文件系统实习报告

姓名张颜 学号 2001210541
日期 2020/12/20

目录

内容一：总体概述

本次 lab 主要是改进 nachos 的文件系统。Nahcos 实现了一个小型的文件系统，但是这个系统很小，而且支持的功能不多，本次实验的主要目的是扩展 nachos 的文件系统。扩展的文件系统，将支持更多的功能，打破文件名长度的限制，打破文件大小的限制等等。除此之外，还要求文件系统支持多线程的访问，最终将之完善成为一个可用的文件系统 demo。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise7	Challenge
	Y	Y	Y	N	Y	Y	Y	Y

具体 Exercise 的完成情况

第一部分

Exercise1 源代码阅读

Nachos 实现了一个简单的文件系统。

在硬件模拟方面，这个文件系统用一个 UNIX 文件来模拟磁盘，用 UNIX 提供的文件读写函数来模拟磁盘的读写。在次基础上，使用同步互斥锁和信号量来实现对磁盘的异步互斥访问。

然而这个文件系统有着诸多的限制。对于一个 nachos 文件来说，一个文件包含有一个文件头，该文件头存储在磁盘的一个扇区中，通过这个文件头，可以找到文件的存储位置。通过文件系统中的目录文件，可以访问一个文件目录，目录记录着目录项，目录项指向一个文件的文件头所在的扇区。Nachos 只有一个目录文件，该文件记录了 nachos 系统中所有文件的目录项。对于每个目录项来说，记录了文件的名字，文件的文件头所在扇区。通过全局唯一的文件名进行文件的检索和查找。

Exercise2 扩展文件属性

需要扩展的文件属性包括文件类型，创建时间，上次访问时间，上次修改时间，路径。

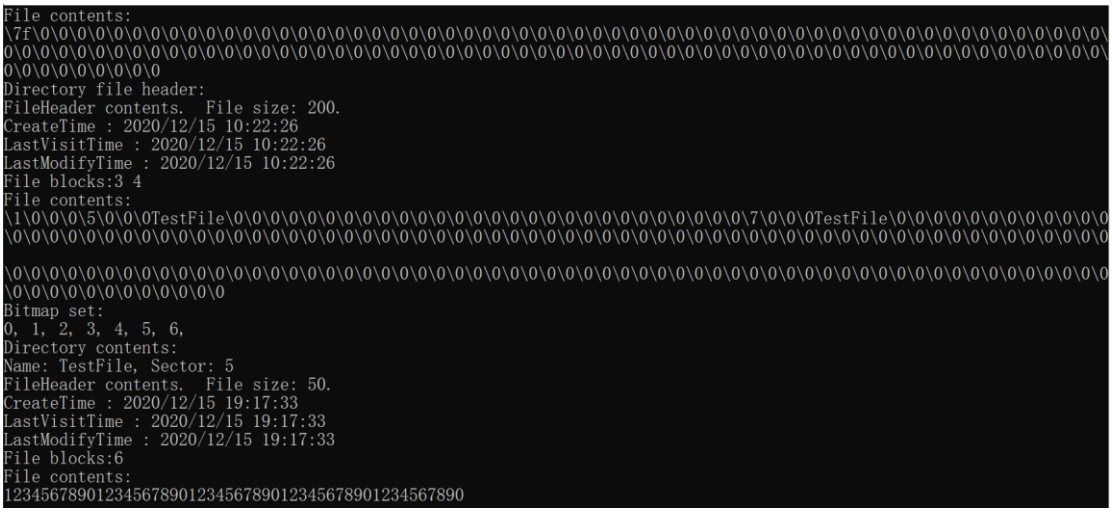
对于文件的创建时间，上次访问时间，上次修改时间这三个文件属性，在文件头中分别定义 createTime，lastVisitTime，lastModifyTime 三个 time_t 类型的变量。这个变量的定义在<time,h>函数中，这个变量类型本质上是 long 类型。调用 time(NULL)函数返回当前的系统时间，系统的时间是从 1970 年开始经历的秒数。用一个 time_t 变量记录这个时间，但是当需要显示这个时间的时候，将这个时间打印成为年月日时分的形式。在文件的创建函数调用修改文件创建读取修改的时间，在写操作中调用文件修改时间更新函数，在读操作中调用文件访问时间更新函数。

在目录文件的每个目录项中添加整形变量 type，和 char[20]数组 path。在每次创建文件的时候，同时给 type 和 path 赋值，文件为目录文件，type 的值为 1，普通文件，type 的值

为 0。在这里需要注意的是，文件名和路径都是一个数组，而不是指针变量。那是因为如果只存一个指针的话，写入文件的就是写入时，name 和 path 变量所在的内存地址。当第二次读出来的时候，读出来的是内存地址，这时候内存经历很多程序的运行，该地址存的内容根本无法预测是什么，再次读取只会得到错误的文件属性，所以，这里需要使用字符型数组来存储文件名和文件路径。

通过修改宏定义#define FileNameMaxLen　9 来修改最大文件的长度。

测试结果：



Exercise3 扩展文件长度

Nachos 规定了文件的最大长度，文件的最大长度使用宏定义

#define MaxFileSize　　(NumDirect * SectorSize)。

在这个宏定义中，SectorSize 代表每个扇区的大小，该值是固定为 128 字节的，在逻辑上，这个值是不应该被改变的，所以，文件长度扩展的关键就在于 NumDirect 的改变。NumDirect 的值是由宏

#define NumDirect　　((SectorSize - 6 * sizeof(int)) / sizeof(int))

定义而来，这个宏的大致意思是，一个文件头只存在一个扇区里，每个扇区的大小是 128 字节。这时候，需要用 128 字节来储存这个文件头，文件头中的内容有变量 createTime 表示文件创建时间，占一个 int 的大小，lastVistTime 表示文件上次被访问时间，占一个 int 的大小，lastModifyTime 表示文件上次被修改的时间，占一个 int 的大小，hdrSector 表示文件头存储所在的扇区，占一个 int 的字节。numBytes 表示文件的大小，占一个 int 的字节。numSectors 表示文件占的扇区数量，占一个字节。除去这 6 个 int 大小的变量，剩余的大小都可以用来存储文件所在的扇区，一个扇区需要一个 int 来表示，最终的换算结果就是存储

文件体所在扇区所用的数组大小。

为了不破坏一个文件头只占一个扇区的规定，同时希望扩大文件的大小，那么最好的办法就是实现多级索引。本实验实现了一个二级索引。

保持文件头中记录文件体所用的数组大小，但是把数组的最后一个元素单拎出来，不用于存储文件的内容，而是用这个元素指向一个索引扇区，在这个索引扇区中存储文件体所在扇区的扇区号。这样一个文件就能扩展（secsize/int-1）个扇区大小的尺寸。

具体来说，对于分配的文件大小，如果使用一级索引可以直接存储的话，那么不需要额外操作，直接使用一级索引记录文件体。文件大小超过限制，这时候，使用前文件头记录文件体数组的 NumDirect-1 个元素指向的扇区存储文件体，第 NumDirect 个元素指向的扇区存储一个索引表，该表的每一个元素代表一个扇区号，指向一个扇区。通过二级索引，扩展了文件的长度。当然此时文件仍然有长度的限制，为

((NumDirect-1)+(sectorSize/sizeof(int)))*sectorSize

新建大文件并写入：

```
Directory file header:
===========================
numDirect :27
===========================
FileHeader contents.  File size: 200.
CreateTime : 2020/12/16 14:48:8
LastVisitTime : 2020/12/16 14:48:8
LastModifyTime : 2020/12/16 14:48:8
File blocks:
3 4
File contents:
\1\0\0\0\5\0\0\0TestFile\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\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\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\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
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
Directory contents:
Name: TestFile, Sector: 5, Type: 0
===========================
numDirect :27
===========================
FileHeader contents.  File size: 5000.
CreateTime : 2020/12/16 14:57:24
LastVisitTime : 2020/12/16 14:57:24
LastModifyTime : 2020/12/16 14:57:24
File blocks:
Direct index :6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Second index Block : 32
Second index :33 34 35 36 37 38 39 40 41 42 43 44 45 46
File contents:
Sector:6
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901
2345678
Sector:7
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
0123456
Sector:8
7890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567
8901234
Sector:9
5678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
6789012
Sector:10
3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123
4567890
Sector:11
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901
2345678
Sector:12
```

Exercise5 动态调整文件长度

Nachos 的文件长度一经定义就无法再次更改，如果要读取或者写入大于文件长度的内容，nachos 会阻止这次读写，最多只能写入到文件最大长度的位置。要打破这个限制就需要对文件大小进行扩展。就文件系统来说，对文件的扩展无非就是修改文件长度，为文件分配新的扇区，将新扇区的扇区号记录在文件头中。

在具体的实现中，在文件头类中定义 ExtendFile 函数，这个函数最主要的功能就是为文件分配新的扇区，将新分配好的扇区号对应填写在文件头中，在文件头中重新定义了文件的大小，扇区数量。为了支持这个 ExtendFile 函数，还在文件头中定义了 hdrSector，这个变量主

要是记录了文件头所被存储在的扇区号，便于 ExtendFile 函数回写时调用。

```
bool
FileHeader::ExtendFile(BitMap *freeMap , int bytes)
{
    int newFileLength = numBytes + bytes;
    int preSectorNum = numSectors;
    int newNumSectors = divRoundUp(newFileLength,SectorSize);
    if(newNumSectors == preSectorNum)
    {
        numBytes = newFileLength;
        return true;
    }
    if (freeMap->NumClear() < newNumSectors - preSectorNum)
        return false;
    printf("\nNeed Extend %d Sectors" , newNumSectors - preSectorNum);
    printf("New Allocate Sectors Index :");
    for (int i = preSectorNum; i < newNumSectors ; i++)
    {
        int temp = freeMap->Find();
        dataSectors[i] = temp;
        printf(" %d\n", temp);
        numBytes = newFileLength;
        numSectors = newNumSectors;
        return true;
    }
}
```

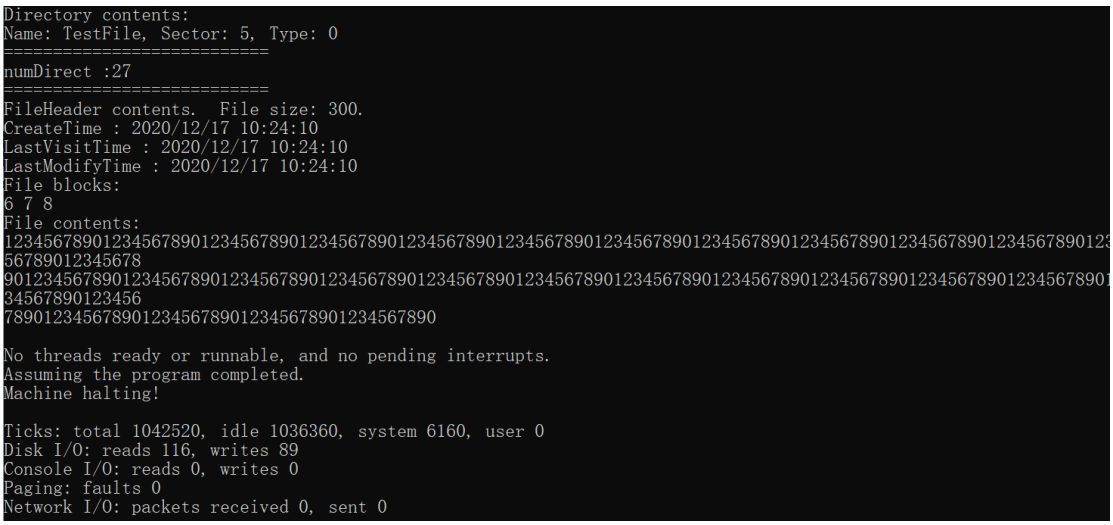
测试，定义小文件，但是再写入的时候，超额写入。

```
#define FileSize        ((int)(ContentSize *5))
```

```
for (i = 0; i < 300; i += ContentSize) {
    {
        numBytes = openFile->Write(Contents, ContentSize);
        //printf("i : %d == contentSzie: %d == numBytes: %d ==\n" , i , ContentSize,numBytes);
```

测试结果：

```
Starting file system performance test:
Sequential write of 50 byte file, in 10 byte chunks
Creating file TestFile, size 50

Nedd Extend 1 SectorsNew Allocate Sectors Index : 7

Nedd Extend 1 SectorsNew Allocate Sectors Index : 8
Sequential read of 50 byte file, in 10 byte chunks
```



第二部分 文件访问的同步与互斥

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

Synchdisk 主要是定义了一把锁和一个信号量。锁的作用是加在读取扇区的操作之前，让读取扇区这个操作，变成互斥的操作。一次只能有一个线程对一个扇区进行读写。信号量的作用主要是实现异步磁盘。在发出磁盘请求之后通过信号量，让当前的线程 sleep 在该信号量上，直到磁盘请求完成后，磁盘中断执行 V 操作，唤醒这个线程。通过一个锁和一个信号量实现了磁盘操作线程之间的同步，让整个过程成为一个异步的过程。

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。

定义了 Synchconsole 类：

```
//-----------------------
//class synchconsole
//synchconsole
//putchar
//getchar
//-----------------------
class SynchConsole
{
  public:
    SynchConsole (char *readFile, char *writeFile);
    ~SynchConsole();
    void PutChar(char ch);
    char GetChar();
  private:
  Console *console;
  Lock *lock;
};
```

这个类主要是对 console 进行同步控制。
具体的，在 PutChar()中：

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

在 GetChar()中:

```
char
SynchConsole::GetChar()
{
    lock->Acquire();
    readAvail->P();
    char ch = console->GetChar();
    lock->Release();
    return ch;
}
```

测试结果

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -c
Successfully Add Thread main to GlobalList!
This is a synchconsole test pro
This is a synchconsole test pro
exit
exit
q
qMachine halting!

Ticks: total 1699812420, idle 1699810070, system 2350, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 39, writes 38
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$
```

Exercise 7 实现文件系统的同步互斥访问机制，达到如下效果：

a) 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。

当前 Nachos 系统中，每个线程每打开一个文件的时候都会 new 一个 openfile 类，openfile 类中定义了文件的读写指针，每个线程都有自己的读写指针，通过这种方式，nachos 线程在访问文件的时候互不干扰。

b) 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。

Nachos 系统实现了 SynchDisk 表示异步磁盘，这个异步磁盘只保证了对某个扇区的互斥访问。对某个扇区的互斥访问，并不代表对文件的互斥访问，一个文件可能会分布在不同的扇区，当一个文件的两个扇区被同时读写的时候，这就存在着一个中间状态。

解决的方法就是对整个文件实现互斥的访问。通过控制对文件头的访问，就可以控制对文件的访问。而文件头是存在在一个扇区中的，那么对每个扇区分配一个信号量，该信号量允许多个读者，或者一个写者。每个扇区中分配一个读者变量标志着当前进入扇区的读者数量，表现形式为一个读者数组。再为这个读者数组分配一把锁，让这个读者数组被线程互斥访问。

```
//读写同步相关变量  文件计数变量
Semaphore *readerWtiterSemap[NumSectors];
int readerNum[NumSectors];
Lock *readerLock;
int visitorNum[NumSectors];
```

定义读写同步的相关函数。

```
//读写同步相关函数
void SynchReaderStart(int sector);
void SynchReaderExit(int sector);
void SynchWriterStart(int sector);
void SynchWriterExit(int sector);
```

这四个函数用于实现读者写者的同步。这是一个典型的读者写者进程同步问题，具体原理就不再赘述。

```
void
SynchDisk::SynchReaderStart(int sector)
{
    readerLock->Acquire();
    readerNum[sector]++;
    if (readerNum[sector] == 1)
        readerWtiterSemap[sector]->P();
    printf("SynchReaderStart The reader num : %d read sector : %d\n" , readerNum[sector] , sector);
    readerLock->Release();
}
```

```
void
SynchDisk::SynchReaderExit(int sector)
{
    readerLock->Acquire();
    readerNum[sector]--;
    if(readerNum[sector] == 0)
        readerWtiterSemap[sector]->V();
    printf("SynchReaderExit The reader num : %d read sector : %d\n" , readerNum[sector] , sector);
    readerLock->Release();
}
```

```
void
SynchDisk::SynchWriterStart(int sector)
{
    readerWtiterSemap[sector]->P();
    //printf("Writer is writing\n");
}
```

```
void
SynchDisk::SynchWriterExit(int sector)
{
    //printf("Writer is Exiting\n");
    readerWtiterSemap[sector]->V();

}
```

调用方法如下：

```
int
OpenFile::Read(char *into, int numBytes)
{
   synchDisk->SynchReaderStart(fileSector);
   int result = ReadAt(into, numBytes, seekPosition);
   seekPosition += result;
   //hdr->SetLastVisitTime();
   synchDisk->SynchReaderExit(fileSector);
   return result;
}
```

```
int
OpenFile::Write(char *into, int numBytes)
{
    synchDisk->SynchWriterStart(fileSector);
   int result = WriteAt(into, numBytes, seekPosition);
   seekPosition += result;
   //hdr->SetLastVisitTime();
   synchDisk->SynchWriterExit(fileSector);
   return result;
}
```

如此实现了读写的互斥访问。

测试：

在 filetest 文件中：

```
void
PerformanceTest()
{
    printf("Starting file system performance test:\n");
    //stats->Print();
    //fileSystem->Print();
    FileWrite();
    read(3);
    Thread *t1 = new Thread("Reader1");
    Thread *t2 = new Thread("Reader2");
    t1->Fork(read ,1);
    t2->Fork(read ,2);
    //FileRead();
    read(3);
    fileSystem->Print();
    if (!fileSystem->Remove(FileName)) {
      printf("Perf test: unable to remove %s\n", FileName);
      return;
    }
    //fileSystem->Print();
    //stats->Print();
}
```

定义了两个线程以及主线程同时访问一个文件。
结果如下：

```
main Go to sleep
main Go to sleep
main Go to sleep
main Go to sleep
main Go to sleep
main Go to sleep
Writer is Exiting
Writer is writing
main Go to sleep
main Go to sleep
main Go to sleep
main Go to sleep
main Go to sleep
main Go to sleep
Writer is Exiting
=================
Successfully Add Thread Reader1 to GlobalList!
Successfully Add Thread Reader2 to GlobalList!
Thread main is going to read the testfile
Sequential read of 500 byte file, in 10 byte chunks
main Go to sleep
Thread Reader1 is going to read the testfile
Sequential read of 500 byte file, in 10 byte chunks
Reader1 Go to sleep
Thread Reader2 is going to read the testfile
Sequential read of 500 byte file, in 10 byte chunks
Reader2 Go to sleep
main Go to sleep
Reader1 Go to sleep
main Go to sleep
```

```
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
SynchReaderStart The reader num : 3 read sector : 5
SynchReaderExit The reader num : 2 read sector : 5
```

c) 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。

如上，定义一个 VisitorNum 的数组，标志着每个扇区当前的访问线程数量，当删除一个文件的时候，该文件如果被其他线程打开，那么该文件的文件头所对应的扇区计数必然不是 0，这个时候就无法对文件进行删除操作。

Challenges

Challenge 1　性能优化

a)　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

实现一个文件数据块尽量放在磁盘的同一个磁道上，需要在分配算法上做文章，分配算法主要需要判断有无符合大小的连续块，如果有，那么分配，如果没有，那么直接按照原来的算法分配。

在分配之前增加 FirstFind()函数来进行连续的块查找。

```
int
BitMap::FirstFind( int size)
{
    for(int i = 0 ; i < numBits ; i++)
    {
        int flag = 1 ;
        for(int j = 0 ; j < size ; j++)
            if(Test(i + j))
                flag = 0;
        if(flag)
        {
            for(int j = 0 ; j < size ; j++)
                    Mark(i+j);
        return i;
        }
    }
        return -1;
}
```

测试：

先创建两个大小 5 个扇区的文件，而后删除第一个文件。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -l
Successfully Add Thread main to GlobalList!
======== File List==========
TestFile
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2550, idle 2420, system 130, user 0
Disk I/O: reads 4, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

```
FileHeader contents.  File size: 500.
CreateTime : 2020/12/20 19:3:31
LastVisitTime : 2020/12/20 19:3:31
LastModifyTime : 2020/12/20 19:3:31
File blocks:
6 7 8 9
File contents:
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
12345678
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
90123456
7890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234
5678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 9440, idle 8920, system 520, user 0
Disk I/O: reads 17, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

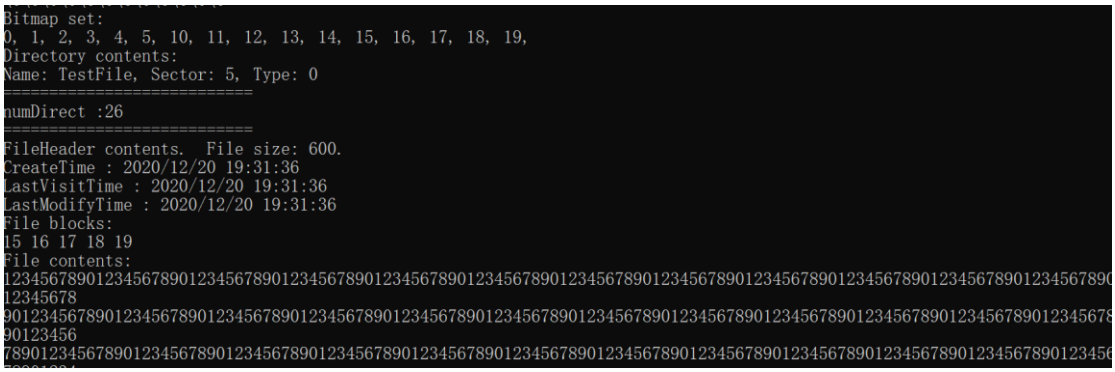
```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -l
Successfully Add Thread main to GlobalList!
======== File List==========
TestFile
TestFile2
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2550, idle 2420, system 130, user 0
Disk I/O: reads 4, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

```
Name: TestFile2, Sector: 10, Type: 0
=========================
numDirect :26
=========================
FileHeader contents.  File size: 500.
CreateTime : 2020/12/20 19:7:48
LastVisitTime : 2020/12/20 19:7:48
LastModifyTime : 2020/12/20 19:7:48
File blocks:
11 12 13 14
File contents:
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
12345678
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
90123456
7890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234
5678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
```

删除第一个文件，此时 6789 扇区都是空的。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -l
Successfully Add Thread main to GlobalList!
======== File List==========
TestFile2
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

再次分配一个大小为 5 个扇区的文件，占用的是 15 16 17 18 19 五个扇区。



b) 使用 cache 机制减少磁盘访问次数，例如延迟写和预读取。

定义 cahe，每次访问的时候先查看 cache，如果 cache 有，那么直接读 cahe，如果没有，那么直接读盘，把内容拷贝进入 cache。Cache 采用先进先出的置换方法。

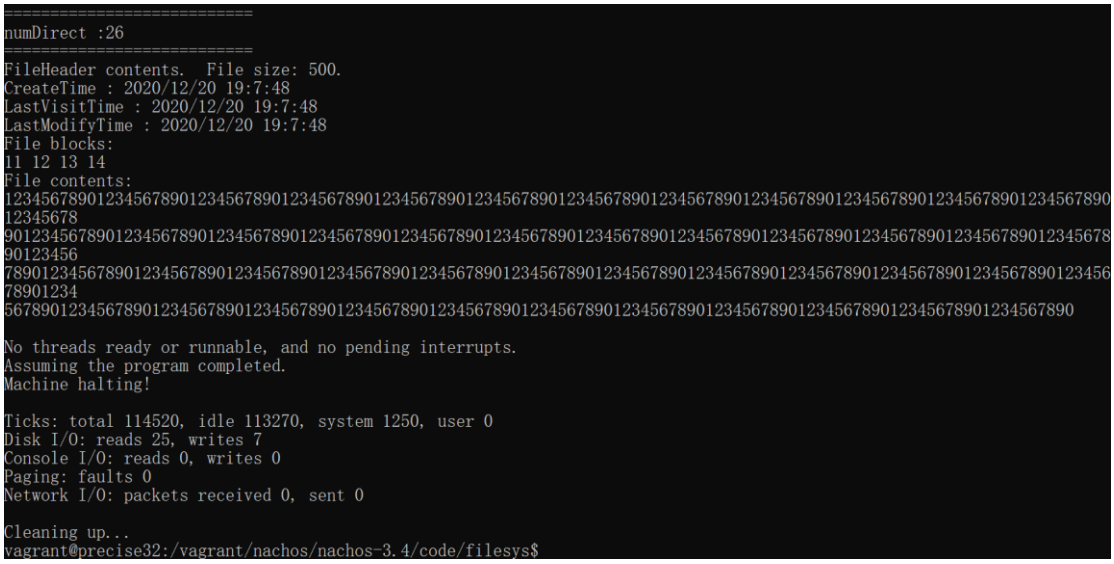
```
Cache cache[4] ;
int FiFoPointer;
```

```
class Cache
{
  public:
    int valid;
    int sector;
    char data[SectorSize];
};
```

测试：

不带 cache:

带 cache:



可以看见带上 cache，磁盘 IO 次数明显减少。

内容三：遇到的困难以及解决方法

困难 1

对文件系统的不熟悉，导致理清代码思路的时候花费了很多时间。

困难 2

对系统的整体性思考不够全面，导致调 BUG 很难。

内容四：收获及感想

文件系统为操作系统中很重要的一部分确实需要我们深入了解。通过实验，更加直观深刻了解了文件系统在实际的设计和实现过程。对理解建立在文件系统上的其他功能，有了更加深刻直观的了解。

内容五：对课程的意见和建议

暂时无。

内容六：参考文献

[1]《nachos 中文教程》
https://wenku.baidu.com/view/905197a9e209581b6bd97f19227916888486b90b.html
[2]《nachos 学习笔记（五）》https://blog.csdn.net/darord/article/details/83303765