操作系统课设实验五---Nachos文件系统扩展

原创

zekdot 于 2018-11-04 14:14:23 发布 4654 收藏 24

分类专栏： Nachos 文章标签： nachos 操作系统课设 文件系统

本文链接：https://blog.csdn.net/zekdot/article/details/83627721

这次的实验让我想起了上学期被操作系统的九个实验支配的恐惧，因为这次可能也是前五次试验中最难的一次了，读源码加实现花了可能一两天的时间，所以也有必要记录一下，有些地方做的不是很好，比如Makefile自己写的话可能不需要把所有文件都从filesys复制到lab5，可是我太菜做不到，所以先这样凑合着吧emmm，毕竟还有一堆其他实验。

一、实验要求重述

目标

在lab5目录下，通过对nachos源代码的修改，实现可扩展的文件系统，即满足以下需求

- 当一个文件创建的时候，它的初始大小为0
- 文件的大小可以在文件内容增加时改变

如：100 bytes大小的文件在50 bytes的位置往后再写100个byte，文件大小变为150 bytes.

设计与实现的提示

1. 在lab5目录下进行工作，其下的test文件夹中存放着测试文件。
2. 需要创建arch子目录以及自己的Makefile文件到lab5。
3. 考虑filesys中的哪些文件需要拷贝到lab5下并进行修改。
4. 不需要修改lab5目录下的main. cc，但是需要将fstest.cc中的Append与NAppend函数中几行注释取消掉：

```
// Write the inode back to the disk, because we have changed it
// openFile->WriteBack();
// printf("inodes have been written back\n");
```

取消最后两行的注释即可。此外，在取消注释之后还需要在OpenFile的类中实现Writeback方法。

二、分析

首先我们需要分析nachos文件部分的源代码，一方面是要了解那七个文件的用法，另一方面要阅读测试使用的main. cc和fstest. cc，通过阅读源码，我们才能达到尽可能少的修改源码来实现功能的目的。

通过对这些源码的阅读以及实验指导书的提示，首先我们需要修改OpenFile这个类，要实现它的WriteBack方法，这个方法的作用是将大小改变的文件头信息写回磁盘。其次在写的方法上我们也要进行修改，那就是该类的WriteAt方法，而因为这里的逻辑是假设文件大小固定，因此我们需要为它添加一个申请新的磁盘空间的方法，这里我在OpenFile类新建了一个AllocateSpace方法。但在实际申请磁盘空间的时候，应该需要在FileHeader类中进行，因此我们为其增加一个ExtendSpace方法。最后，在文件大小变化的时候，应该修改FileHeader类中的numBytes的值，因此我们需要添加一个set方法SetLength。这一切就绪之后，就可以依次实现这些方法了。

ps:原来我在AllocateSpace方法中是直接先释放所有磁盘空间再申请新空间来实现扩容的，经过小伙伴指正我也发现这样确实不行，如果内容存储扇区之前的扇区空间都被别的内容占用的话的没有问题，可是如果有空扇区的话，申请之后文件头信息中指向的内容扇区就会和实际存储的扇区有偏差。
至少之前的设计是有问题的，可以用以下命令实验下试试

```
rm DISK
./nachos -f
./nachos -cp test/big big
./nachos -cp test/small small
./nachos -r big
./nachos -ap test/small small
./nachos -D
```

之前的结果：

```
0, 1, 2, 3, 4, 5, 6, 9,
Directory contents:
Name: small, Sector: 9
FileHeader contents.  File size: 168.  File blocks:
5 6
File contents:
T\1\0\0\3\0\0\0\6\0\0\0\7\0\0\0\8\0\0\0\b0\b7\de\b7\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\80\0\0\0ile
small file small file\a***end of file***\a

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

Ticks: total 6900, idle 6500, system 400, user 0
Disk I/O: reads 13, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up
```


现在的结果

```
\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, 9, 10,
Directory contents:
Name: small, Sector: 9
FileHeader contents.  File size: 168.  File blocks:
10 5
File contents:
small file small file small file\asmall file small file small file\a***end of fi
le***\asmall file small file small file\asmall file
small file small file\a***end of file***\a

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

Ticks: total 6900, idle 6500, system 400, user 0
Disk I/O: reads 13, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up
```


三、实现

首先来说FileHeader类中的SetLength方法，我们只需要简单的设置即可，因此它的代码也非常简单，如下：

```
void
FileHeader::SetLength(int length)
{
    this->numBytes=length;
}
```

下一步来实现相对简单一点的OpenFile类中的WriteBack方法，这里我们要把其内部的FileHeader类型的成员hdr写回到磁盘中，FileHeader内部实现了写回磁盘的方法，但是它需要扇区参数，而在这个OpenFile类中这个值并不好取得，因此我们再给OpenFile类增加一个私有成员sector，并在其构造函数中利用传入的头信息扇区号来初始化。如下：

```
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
    this->sector=sector;
}
```

这样，我们就能在WriteBack方法中拿到扇区号了。如下：

```
void
OpenFile::WriteBack()
{
    hdr->WriteBack(sector);
}
```

然后，我们就只剩下OpenFile类中的AllocateSpace与WriteAt函数和FileHeader类中的ExtendSpace函数需要完成了。

由于OpenFile类里面的两个函数的实现都需要使用到ExtendSpace函数的功能，因此这里首先来实现这个函数，这个函数模仿Allocate函数实现，需要做的就是根据要申请的空间大小和比特图来申请新的磁盘扇区空间。如下：

```
bool
FileHeader::ExtendSpace(BitMap *freeMap,int appendSize)
{
    int oriSectors=numSectors; //记录原空间大小
    numSectors  = divRoundUp(appendSize, SectorSize)+oriSectors; //计算新扇区大小
//    printf("newSectorsNum is %d\n", numSectors);
    if (freeMap->NumClear() < numSectors-oriSectors)
 {
  numSectors=oriSectors;//将空间大小复原
  return FALSE;  // not enough space
 }

    for (int i = oriSectors; i < numSectors; i++) //申请新的扇区
 dataSectors[i] = freeMap->Find();
    return TRUE;
}
```

在WriteAt函数中需要调用AllocateSpace函数，所以首先应该来实现AllocateSpace函数：
这个函数的功能应该是给它一个需要扩展的文件长度，它就能申请到这个长度对应的扇区来进行以后的存储，而具体的申请操作在OpenFile类的FileHeader类型的hdr成员已经实现了，这里可以直接调用它来做：

```
void
OpenFile::AllocateSpace(int size)
{
    BitMap *freeMap;
    freeMap=new BitMap(NumSectors); //新建一个BitMap对象
    OpenFile *freeMapFile;
    freeMapFile=new OpenFile(0); //新建一个比特图对应的OpenFile对象
    freeMap->FetchFrom(freeMapFile); //从磁盘中取出比特图的信息

    hdr->ExtendSpace(freeMap,size); //实际的扩展操作
    freeMap->WriteBack(freeMapFile); //写回比特图的信息
    delete freeMap;
}
```

完成这些之后，可以来实现WriteAt方法了，这个方法在实现的时候已经考虑到了不从头开始写的情况，因此我们也不需要太担心ap和hap命令的实现，在实现这个函数之前，我们应该先进行一些思考：

如果是cp模式，即直接复制UNIX文件的内容到nachos文件系统中去，则当前的代码已经不需要修改了。如果我们要实现ap或者hap模式，则需要申请新的空间，而剩余的部分也不需要进行修改，因此，我们首先需要判断一下是不是追加模式，如果是，则去申请新的空间。而判断是否为追加模式，我们可以用当前文件指针的位置是否大于文件大小来确定。如果是追加模式，我们则分两种情况讨论：

1.加上新的字符后的文件大小还没有占满整个扇区，这时只需要设置文件大小即可。

2.加上新的字符后文件大小已经超过了整个扇区，这时候就需要申请新的磁盘空间来对文件进行存储了。

此外再排除一些WriteAt本来的关于文件长度的一些限制之后，修改的WriteAt方法如下：

```
int
OpenFile::WriteAt(char *from, int numBytes, int position)
{
    int fileLength = hdr->FileLength();
    int i, firstSector, lastSector, numSectors;
    bool firstAligned, lastAligned;
    char *buf;
//    printf("seekPosition is %d,fileLength is %d\n",seekPosition,fileLength);
    if (numBytes < 0)
 return 0;    // check request
    if(seekPosition>=fileLength) //如果文件指针已经超过了文件大小
    {
        numSectors=divRoundUp(fileLength,SectorSize); //计算当前需要的扇区
 int numPos=seekPosition+numBytes; //计算添加新字节之后的指针位置
        if(numPos>numSectors*SectorSize) //如果文件为空或者超过了已有扇区空间
        {
            AllocateSpace(numPos-numSectors*SectorSize);//申请新空间
            fileLength+=numBytes; //增大文件空间
        }
        hdr->SetLength(numPos); //根据新指针位置设置新的文件大小
    }
    //写入的部分已经考虑了从中间写入的情况

    if(fileLength==0) return 0;
    DEBUG('f', "Writing %d bytes at %d, from file of length %d.\n",
   numBytes, position, fileLength);

    firstSector = divRoundDown(position, SectorSize);
    lastSector = divRoundDown(position + numBytes - 1, SectorSize);
    numSectors = 1 + lastSector - firstSector;
//    printf("firstSector is %d,lastSector is %d\n",firstSector,lastSector);
    buf = new char[numSectors * SectorSize];

    firstAligned = (bool)(position == (firstSector * SectorSize));
    lastAligned = (bool)((position + numBytes) == ((lastSector + 1) * SectorSize));

// read in first and last sector, if they are to be partially modified
    if (!firstAligned)
        ReadAt(buf, SectorSize, firstSector * SectorSize);
    if (!lastAligned && ((firstSector != lastSector) || firstAligned))
        ReadAt(&buf[(lastSector - firstSector) * SectorSize],
    SectorSize, lastSector * SectorSize);

// copy in the bytes we want to change
    bcopy(from, &buf[position - (firstSector * SectorSize)], numBytes);

// write modified sectors back
    for (i = firstSector; i <= lastSector; i++)
        synchDisk->WriteSector(hdr->ByteToSector(i * SectorSize),
     &buf[(i - firstSector) * SectorSize]);
    delete [] buf;
    return numBytes;
}
```

最后，我们则是按照实验指导书上的提示，在fstest.cc文件中取消掉调用WriteBack的部分的注释。

```
// Write the inode back to the disk, because we have changed it
  openFile->WriteBack();
  printf("inodes have been written back\n");
```

源码部分处理完成以后就要进行编译与测试了，这里实验要求自己编写Makefile文件，由于nachos的编译模块过于复杂，这里我先把filesys中的所有除fstest.cc之外的文件复制到lab5目录下，然后修改Makefile文件来实现的编译，两个相关的Makefile代码如下：

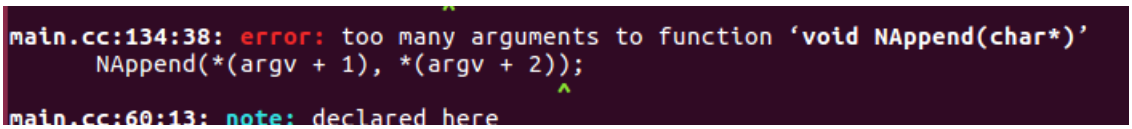
Makefile.local:

```
ifndef MAKEFILE_LAB5_LOCAL
define MAKEFILE_LAB5_LOCAL
yes
endef

# Add new sourcefiles here.

CCFILES +=bitmap.cc\
        directory.cc\
 filehdr.cc\
 filesys.cc\
 fstest.cc\
 openfile.cc\
 synchdisk.cc\
 disk.cc\
 main.cc

ifdef MAKEFILE_USERPROG_LOCAL
DEFINES := $(DEFINES:FILESYS_STUB=FILESYS)
else
INCPATH += -I../userprog -I../lab5
DEFINES += -DFILESYS_NEEDED -DFILESYS
endif

endif # MAKEFILE_FILESYS_LOCAL
```

Makefile:

```
ifndef MAKEFILE_LAB5
define MAKEFILE_LAB5
yes
endef

# You can re-order the assignments.  If filesys comes before userprog,
# just re-order and comment the includes below as appropriate.

include ../threads/Makefile.local
include ../lab5/Makefile.local
#include ../userprog/Makefile.local
#include ../vm/Makefile.local
#include ../filesys/Makefile.local

include ../Makefile.dep
include ../Makefile.common

endif # MAKEFILE_FILESYS
```

这里有一个源码中的坑，就是在测试要用的main.cc中，如果不改的话编译会报如下错误：



只需要进入lab5下的main. cc，把NAppend函数的声明修改一下即可：

```
extern void NAppend(char *from, char *to);
```

四、测试

测试可能需要若干命令，这里我们可以写一个Shell脚本来保存连续的命令，需要的时候调用然后观察最后结果即可，这里我准备了两个测试脚本，修改完源码并且编译之后执行即可：

测试ap参数：

```
rm DISK
./nachos -f
./nachos -cp test/small small
./nachos -ap test/small small
./nachos -cp test/empty empty
./nachos -ap test/medium empty
./nachos -D
```

测试hap参数：

```
rm DISK
./nachos -f
./nachos -cp test/small small
./nachos -cp test/big big
./nachos -hap test/big small
./nachos -hap test/small big
./nachos -D
```

将上述两个脚本分别另存为ap. sh和hap. sh，然后在命令行中执行如下命令给予其执行权限：

```
$chmod a+x ap.sh
$chmod a+x hap.sh
```

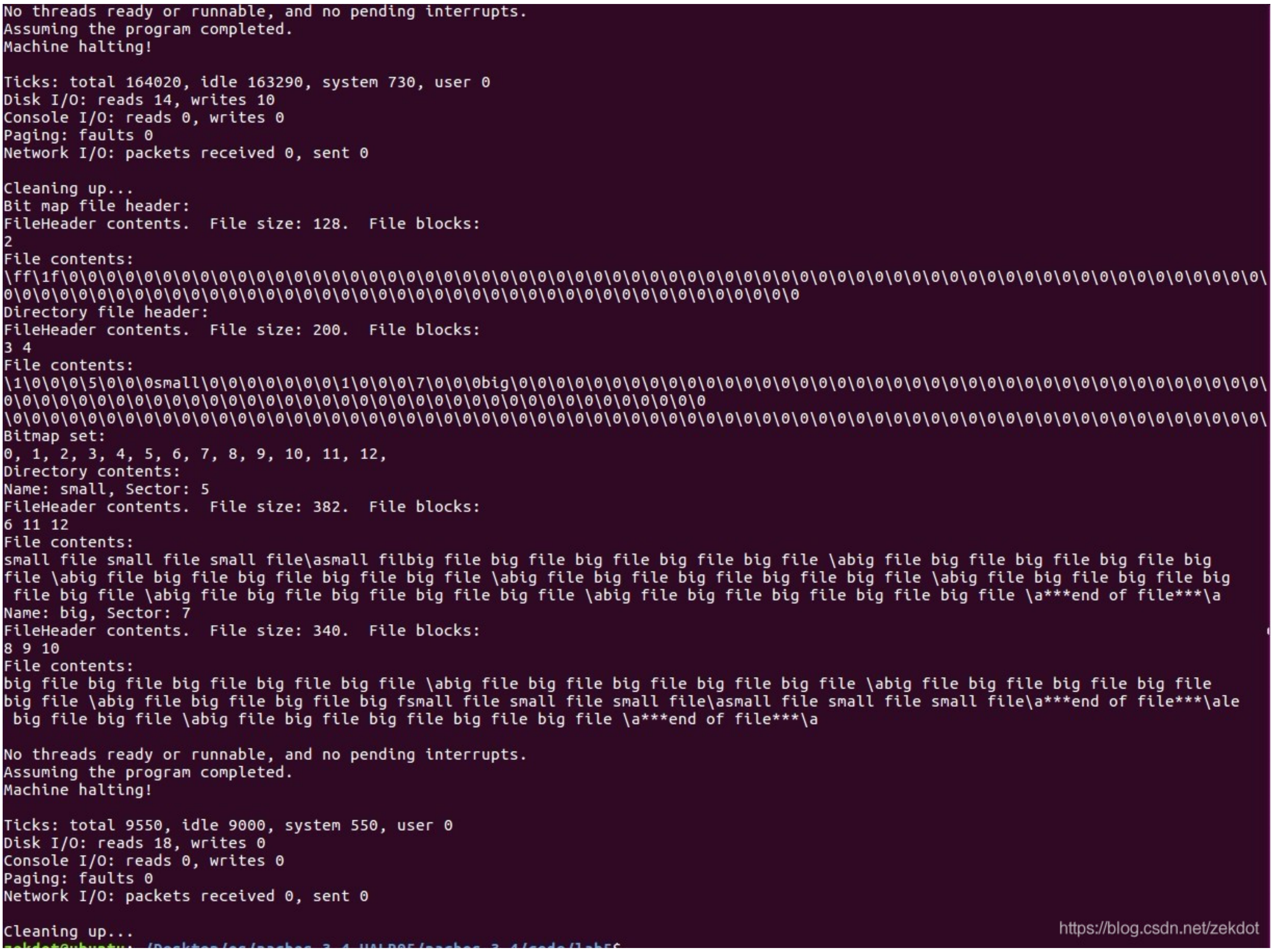
然后首先我们测试ap参数的运行情况，运行结果如下：

```
Cleaning up...
inodes have been written back
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 308520, idle 307070, system 1450, user 0
Disk I/O: reads 27, writes 21
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Bit map file header:
FileHeader contents.  File size: 128.  File blocks:
2
File contents:
\ff\7\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
Directory file header:
FileHeader contents.  File size: 200.  File blocks:
3 4
File contents:
\1\0\0\0\5\0\0\0small\0\0\0\0\0\0\0\1\0\0\0\8\0\0\0empty\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,
Directory contents:
Name: small, Sector: 5
FileHeader contents.  File size: 168.  File blocks:
6 7
File contents:
small file small file small file\asmall file small file small file\a***end of file***\asmall file small file small fi
small file small file\a***end of file***\a
Name: empty, Sector: 8
FileHeader contents.  File size: 162.  File blocks:
9 10
File contents:
medium file medium file medium file\amedium file medium file medium file\amedium file medium file medium file\amedium
ile medium file\a***end of file***\a

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

Ticks: total 8490, idle 8000, system 490, user 0
Disk I/O: reads 16, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
zekdot@ubuntu:~/Desktop/os/nachos-3.4-UALR05/nachos-3.4/code/lab5$
```


可见，与实验指导书预计的结果一致，即ap功能得以实现。

接下来再来测试一下hap命令。

这里测试了两种情况，即从大文件的中间写小文件和从小文件的中间写大文件。结果如下：运行结果如下：



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

Ticks: total 164020, idle 163290, system 730, user 0
Disk I/O: reads 14, writes 10
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Bit map file header:
FileHeader contents.  File size: 128.  File blocks:
2
File contents:
\ff\1f\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
Directory file header:
FileHeader contents.  File size: 200.  File blocks:
3 4
File contents:
\1\0\0\0\5\0\0\0small\0\0\0\0\0\0\0\1\0\0\0\7\0\0\0big\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,
Directory contents:
Name: small, Sector: 5
FileHeader contents.  File size: 382.  File blocks:
6 11 12
File contents:
small file small file small file\asmall filbig file big file big file big file big file \abig file big file big file big file big
file \abig file big file big file big file big file \abig file big file big file big file big file \abig file big file big file big
 file big file \abig file big file big file big file big file \abig file big file big file big file big file \a***end of file***\a
Name: big, Sector: 7
FileHeader contents.  File size: 340.  File blocks:
8 9 10
File contents:
big file big file big file big file big file \abig file big file big file big file big file \abig file big file big file big file
big file \abig file big file big file big fsmall file small file small file\asmall file small file small file\a***end of file***\ale
 big file big file \abig file big file big file big file big file \a***end of file***\a

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

Ticks: total 9550, idle 9000, system 550, user 0
Disk I/O: reads 18, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```


从结果来看，hap的功能也得到了实现。

五、结论分析与体会

从输出结果来看，拓展之后的文件系统在保证了基本的cp命令执行不出错的情况下，增加了ap和hap两个命令，让原来固定大小的文件系统变成了大小可变的文件系统。

本次实验在掌握了nachos基本文件命令用法的情况下，对其功能进行了拓展，增加了文件可变的功能，而实现这个功能的前提也是足够了解nachos磁盘系统相关的源代码，因此通过扩展这个功能，我对nachos文件系统相关的部分有了更深的认识，也对操作系统中所讲的文件系统相关的内容进行了很好的回顾。