

目录

内容一：总体概述.....	3
内容二：任务完成情况.....	3
任务完成列表（Y/N）	3
具体 Exercise 的完成情况.....	3
内容三：遇到的困难以及解决方法.....	17
内容四：收获及感想.....	18
内容五：对课程的意见和建议.....	18
内容六：参考文献.....	18

内容一：总体概述

本次 lab 主要是改进 nachos 的文件系统。Nachos 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，记录了文件的名称，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:

[illegible][illegible]

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

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
```



```

//-----
//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 *readerWriterSemap[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：

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...
```

```
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
```

```
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!
```

```
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!
```


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>