编号

**南京航空航天大学**

**毕业设计**

|  |  |
| --- | --- |
| 题目 | Nachos文件系统扩展 |

|  |  |
| --- | --- |
| 学生姓名 | 董一峰 |
| 学号 | 040930223 |
| 学院 | 计算机科学与技术学院 |
| 专业 | 计算机科学与技术 |
| 班级 | 0409302 |
| 指导教师 | 刘宁钟教授 |

二〇一三年六月

**南京航空航天大学**

**本科毕业设计（论文）诚信承诺书**

本人郑重声明：所呈交的毕业设计（论文）（题目：）是本人在导师的指导下独立进行研究所取得的成果。尽本人所知，除了毕业设计（论文）中特别加以标注引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名：年月日

（学号）：

Nachos文件系统扩展

摘要

目前Nachos的文件系统中实现的是单级目录，目录中的文件个数有限。文件的索引结构采用了直接索引，使得文件的长度有限。文件大小在创建时指定，并不是根据文件内容动态分配，而且文件大小不可变。因此了解当前文件系统的缺陷，可以很好的帮助我们。

就论文中的文件系统而言，Nachos3.4自带的文件系统就功能而言却是比较弱，我们可以使用“一目录十文件”描述默认的文件系统。缺少多级目录、目录下文件入口数量过少、单个文件容量太小都是当前文件系统中很致命的缺陷；当然从多用户文件系统的层面而言，当前文件系统拥有一定的处理能力，但是对于一些临界资源的多用户的访问系统表现的多少有些不尽如人意；根据论文中的论述，在文件系统中某些类不适合拥有多个实例，因为多个实例的类会造成数据的不一致。

因此，我们需要做的就是对Nachos3.4的文件系统进行一个更加完善的升级，具体的升级项目包括上面所说的项目。对于以上的问题，我在论文中给出了相应的解决方案。比如：对于多个实例造成数据不一致的问题，我给出的解决方案是在类中引用单例设计模式，从类初始化的时候就限制多实例的生成，而不是使用没有任何约束的协议；对于多线程访问临界资源造成数据不一致的问题，我的处理方案是对临界资源使用PV操作，并且将暂时请求不到临界资源的进程放到一个等待队列中。当临界资源被释放的时候，在将等待队列中的线程激活；对于突破文件尺寸，我采用的方法是更新文件存储扇区索引的方法，在理论上，只要虚拟盘够大文件的尺寸就可以无限增长。

关键词**：**Nachos文件系统，多实例，数据不一致，分布式

The development of Nachos file system

Abstract

Nachos file system is a single-level directory, a limited number of files in the directory. The index of the file structure of a direct index, making the length of the file. File size specified at creation time, not dynamically allocated based on the contents of the file, and the file size is immutable. Therefore, understanding the defects of the current file system, you can help us.

Paper file system, the file system Nachos 3.4 comes with function Que Shibi weak, we can use the 12 files "directory" describes the default file system. Lack of multi-level directory, the directory file entry number is too small, single file size is too small is the current file system fatal flaw; course, from the level of the multi-user file system, the file system has a certain amount of processing power, but the performance of multi-user access to the system for some critical resources somewhat unsatisfactory; discussed in the paper, some classes in the file system is not suitable to have multiple instances, multiple instances of the class will result in data inconsistencies .

Therefore, we need to do a more complete upgrade the Nachos3.4 file system specific upgrade projects, including the above-mentioned project.

For the above problems, I have given in the paper corresponding solutions. For example: For multiple instances result in data inconsistency problems, I give the solution is referenced in the class singleton design pattern from the class initialization to limit the generation of multi-instance, instead of using the protocol without any constraints; multi-threaded access to critical resources causing data inconsistency problem, my treatment program is the use of critical resources PV operation, and will temporarily not request the process of critical resources into a waiting queue. When critical resources are released, the thread will wait queue activation; the breakthrough file size, the method I used is the update file storage sector index method can, in theory, as long as the size of the virtual disk is large enough file unlimited growth.

Key Words**：**the Nachos file systems, multi-instance, data inconsistencies, distributed

# 第一章引言

## Nachos的简介：

Nachos是一个教学用的模拟操作系统。Nachos使用了纯软件的方法模拟了整个操作系统。包括相关的虚拟硬件，进程管理系统，文件管理系统，网络管理器等一系列的模块。虽然Nachos已经成功的实现了一个“操作系统”，但是从管理方面而言，Nachos的各个模块还不是很完善，大部分模块只实现了很简单的功能。因此对于一个真正意义上的操作系统而言，Nachos的很多模块都需要程序员进行必要的完善，当然啊这也是初级程序员了解操作系统工作原理的一个很好的机会。

在Nachos中，系统使用了通用虚拟机，模拟相关的硬件部分，诸如：网络模块，中断系统，相关的指令等等。Nachos的大部分源代码采用了C++代码书写，模块之间具有很强的逻辑性，方便代码的重用和模块之间的解耦。

## Nachos文件系统简介

### Nachos文件系统

目前Nachos的文件系统中实现的是单级目录，目录中的文件个数有限。文件的索引结构采用了直接索引，使得文件的长度有限。文件大小在创建时指定，并不是根据文件内容动态分配，而且文件大小不可变。

在Nachos文件系统中，许多数据结构既可存放在宿主机内存里，又可存放在磁盘上。为了一致起见，文件系统中Synchdisk以上的类都有一个FetchFrom成员方法，它把数据结构从磁盘读到内存；还有一个WriteBack成员方法，与FetchFrom相反，它把数据结构从内存写回磁盘。在内存中的数据结构与磁盘上的完全一致，这给管理带来了不少方便。

Nachos文件系统一共由6个部分组成，分别为bitmap，directory，filehdr，filesys，openfile，synchdisk。

|  |  |
| --- | --- |
| **模块名称** | **模块功能** |
| bitmap | 用来指示当前硬盘的扇区的分配情况，通常意义上，当置1的时候，说明当前扇区被分配出去了；当为0的时候，说明当前扇区还没有被分配出去。 |
| directory | 这是Nachos管理目录的类。定义了目录的一系列基本操作。 |
| filehdr | 这个类在类Linux操作系统中，主要充当了inode的作用，主要定义了文件、目录等的其实索引节点。方便程序员管理文件。 |
| filesys | 当前文件系统的对象，主要定了文件的一系列函数。如在文件系统中创建文件，或者删除当前系统的文件。 |
| openfile | 主要定义了对一个单独文件的一系列操作。 |
| synchdisk | 对虚拟盘进行写入和读取操作。这个类是线程安全型的。 |

表1.1Nachos文件系统中各个类的功能

当前六个类的层次示意图如下所示：



图1.1Nachos文件系统的类间关系

在这六个类中，类Synchdisk是软件和硬件之间打交道的类。它是虚拟盘和文件系统交换数据的接口，因此这是整个文件系统的基础类；而类filesys代表了当前的Nachos文件系统，因此在文件系统中需要一个类filesys的实例，并且需要增加线程安全机制；对于类bitmap来说，文件只要需要分配或者归还硬盘空间，就需要使用该类的实例，当然当前bitmap实例也只需要一个即可，并且由于是临街资源，因此需要为当前的实例增加线程安全方面的机制；filehdr是文件系统的索引节点，每一个文件或者是目录都会有唯一一个节点与之相对应；directory和openfile分别对应了目录和文件的基本读写操作，需要为两个类的基本操作增加线程安全机制，防止不同线程修改文件的时候，破坏文件中的内容。

先从Nachos虚拟盘进行分析，Nachos虚拟盘共有32\*32个Sector，每个Sector的大小为128B，因此虚拟盘的大小为32^2\*128B=128KB。每个文件的大小必须是Sector的整数倍，即128B\*n(n=1,2,3,4…)，并且在默认情况下文件最多有30个索引。

在Nachos文件系统中，有这样的一种结构bitmap，这个结构主要是告诉程序员硬盘的Sector的分配情况。bitmap结构体单独占用一个Sector，而每个字节可以表示8个Sector的分配情况，因此bitmap可以指示的Sector的数目为128\*8=2^7\*2^3=1024，正好是虚拟盘上Sector的总数量。在显示方面，当某一个比特位是0的时候，证明当前的Sector处于待分配的状态；当为1的时候，证明当前的Sector处于以分配的状态。bitmap是一个临界资源，但是在Nachos默认情况下，bitmap是一个非线程安全的资源。因此需要添加线程安全的机制，并限制bitmap的实例个数。在为文件分配存储空间的时候，需要先查看bitmap中是否有足够的sector分配给程序，如果没有那么返回error；如果有的话将相关sector的bitmap为置1。

最后，对文件进行的所有修改都是在基于内存的基础上，因此在退出前需要将相关的修改写回虚拟盘（WriteBack）。

### Nachos文件系统的缺陷

1. 文件尺寸的限制

在Nachos文件系统中，在默认情况下，一个文件的大小完全由索引的个数决定而在Nachos中索引只能存储在一个Sector中，而一个Sector大约是128Byte。下面为文件系统类FileHeader的定义：

classFileHeader{

public:

.....

private:

intnumBytes;

intnumSectors;

};

由这个类型我们可以得到文件的大小：(128-4\*2)\*128b<4K，因此我们可以知道默认文件的大小小于4K。

1. 文件目录的单级化

当前nachos文件系统某人只有一个文件目录，并且在文件目录下只有10个文件保存点，也就是说当前文件的系统只能保存10个大小小于4K的文本文件。因此大量的文件虚拟盘空间得不到使用，对存储设备是一个很大的浪费。查看一下文件保存点定义：

class DirectoryEntry{

bool inUse;

int sector;

char name[FileNameMaxLen+1];

};

当前程序默认程序在主目录下的保存节点都是文本文件保存点，因此，这也是当前Nachos文件系统没有多级目录的最大限制因素。

1. 文件系统的多个实例造成文件系统的损坏

查看类FileSystem以后，我们会得到这样的一个结论，就是每次对FileSystem实例化以后，会产生一个当前文件系统状态的一个映射。因此如果存在多个FileSystem的实例，那么可能会对文件系统的数据完整性造成一定的破坏。所以，我们需要对FileSystem实例对象的初始化进行必要的限制。

# 第二章Nachos的安装和使用

* 1. Nachos的安装

安装Nachos需要以下的几样东西：

1. Nachos3.4源代码，源代码可以在[download.csdn.net/download/nudt\_bubble/4109063](http://download.csdn.net/download/nudt_bubble/4109063)获得。
2. g++编译器4.6.3。
3. gcc编译器4.6.3
4. 交叉编译器gnu-decstation-ultrix。

首先将Nachos源代码(nachos)放置在~/code/下，即Nachos源代码的位置是~/code/nachos,然后将交叉编译器gnu-decstation-ultrix放置在~/code/gnu-decstation-ultrix，并修改nachos的Makefile文件，将交叉编译器的路径修改为~/code/gnu-decstation-ultrix。

进入路径~/code/nachos/code下，输入make命令。编译时间大概为2-3分钟，编译完成后发现在目录machine、network、threads和filesys下分别出现了名为nachos的可执行文件。到此nachos的安装已经完成。我们可以分别进入不同的目录运行可执行文件nachos，以检测不同的程序模块。

* 1. Nachos源代码目录分析

Nachos源代码目录结构如下：



图2.1Nachos源代码的目录结构

1. bin目录：目录主要安装了文件格式解释器。Nachos系统中有属于自已的文件系统，因此需要文件格式解释；
2. Machine目录：主要定义了虚拟机的相关的属性，主要包含如下的类型：硬盘，终端系统，输入输出等等；
3. Makefile.common文件：存储了Make文件所需要一系列参数；
4. Network目录：主要定义了Nachos系统的网络模块；
5. Threads目录：主要定义了Nachos系统的线程模块，例如Thread属性和线程的一系列同步操作，整个系统的Main函数定义在该文件目录下；
6. vm目录：可以编译生成虚拟机；
7. Filesysten目录：主要定义了Nachos系统的文件系统模块；
8. Makefile文件：用来执行make命令的文件；
9. Makefile.dep文件：定义了编译器在编译的时候默认的参数；在模块Makefile文件中需要包含这些参数的。
10. Test目录：主要定了以Nachos文件系统的测试项目；
11. Usrprog目录：里面可以放置用户基于Nachos平台书写的代码；除此以外，里面定义了一个很重要的结构bitmap，这个结构在nachos文件系统中发挥着至关重要的作用。

并且每个模块是可以单独编译的，比如如果当前程序员只想调试文件模块，那么只需要进入目录Filesys执行make命令，单独执行这一个模块就可以了。

* 1. Nachos文件系统的使用

在Filesys目录下执行当前可执行文件，具体的执行的命令如下：./nachos。在对应的命令选项中有很多的选项，其中包含了创建文件，修改文件，追加文件内容等等一系列操作。每次只能执行一个命令，因此在命令执行效率方面显得比较低。

第三章 模块设计和升级

Nachos文件扩展主要是基于原有的代码进行扩展和升级。因此设计和升级部分主要分为以下的几个部分分别阐述：

* 1. 突破文件尺寸的限制

Nachos的文件是这样存储数据的，如下图所示：



图3.1Nachos默认文件存储形式

文件大小的限制原因可以查看1.2.2，里面有相关的解释。在这里论文主要论述相关的解决方案。在Nachos中类FileHdr的数据成员是这样定义的，如下面的代码所示：

class FileHeader {

public:

…..

private:

int numBytes ;

int numSector ;

int dataSectors[NumDirect] ;

} ;

其中NumDirect在代码中定义的如下：

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

Nachos默认每个文件使用dataSectors存储文件的Sector索引。因此我们需要在保留原来结构的基础上，增加新的存储空间用来保存索引。因此新的结构体定义如下（红色方框中的是新增加的部分）：

classFileHeader{

public:

…..

private:

int numBytes;

int numSector;

int nextSector;

int dataSectors[NumDirect] ;

}

新的数据结构使用了额外的空间去存储索引。即当索引数大于30的时候，程序将再申请一个Sector，并将对应的Sector号存储在nextSector中，当dataSector的存储量达到饱和的时候，我们将大于30的索引存储在Sector号为nextSector的Sector上，并且约定在新的Sector上有两个固定的数据域，如图Figure3中的红色框所示，第一个数据是文件在当前Sector上的索引个数；第二个数据是下一个Sector的号码，记做SectorNum，其默认值为-1，代表这是文件最后一个Sector了；其余的部分用来存储相关的索引。

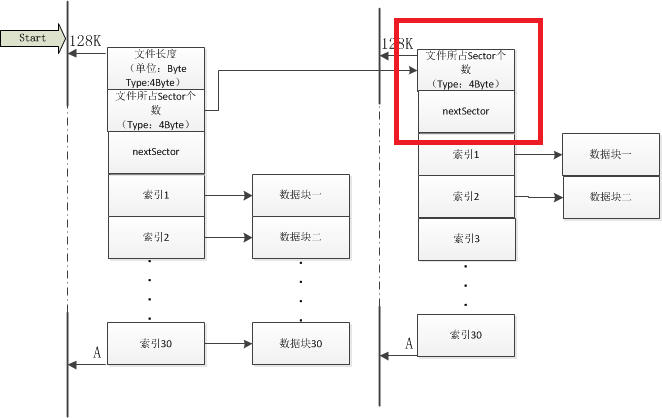


图3.2更新后的Nachos文件系统

此时NumDirect的定义需要进行修改了，具体的定义如下：

#define NumDirect 128-3\*sizeof(int)

而对于扇区号为nextSector的Sector而言，我们需要这样定义：

#define NextNumDirect 128-2\*sizeof(int)

以上就是突破文件大小限制的方法。

查看一下类FileHeader中的相关函数，与文件大小的函数有一下三个：

1. bool Allocate(BitMap\* bitmap,int fileSize)：为文件分配相关的虚拟盘空间；
2. void Deallocate(BitMap\* bitmap)：回收文件对应的虚拟盘空间，这个函数与上面的函数在功能上市相反的。
3. int ByteToSector (int offset ) ：将相应的偏移位置转化为对应的扇区号。

因此我们需要重写这3个函数。重写函数的形式如下：

bool Allocate(BitMap \*bitmap, int fileSize ) /\*为文件分配对应的空间\*/

{

计算当前的数据需要的扇区数目NumSector。

if ( NumSector < 30)

{

采用默认的方式进行数据的存储

}

else

{

前30个扇区采用默认的方法，后面的扇区被定义在nextSector上；如果存储空间被填满了那么在申请一个空间，知道可以将全部数据存储在磁盘上。

if(没有足够的磁盘空间){

那么放弃为文件分配相关的空间。

}

}

}

void Deallocate ( BitMap \*bitmap ) /\*回收文文件的空间\*/

{

获得当前扇区上的索引数目indexNum；

while(true)

{

for(int i=0; I < indexNum; i++)

{

释放对应的空间。

bitmap.Clear(SectorNum);

}

if（nextSector == -1）break;

else读取nextSectorNum上的数据

}  
}

int ByteToSector ( int offset ) ；通过偏移地址，找到对应的磁盘号

{

//计算当前的offset对应的是第几个Sector

offsetSectorNum = Math.Floor(offset/128)

while ( 找到对应的磁盘 ) {}

return返回对应的硬盘号。  
}

* 1. 突破Nachos的单目录结构

Nachos默认的文件结构如下所示：



图3.3Nachos文件系统默认的目录结构

简要的说一下Nachos文件的文件和目录结构。在默认情况下，整个系统只有一个目录，在这个目录下有10个进入点（DirectoryEntry）。根目录所在扇区号被定义在DirectorySector这个宏里，也就是说根目录所在扇区号是全局的，任何人都可以访问和查看。

在Nachos中，目录下的文件节点被定义在如下的结构体中：

class DirectoryEntry {

public:

bool inUse;

int sector;

char name[FileNameMaxLen+1] ;

} ;

由这个结构可见，在结构DirectoryEntry中，没有区分当前节点的属性，即没有告知程序员这个节点代表目录还是文件，由此可见我们需要为程序员指明节点的相关的属性。作为一种解决方案，改进方案中定义了如下的枚举类型：

typedef enum NodeType {

FILE\_TYPE=1,

DIR\_TYPE

} NodeType ;

并且在DirectoryEntry中增加了新的数据区域，展示如下：

class DirectoryEntry {

public:

bool inUse ;

int sector ;

NodeType fileType ;

char name[FileNameMaxLen+1] ;

} ;

通过改进结构体，我们可以区分文件节点的属性，具体的属性可以访问NodeTypefileType这个变量。这样我们的文件结构就可变为常见的模式，即在同一个层面可以同时存在文件和目录。当我们进入目录的时候，可以访问到目录里面的文件节点；与之相反的时候，当我们访问一个文件的节点的时候，那么我们将会获得文件的相关的内容。

因为有了多目录这样的结构，因此我们会面对这样的一个问题，我们当前在那个目录下？简单说就是当前的文件路径是什么。

我们可以这样看待问题，对于多线程文件系统而言，每个线程的当前的文件路径不尽相同，因此我们知道对于每个单独的线程而言，保存文件路径的变量应该拥有这样的性质，首先，当线程被销毁的时候，这个变量需要被回收；当线程还有被销毁的时候，我们必须保存这个变量，并拥有可以修改这个变量的权限。在这里，我们假设这个变量为char\*path;在默认情况下，path的值为”/”，代表当前路径在根目录。如果当前我进入了目录A，那么当期的path值为”/A”,在上面的基础上又进入了目录B，那么当前的path为”/A/B”。进入不同的目录，进行相同操作的结果是不同的。这点值得我们所有的人注意。

* 1. 防止Nachos实例多个FileSystem对象导致数据不一致

在用户访问文件系统的时候，不可避免的需要实例化FileSystem这个类型，因为这个类是文件系统的基础类。它负责和底层虚拟盘打交道，并且为上层部分提供一些必要的基本操作。但是我们也要注意到这样的事实，当系统存在多个FileSystem实例的时候，可能会导致数据不一致，从而造成数据损坏和丢失，下面我们讨论一下多个FileSystem实例给系统造成的破坏。示意图如下：

1. 多个实例写回的时机不同，导致数据的丢失。

如图2.4所示，当系统中存在FileSystem\_1和FileSystem\_2，两个都是向扇区号为10的扇区写入数据。当用户先实例化FileSystem\_1,并向扇区号为10的扇区写入数据1；而后又实例化FileSystem\_2，又向扇区号为10的扇区写入数据2。当FileSystem\_1后写入的时候，虚拟盘最后保留的数据为1；当FileSystem\_2后写入的时候了，虚拟盘最后保留的数据为2。这样会导致程序数据的丢失和不一致，因此多个实例的FileSystem会引起数据竞争。

1. 多个实例同一个的扇区追加数据，导致对应的数据被冲洗或更新失败。

如图2.4所示，当存在实例FileSystem\_1和FileSystem\_3的时候，我们会发现这样的事情：

1. 我们先考虑相同数据的情况，如果扇区号为10的扇区中原来的数据为1，那么FileSystem\_1将该扇区的数据更新为1，保持和原来一样；FileSystem\_3现在向扇区追加数据那个扇区中的数据为13。如果二者的写入顺序是FileSystem\_3先写入，而FileSystem\_1后写入，那么虚拟盘上的最终数据结果是1。这也就是说，程序的追加操作相当于没有执行；当然如果FileSystem\_1先写入，而FileSystem\_3后写入，那么结果是正确的。
2. 现在我们考虑不同的数据情况，如果当前扇区号为10的扇区中又来的数据为4，那么FileSystem\_1将该扇区的数据更新为1；而FileSystem\_3将扇区的数据更新为43。这样也会有两种情况，当FileSystem\_1先写入，FileSystem\_3后写入，那么最终的数据为43，数据错误；当FileSystem\_3先写入，FileSystem\_1后写入，最终的数据时1，同样数据也是错误的。



图3.4FileSystem多实例示意图

我们可以看到FileSystem多实例会对系统造成数据方面的破坏。在Nachos系统中，Nachos的作者在system.h文件中声明了这样的一个外部变量：

#ifndef FILESYS\_NEEDED

#include “ filesys.h ”

Extern FileSystem \* fileSystem;

#endif

这样只要可以包含system.h这个头文件，程序员就可以在程序中访问这个FileSystem类型的实例，也就是说程序员不需要自己实例化FileSystem这个类型。但是在当前的编程框架下，代码对是否可以对FileSystem进行多次实例没有严格的限制，更多的限制只是被放置在“口头”上。因此为了彻底解决这个问题，我们需要在代码中限制类型的实例个数，这样无论程序员是否访问那个全局变量，都可以防止多个实例共存的情况。我的解决方案如下：

class FileSystem {

public:

static inline FileSystem\* getInstance()

{

if ( fileSystem == NULL )

{

this->fileSystem = newFileSystem();

}

returnthis->fileSystem;

}

.......

private:

FileSystem\* fileSystem = NULL;

FileSystem();

.....

};

在这里采用了单例模式，由于当前FileSystem类的构造函数是私有成员，因此确保了不论程序在什么地方、在什么时候只要想使用FileSystem实例，那么只能采用如下的方式进行调用：

FileSystem \* current=FileSystem :: getInstance();

都只会在程序中产生一个对象实例，所有的引用都是对当前的对象的一个引用。

* 1. 多个bitmap实例引起数据不一致

在这里可以再一次解释这个问题。当程序员在自己的代码中，实例化了多个类型为BitMap的变量，并且一次在上面做了不同的修改，在依次写入到Nachos文件的系统的硬盘中。

* 1. 在目录下搜索相应的文件或是目录

一般意义而言，我们将若干目录的结构看做是一颗长“乱”的树。在搜索开始的时候，我们将当前的路径（有关当前路径的定义参考2.2）指向的节点看做是搜索树的“根节点”。在我们的解决档案中，采用DFS的方式查找相应方法的名字。当我们需要查的名字为filename，那么程序依次访问当前搜索路径下的DirectoryEntry节点，当遇到目录的时候，检测当前目录是否是我们要找的，然后进入目录；否则检测文件是否是我们需要查找的。依次迭代，直到当前没有更新的路径并且所有节点都检测完毕。

伪代码如下：

void search(char\* filename, char\* currentPath)

{

得到当前currentPath下的DirectoryEntry，并保存到FileArray[]下

for(int I = 0;I < length(FileArray); i++)

{

if(strncmp(FileArray[i],filename,strlen(filename))==0)

{

将当前的路径打印出来。

}

if ( FileArray[i]是目录 )

{

search(filename,currentPath+"\"+FileArray[i]);

}

}

}

* 1. 多线程引起的虚拟盘数据丢失或不一致

文件系统是一个为用户提供文件管理的工具，用户可以通过文件系统完成基本的操作，诸如：新建，删除，修改，重命名等一系列的操作。这一些列的操作在单线程下基本不会引发数据错误。但是这一系列操作在多线程下会引发很多的错误。举个例子来说，如果有如下的代码片段：

1. ....

2. Bitmap\* bitmap=Bitmap::getInstance();

3. int sectorNum=3;

4. if(bitmap->Test(sectorNum))

5. {

6. bitmap->Mark(sectorNum);

7. }

//开始在扇区号为sectorNum的扇区分配写入内容。

...

现在有两个线程分别为thread1和thread2，两段代码都在执行上面的代码。当thread1线程执行到5的时候，判定当前扇区号为sectorNum的扇区还没有分配出去，因此决定使用当前扇区，此时暂时停止thread1的执行，开始执行thread2，thread2在判断以后发现扇区号为sectorNum的扇区也没有分配出去，因此也使用了这个扇区。现在可以发现一个很大的问题，那就是有两个线程同时决定使用一个扇区，并决定向扇区写入自己的数据，因此这块扇区的数据是不安全的，上面的代码也是非线程安全的代码。因此我们需要修改上面的代码，将上面的代码片段变为一个原子操作，防止代码在执行的时候，发生不必要的中断从而导致数据发生安全问题。具体代码如下：

1. ....

2. Semaphore \* pv = new Semaphore();

3. pv->p();

4. Bitmap \* bitmap=Bitmap :: getInstance();

5. int sectorNum = 3;

6. if ( bitmap->Test(sectorNum) )

7. {

8. bitmap->Mark(sectorNum);

9. }

10. pv->v();

//开始在扇区号为sectorNum的扇区分配写入内容。

...

在代码两端加一个PV操作，并设临界资源数为1，那么当程序进入主要执行点的时候，就不会因为中断造成数据的不完整了。当然，为了完整的写完代码，我们还要设置一个多进程的等待队列，当发现临界资源数为0的时候，就需要将当前的进程加入等待队列，当资源被释放以后，重新唤起进程的等待队列。等待队列的定义可以在~/code/thread/synch.h中找到相关的定义。

第四章文件管理器的设计与实现

前一章主要论述了在升级Nachos文件系统时需要注意的几个部分。这一章主要是在论述如何在完成升级以后，将各个文件模块进行整合，完成一个文件管理系统。当前文件系统的模块图主要由以下几个部分构成：



图4.1文件管理器的模块

在类shell的文件管理器中，一般上将从命令的输入，再到分割，知道最后的解析分割上上面的两个部分。模块的调用顺序是一次从上到下的，数据传输是单向的并且也是从上到下。最后将数据打印在Linux终端上。

* 1. 文件管理器基本命令一览表

|  |  |  |
| --- | --- | --- |
| 命令 | 命令功能 | 命令类型 |
| cd | 改变当前用户的路径，  使用格式：cd filename | 内置命令 |
| mkdir | 在当前用户路径下，新建目录,  使用格式：mkdir dirname | 外置命令 |
| rmdir | 在当前用户路径下，删除用户目录  使用格式:rmdir dirname | 外置命令 |
| rmfile | 在当前用户路径下，删除用户文件  使用格式：rmfile filename | 外置命令 |
| exit | 退出当前文件管理器，返回终端默认状态  使用格式:exit | 内置命令 |
| touch | 新建空文件，默认大小为一个扇区的大小。  使用格式:touch filename | 外置命令 |
| search | 在当前文件路径以及所包含的目录下寻找同名的文件。  使用格式:search filename | 外置命令 |

表4.1文件管理器中的基本命令

命令类型的具体解释将在3.2中给出

* 1. 具体的模块的设计

### 命令接收模块

模块图如下所示：



图4.2文件管理程序具体的模块组成

命令接收模块说白了就是，获得用户输入的命令，并将用户输入的部分拆分基本的命令格式，即：命令(命令选项命令参数)\*。

在C语言中这个格式可以被以下的形式表示：

typedef struct

{

char cmd[CMD\_MAX\_LENGTH];

char params[MAX\_PARAM\_NUM][10];

char param\_options[MAX\_PARAM\_NUM][10];

}CodeFormat;

当然用户输入的时候是按照一个字符串进行输入的，这个字符串包含了命令、参数以及参数选项。因此我们需要过滤一些无用的字符，比如空格、Tab等键并将获得的部分分类，以此对用户的命令进行判断和解释。在这个部分最重要的部分就是区分命令的类型，在这里需要解释一下命令类型这个概念。

在命令解释器中一般意义而言，有两个进程。程序示意图如下：



图3.3文件管理系统运行原理

根据示意图，我们可以很清楚的看到在父线程除了要维护管理器以外，它还有一个很重要的功能就是用来解释用户命令中的内置命令。所谓内置命令，就是这个命令必需要在父线程中进行解释，并且命令所产生的效果是全局的，换句话说，内置命令产生的效果不仅会波及当前线程并且会波及子进程。在Linux体系下，采用fork函数的时候，会在当前的基础上产生一个新的进程，为了区分我们将基础进程成为父进程；新产生的进程称作子进程。在程序执行过程中，父进程会一直存在，直到程序结束；子进程则不同，执行结束后就被销毁。因此，如果想要保存全局运行数据，必须在父进程中执行。

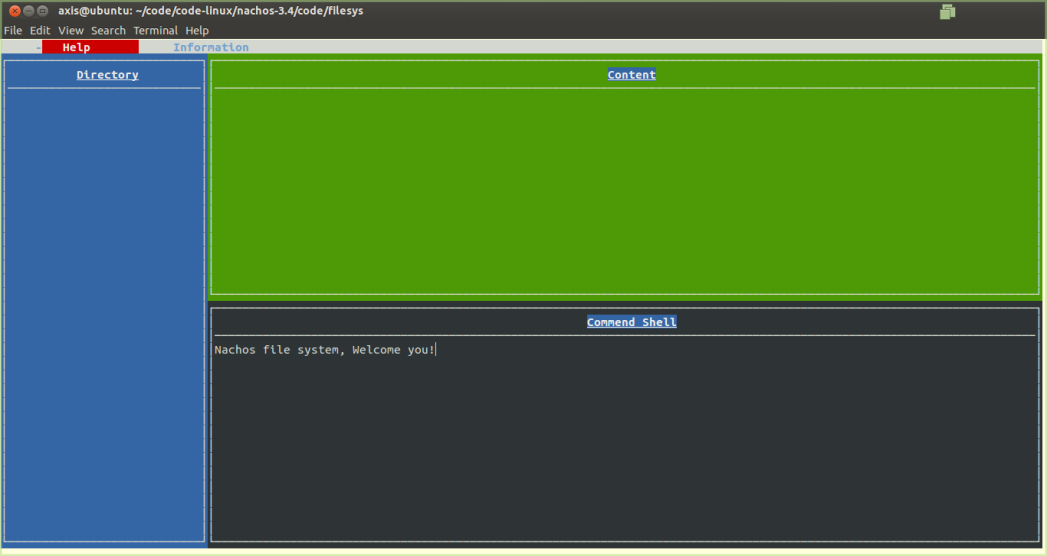
举一个例子，在3.1中给出了exit是一个内置命令，如果这个exit在子进程中解释，那么它会终止子进程，但是对父进程没有任何的影响。父进程仍然处于循环中，因此exit没有起到对应的作用。因此必须将其放在父进程中，这样才能起到终止作用。

### 命令调用模块

当前的这个模块，主要完成的工作是根据命令的不同将用户命令交予不同的处理程序予以实现。在编写程序方面，我们将每个命令对应一个.c文件，当程序编译的时候，每个.c文件对应生成一个可执行文件，而当前模块就采用linux中系统函数execl,调用不同的处理函数。这样基本就完成了一个Nachos文件管理程序。在这个模块可能要做好多进程对临界资源的互斥访问。在两个进程中，主要是子进程在解释外部命令，当然同内置命令相比之下，外部命令在数量上会比较多一些，因此子进程运行的机率会很大，我们必需要做好父进程和子进程之间的协同，防止僵尸进程的产生。

* 1. 文件管理器运行界面以及相关的操作步骤

### 文件管理器运行界面及相关的技术



当前文件系统的前端界面运行在Ubuntu12.0.4的终端上，界面基于的前端技术是Linux发行版Ubuntu 12.0.4自带的图形库NCurses。NCurses是一个基于C语言的图像库，图形库中自带panel、menu等一系列的组件，可以直接在绘制前端界面的时候进行使用，节约了开发者的大量时间。我们需要在Makfile.dep文件中加入如下的编译选项：LDFLAGS=-lpanel –lmenu –lncurses,确保当前库文件可以参与文件最后的编译工作。

### Nachos文件管理器的使用方式

界面主要分为四个部分了， 下面分别介绍每一个部分的作用：

1. 灰色的部分：

这是这个软件部分的菜单部分， 这个部分主要定了菜单的一些定义。诸如：帮助， 退出等等一系列操作；

1. 蓝色的分部：

主要展示当前路径下的内容。将当前路径下的文件和目录的名字显示出来， 当用户选取的是目录的时候， 更换当前的路径， 并重新显示路径下的信息；当用户选取的是文件的时候， 程序将在绿色区域显示当前文件的内容；

1. 绿色的部分：

这个区域主要是显示区域。当用户在蓝色区域选取某一文件的时候，该区域将显示该文件的内容。同时在光标在该区域的时候， 支持文件内容的编辑和修改。

1. 黑色的区域：

这个区域是一个命令解析区域。用户可以将这个区域当做类似于Linux shell之类的东西。用户可以在当前黑色区域输入相关的命令（具体的命令可以在4.1这个部分找到）， 程序经过解析以后可以将相关的结果数据显示在当前的区域， 以供使用者分析。

# 第五章总结与展望

### 对Nachos文件升级的总结

Nachos作为一个教学系统而言，对于我们这样的操作系统初学者而言有很大的帮助。Nachos的源代码模拟了整个计算机系统，包含了一整套的指令，以及硬盘、内存等一些列的。当然Nachos也有它自己的缺陷和不足，很多模块功能比较简单，在真实环境中没有办法使用。当然正是因为这些缺陷，给程序员了很大的发展的空间，程序员可以给予Nachos框架对其中的模块进行任意合理的升级和更新。

就论文中的文件系统而言，Nachos3.4自带的文件系统就功能而言却是比较弱，我们可以使用“一目录十文件”描述默认的文件系统。缺少多级目录、目录下文件入口数量过少、单个文件容量太小都是当前文件系统中很致命的缺陷；当然从多用户文件系统的层面而言，当前文件系统拥有一定的处理能力，但是对于一些临界资源的多用户的访问系统表现的多少有些不尽如人意；根据论文中的论述，在文件系统中某些类不适合拥有多个实例，因为多个实例的类会造成数据的不一致。

对于以上的问题，我在论文中给出了相应的解决方案。比如：对于多个实例造成数据不一致的问题，我给出的解决方案是在类中引用单例设计模式，从类初始化的时候就限制多实例的生成，而不是使用没有任何约束的协议；对于多线程访问临界资源造成数据不一致的问题，我的处理方案是对临界资源使用PV操作，并且将暂时请求不到临界资源的进程放到一个等待队列中。当临界资源被释放的时候，在将等待队列中的线程激活；对于突破文件尺寸，我采用的方法是更新文件存储扇区索引的方法，在理论上，只要虚拟盘够大文件的尺寸就可以无限增长。

在这次升级Nachos文件系统的项目中，给我最大的启示就是知道了操作系统的文件系统的结构、知道了如何有效的防止零碎存储的产生以及合理的使用硬盘空间。一般简单的来说，在硬盘上存储数据最简单的方式就是顺序存储，例如：我现在硬盘上存储一个大小为100M的文件，紧接着我又在其后存储一个文件的小为4k的文件，最后我又在其后面存储一个大小为1G的文件，现在我删除这个4k的文件，那么在存储空间上就产生了一个大小为4k的碎片。如果后面需要存储的文件的大小都大于4k，那么就意味着这个4k碎片始终处于没有被使用的状态。等时间一长，硬盘会出现成千上万的这样的碎片，可能碎片加在一起可能有数GB，但是这些空间没有办法被有效的利用。因此更加行之有效的方式被发明出来，那就是将整个存储空间划分为若干大小为128B（对Nachos文件系统而言）的段，每个文件都有若干的段组成，因此文件的大小都是段的整数倍，这样虽然可能在一个段浪费了部分空间，但是从整体而言，还是节约了大部分空间。除了这个知识点外，我还知道了类Linux是组织文件系统的方式，在类Linux体系下，有一个很重要的结构体iNode，每个文件或者目录都是一个iNode节点，可以说iNode节点是每个文件的起点，我们可以通过iNode节点访问相关的文件内容和相关的属性。可以说iNode是文件对外的一个数据接口。

以上就是我在Nachos文件系统这个项目中获得的感悟。

### 对文件系统的展望

当今我们处于一个大数据时代，因此当今的文件系统应该是分布式多用户文件系统的天下。现如今，人们每天的活动都会产生难以计数的用户数据，如何有效的保存和使用数据成为对文件系统的一个很大的考验。其实我们现在考虑如何制造更小但存储量更大的硬盘，意义不算是特别大，因为面对如此庞大的用户数据，单机的硬盘或者单个服务器的硬盘式无论如何都无法存储这些庞大的数据，因此分布式文件系统可以解决这个问题，我们可以这样理解分布式文件系统：系统将若干不同计算机上的硬盘用某种方式进行互联，对程序员看来就好像是同一块硬盘。

就长期的观察而言，大多数的文件操作都是追加操作，而不是重写操作。因此，一个好的文件系统应该可以解决这样的一个问题，保证成千上万的用户在对文件进行正确的追加操作，并且这些文件可能分部在不同的服务器上，因此必须做好并发的处理多用户文件操作。在实际的环境下，可能用户不止数万，早在前几年腾讯QQ同时在线用户数就已经超过一亿。因此可以想象腾讯的服务器所承担的压力，也可以看到分布式文件系统在实际中的使用意义。我觉得分布式文件系统应该具有如下的属性：

1. 具有多用户并发访问机制：

不论现在还是未来，多用户访问一定是一个成熟的文件系统所具备的的特点之一。文件不可能每次只让一个用户使用文件系统，否则文件的系统的响应时间太长并且系统利用率都太低了。因此，必须做好文件系统的并发机制，让很多用户可以访问文件系统。

1. 将用户层和底层明显的区分开。

在这里用户层指的是上层程序员的编程接口。底层指的是将若干服务器连接的体系。

无论底层如何变化，程序员的编程接口尽量保持不变。上下部分完全解耦，节约了上层程序员的学习时间，可以让其快速的开发相关的软件，而不用将过多的精力花在学习框架上。

1. 文件需要有强大的冗错能力。

大部分多用户文件系统是架设在廉价的商业服务器上，因此设备自身有很多的错误。因此架设在其上的文件系统必须有很强大的冗错的能力。系统不会因为用户的输入或者传输的错误而崩溃。导致服务器无端死机。

1. 对文件管理有优化的机制。

在一个文件系统中，可能有高达几GB的文件，也可能有只有数十KB的文件。因此如何有效的组织这些文件，对文件系统而言，将是一个很大的挑战。就一般的文件系统而言，访问一个几十KB的文件只需要很短的时间；相比之下几GB的文件的访问时间可能高达数十秒。如何缩短访问文件的时间，除了提高硬盘寻道时间外，在寻找算法上应该有相应的突破点。

以上就是我对分布式冰法文件系统的一个展望。

参考文献

[1]KennethRosen,UNIX参考大全,北京希望电子出版社，2004.

[2]Robbins，UNIX系统编程，人民邮电出版社,2004.

[3]史蒂文斯,UNIX环境高级编程(英文版第2版),人民邮电出版社,2003.

[4]赵炯.Linux内核完全注释[M].第1版,北京:机械工业出版社,2004.

[5]Kerninghan,B.W.,R.Pike.TheUNIXProgrammingEnvironment[M],1stNewJersey:Prentice-Hall,1984.

[6]MauriceJ.Bach著,陈葆钰、王旭、柳纯录、冯雪山等译.UNIX操作系统设计[M].第1版,北京:北京大学出版社,2000.

[7]于渊,自己动手写操作系统.第1版，电子工业出版社，2005

[8]WayneA.Christopher,StevenJ.ProcterandThomasE\_Anderson.TheNachosInstructionalOperatingSystem.

[9]史杏荣，杨寿宝.操作系统原理与实现技术.第1版，合肥：中国科学技术大学出版社，1997.

[10]汤子瀛，哲凤屏，汤小丹.计算机操作系统.修订版.西安：西安电子科技大学出版社，2006

[11]殷人昆.数据结构（用面向对象方法与C++语言描述）.第2版，北京：清华大学出版社，2007

[12]徐谡，陈洪彬，彭珊珊，谢晓娜.Linux命令行实用技术详解.第1版，北京:人民邮政出版社，2008

[13]NeilMatthew,Ric.Linux程序设计.第1版,北京:人民邮政出版社，2007

[14]SanjayGhemawat,TheGoogleFileSystem,2003

致谢

这次毕业设计可以圆满的完成,很感谢刘宁钟教授给我的帮助。刘宁钟教授在程序设计方面以及系统搭建方面给了我很大的帮助，并且在书籍和资料方面给了我很多的启示；感谢我的父母在写论文这段时间给我的鼓励和支持，没有你们，我没有办法坚持下来；感谢我未来的公司，是公司给了我无限的包容，让我们可以安心度过这段坎坷的时光，令我静下心来安心进行书写论文的工作。