文件系统实习报告

姓名 张煌昭 学号 1400017707
日期 2017.11.26

目录

内容一：总体概述

Nachos 系统中包含了一个非常简单的文件系统，有磁盘、文件目录、文件头（也就是FCB）和管理空闲区的位图。Nachos 还支持对文件的各种操作，创建、读写、删除等。本次实习的各个练习就是为了完善 Nachos 的文件系统，使其更接近于真实操作系统中的文件系统。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise 1	Exercise 2	Exercise 3	Exercise 4	Exercise 5	Exercise 6	Exercise 7	Challenge 1	Challenge 2
Y	Y	Y	Y	Y	Y	Y	Y	--

具体 Exercise 的完成情况

第一部分. 文件系统的基本操作

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**

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

有两个一模一样的 FileSyste 类，说明现在的 nachos 中有两个一样的文件系统，其中加了 FILESYS_STUB 选项的是建立在宿主机 Linux 之上的文件系统，另一个是 nachos 自己的文件系统。建立在 Linux 上的文件系统目的是调用 Linux 的文件系统暂时实现 nachos 文件系统的功能，直到 nachos 自己的文件系统变得可用。在 FileSystem 类中定义了文件系统的初始化函数，以及若干个对文件系统中单个文件进行操作的函数。构造函数传入了一个布尔类型的参数 format，当该参数为 true 时，构造函数初始化新的文件系统，使用新的空闲磁盘块位图以及新的文件目录。否则，载入旧的空闲位图和目录文件即可。

create 函数用来创建文件。传入参数为文件名和初始化长度，因此 nachos 的文件长度不能动态增长。当要创建的文件不存在时，才继续创建文件。创建文件的操作有：载入目录文件和位图文件；为文件头分配磁盘块并加入到目录文件中；为文件数据分配磁盘块并将信息记录在文件头部；将文件头、目录文件和空闲位图写回磁盘。

open 函数用于打开文件。传入参数为文件名，返回值为 openfile 类指针。操作顺序为：从磁盘中载入目录文件；在目录文件中找到文件名对应的文件返回文件头对应的磁盘块号；打开文件头部。

remove 函数用于从文件系统中删除文件。传入参数为文件名，返回值为布尔类型。操作顺序为：从磁盘中载入目录文件和空闲位图；找到要删除文件的文件头；删除文件的数据块，通过 deallocate 实现；删除文件头，清除对应的位图位和目录项；将位图和目录文件写回磁盘。list 和 print 函数用于打印文件系统的信息。比如磁盘块的分配状况、文件目录的情况等等。

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

定义了文件头部类 FileHeader 的数据结构。私有变量包括文件的字节数、文件数据占用的磁盘块的个数以及记录每个磁盘块的数组（共 128 个元素，因此目前支持的文件最大长度约为 4KB 大小）。数据结构中还定义了对文件块的操作函数。

Allocate 函数为文件块分配磁盘块。传入的参数为位图指针和文件大小。操作顺序为：更新文件头的文件长度和磁盘块长度两个变量；为文件块分配磁盘块，此时磁盘块仅仅被分配了，还没有写入。

Deallocate 函数释放为该文件分配的所有的磁盘块。传入参数为位图指针。函数中将磁盘块的占用标清除即可。

FetchFrom 函数用于从磁盘中载入文件头。传入的参数为要载入的磁盘块号。调用 synchDisk 类中的 ReadSector 函数，将磁盘中的内容覆盖自己，类似于构造函数。

WriteBack 函数用于将文件头的内容写回磁盘。传入的参数为要写入的磁盘块号。操作类似于上一个函数。

ByteToSector 函数读入文件中偏移量为 offset 的磁盘块。传入参数为偏移量。计算出文件的偏移量对应的文件块，通过 dataSector 数组返回对应的磁盘块号。

FileLength 函数和 Print 函数用于返回文件的大小以及打印文件的信息。

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

定义了 DirectoryEntry 类和 Directory 类。前者相当于一个目录项，记录了文件名对应的文件头部所在的磁盘号，变量有：inUse 表示该项是否被使用，sector 记录文件头所在的磁盘号以及 name 数组记录文件名。后者则模拟了文件目录，私有变量有：目录项的个数、文件目录指针以及 FindIndex 函数用于寻找文件名对应的目录项的索引。

构造函数传入的参数为 size，该函数在刚才的 FileSystem 的构造函数中被调用，参数大小为 10，也就是说 nachos 文件系统支持的文件数目不超过 10 个。在构造函数中初始化目录表并初始化每项的 inUse 位为 False。析构函数将目录表释放。

FetchFrom 函数用于从目录文件中载入文件目录。WriteBack 函数将文件目录写回目录文件。

Find 函数用于根据文件名查找文件头存放的磁盘块号。Add 函数用于在目录中增加文件目录项。Remove 函数从目录中删除一个文件目录项。List 和 Print 函数打印出每个目录项对应的文件名以及文件头的信息。

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

定义了 OpenFile 类，用于对文件执行基本的读写操作。同样，nachos 定义了两个 OpenFile 类，原因同两个 FileSystem 类的原因类似。对于 nachos 自己的文件系统类，私有变量为文件头指针，以及当前的文件光标指针。

构造函数用于初始化私有变量，传入参数为文件头部的磁盘号，磁盘上的内容覆盖到文件头指针指向的内容。

析构函数用于回收私有变量头部指针。

Seek 函数用于将当前的光标位置设置为传入的参数。

Read 函数用于将 numBytes 个字节读入一个缓冲区数组中，调用自身的 ReadAt 函数，返回成功读到的字节数。Write 函数与此对应。

ReadAt 函数用于从磁盘上读出指定字节到 into 数组中。传入的参数为 into 数组指针，numBytes 以及开始读的位置 position。操作顺序为：调用文件头类中的 FileLength 函数获取文件长度；确定当前位置 position 所在的文件块；计算总共需要的文件块的数目；将这些文件块的内容全部取出放入 buf 数组中，注意这时候 buf 数组中的并不全是我们需要的；将 buf 数组中我们想要的内容复制到 into 数组中。

WriteAt 函数将文件的内容写回磁盘，操作类似于 ReadAt 函数。

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

这两个文件在上一个 lab 中已经用到，定义了 BitMap 类，用来模拟位图。在本次 lab 中，该数据结构用于记录空闲磁盘块，以文件的形式存在于磁盘中。每一位标志了磁盘块的使用情况，0 代表已经分配的磁盘块。使用 BitMap 中的函数实现管理磁盘块的功能。

Exercise2　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。

在 FileHeader 类中添加 createTime，lastAccessTime 和 lastModifyTime 三个 time_t 类型的成员变量分别记录创建时间，上次访问时间和上次修改时间；并添加相应的方法通过 Linux 的 time.h 下的 time 函数设置这三个时间。由于 FileHeader 类中添加了成员，因此可用的 DirectoryEntry 就会减少，需要修改 NumDirect 宏定义满足 FileHeader 的大小限制。

在 DirecrotyEntry 中添加 bool 类型的 isDirecroty 成员用于记录该 Entry 是文件还是目录；在 Directory 类下的 Add 成员方法中进行设置，由于 Add 方法添加的都是文件，因此 isDirectory 设置为 FALSE。

文件名和路径的处理在 Exercise 4 多级目录中进行实现。

修改 FileHeader 类下的 Print 方法，使其打印添加的三个时间成员；之后修改 fstest.cc 下的 PerformanceTest 函数，将文件大小缩小至 1000B，并使之不删除创建的文件。运行./nachos -f 初始化文件系统，之后运行./nachos -t 测试创建并读写文件，最后运行./nachos -D 查看文件系统下各个文件信息。结果如下图所示，文件按照预期创建，添加的时间信息均按照预期显示。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
Ticks: total 1080, idle 1000, system 80, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Sequential write of 1000 byte file, in 10 byte chunks
Sequential read of 1000 byte file, in 10 byte chunks
Ticks: total 8563030, idle 8525140, system 37890, user 0
Disk I/O: reads 324, writes 825
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

```
Name: /TestFile, Sector: 5
FileHeader contents:
File size: 1000
Create Time: Sun Nov 26 14:31:47 2017
Last Access Time: Sun Nov 26 14:31:47 2017
Last Modify Time: Sun Nov 26 14:31:47 2017
File blocks:
6 7 8 9 10 11 12 13
File contents:
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456

12345678
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234

90123456
7890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
```

Exercise3　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。

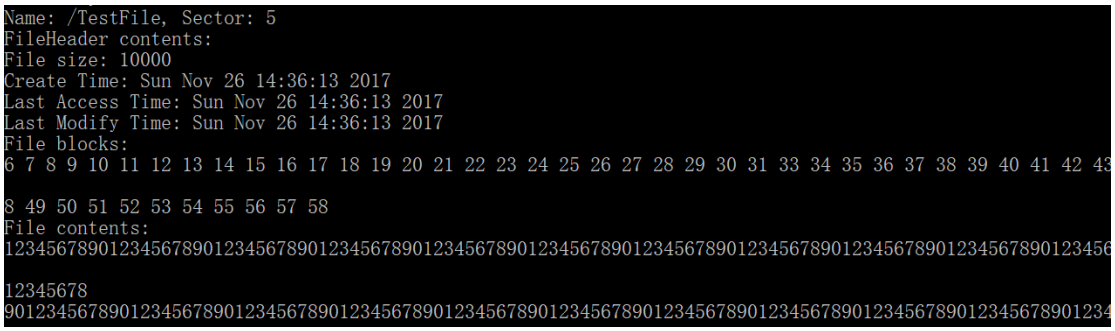
为了简单起见，采用链表结构，当文件不足以放在一个 FileHeader 下时，通过一个指针指向下一个 FileHeader，如此完成对文件长度的扩展。采用链表结构会导致无法实现随机访问，但是链表结构易于扩展且简单可行。

需要向 FileHeader 类中添加 nextSectorOfHeader 成员指明紧接着的下一个 FileHeader，此外由于向 FileHeader 中增加了成员，NumDirect 也会减少，同步地修改 NumDirect 宏定义。

此外还需要修改 Allocate 方法和 Deallocate 方法，Allocate 方法需要在文件长度超限时递归地分配新的文件头与 nextSectorOfHeader 中，若未超长则令其为-1 指明尾部；Deallocate 方法同理递归地释放 nextSectorOfHeader 直至尾部。

修改 fstest.cc 中的文件大小为 10000B（超出 4KB 限制），运行./nachos -f 初始化文件系统，之后运行./nachos -t 测试创建并读写文件，最后运行./nachos -D 查看创建的文件。结果如下图所示，超长文件正常创建，该文件信息也正常。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
Ticks: total 1080, idle 1000, system 80, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Sequential write of 10000 byte file, in 10 byte chunks
Sequential read of 10000 byte file, in 10 byte chunks
Ticks: total 109715030, idle 109231680, system 483350, user 0
Disk I/O: reads 6420, writes 8220
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```



Exercise4　实现多级目录

首先需要将 Directory 类中的所有方法扩展为递归实现，即 Find 扩展为递归的 RecursivelyFind，Add 扩展为 ResursivelyAdd，Remove 扩展为 ResursivelyRemove，List 扩展为 RecursivelyList，Print 扩展为 RecursivelyPrint。

Recursively Find 方法需要根据参数文件名 name 查找文件，需要增加 full path 参数用于传递文件的绝对路径以便打开该文件。若其在本级目录之中则使用 FindIndex 方法放回该文件的 DirectoryEntry；否则递归地向下层目录搜索。由于绝对路径中天然地使用'/'进行划分，因此可以通过 name 中是否有'/'来判断是否搜索到叶节点，若搜索到叶节点则说明该文件只可能在当前目录下，否则该文件不存在；若 name 中有'/'，则找到第一个'/'，之前的部分为下一级目录的名字，在本级目录下搜索该名字，若找不到，说明该目录不存在，则文件也自然地不存在；否则可以找到，找到后需要检查该 DirectoryEntry 是否是目录，若否说明路径出错，若是则取出该 DirectoryEntry，向下继续递归搜索。

原始的 Find 方法改为调用 RecursivelyFind 进行查找。

Recursively Add 方法仿照原本的 Add 方法和上面的递归策略，需要增加 fullPath 和 isDirectory 两个参数以便进行递归地添加，方法同上。若 name 为文件，则找到该文件进行添加；否则 name 为目录，找到并检查是否为目录，之后进行递归添加。

原始的 Add 方法改为调用 RecursivelyAdd 进行添加。

Recursively Remove 方法同上。

原始的 Remove 方法改为调用 RecursiveRemove 进行删除。

Recursively List 方法按照广度优先进行打印。根据参数 layer 打印空格表示层级，之后若为文件则直接打印文件名，否则为目录，打印目录名后，广度优先递归调用打印目录中下一层内容。

原始的 List 方法改为调用 RecursivelyList 进行打印。

Recursively Print 方法按照深度优先进行打印，根据路径向下打印，若 DirectoryEntry 为文件，则调用该文件头的 Print 方法打印文件信息；否则为目录，需要先调用文件头的 Print 方法打印目录信息，之后打开打开该目录，从中递归地按路径向下打印。

原始的 Print 方法改为调用 RecursivelyPrint 进行打印。

在 FileSytem 中使用原先的所有 Directory 方法，现在都可以进入递归的扩展方法之中，从而实现多级目录的递归。

修改 fstest.cc 创建多级目录，并在每级目录中创建文件用于测试该多级目录系统，运行./nachos -f 初始化文件系统，之后运行./nachos -t 测试创建多级目录，最后运行./nachos -l 查看多级目录。结果如下图所示，正常创建多级目录，多级目录层级结构打印也符合预期。

```
    //  printf("Perf test: unable to remove %s
    //  return;
    //}
    fileSystem->Create("file0", 0);
    fileSystem->Create("Dir0/", 0);
    fileSystem->Create("Dir0/file1", 0);
    fileSystem->Create("Dir0/Dir1/", 0);
    fileSystem->Create("Dir0/Dir1/file2",0);
    fileSystem->Create("Dir2/", 0);
    fileSystem->Create("Dir2/file3", 0);
    stats->Print();
}
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
Ticks: total 1080, idle 1000, system 80, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Ticks: total 1632520, idle 1625780, system 6740, user 0
Disk I/O: reads 74, writes 130
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/filesys$ ./nachos -l
 file0
|Dir0
   file1
  |Dir1
     file2
|Dir2
   file3
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

Exercise5　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，以使其符合实习要求。

考虑动态长度调整，会在文件写入而原始文件大小不够时发生。因此需要的改动如下。

首先在 FileHeader 中添加 Realloccate 方法，仿照 Allocate 方法扩展文件长度。检查新的文件长度是否需要扩展链表节点，若是则递归调用 Reacllocate 方法创建新的链表节点；否则按照 Allocate 方法中的分配方法进行分配。

之后在 FileSystem 中添加 Reallocate 方法，分配 BitMap 并调用该文件头的 Reallocate 方法进行动态扩展分配。

最后在 OpenFile 中的 WiteAt 添加动态分配，当检查到需要写的内容会超出原本文件长度时，调用 fileSystem->Reallocate 进行动态扩展，将文件长度扩展至恰满足写入，之后再正常写入。

修改 fstest.cc 中的 FileWrite，不再创建文件，使之接受 dummy 参数，若 dummy 为 0

则进行原来的操作；否则将写两倍的文件长度。在 PerformanceTest 中检查 FileName 文件是否存在，若存在则调用 FileWrite(1)，否则创建 FileName 文件后调用 FileWrite(0)。

运行./nachos -f 初始化文件系统，之后第一次运行./nachos -t 创建 TestFile 文件，运行./nachos -D 查看文件大小，再次运行./nachos -t 写两倍长度到 TestFile 文件中，再运行./nachos -D 查看文件大小是否发生动态变化。结果如下图所示，文件长度在第二次写入时发生了动态调整。

```
File size: 1000
Create Time: Sun Nov 26 15:52:35 2017
Last Access Time: Sun Nov 26 15:52:35 2017
Last Modify Time: Sun Nov 26 15:52:35 2017
File blocks:
6 7 8 9 10 11 12 13
File contents:
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

12345678
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
File size: 2000
Create Time: Sun Nov 26 15:58:59 2017
Last Access Time: Sun Nov 26 15:58:59 2017
Last Modify Time: Sun Nov 26 15:58:59 2017
File blocks:
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
File contents:
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

12345678
9012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
```

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 是在磁盘的基础上包装了一个信号量和一把锁，通过信号量和锁实现对磁盘的互斥访问。锁的作用是保证线程之间对磁盘的读或者写操作是互斥进行的。当一个线程完成读或者写请求后，通过产生一个中断，然后调用中断处理函数 DiskRequestDone 将信号量释放，从而使得一个在等待中的线程继续执行读或者写磁盘操作。

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。

模仿 SynchDisk 的实现，将 Semaphore 和 Lock 包装在 Console 类之中，当用户调用 Console 的功能时，不再需要自己设计信号量和锁，可以由 Console 保证互斥。

SynchConsole 类内包括一个私有的 console 指针和两个锁（一个读锁，一个写锁）。构造函数仿照 Progtest 中的 console 建立，使用两个信号量来进行中断处理，其余方法均仿照 Console 类和 ConsoleTest 函数实现即可。

编写 SynchConsoleTest 函数用于测试，在 main 函数中添加-sc 命令行参数，调用 SynchConsoleTest。运行./nachos -sc 结果如下，与./nachos -c 的行为完全相同。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -sc
a s
a s
dsafdsafsd
dsafdsafsd
gfdgdf
gfdgdf
vxzcvx
vxzcvx
fdsafsadfsad
fdsafsadfsad
```

Exercise7　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。

由于 Nachos 通过 OpenFile 类对文件进行各类操作，而 OpenFile 类中，读写指针为私有变量。因而每个线程创建一个 OpenFile 来打开一个文件，彼此之间相互独立，并不会干扰。

修改 PerformanceTest 函数，先创建一个文件并写 1000B，之后创建两个线程打开该文件，并轮流地读。测试结果如下，读取行为正常。

```
thread 1 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 2 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 1 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 2 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

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。

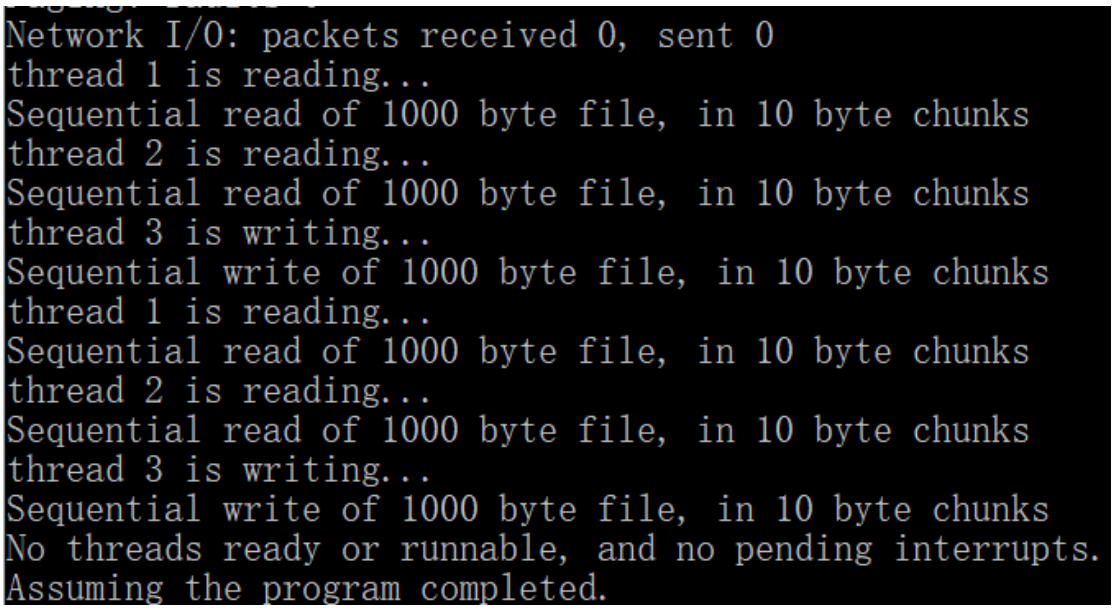
采用第一类读写锁的思路。在 FileHeader 中添加读互斥信号量 read_mutex，读计数 rc 和写锁 write_lock。在 OpenFile 的 Read 和 Write 方法中实现读写锁。

Read 方法首先通过 read_mutex 保护临界区，在临界区内检查 rc，若为 0 说明这是第一个读者，获取写锁禁止写者进入，之后 rc++出临界区进行读操作。读操作完成后，再进入 read_mutex 保护的临界区，rc--后检查 rc，若为 0 说明是最后一个读者，释放写锁允许写者进入，最后出临界区。

Write 方法首先首先尝试获取写锁，若成功则进行写操作，否则被挂起等待，写操作完成后释放写锁。

在 fstest 中创建三个线程，两个读 TestFile 一个写 TestFile，启动顺序为“读写读”。为了进一步说明读写锁有效，在 Read 中出缓冲区后读操作前添加 currentThread->Yield()改变顺

序。结果如下图，最终的顺序为“读读写”，符合第一类读写锁的特征。



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。**

在 FileHeader 中添加打开计数 count，同时修改宏定义，在构造函数中初始化为 0，由于需要保护对 count 的互斥访问，因此所有对其的访问都要收到 read_mutex 和 write_lock 的保护。

OpenFile 的构造函数中添加对 count 的受保护的加一，表示有一个线程尝试打开该文件；同理，析构函数中添加对 count 的受保护的减一。FileSystem 中的 Remove 函数，在删除之前，必须查看 count 是否为 0，若不为 0 则不进行删除操作，直接返回。

修改测试，每个线程在结束之后都尝试删除 Test File 文件。测试结果如下图，只有所有打开的文件都关闭后，最后一个线程才成功地删除了文件。

```
Sequential write of 1000 byte file, in 10 byte chunks
thread 1 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 1 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 2 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 2 is reading...
Sequential read of 1000 byte file, in 10 byte chunks
thread 3 is writing...
Sequential write of 1000 byte file, in 10 byte chunks
thread 3 is writing...
Sequential write of 1000 byte file, in 10 byte chunks
Delete!
Ticks: total 113030, idle 112230, system 800, user 0
Disk I/O: reads 16, writes 7
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

第三部分　Challenge 题目

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

通过阅读 disk 类的代码发现，现有的 Nachos 的访问磁盘时间模拟了寻道延迟和旋转延迟。每次分配磁盘块也是从头到尾扫描磁盘块，若存在未分配的则分配即可。因为每次测试的文件都比较少，也没有在测试过程中删除文件的行为，因此磁盘的访问时间都是比较少的，因为所有的文件都是顺序存放，一个磁道放满之后再放另一个。

因此我的优化方式是，分配文件的第一个数据块时，需要尽量保证后续相邻的数据块可以紧挨着依次放入磁盘上，分配后面的数据块时可以每次从上一个数据块分配的位置开始查找，这样保证了同一个文件的数据块尽量连在一起。

b) 使用 cache 机制减少磁盘访问次数，例如延迟写和预读取。

在 SynchDisk 类中，添加 cache 机制，cache 的大小为 4 个数据块。每当用户调用 ReadSector 时，先在 Cache 中进行查找，若找到了则直接返回，否则才进入磁盘进行查找并替换 cache；在调用 WriteSector 时，现在缓存中查找该磁盘块，若存在则直接修改，同时修改磁盘，否则修改磁盘并替换 Cache。

创建一个大小为 400B 的文件，并读取其中内容，记录访问磁盘次数。在使用 Cache 前需要访问 96 次磁盘，使用 Cache 之后降低至 52 次。说明 Cache 机制可以有效地减少访问磁

盘的次数。

内容三：遇到的困难以及解决方法

1. -DTHREAD 宏定义导致无法测试

在 Excersice 1 之中，使用./nachos -t 无法进入测试函数，经过查看代码发现是因为 -DTHREAD 导致，去除 Makefile 中的宏定义后可以正常进行测试。

2. 宏定义使用出错

修改 FileHeader 的宏定义时由于为添加括号，导致运算优先级出错，经过长时间 Debug 后发现，修改后恢复。

3. Cache 写回

本来试图使用定期写回 Cache 的方法，在时钟中断处理函数之中进行 Cache 写回，但由于系统时钟的某些 bug，并未实现这一想法。最终退而求其次改为使用 Write through 的写回策略。

内容四：收获及感想

本次 Lab 十分复杂，不仅仅需要文件系统，同时还涉及到同步机制、缓存等很多内容。对于知识整合和能力提升很有帮助。

内容五：对课程的意见和建议

暂无。

内容六：参考文献

暂无。