|  |
| --- |
|  |
| 文件系统实习报告 |
|  |

|  |
| --- |
| 姓名 徐力有 学号1200012903  日期 4/25 2015 |

目录

[内容一：总体概述 3](#__RefHeading___Toc286402948)

[内容二：任务完成情况 3](#__RefHeading___Toc286402949)

[任务完成列表（Y/N） 3](#__RefHeading___Toc286402950)

[具体Exercise的完成情况 3](#__RefHeading___Toc286402951)

[内容三：遇到的困难以及解决方法 4](#__RefHeading___Toc286402952)

[内容四：收获及感想 5](#__RefHeading___Toc286402953)

[内容五：对课程的意见和建议 5](#__RefHeading___Toc286402954)

[内容六：参考文献 5](#__RefHeading___Toc286402955)

* 1. 内容一：总体概述

* 1. 内容二：任务完成情况
     1. 任务完成列表（Y/N）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Exercise1 | Exercise2 | Exercise3 | Exercise4 | Exercise5 | Exercise6 | Exercise 7 | Challenge 1 |
|  | Y | Y | Y | Y | Y | Y | Y | Y |

* + 1. 具体Exercise的完成情况
       1. 第一部分(第一周)

**Exercise 1 源代码阅读**

**阅读Nachos源代码中与文件系统相关的代码，理解Nachos文件系统的工作原理。**

**code/filesys/filesys.h和code/filesys/filesys.cc**

**code/filesys/filehdr.h和code/filesys/filehdr.cc**

**code/filesys/directory.h和code/filesys/directory.cc**

**code /filesys/openfile.h和code /filesys/openfile.cc**

**code/userprog/bitmap.h和code/userprog/bitmap.cc**

先看filesys.h

文档介绍先说了下我们的之前做的多线程和虚拟内存都是在用unix本身的filesystem。

而现在我们要做的文件系统lab则是在模拟的disk的基础上做的。

模块FILESYS\_STUB 中定义的就是临时unix文件系统调用的包装，我们略去。直接进入自己模拟的文件系统。类FileSystem 有两个成员变量freeMapFile，里面存的是freedisk的bitmap；还有directoryFile，里面存的是根目录，里面存的是文件的名字。

然后看filesys.cc构造函数 假如format为 true：

新建一个numsectors的bitmap。然后新建一个有NumDirEntries个的directory；

然后新建两个fileheader的数据结构分配给mapHdr和dirHdr两个fileheader指针。

我们先看directory这个类。

DirectoryEntry 结构里有bool inuse，int sector， char name[FileNameMaxLen + 1];

然后看Directory类，里面有成员变量 int tableSize； DirectoryEntry \*table；成员函数有构造析构函数略去，

FetchFrom（OpenFile \*file）函数，从disk中读取目录的内容

WriteBack（Openfile \*file）函数，将对directory的修改写回到disk

int FindIndex（char \*name），根据name来寻找这个文件在directory中的序号。

Int Find（char \* name），根据name寻找目录中的对应文件并返回该文件的sector号

Add（char \*name，int newSector）在目录项中插入一个文件

Remove（char \*name）目录中删除一个文件

List（） 打印该目录下的所有文件名字。

Print（）打印目录中每个文件的详细信息，还有文件的的具体内容。

接下来我们来看filehdr.h和filehdr.cc中的内容来仔细看看filehdr类到底实现了哪些东西。

FileHeader的成员变量有int numBytes；int numSector；int dataSectors【NumDirect】

numBytes就是文件总共有多少bytes，numsector是指占用了多少个sector，dataSectors数组里存了每个数据sector的编号。

成员函数解析：

Allocate（bitmap\* freeMap， int fileSize）初始化一个fileheader，并在freemap中找出足够大的空间来放此file的内容

Deallocate（Bitmap \*freemap）释放文件的所有datasector资源

FetchFrom（int sector）在synchDisk中具体某个sector中读取fileheader写到自身（这样竟然都可以！！！！！！！！！！！！！！！！！）

WriteBack（int sector）把自身写入到synchdisk

ByteToSector(int offset)根据文件整体的offset计算出这个byte应该在disk的哪个sector上。

FileLength（）返回文件大小

print（）输出文件信息 包括文件多大，分别用了哪几个disk sector 并且打印文件内容

接下来看openfile.h中的内容。我们查看openfile.h 和openfile.cc（我们看的是FILESYS模块部分）

成员变量有 fileheader \* hdr 和int seekPosition 现在已经处于文件中的哪个位置

OpenFile(int sector) 抓取文件的fileheader 将seekPosition置零。

析构函数略去不写

Seek（int position）查找到相应位置

Read（char \*into， int numBytes）调用readAt函数，就是在当前位置继续往后读取numBytes个字节数据到into的位置

Write（char \*into， i那天numBytes）同Read

Length（）返回这个文件的长度

ReadAt（char \*into， int numBytes， int position）

先用一个buff把可能用到的sector全部取出来，然后bcopy拷贝要用到的部分到into指针

WriteAt（char \*from， int numBytes， int position）

同上把from内容写在一个buff上，然后将buff写回到sector上

现在我们重新回到filesys.cc中，我们此时已能看懂构造函数其实就是初始化了bitmap和directory，给这两部分在磁盘里预留了空间，并且修改了fileheader并且写回到磁盘里

Create（），创建一个文件，注意一定要在所有操作显示成功后，才可以flushback to disk

Open（），打开一个文件 Remove（）删除一个文件 List（）打印目录下所有文件信息

**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。**

首先我们要决定我们打算怎么去做这几项内容。创建时间、上次访问时间、上次修改时间、类型都应该跟随fileheader对象，fileheader其实就相当于inode，我们在此后实现扩展文件长度时，可以将dataSector【】数组的最后一个作为一级索引。本来仿照unix的设计应该是3级索引，但是我们既然是实习，所以就实现一级索引测试即可

类型主要是需要记录这个文件是普通文件还是目录文件。创建时间、上次修改时间、上次访问时间，全部用totalTicks来表示即可。然后还有文件名的长度限制，实现方法1就是给每个文件的名字一个单独的磁盘块，然后directory里的name成员变量填sector号。这样做就是无论name长短，都会占用一个sector。我们都知道无论linux还是windows文件名长度是肯定会有长度限制，而且一般上限是128bytes，又由于nachos的sector size刚好是128bytes，所以我们在name处就填sector号。方法二，此方法需要结合之后突破文件长度的作业，直接令name为一个128位的字符串。这样的话，一个目录文件的大小就不可控了，肯定超过一个sector。

路径的添加，我们打算添加在目录文件上一个绝对路径的成员变量，这样每次可以使用父目录文件的绝对路径加上现在的文件名构造出现在文件的绝对路径。

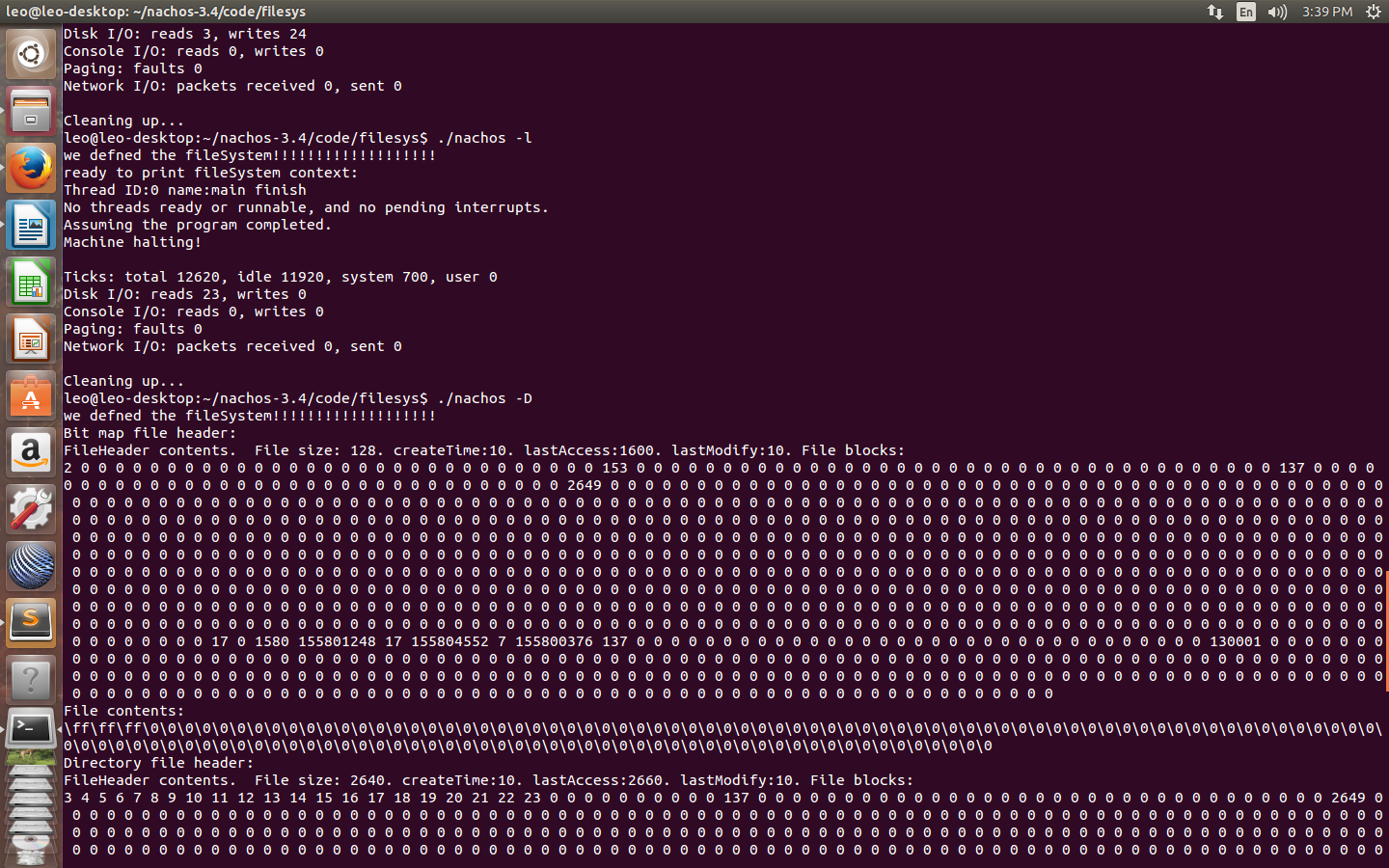
**Exercise 3 扩展文件长度**

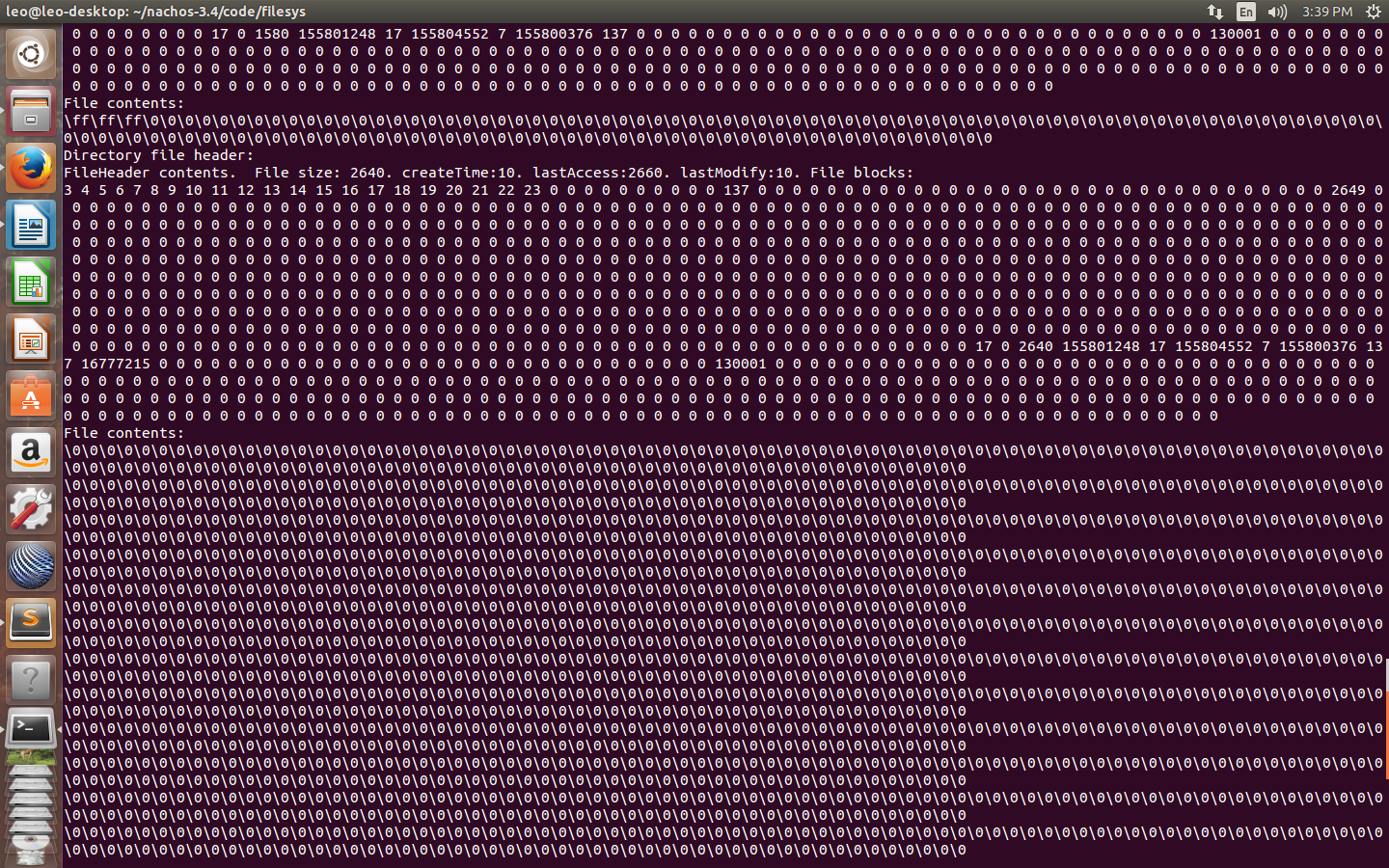
**改直接索引为间接索引，以突破文件长度不能超过4KB的限制。**

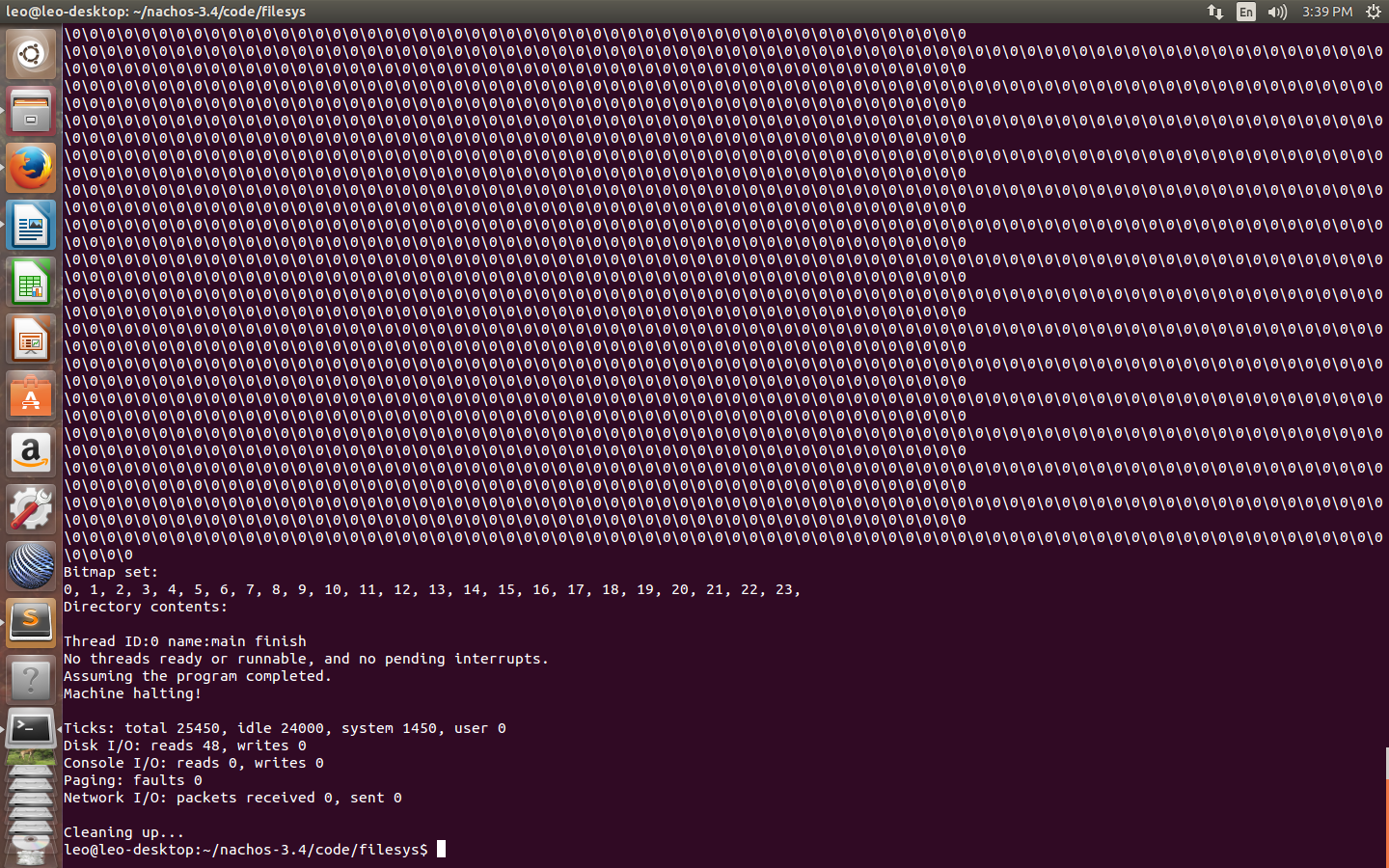
修改filehr的最后一个索引为一级索引。这样做之后需要修改fileheader类中的allocate函数deallocate函数、print函数。

我们还需要修改openfile类中的一些函数，间接索引能够得以运行。主要是readat和writeat，因为我们需要让openfile知道我们使用了一级索引的这种结构。我们发现我们只需要更改bytetosector函数，使这个函数识别一级索引结构，这样就不用跟改readat和writeat函数。

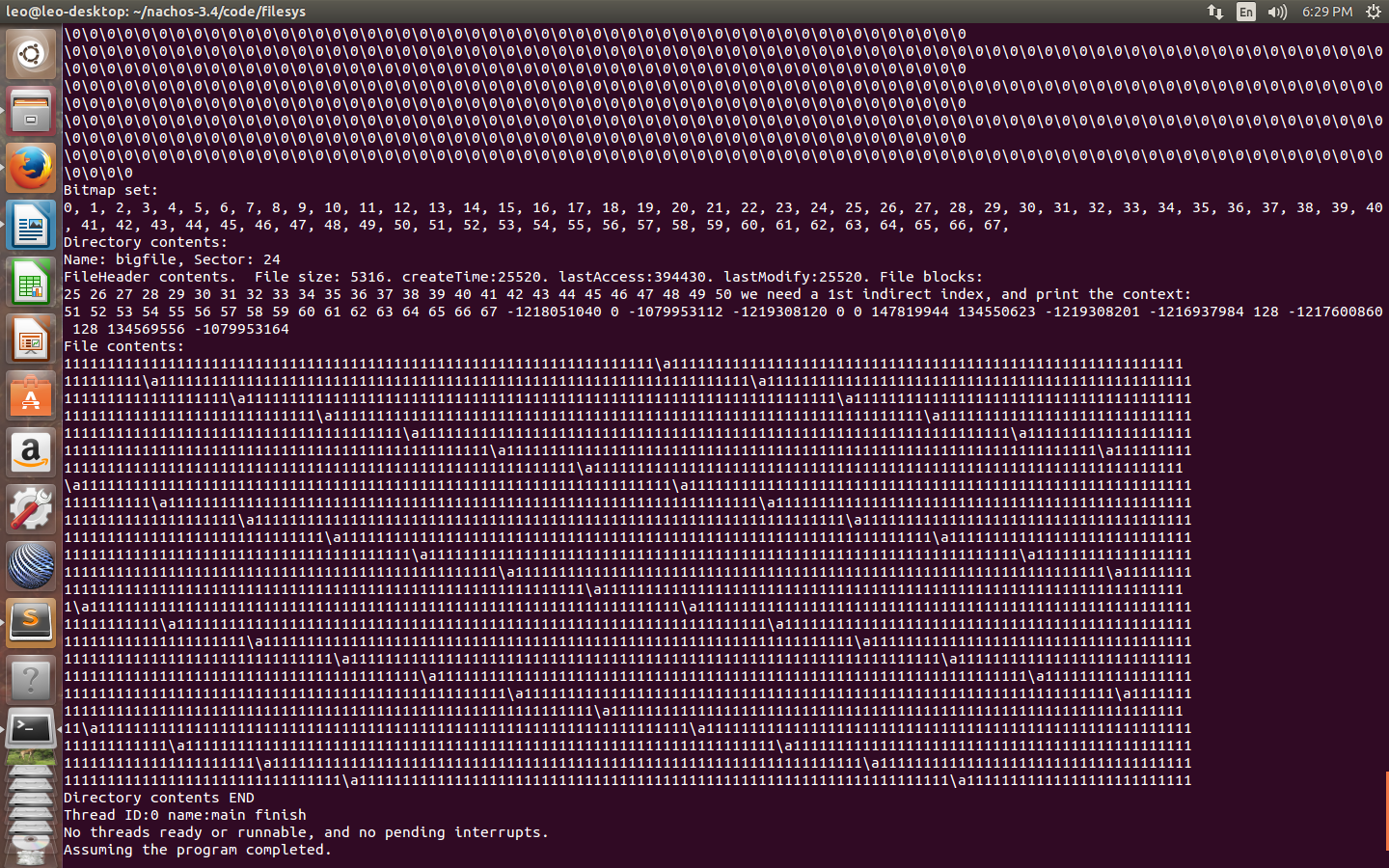
这里我们已经可以做测试了。首先我们需要把filesys文件下的makefile中的-DTHREAD去掉，因为main.cc 中thread模块的操作会改变argv的参数，使得文件系统操作的参数变得不对。





这3张图显示了nachos -D的内容 我们可以看到bitmap文件128byte，所以文件头加内容共2sectors，directory2640bytes，所以占了21个sector加上fileheader共22个，两者合计24个sector，这与bitmap print中0~23恰好吻合。在这里我们可以看到文件的exercise中的创建时间等属性。

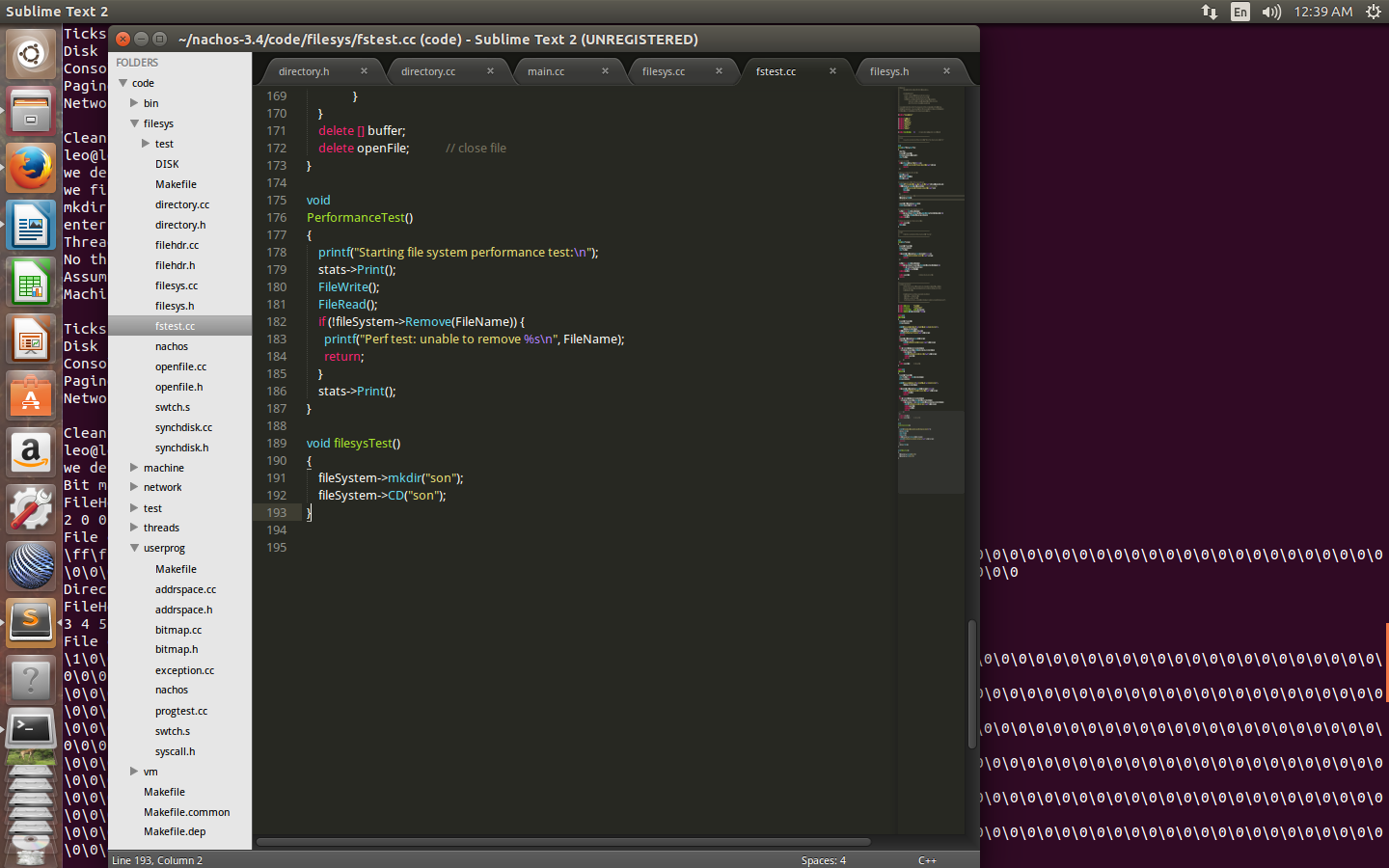
我们尝试复制一个超出4kb的文件到nachos的文件系统。

我们在filesys/test下构造了一个5KB的文件，用来测试一级索引的正常使用情况。结果是可以正常使用。

**Exercise 4 实现多级目录**

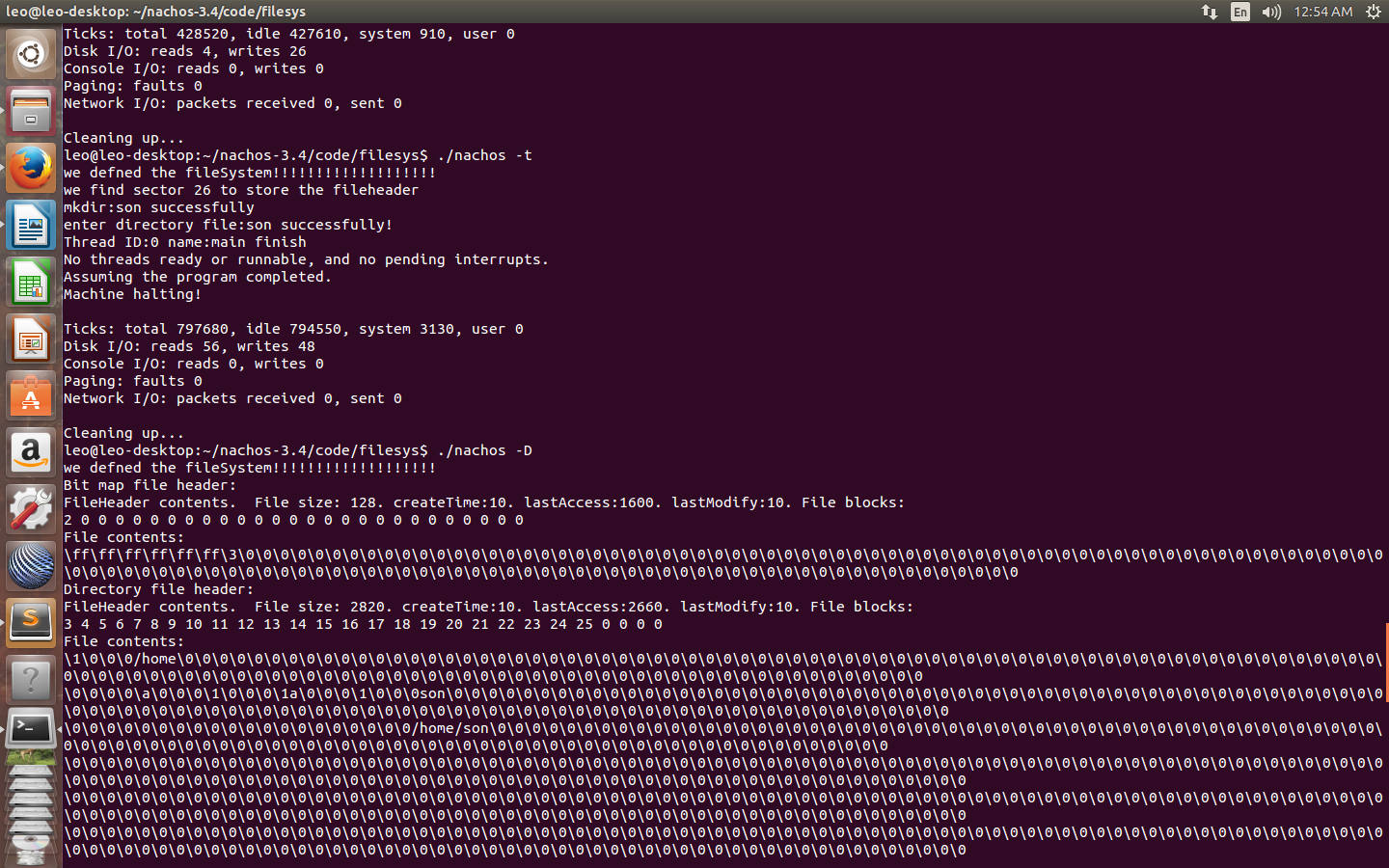
**要实现多级目录，我们必须把目录文件也当做一种文件而且我们最好能在目录文件中就能识别出来。所以这样我们之前把文件类型这个属性加在fileheader这个类上就不太合理了，应该在directorEntry里就加上filetype。这样我们就可以直接决定应该用new fileheader 还是 new director来拉取文件了。**

**然后我们在filesystem中添加一个成员变量记录当前在哪个directory\*中， 并且记录一个父亲directory\***



**这里我写了测试函数filesysTest，在nachos -t的时候会运行。**

**我们尝试建立多级目录，并将当前目录前进到新建的的son目录下。**



leo@leo-desktop:~/nachos-3.4/code/filesys$ ./nachos -l

we defned the fileSystem!!!!!!!!!!!!!!!!!!!

ready to print fileSystem context:

daughter

sons

###########Sub Directry###################

bill

leo

###########End Sub Directory###############

以上是测试结果。改写list函数后，我们完美的打印出了多级目录的情况。

**Exercise 5 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**

动态調整文件长度，首先我们要明白我们的readat的功能是在file的对应地方写上要写的内容，但是有可能会写超过，原来的nachos会自动忽略超出的部分，现在我们要解决的就是写入的时候，加入发现不够大，就要动态继续分配sector，扩大文件长度。

关键就是对openfile的writeat函数的改造。

测试结果：

Directory contents:

Name: bill, Sector: 52

FileHeader contents. File size: 105. createTime:1233520. lastAccess:42590. lastModify:1233520. File blocks:

53 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

File contents:

11111we try to write bytes more than the file can hold !\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

Name: leo, Sector: 54

FileHeader contents. File size: 10. createTime:1665520. lastAccess:43650. lastModify:1665520. File blocks:

55 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

这里注意bill这个文件，初始文件大小10,后来在5的位置直接写100bytes，所以现在文件大小为105.所以得以验证实现的正确性。

**Exercise 6 源代码阅读**

**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**

**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。**

Synchdisk 其实就是在disk的基础上包装了一个信号量和一个互斥锁。我们具体看synchdisk.cc中代码。

利用disk在readRequest或者writeRequest后隔一定时间会触发diskinterrupt ，从而执行requestdone（）函数。requestdone会释放信号量，从而使readsector和writesector必须一个接一个执行，lock则保证了synchdisk的read和write的互斥执行。

接着我们看console类。我们想要写synchConsole，就是要让读和写分别都是逐个进行的。

那么我们相当于在读和写操作时要进行p操作，而在完成读和写的时候利用interrupt触发的handler实现V操作，从而达到同步的目的。

synchconsole.h：

#include "copyright.h"

#include "utility.h"

#include "console.h"

#include "synch.h"

#include "system.h"

// The following class defines a hardware console device.

// Input and output to the device is simulated by reading

// and writing to UNIX files ("readFile" and "writeFile").

// Since the device is asynchronous, the interrupt handler "readAvail"

// is called when a character has arrived, ready to be read in.

// The interrupt handler "writeDone" is called when an output character

// has been "put", so that the next character can be written.

class SynchConsole {

public:

SynchConsole(char \*readFile, char \*writeFile, int callArg); // initialize the hardware console device

~SynchConsole(); // clean up console emulation

void SynchPutChar(char ch);

char SynchGetChar();

private:

Console \*console;

Lock \*readLock;

Lock \*writeLock;

};

synchconsole.cc：

#include "copyright.h"

#include "console.h"

#include "synchconsole.h"

#include "system.h"

//#include "synch.h"

// Dummy functions because C++ is weird about pointers to member functions

static Semaphore \*readAvail;

static Semaphore \*writeDone;

static void ReadAvail(int arg) { DEBUG('a', "ReadAvail:readAvail->V()\n"); readAvail->V(); }

static void WriteDone(int arg) { DEBUG('a', "WriteDone:writeDone->V()\n"); writeDone->V(); }

//----------------------------------------------------------------------

// SynchConsole::SynchConsole

// Initialize the simulation of a hardware console device.

//

// "readFile" -- UNIX file simulating the keyboard (NULL -> use stdin)

// "writeFile" -- UNIX file simulating the display (NULL -> use stdout)

// "readAvail" is the interrupt handler called when a character arrives

// from the keyboard

// "writeDone" is the interrupt handler called when a character has

// been output, so that it is ok to request the next char be

// output

//----------------------------------------------------------------------

SynchConsole::SynchConsole(char \*readFile, char \*writeFile, int callArg)

{

console = new Console(readFile, writeFile, ReadAvail, WriteDone, callArg);

readAvail = new Semaphore("read avail", callArg);

writeDone = new Semaphore("write done", callArg);

readLock = new Lock("readLock");

writeLock = new Lock("writeLock"); //printf("init console done\n");

}

SynchConsole::~SynchConsole()

{

delete console;

delete readAvail;

delete writeDone;

delete readLock;

delete writeLock;

}

char SynchConsole::SynchGetChar()

{

char ch;

readLock->Acquire();

readAvail->P();

ch = console->GetChar();

readLock->Release();

return ch;

}

void SynchConsole::SynchPutChar(char ch)

{

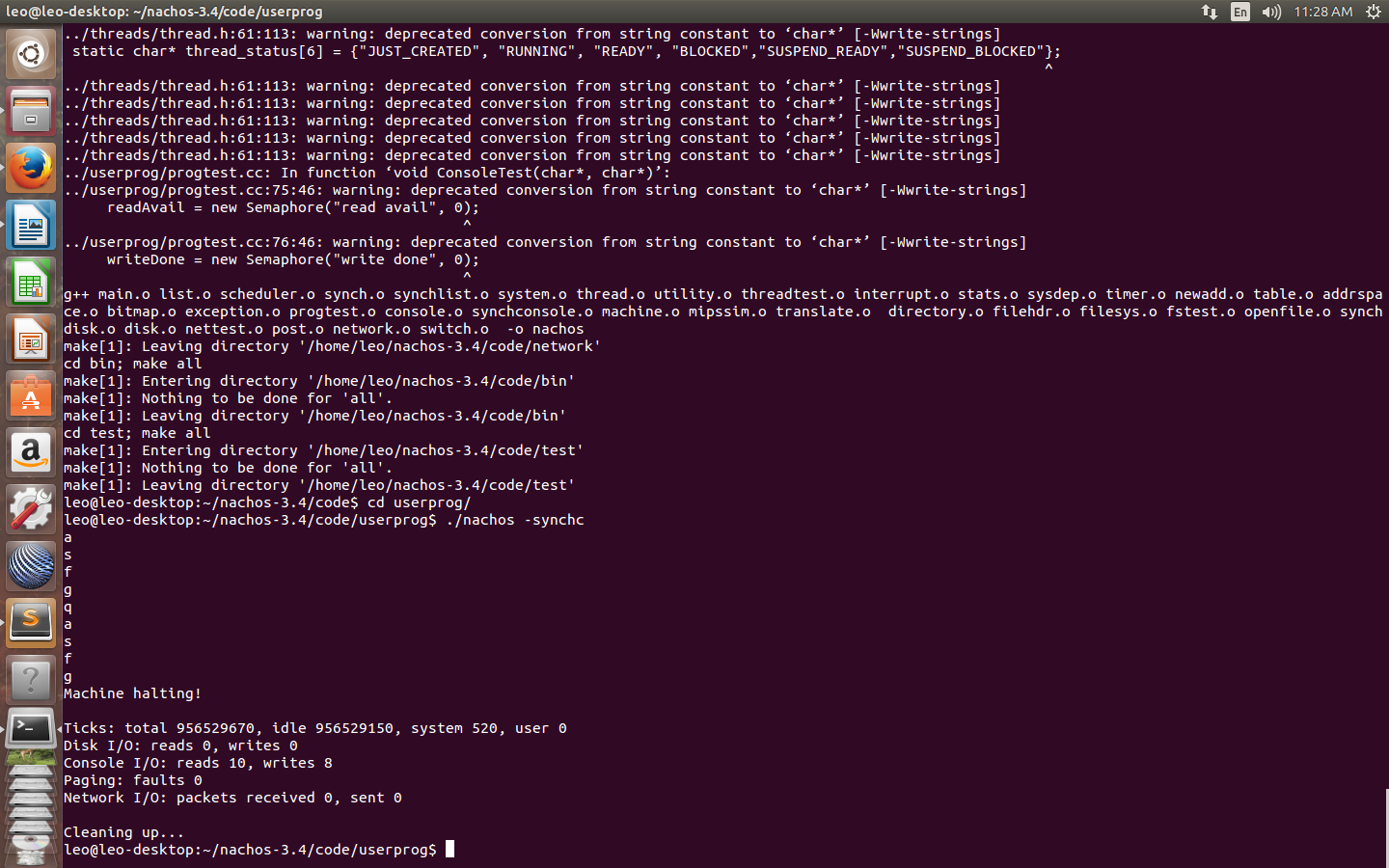
writeLock->Acquire();

console->PutChar(ch);

writeDone->P();

writeLock->Release();

}



synchconsole的简单测试结果：

可以正常使用，虽然不知道怎么测试它的互斥性。。。。。orz

**Exercise 7 7 实现文件系统的同步互斥访问机制，达到如下效果：**

**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。**

**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。**

对文件系统的操作必须是原子操作和序列化的，所以我们需要把filesystem 的各个成员函数加上开关中断进行保护。将filesystem的构造函数，create函数、remove函数、mkdir函数、CD函数。

现在已经能解决每个线程独自打开文件，且独自拥有一个当前文件访问位置，这个是openfile类已经保证了的。彼此之间不会干扰。通过开关中断的保护，再加上synchdisk的实现，我们也已经保证了所有对文件系统的操作都是原子化的和序列化的，不会出现不可预计的中间状态。

但是我们还需要解决一个问题，就是当某一线程打算删除一个文件时，需要判断当前文件没有被其他线程打开，若有被其他线程打开，那么删除操作就不能正确执行，需要报错警告。我们如何知道当前这个文件有没有被其他线程打开，我们只需要有一个全局的系统打开文件表即可。我们记录一个文件的文件头的sector号来代表文件，然后要记录一个对应的referCount 引用计数。这样我们只需要检查referCount是否为1来判断知否只有当前线程打开此文件。在openfile的构造函数中我们需要将技术+1 在析构函数中，将计数-1.

Ok 简便起见 我们在system.h 中声明1个numsectors的数组，用来记录每个文件的refercount。

**Challenge 1 性能优化**

**例如，为了优化寻道时间和旋转延迟时间，可以将同一文件的数据块放置在磁盘同一磁道上**

**使用cache机制减少磁盘访问次数，例如延迟写和预读取。**

实现challenge1 将同一文件的数据尽量放在磁盘同一个track上，要做到这一点我们需要对bitmap的find函数进行适当修改，或者我们在里面添加新的函数。在这里我的方法是在bitmap里添加FindBlock（int size）函数，这个函数会在disk的track中挑一个能能容纳下size的track，然后在上面连续分配sector。

使用cache机制，就是在内存中分配一部分空间，对应sector。每一次读或写一个sector的时候，总是先在cache中查找，类似tlb的写法。加入发生cache满了，那么采用剔除算法，剔除一个sector写回到disk。这里的cache的组织方式就有很多种，为了方便起见就使用全相连。剔除算法用LRU。

**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。**

* 1. 内容三：遇到的困难以及解决方法
     + 1. 困难1

在添加fileheader的成员变量后，忘记更改NumDirect的值导致实现间接索引的exercise时出现奇怪的错误。找了好久才找出来。。。。。。orz

* + - 1. 困难2

directory由于本来只有一个directory， 所以directory的fetchfrom和writeback都只有对table内容的操作，我们的写入必须把directory的其他内容也写到磁盘里。所以我们需要对fetchfrom 和writeback函数进行修改。

**困难3**

时间真的不够啊，感觉工作量跟3周的有得一拼。但时间只有2周。。。。。。。。。。。orz

* 1. 内容四：收获及感想
  2. 内容五：对课程的意见和建议
  3. 内容六：参考文献

教学网上nachos参考资料 nachos study book , The University of Southern Queensland Faculty of Sciences, Version 1.2

|  |
| --- |
|  |
| 文件系统实习报告 |
|  |