|  |  |
| --- | --- |
| 实验编号：Lab4 | |
| 实验题目：扩展文件系统 | |
| 实验学时：4 | 实验日期：2022.12.4 |
| 实验目的：  1. 扩展Nachos的基本文件系统。Nachos的文件系统是一个简单并且能力有限的系统，限制之一就是文件的大小是不可扩展的。通过扩展，使得文件的大小是可变的。在扩展写入文件内容时，一边写入，一边动态调整文件的长度及所占用的数据扇区。  2. 增加Nachos文件的最后修改时间，并在执行./nachos -D命令时显示。Nachos文件头中存储文件最后修改时间，时间值是从UTC 1970年1月1日00:00:00来的秒数(精确到1秒)，占用原来numSectors的存储位置(从磁盘存储空间效率上考虑，文件头中已经有了文件长度字节数，无需再存储文件内容占用的扇区数)。  注1：仅普通文件的文件头最后修改时间字段有意义，并在执行./nachos -D命令时显示其时间。对其他文件头对象，在执行./nachos -D命令时不显示时间即可。  注2：若Lab4全部完成，演示提交的代码为带有文件最后修改时间的。  注3：对一般的OS，一个100字节的文件，open后lseek到偏移50处，write 10字节，close后，文件长度还是100字节，不会截短到60字节。这在实现Nachos的-hap命令行选项时需要注意。 | |
| 硬件环境：  DELL品牌DESKTOP-3CCOK51型号笔记本  Intel Core i5-10210U CPU  8GB内存  1T SSD | |
| 软件环境：  宿主机：Windows 10 21H2 64位  虚拟机软件：VMware Workstation Pro 16.1.2 build-17966106  Linux：Ubuntu 14.04.6 LTS Desktop i386 (Trusty Tahr)  gcc/g++：(Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4  MIPS交叉编译器：gcc-2.8.1-mips.tar.gz  Nachos：Nachos-3.4-UALR-2022 | |
| 实验步骤与内容：  进行扩展Nachos的基本文件系统的修改前，先要了解Nachos文件系统的组织结构：  Nachos文件系统中，使用一个名为Disk的文件模拟磁盘空间，synchdisk用于控制磁盘的“写锁”，保证同步硬件。main和fstest用于封装方法供命令行调用。filesys用于控制文件系统，directory用于进行文件目录的操作。openfile用于对文件进行读写等操作，filehdr用于维护文件头，即文件所占用的扇区、文件大小和文件的最后修改时间等信息，也是本次实验的重点。  接着认识为什么原先Nachos的文件是无法扩展的：  在nachos系统中新建一个文件时，由文件系统filesys调用create方法，这一步传入了文件的名字和大小，directory依据这两个参数在文件目录中新增这一项，通过磁盘位图找到一个空闲扇区作为文件的头，并依据文件大小分配合适数量的扇区写入文件，再更新文件的头部和磁盘的位图。在这个过程中，文件的大小是在创建时定的，在header，allocate时候，传入参数fileSize，按fileSize分配恰好合适大小的扇区数量。  而在进行文件修改时，当写的内容超出一开始创建时的fileSize时便会切掉超出的部分  只能写入前fileSize个字符。所以一个文件的大小从创建开始是不会改变的。  其次考虑什么情况下需要对文件大小进行扩展：主要是在修改文件保存时，因此需要在将文件写回到硬盘上之前判断文件大小的变化，以及是否需要分配新的扇区。若需要分配新的扇区则需要找到空闲扇区并填入文件头部的索引表中，同时将磁盘用于记录各个扇区空闲状态的位图进行更新，即为directory.cc文件中新增的extend方法：   1. **bool** 2. FileHeader::Extend(**int** newNumBytes){//新增扩展方法 3. **if**(newNumBytes<numBytes)**return** **false**;//wrong param 4. **if**(newNumBytes==numBytes)**return** **true**;//无需扩展 5. **int** newNumSectors=ceil((**double**)newNumBytes/(**double**)SectorSize);//ceil函数：取上整 6. **if**(newNumSectors==numSectors()){//same num of sectors 7. numBytes=newNumBytes; 8. **return** **true**; 9. } 10. **int** deltaSectors=newNumSectors-numSectors(); 11. OpenFile \*openFile=**new** OpenFile(0); 12. BitMap \*bitMap=**new** BitMap(NumSectors);//NumSectors:常数 磁盘上的所有扇区数量 13. bitMap->FetchFrom(openFile); 14. //剩余扇区数量不足以满足文件的存储 15. **if**(newNumSectors>NumDirect||deltaSectors>bitMap->NumClear()){ 16. printf("disk is full/ file is too big\n"); 17. printf("old size:%dB--new size:%dB\n",numBytes,newNumBytes); 18. printf("new sectors:%d   delta:%d   direct:%d   clear:%d\n",newNumSectors,deltaSectors,NumDirect,bitMap->NumClear()); 19. bitMap->Print(); 20. **return** **false**; 21. } 22. //allocate 23. **for**(**int** i=numSectors();i<newNumSectors;i++)dataSectors[i]=bitMap->Find(); 24. //dataSectors[i]该文件第i个扇区的扇区号 25. //bitMap->Find()  找到一个可用扇区 26. bitMap->WriteBack(openFile);//更新DISK的位图 27. numBytes=newNumBytes; 28. **return** **true**; 29. }   在extend方法中，首先判断原先的字节数numBytes是否小于新字节数newNumBytes，如果小于则说明需要进行扩展，于是计算修改后文件需要的扇区数newNumSectors，并和原先的占用扇区数numSectors进行比较，若需要新增扇区则通过for循环在位图中找到可用扇区进行分配，从而逐个新增直到给文件分配了对应数量的扇区并将新分配的扇区块号写入文件头的扇区数组dataSectors中，之后再将磁盘的位图进行更新、将文件修改后的字节数newNumBytes赋值给文件头的numBytes属性。此外，剩余扇区数量不足以满足文件的存储，则打印相关信息并返回代表扩展失败的“false”。  将openfile.cc的writeAt方法修改为可扩展大小的：   1. **if** ((numBytes <= 0) || (position > fileLength))  //修改：可以从文件结尾处（EOF）开始写 2. return 0;              // check request 3. **if** ((position + numBytes) > fileLength) 4. **if**(!hdr->Extend(position+numBytes))**return** 0;//扩展没有成功   在writeAt方法中加上上面的程序语句，通过检查当前位置加上需要增加的字节数是否超出了文件长度来判断是否扩展大小，若扩展则调用Extend方法，并返回代表了是否扩展成功的布尔值。  通过搜索，得知了获取linux文件系统中文件最后修改时间的方法，结合sys/stat.h和time.h，在头文件中包含它们后即可实现：   1. struct stat buf; 2. stat(from,&buf); 3. （此时，(int)buf.st\_mtime就是最后修改时间了）   而time(NULL)就是系统当前的时间了。  在fstest.cc中的追加方法Append/NAppend等函数中，增加了与时间有关的逻辑：  1、文件不存在时，创建新文件并将最后修改时间设置为和源文件相同；  2、文件存在时，将目标文件最后修改时间改为当前时间。   1. **if** ( (openFile = fileSystem->Open(to)) == NULL) 2. { 3. // file "to" does not exits, then create one 4. **if** (!fileSystem->Create(to, 0)) 5. { 6. printf("Append: couldn't create the file %s to append\n", to); 7. fclose(fp); 8. **return**; 9. } 10. openFile = fileSystem->Open(to); 11. //new file, modified time should be the same as original 12. **struct** stat buf; 13. stat(from,&buf); 14. openFile->SetModifiedTime((**int**)buf.st\_mtime); 15. }**else** openFile->SetModifiedTime((**int**)time(NULL));//existing file, change time   在fstest.cc中的复制方法Copy中增加了将最后修改时间设置为与原文件相同的逻辑：   1. **struct** stat buf; 2. stat(from,&buf); 3. openFile->SetModifiedTime((**int**)buf.st\_mtime);   扩展要求是记录下文件的最后修改时间。这部分的存储逻辑我们选择了方案a，即在openfile.cc中增加WriteBack方法，并在SetModifiedTime手动调用writeBack方法即可。此外，在对文件做出修改时也可调用此方法更新文件头。   1. **void** 2. OpenFile::WriteBack(){//新增写回文件头方法 3. hdr->WriteBack(sector); 4. } 6. **void** 7. OpenFile::SetModifiedTime(**int** modifiedTime){ 8. hdr->SetModifiedTime(modifiedTime); 9. WriteBack(); 10. }   由于实验要求中说仅普通文件的文件头最后修改时间字段有意义，并在执行./nachos -D命令时显示其时间。对其他文件头对象，在执行./nachos -D命令时不显示时间，因此在filehdr.cc的print方法中，打印最后修改时间之前先对时间进行是否为0的判断。位图和磁盘目录的最后修改时间都被置为0，因而可以用这种方式判断是否为普通文件。   1. **if**(modifiedTime){//仅普通文件显示最后修改时间系统文件的最后修改时间设置为0（位图、目录） 2. **char** s[100]; 3. **time\_t** tmpTime=(**time\_t**)modifiedTime; 4. strftime(s, **sizeof**(s), "%Y-%m-%d %H:%M:%S", &\*localtime(&tmpTime)); 5. printf("\nLast modified time:%s", s); 6. }   Readme中指令运行结果：  rm -f DISK  ./nachos -f  ./nachos -cp test/big big    ./nachos -D      ls –full-time test/big  ./nachos -ap test/small big  ./nachos -D      ./nachos -hap test/small big  ./nachos -D      ./nachos -ap test/small small  ./nachos -nap small small2    ./nachos -D      ./nachos -nap small small2  ./nachos -D | |
| 结论分析与体会：  此次课设使我收获了许多。在一开始进行实验时感到无从下手，后来通过阅读nachos源码以及老师课堂上所讲解的代码思路，我逐步看懂了原先的管理流程，并在理解的基础上进行了实验要求的功能扩展。在理解了源码后，结合实验要求也很自然地想出来了解决方法并上手调试，在真正看懂了源码之后实验其实并不难。  我也体会到了nachos系统完整详细的注释和优良的扩展性。整体架构设计明晰且易于修改，比如本次实验中增加了extend方法来对文件大小进行扩展，通过代码封装进行调用，没有对源程序做出太多的改动，易于调试。  此外，这个过程促使我对操作系统中文件存储管理等操作的具体实现进行复习，而经过了实验中的多次尝试又加深了理解。这也启示我在今后的学习中也要注重理论实践相互结合，达到对知识的掌握和运用。  最后我要感谢为我提供技术支持的组员，在我看不懂源码时组员很认真地为我讲述他的理解，并和我讨论思路、一起查资料共同学习。调试过程中linux时常出现段错误的返回，多亏了组员们的支持鼓励让我最终坚持尝试并完成了实验要求。 | |