory \*host)：

Directory::Directory(int size, Directory\* parent)

{

table = new DirectoryEntry[size];

tableSize = size;

for (int i = 0; i < tableSize; i++)

{

table[i].inUse = FALSE;

table[i].host = this;

}

parent = parent;

if(parent == NULL)

{

child = parent;

}

else

{

parent.child = this;

}

}

然后定义一个根目录 root = new Directory(10, NULL)

所有在调用Directory构造函数的地方，都要进行修改， 比如:

FileSystem::Create

FileSystem::Open

FileSystem::Remove

这个时候所有的文件将是一个路径：“/usr/tmp/test.txt”我们需要解析这些路径，将其放到root为根的树中。目前为止，我们可以将table作为children节点，这样就可以通过root完整遍历该树。

另外，我们并不需要每次执行操作都去动态的生成这些目录了， root为根节点的目录树将是一个全局的变量，哪些delete和new就应该删除了， 除非这个目录真的被用户显著的删除了。

**Exercise5 动态调整文件长度**

我们查看了Directory的代码，发现它的每个directory都是一个固定的数据，数组大小有宏NumDirEntries(10)来决定。我们可以根据exercise 4的基础上，做一些调整。 不在生成一个DirectoryEntry\* table, 每个Directory将只对应一个DirectoryEntry， 然后加入兄弟指针，则可以构成一个完整的树目录结构。需要做的调整为：

class Directory {

public:

…

void GetPath(char \*path);

private:

Directory \* parent;

Directory \* children;

Directory \* prev\_sibling;

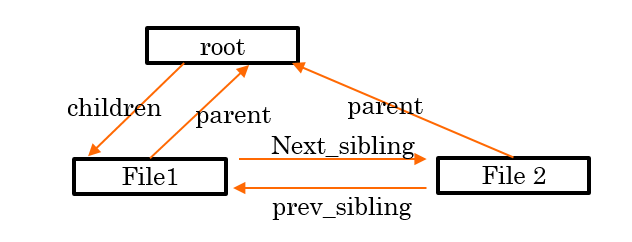
Directory \* next\_sibling;

DirectoryEntry inode;

…

}

不同层的目录通过parent, children来标识， parent顾名思义存的是该文件或目录的父， children如果指向的是该文件子目录的链表的首节点。 对于同一层目录下的文件，通过prev\_sibling和next\_sibling来标识关系。如果prev\_sibling为空，那么这个文件是该层目录下的首节点，如果next\_sibling为空，那么该节点是改层目录下的最后一个节点。关系示意图为：



**Exercise6 文件互斥源码阅读**

## 内容三：遇到的困难以及解决方法

暂无

## 内容四：收获及感想

我可能并没有真正的去把结果调试出来，但是我画了些时间去理解和实现这些问题，让我清楚了一些细节的概念。比如磁盘的一级索引，我一直以为是指向一个新的inode节点，但是在做实验的过程中，我发现其实还是指向的磁盘扇区块，然后该块存放的是指向新的磁盘扇区块的指针。Nachos一个扇区的大小是128(B)， 一个指针就是一个int(4B), 那么就可以指向128/4=32个扇区。

## 内容五：对课程的意见和建议

暂无

## 内容六：参考文献

1. <https://github.com/srijanshetty/nachos-syscalls>
2. <https://blog.csdn.net/maerdym/article/details/7470493>
3. <https://blog.csdn.net/small_snowflake/article/details/51407554>