文件系统实习报告

姓名 马少鹏　　学号 1500012893

日期 2018.5.8

目录

内容一：总体概述

本实习希望通过修改 Nachos 系统平台的底层源代码，达到“实现虚拟存储系统”的目标。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise6	Exercise7	Challenge
Y	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

Exercise1 源代码阅读

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

Nachos 中实现了两套文件系统。STUB 是建立在 Linux 文件系统基础之上的，直接使用 Linux 的系统调用完成相关的功能，可以保证结果的正确性。还有一个就是 Nachos 原生的文件系统。两套文件系统有 相同的接口。

原生的文件系统数据结构中，Create 方法创建一个固定大小的文件。Open 方法打开一个已有的文件。Remove 方法删除一个已有的文件。

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

定义了 Nachos 文件头的数据结构。

成员变量有文件的字节数 numBytes，数据块数 numSectors，磁盘块的索引 dataSectors[]。

Nachos 中一个文件头结构的大小和一个 Sector 相同，也就是 128bytes，除掉两个 int 的空间，索引共有（128 – 2 * 4）/ 4 = 30 个。一个 sectorsize 是 128，所以 Nachos 现有的文件系统一个文件最大为 30 * 128 = 3840 bytes。

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

Nachos 中的目录结构。

DirectoryEntry 数据结构是目录中的项，每个指向一个文件头的位置，并记录了文件的名字。

Directory 数据结构是目录，有 Find、Add、Remove 等方法对目录进行操作。

Nachos 现有的目录结构非常简单，只有一级根目录。并且在 FileSystem 的构造函数中，

初始化目录，只有 10 个项。

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

OpenFile 数据结构用来打开、读写文件。此处也有 STUB 和 Nachos 原生两套结构。直接从文件指针位置开始的读写是 Read 和 Write 方法。此外，还有 ReadAt 和 WriteAt 方法，是从指定的位置开始读写。

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

定义了位图的数据结构，在上一次内存管理的实习中已经使用过。

Exercise 2 扩展文件属性

只需要在 DirectoryEntry 数据结构中增加相应属性的成员变量即可：

Char type;　　// 类型

Int create_time;　// 创建时间

Int last_visit_time; // 上次访问时间

Int last_change_time;　// 上次修改时间

Char *path;　// 路径

而突破文件名长度的限制只需要更改：

#define FileNameMaxLen 即可。

修改 Add 成员函数：

```
146  bool
147  Directory::Add(char *name, int newSector, char *path, char type)
148  {
149      if (FindIndex(name) != -1)
150      return FALSE;
151
152      for (int i = 0; i < tableSize; i++)
153          if (!table[i].inUse) {
154              table[i].inUse = TRUE;
155              //strncpy(table[i].name, name, FileNameMaxLen);
156              table[i].name = name;
157              table[i].sector = newSector;
158              table[i].path = path;
159              table[i].type = type;
160          return TRUE;
161      }
162      return FALSE;   // no space.  Fix when we have extensible files.
163  }
```

初始化增加的文件属性。

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。

在 filehdr 类中，修改原有的直接索引（30 个），拆成 20 个直接索引和 10 个二级索引。每个二级索引指向下一级索引块，而每个下一级索引块有 32 个直接索引，所以一个文件最多对应 10 * 32 + 20 = 340 个磁盘块。共 42.5KB。

需要修改代码：

```
// lab5
    int left;
    int second_num;    // how many second indexes we need
    if(numSectors < NumDirect)  // only need direct
    {
        for (int i = 0; i < numSectors; i++)
        dataSectors[i] = freeMap->Find();
        return TRUE;
    }
    else
    {
        for(int i=0;i<NumDirect;++i)
        {
            dataSectors[i] = freeMap->Find();
        }
        left = numSectors - NumDirect;
        second_num = left/32 + 1;
        for(int i=0;i<second_num;++i)
        {
            dataSectors[NumDirect + i] = freeMap->Find();
            int num = (left < 32)? left : 32;
            int *second = new int[num];
            for(int j=0;j<num;++j)
            {
                second[j] = freeMap->Find();
            }
            synchDisk->WriteSector(dataSectors[NumDirect + i], (char*)second);
            delete second;
            left -= 32;
        }
        return TRUE;
    }
}
```

如图，是修改 Allocate 方法的部分代码。首先计算文件需要的磁盘扇区的数量，如果不需要间接索引，则直接分配即可。否则，前 10 个直接分配，后面的需要使用间接索引。Deallocate 方法也需要同样的修改，先释放低级的索引，再释放间接索引。

此外，还需要修改 ByteToSector 方法，思想也是当判断地址所在的扇区在间接索引中时，使用 synchDisk 类的 ReadSector 方法读入改间接索引指向的 32 个低级索引，再在这些低级的索引中寻找。

Exercise4 实现多级目录

原有的 Nachos 只有根目录且最多有 10 个文件。

现在实现树状的多级目录，每层最多可以放 10 个文件或者子目录。

之前已经扩展了文件属性，其中有文件路径 path 和文件类型，类型分为两种：F 文件，D 目录。

修改文件的创建形式，需要修改 FileSystem 类的 Create、Open、Remove 方法。

Create 的实现：

首先判断 initialSize，如果是-1，则说明要创建的是文件夹，否则创建文件。

然后加载根目录，根据文件的名字得到它的路径 path：

```
171
172     char *path;
173     int pos = 0;
174     int len = strlen(name);
175     for(int i=len-1;i>=0;--i)
176     {
177         if(name[i]=='/')
178         {
179             pos =i;
180             break;
181         }
182     }
183     path = new char[pos+1];
184     for(int i=0;i<=pos;++i)
185     {
186         path[i] = name[i];
187         path[pos+1] = '\0';
188     }
189
```

然后调用 directory->get_correct_dir(path)，该函数的作用是在树状的目录中递归寻找要创建的文件的路径。返回值为该路径所在的磁盘 sector。找到后判断该路径是否为根目录 root/，如果不是根目录，需要从磁盘中加载该目录。这一步完成后，directory 指向的目录就是要创建的文件的路径了，即文件所在的目录。

```
266         if(correct_sector != 1)
267         {
268 printf("not the root\n");
269             delete directory;
270             dir = new OpenFile(correct_sector);
271             directory = new Directory(NumDirEntries);
272             directory->FetchFrom(dir);
273         }
```

然后和之前的 create 相同，先为文件头分配磁盘块。然后调用 add 将对应条目加入 table 中。然后写回目录文件。不同的是，如果是文件夹的话，需要创建一个新的 directory 给它。

directory->get_correct_dir(path)函数的实现：

```
267  Directory::get_correct_dir(char *name)
268  {
269      int dir_sector = -1;
270      char *root = "root/";
271      int test = strcmp(root, name);
272
273      if(test == 0)
274      {
275          return 1;
276      }
277      else
278      {
279          int index = Find(name);
280  //printf("index in get: %d\n",index);
281          if(index != -1)
282          {
283              return index;
284          }
285          else
286          {
287  printf("did not find it, now next layer\n");
288              for(int i=0;i<tableSize;++i)
289              {
290                  if(table[i].inUse && (table[i].type == 'D'))
291                  {
292                      int dir_index = table[i].sector;
293                      OpenFile *tmp = new OpenFile(dir_index);
294                      Directory *dir = new Directory(200);
295                      dir->FetchFrom(tmp);
296  //printf("di gui le!!!!!!\n");
297                      dir_sector = dir->get_correct_dir(table[i].path);
298                      delete tmp;
299                      delete dir;
300                      return dir_sector;
301                  }
302              }
303          }
304      }
305      return dir_sector;
306  }
```

首先判断路径是否是根目录，是的话直接返回根目录所在 sector 1。否则在当前目录下查找，如果找到，返回所在 sector，否则递归到下一层目录查找（需要判断是文件夹还是文件，只有文件夹可以查找下一层目录）。如果没有找到，返回-1。

测试：

修改 ftest.c 中的 PerformanceTest:

```
    fileSystem->create_dir("root/A",-1);
    fileSystem->create_dir("root/A/B",-1);
    fileSystem->create_dir("root/A/B/C",-1);
}
```

如图，创建三个文件夹。

结果：

```
msp@msp-virtual-machine:~/下载/nachos_dianti/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
name: root/A    path: root/
find the dir path in sector 1
add name: root/A/ path : root/
successfully create root/A

name: root/A/B    path: root/A/
find the dir path in sector 6
not the root
add name: root/A/B/ path : root/A/
successfully create root/A/B

name: root/A/B/C    path: root/A/B/
did not find it, now next layer
find the dir path in sector 1
add name: root/A/B/C/ path : root/A/B/
successfully create root/A/B/C

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

可以看到，成功的创建了三个文件夹。创建第三层时，在第一层没有找到路径，递归的在下一层找到了。

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。

文件创建时，分配了一部分磁盘 sector，当文件写入时，如果写入文件大小大于初始分配的大小，则需要动态增加文件长度。

创建文件时，设置初始大小最小为 10bytes。

修改 WirteAt 函数，nachos 原有的实现方式：当写入的文件大于原有文件大小时，直接截断为文件所剩空间的大小。修改如下：

```
166     if ((position + numBytes) > fileLength)
167     {
168         //numBytes = fileLength - position;
169 // lab5
170 printf("need more space!\n");
171         BitMap *bitMap = new BitMap(NumSectors);
172         bitMap->FetchFrom(fileSystem->freeMapFile);
173         int size = position + numBytes - fileLength;
174 printf("fileLength before extend: %d\n", hdr->FileLength());
175         hdr->extend(bitMap, size);
176 printf("fileLength after extend: %d\n\n", hdr->FileLength());
177         hdr->WriteBack(_sector);
178         bitMap->WriteBack(fileSystem->freeMapFile);
179     }
```

使用 extend 函数为该文件再申请至少一个 sector 的磁盘空间。注意执行完 extend 后要把相关信息写回磁盘。

Extend 函数实现为 FileHeader 的成员函数：

```
286  bool
287  FileHeader::extend(BitMap *bitMap, int fileSize)
288  {
289      int sector = divRoundUp(fileSize, SectorSize);
290  //printf("sector in extend: %d\n",sector);
291  //printf("numBits: %d\n",bitMap->numBits);
292      //bitMap->Print();
293      if(bitMap->NumClear() < sector)
294      {
295          printf("not enough space!\n");
296          return FALSE;
297      }
298      int already_sector = FileLength();
299      already_sector = divRoundUp(already_sector, SectorSize);
300      if(already_sector + sector > NumDirect)
301      {
302          printf("too big file!\n");
303          return FALSE;
304      }
305      for(int i=0;i<sector;++i)
306      {
307          numBytes += SectorSize;
308          numSectors++;
309          dataSectors[numSectors-1] = bitMap->Find();
310      }
311  //printf("numBytes in extend: %d\n\n",numBytes);
312  //printf("\n");
313      return TRUE;
314  }
```

首先判断磁盘中有没有足够空间，然后判断文件大小是否超过上限。如果以上都符合条件的话，为该文件申请需要的空间。

验证：修改 ftest，创建一个文件，初始大小为 10bytes，分 15 次向文件中写入，每次写入 10bytes，每次写入完成打印文件信息：

```
msp@msp-virtual-machine:~/下载/nachos_dianti/nachos-3.4/code/filesys$ ./nachos -t
Starting file system performance test:
Sequential write of 150 byte file, in 10 byte chunks

Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 10 bytes
file_sector: 7

need more space!
fileLength before extend: 10
fileLength after extend: 138

Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 138 bytes
file_sector: 7 8

Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 138 bytes
file_sector: 7 8
```

可以看到，第二次写入时，文件大小不足了，执行 extend 函数，分配给该文件一个 sector 的磁盘空间，即 128bytes。

```
Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 138 bytes
file_sector: 7 8

need more space!
fileLength before extend: 138
fileLength after extend: 266

Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 266 bytes
file_sector: 7 8 9

Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 266 bytes
file_sector: 7 8 9

Directory contents:
Name: TestFile, file_header_Sector: 6
file_size: 266 bytes
file_sector: 7 8 9
```

第 13 次写入时，再次空间不足，同样再分配一个 sector 的磁盘空间。

第二部分 文件访问的同步与互斥

Exercise6 源代码阅读

filesys/synchdisk.h 和 filesys/synchdisk.cc

通过一把锁，使线程之间对同一个磁盘块的读写是互斥进行的。

通过一个信号量，实现对磁盘读写的同步。一个线程完成操作后，产生一个中断，使得其他等待的线程可以执行。

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。

类似，通过锁，实现读字符和输出字符的互斥。

定义读锁和写锁，可读信号量和可写信号量。

```
char
synch_console::get_char()
{
    char c;
    read_lock->Acquire();
    read_char->P();
    c = console->GetChar();
    read_lock->Release();
    return c;
}

void synch_console::put_char(char c)
{
    write_lock->Acquire();
    console->PutChar(c);
    write_done->P();
    write_lock->Release(;
}
```

实现 read_handler 函数，进行读信号量的 V 操作，write_handler 函数进行写信号量的 V 操作。

Exercise7 实现文件系统的同步互斥访问机制

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，彼此之间不会互相干扰。

Nachos 原有的 OpenFile 类中有成员变量 seekposition，为当前文件访问位置，本来就是线程独自拥有的。

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。

需要修改 read 和 write 函数。

在 SynchDisk 类中增加成员变量和成员函数：

```
45  // lab5
46      void rw_p(int sector);
47      void rw_v(int sector);
48      void cnt_p(int sector);
49      void cnt_v(int sector);
50
51      Lock *rw_locks[NumSectors];
52      Lock *locks[NumSectors];
53      int read_cnt[NumSectors];
```

其中，rw_locks 是每个磁盘块的读写的互斥锁。Read_cnt 是每个磁盘块的读者的数目。Locks 是保护 read_cnt 的锁。

当有线程读磁盘块时，如果该线程时第一个读者，则加锁，否则直接读，允许同时有多个线程读。当线程结束读操作后，如果时最后一个结束的读者，则解锁。当线

程写操作时，加锁，写结束后，解锁，同一时刻只能有一个写者。

```
78  OpenFile::synch_Read(char *into, int numBytes)
79  {
80  //printf("synch_read\n");
81      int start = seekPosition;
82      int last = hdr->ByteToSector(start);
83      for(int i=0;i<numBytes;++i)
84      {
85          int sector = hdr->ByteToSector(start+i);
86          if(sector!=last || i==0)
87          {
88              synchDisk->cnt_p(sector);
89              synchDisk->read_cnt[sector]++;
90              if(synchDisk->read_cnt[sector] == 1)
91              {
92                  synchDisk->rw_p(sector);
93              }
94              last = sector;
95              synchDisk->cnt_v(sector);
96          }
97      }
98
99      int result = ReadAt(into, numBytes, seekPosition);
100     seekPosition += result;
101
102     last = hdr->ByteToSector(start);
103     for(int i=0;i<numBytes;++i)
104     {
105         int sector = hdr->ByteToSector(start+i);
106         if(sector!=last || i==0)
107         {
108             synchDisk->cnt_p(sector);
109             synchDisk->read_cnt[sector]--;
110             if(synchDisk->read_cnt[sector] == 0)
111             {
112                 synchDisk->rw_v(sector);
113             }
114             last = sector;
115             synchDisk->cnt_v(sector);
116         }
117     }
118
119     return result;
```

```
122  int
123  OpenFile::synch_Write(char *into, int numBytes)
124  {
125  printf("synch_write\n");
126      int start = seekPosition;
127  printf("seekPosition of thread %s(tid: %d): %d\n",currentThread->getN
128      int last = hdr->ByteToSector(start);
129      for(int i=0;i<numBytes;++i)
130      {
131  //printf("i: %d\n",i);
132          int sector = hdr->ByteToSector(start+i);
133          if(sector!=last || i==0)
134          {
135              synchDisk->rw_p(sector);
136  printf("i: %d  acquire!!!!!!!!!!!!!!!!\n",i);
137          }
138      }
139
140  //printf("numBytes start write: %d\n",numBytes);
141      int result = WriteAt(into, numBytes, seekPosition);
142      seekPosition += result;
143  //printf("return value in write: %d\n",result);
144
145      last = hdr->ByteToSector(start);
146      for(int i=0;i<numBytes;++i)
147      {
148  //printf("i2: %d\n",i);
149          int sector = hdr->ByteToSector(start+i);
150          if(sector!=last || i==0)
151          {
152              synchDisk->rw_v(sector);
153          }
154      }
155  //printf("result: %d\n",result);
156      return result;
157  }
```

如图，需要对读写操作的每一个磁盘进行加锁解锁操作。

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。

在文件头的数据结构中增加成员变量：int counts。表示当前打开该文件的线程个数。当执行 Remove 操作时，判断 counts 是否为 0，如果不为 0，则当前线程退出，不删除文件，否则删除文件。

第三部分

Challenge 性能优化

a. 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。

修改 FileSystem 类，把之前整个磁盘扇区的 bitmap 修改为 NumTracks 个 bitmap（nachos 默认磁盘有 32 个磁道），每个磁道有自己的 bitmap，每个 bitmap 的大小为 SectorPerTrack，默认为 32。每个磁道的 bitmap 对应一个 freeMapFile。

还需要修改 create 函数，创建文件时，按顺序读取各个磁道的 bitmap，如果空间不够，就读取下一个，否则直接分配磁盘空间。

此外，为了适应之前写的动态增加文件长度，还需要在文件头中记录该文件存储在哪一个磁道上，在执行 extend 的时候，继续在这个磁道上为该文件分配空间。

b. 使用 cache 机制减少磁盘访问次数，例如延迟写和预读取。

在内存中定义一个缓冲区大小为一个磁道，当读写操作时，把读写位置所在磁道的所有 sector 都存到缓冲区中。之后的读写操作都只对缓冲区进行操作。当需要读写的文件不在这个磁道时，再把缓冲区中的数据写回磁盘。

内容三：遇到的困难以及解决方法

困难 1

这次的实习流程很长，而且代码量比之前的都要大，总体还是很有挑战性的。

开始测试的时候，总是运行 thread 的测试程序，后来发现这次测试和 thread 无关，修改 makefile，解决了。

内容四：收获及感想

这次 lab 代码量大，如果不提前规划好直接上手就写代码的话，很容易乱成一团。通过这次实习，锻炼了我写代码之前的规划能力。

深入理解了文件系统，尤其是多级目录。

内容五：对课程的意见和建议

目前对课程非常满意，没有意见。

内容六：参考文献

[1] Andrew S. Tanenbaum 著．陈向群 马洪兵 译 .现代操作系统［M］. 北京：机械工业出版社，2011：47-95

[2] 和队友刘雨曦同学的讨论