山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300063 | 姓名：李彦浩 | | 班级：工软 21 |
| 学号：202100300340 | 姓名：黄幸兒 | | 班级：工软 21 |
| 实验编号：Lab5 | | | |
| 实验题目：具有二级索引的文件系统 | | | |
| 实验学时：4 | | 实验日期：2023.11.10 | |
| 实验目的：  1. Nachos系统原有的文件系统只支持单级索引，最大能存取30 \* 128 = 3840字节大小的文件。本实验将在理解原文件系统的组织结构基础上扩展原有的文件系统，在Lab4的基础上，设计并实现具有二级索引的文件系统。  2. 为Nachos增加命令行选项-DI。执行./nachos -DI时显示Nachos磁盘的以下信息：Nachos磁盘的总体大小，已使用空间大小，空闲空间大小，普通文件数目，全部普通文件的总字节数，全部普通文件占用的空间大小(不包括文件头占用的，但加上普通文件数据扇区的内碎片)，总内碎片字节数(仅计普通文件数据扇区造成的)。  3. 若要求为Nachos文件增加rwx权限(可读，可写，可执行)，请给出在Nachos中实现的具体方法(不要求实现可运行的代码。在实验报告中用文字描述即可，必要时可在文字中结合关键代码片段、数据结构、对象等说明)。 | | | |
| 硬件环境：  联想Thinkpad笔记本  Intel Core i5-8250 CPU 8核  8GB内存  1907GB 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： 目录 [1. 扩展文件系统 2](#_Toc19359)  [1.1. 多级索引结构 2](#_Toc10526)  [1.1.1. 回顾：Nachos原有文件存储结构 2](#_Toc13101)  [1.1.2. 前瞻：多级索引结构 3](#_Toc2507)  [1.2. 任意级索引与链式存储 4](#_Toc21056)  [1.2.1. 为什么不只是二级索引 4](#_Toc21474)  [1.2.2. 任意级索引的实现思路 4](#_Toc15909)  [1.3. DFS算法分配空闲磁盘块 5](#_Toc29010)  [1.3.1. 文件初始化 5](#_Toc926)  [1.3.2. 变长文件的扇区分配 11](#_Toc1074)  [1.4. DFS算法回收磁盘块 14](#_Toc27448)  [1.5. 无限大的Nachos文件 15](#_Toc3666)  [2. 统计选项-DI 16](#_Toc9524)  [3. Nachos文件的rwx权限 19](#_Toc1518)  [3.1. 什么是ACL？ 19](#_Toc16854)  [3.2. 数据结构的更新 20](#_Toc18760)  [3.2.1. ACL的数据结构 20](#_Toc16669)  [3.2.2. ACL作为目录表项的成员字段 20](#_Toc12300)  [3.3. 为文件注册ACL 21](#_Toc7462)  [3.4. 用户与鉴权 21](#_Toc12435)  [3.4.1. 用户类 21](#_Toc3279)  [3.4.2. 用户鉴权表 22](#_Toc133)  [3.4.3. 用户注册 24](#_Toc26709)  [3.5. 用户访问文件 25](#_Toc4984)  [3.5.1. 用户权限验证 26](#_Toc10245)  [3.5.2. 文件权限验证 26](#_Toc12143) 扩展文件系统 Nachos原有的文件头数据结构支持单文件可以占据30\*128=3840B，本实验最基本的要求是支持二级索引，单文件可以占据(29+30)\*128 = 7552B。但我不满足于此，最终实现了理论上可以无限大的Nachos文件，当然，任意集索引自然是二级索引的超集了，因此基本要求我肯定是做到了的，实现任意级索引只是想让Nachos文件系统的适用性更强。 1.1. 多级索引结构1.1.1. 回顾：Nachos原有文件存储结构 之前在Lab4中介绍过Nachos原有的文件存储结构，如下图：  Untitled  每个文件头占一个扇区大小（源码中规定为128字节），其中8个字节分别为文件长度与时间戳（各占4个字节）。而剩下120个字节对应了一个数组，标识了该文件占据的磁盘块。120个字节最多存放120B/4B = 30个整形数，每一个整形数可以标识一个磁盘上的扇区，则单文件最大可以有30\*128=3840字节。显然这和现代操作系统相去甚远，这里我们要做的就是扩展Nachos文件结构。 1.1.2. 前瞻：多级索引结构 下面放一张多级索引的示意图（图来自ZHL的指导书）  Untitled 1  思路简单来说就是把上述的占据磁盘块的数组中抽4个字节出来，用来表示下一个文件头的磁盘扇区号。这样一个文件就有多个文件头了，假设一个文件头中的数组可以标识x个磁盘扇区号，那么y个文件头就可以总计标识xy个磁盘扇区号。如果一个文件头放不下要存储的所有字节，那么就附上下一个文件头，直到可以存放所有字节为止。 1.2. 任意级索引与链式存储1.2.1. 为什么不只是二级索引 之前提到过，做任意级索引是为了让Nachos的文件系统适用性更强，一个文件的大小上限应该以物理极限为准，而不是以逻辑极限为准。也就是说应该依赖于磁盘硬件的大小，而不是程序员自己规定的大小。前者是现实实际的限制，短时间内可能无法改变；但后者是程序员自己编码设计出现了问题。因此我认为文件系统不能仅仅限制在二级索引上，而应该扩展到任意级索引。 1.2.2. 任意级索引的实现思路 实际上ZHL的那张图已经展示了任意级索引的实现思路，这里我画一张更贴合于我的实现的图，可辅以源码查看，理解起来更加清晰。  Untitled 2  本质上其实是个链表，只不过下一个文件头的指针并不是一个实实在在的FileHeader的实例的指针（这一点上，我和实验指导书的实现方式不同），而是一个逻辑上的指针，指针的内容是下一个文件头的磁盘扇区号。最后一个文件头的indirect字段为-1，这是我预先设定好的值，用来表示不合法的多级索引，因为众所周知扇区号没有-1号。当我们想访问一个文件，只需要拿到这个文件的第一个文件头即可。对于剩下的文件头，可以通过indirect字段来读取磁盘上对应的扇区号，拿到下一级文件头的内容，直至indirect字段的值为-1为止。  而在我的实现中，还有一个重要的规定：只有根文件头的文件长度表示整个文件的字节数，之后的文件头中的长度字段仅表示当前文件头所覆盖的字节数！这样做一是为了编码方便，二是这样的约定在后续的统计功能中为我们提供了很大的方便！ 1.3. DFS算法分配空闲磁盘块 Lab5扩展的文件系统并不会影响Lab4中已有的功能，也就是说文件支持变长。即便用户一开始指定了文件的大小，后续他依然有能力修正该文件的大小（比如向其中附加数据）。所以这里分为两部分。第一部分是如何初始化一个指定大小的文件（为该文件分配扇区）；第二部分是如何向一个文件附加数据（重新分配新增扇区）。 1.3.1. 文件初始化 对于这个问题，我们可能无法像Lab4中一样，直接计算出所需的扇区数量，然后直接为dataSectors字段中分配bit map返回的空闲扇区号的值。因为可能用户一开始为这个文件指定的大小已经超出了一个文件头能够控制的范围。比如在任意级索引中，一个文件头可以控制29个扇区，对应3712字节。如果用户一开始指定的大小是5000字节，那么一个文件头就不够了。因此我们要想办法链式存储（就像刚才1.2节中的一样）存储所有的文件头。  链表本质上也是一棵有向树，只不过长得比较简单而已，碰到这种每一级都需要分配一个文件头，然后为当前文件头分配扇区的操作，很容易想到DFS，因此我对于Allocate函数的实现就是一个基于递归的DFS，且每个节点都是1叉的。  其实我们也可以通过迭代来构造一个链式存储，只是这里我选用了上述的方式。  先看一下如何递归地分配每一级文件头：   |  | | --- | | /\*\*  \* recursive allocate the dataSectors  \* @return the next file header's sector number,null if -1  \*/  int  FileHeader::AllocateEachFHdr(BitMap \*bitMap,int startNo,int restSectors,int curSectorNo) {  if(restSectors<=0){  // nothing left to allocate  // there'll be no more external file headers  return -1;  }  //DEBUG('f',"Allocating the original file header\n");  int leftSize = NumDirect - startNo;  // the real number of sectors to be allocated this time  int allocatedNum = restSectors>leftSize?leftSize:restSectors;  // if there's no rest sector to be allocated,then there'll be no more external file headers  int nextFHdrSectorNo = (restSectors-allocatedNum)>0?bitMap->Find():IllegalIndirectSectorNo;  DEBUG('f',"There'll be %d rest sectors after this allocation\n",restSectors-allocatedNum);  if(nextFHdrSectorNo!=IllegalIndirectSectorNo){  DEBUG('f',"Allocate sector %d as an external file header for file header on %d\n",  nextFHdrSectorNo,curSectorNo);  }  FileHeader\* fHdr = new FileHeader;  // fetch the file on the disk  // in case we change the part that has already been allocated  fHdr->FetchFrom(curSectorNo);  fHdr->setLastUpdateTime((int)time(NULL));  fHdr->setFileLength((startNo+allocatedNum)\*SectorSize);  //allocate the data sectors  fHdr->setDataSectors(bitMap,startNo,allocatedNum);  // set indirect field  fHdr->setIndirect(nextFHdrSectorNo);  fHdr->WriteBack(curSectorNo);  DEBUG('f',"Allocate fHdr:Write file header %d back to disk\n",curSectorNo);  delete fHdr;  // recursively allocate the next file header  // ought to start at 0!  AllocateEachFHdr(bitMap,0,restSectors-allocatedNum,nextFHdrSectorNo);  return nextFHdrSectorNo; |   这个递归函数的返回值是下一级文件头的扇区号，如果没有下一级文件头则返回-1。这个有什么用之后在递归入口中介绍。共计四个参数：   1. bitMap：位图的指针，在分配文件头的过程中我们需要查看位图。而位图在这个系统中是单例的，所以不需要每层递归都初始化一个位图再把位图从磁盘拉上来。浪费时间空间。 2. startNo：开始分配数据扇区的位置，最小是0，最大29（也即一个文件头能控制的最大的扇区数），这是为了在1.3.2节中复用这个函数。 3. restSectors：剩下的未分配的扇区数量 4. curSectorNo：当前文件头的扇区号   把DEBUG信息去掉，这个函数的流程如下：   1. 递归基：如果没有任何剩余的未分配的扇区，代表本次分配结束  |  | | --- | | if(restSectors<=0){  // nothing left to allocate  // there'll be no more external file headers  return -1;  } |  1. 计算当前文件头还能容纳多少个扇区，计算方式是用最大扇区数减去当前已有的扇区数，也就是开始分配数据扇区的位置。比如startNo是10，代表已经分配了0-9这10个扇区，那么这个文件头还剩下10-29这（29-10）=19个扇区未分配。  |  | | --- | | int leftSize = NumDirect - startNo; |  1. 计算本次将要分配的扇区数。为什么要算这个呢？是因为剩下的未分配的扇区数可能还没有当前这个文件头剩余的空闲扇区数大。比如当前文件头有20个空闲扇区数，但实际要分配的只有3个，那么我们将只分配这三个。  |  | | --- | | // the real number of sectors to be allocated this time  int allocatedNum = restSectors>leftSize?leftSize:restSectors; |  1. 计算是否还需要下一个文件头。怎么计算呢？通过从剩余未分配的扇区数中减去本次要分配的扇区数，如果为0，证明不需要下一个文件头了；否则我们需要从位图中找一个空闲扇区号分配给下一个文件头。  |  | | --- | | int nextFHdrSectorNo = (restSectors-allocatedNum)>0?bitMap->Find():IllegalIndirectSectorNo; |  1. 创建当前一级的文件头。首先我们可能是更新这个文件头而非新建这个文件头。比如说一个文件可能已经有2个文件头了，第二个只使用了3个扇区，那么下一次我们在这个文件中新增内容时就要把数据放到第二个文件头的空闲扇区中。因此我们要从磁盘中先把这个文件头拉出来，然后更新它，而不是完全新建一个文件头。当然，也存在新建一个文件头的情况，不过我的实现兼容了所有的应用场景。  |  | | --- | | FileHeader\* fHdr = new FileHeader;  // fetch the file on the disk  // in case we change the part that has already been allocated  fHdr->FetchFrom(curSectorNo); |  1. 设置文件头修改时间，这个是为了兼容Lab4的，不再赘述。  |  | | --- | | fHdr->setLastUpdateTime((int)time(NULL)); |  1. 设置文件头长度。这里可能导致一个问题，后面再讲。  |  | | --- | | fHdr->setFileLength((startNo+allocatedNum)\*SectorSize); |  1. 分配数据扇区，从指定的开始位置分配。还是那件事：我们可能在更新一个文件头而不是新建一个文件头！  |  | | --- | | //allocate the data sectors  fHdr->setDataSectors(bitMap,startNo,allocatedNum); |  1. 设置下一个文件头的扇区号。这里可能导致一个问题，后面再讲。  |  | | --- | | // set indirect field  fHdr->setIndirect(nextFHdrSectorNo); |  1. 将该文件头写回。这一点和之前的Lab4以及示例源码均不同，之前我们是不写回的。但为什么这里写回呢？注意，我们在内存中更新/新建了一个文件头，但此时还没有持久化到磁盘。而之前是因为一个文件只有一个文件头，也就是调用Alloacte函数的文件头对象。但现在我们有多级文件头，如果此时不写回，我们将无法在内存中找到这多级文件头，从而永久地丢失了这些信息。  |  | | --- | | fHdr->WriteBack(curSectorNo); |  1. 递归地分配下一级文件头，把这一级计算出来的下一级文件头的扇区号传递给它。  |  | | --- | | AllocateEachFHdr(bitMap,0,restSectors-allocatedNum,nextFHdrSectorNo); |  1. 返回下一级文件头的扇区号。  |  | | --- | | return nextFHdrSectorNo; |   这一部分还是没什么难度的，接下来看递归入口：   |  | | --- | | bool  FileHeader::Allocate(BitMap \*freeMap, int fileSize, int secNo)  {  /\*\*  \* Lab4:filesys extension  \* when the file is creating,we should assign its create time  \*/  lastUpdateTime = (int)time(NULL);  int numSectors = divRoundUp(fileSize, SectorSize);  /\*\*  \* Lab5:filesys extension  \* check if there are enough space for the data and the external file headers  \* except for the original one  \*/  int totalSectors = numSectors + divRoundUp(numSectors,NumDirect);  // ought to be greater than -1!  totalSectors = (totalSectors-1)>=0?(totalSectors-1):0;  if (freeMap->NumClear() < totalSectors)  return FALSE; // not enough space  // recursively allocate dataSectors  int nextSecNo = AllocateEachFHdr(freeMap,0,numSectors,secNo);  // prevent flushing the disk  FetchFrom(secNo);  // in case the indirect has not been written to the disk  //WriteBack(secNo);  numBytes = fileSize;  indirect = nextSecNo;  //DEBUG('f',"updating the last file header's length\n");  updateLastFileHeaderLength(secNo);  return TRUE;  } |  1. 更新文件头的修改时间，用来兼容Lab4。 2. 先计算数据扇区的数量，再根据数据扇区数计算总共需要多少个扇区（含文件头的扇区数）。比如数据扇区总计要占据59个，那么文件头就需要ceil(59/29)=3个文件头。但又由于我们已经为当前这个文件头分配了扇区，所以减去1。如果位图上的剩余空闲扇区数不够，返回FALSE，分配不成功。  |  | | --- | | int numSectors = divRoundUp(fileSize, SectorSize);  /\*\*  \* Lab5:filesys extension  \* check if there are enough space for the data and the external file headers  \* except for the original one  \*/  int totalSectors = numSectors + divRoundUp(numSectors,NumDirect);  // ought to be greater than -1!  totalSectors = (totalSectors-1)>=0?(totalSectors-1):0;  if (freeMap->NumClear() < totalSectors)  return FALSE; // not enough space |  1. 递归地分配文件头，接收首个文件头的下一个文件头所在磁盘扇区号。  |  | | --- | | // recursively allocate dataSectors  int nextSecNo = AllocateEachFHdr(freeMap,0,numSectors,secNo); |  1. 注意，我们在递归函数中把文件头写入磁盘了，但是没有更新当前内存中的这首个文件头，所以我们再从磁盘把首个文件头的内容读回来：  |  | | --- | | // prevent flushing the disk  FetchFrom(secNo); |  1. 然后是一个重点补丁！现在假想一下用户如果一开始指定的文件大小是0B会怎么样？那么进入递归函数中将经过递归基的判断直接返回！不会写入当前根文件头的下一个文件头的索引以及文件大小！而4中又把这个错误的写入结果读到内存中，那么就造成内存中的数据错误。之后如果写入磁盘，则污染了磁盘空间。所以我们这里要热更新文件大小和下一个文件头的索引这两项数据，确保数据正确。  |  | | --- | | numBytes = fileSize;  indirect = nextSecNo; |  1. 之后还是一个重要补丁！现在假想一下当我们的文件存在内碎片时会发生什么。假设一个文件是200字节，而一个扇区128字节，那么该文件占据两个扇区。但是在递归函数中，我们把文件的大小设置为了两个扇区的大小，也即256字节。因此这里要热更新文件的实际大小。这里注意，对于那些没有空闲扇区的文件头，该文件头覆盖的长度一定是29\*128=3712字节，所以这些文件头是不用管的，只需要热更新最后一个文件覆盖的长度即可。这里热更新的逻辑也不复杂，首先先找到最后一个文件头，通过从磁盘读取当前文件头中的下一个文件头所在的磁盘扇区即可，直到最后一个文件头。然后热更新它的长度，写回磁盘就可以了。  |  | | --- | | updateLastFileHeaderLength(secNo);  int  FileHeader::updateLastFileHeaderLength(int secNo) {  FileHeader\* fHdr = new FileHeader;  copyFHdr(fHdr);  //DEBUG('f',"fHdr.fl= % d,fl = %d\n",fHdr->FileLength(),numBytes);  int fl = fHdr->FileLength();  int lastSecNo = secNo;  // get the last file header  //DEBUG('f',"fl :%d,max: %d\n",fl,SectorSize\*NumDirect);  while(fl>SectorSize\*NumDirect){  fl -= SectorSize\*NumDirect;  //DEBUG('f',"fl :%d,max: %d\n",fl,SectorSize\*NumDirect);  ASSERT(fHdr->getIndirect()!=IllegalIndirectSectorNo);  lastSecNo = fHdr->getIndirect();  fHdr->FetchFrom(fHdr->getIndirect());  }  fHdr->setFileLength(fl);  fHdr->WriteBack(lastSecNo);  //DEBUG('f',"update last file header on sector %d,file length %d\n",lastSecNo,fl);  delete fHdr;  return fl;  } |   至此为止我们实现了一个文件初始化时分配扇区的部分。某些地方的实现可能不够优雅，比如那两处热更新补丁。不过考虑到重构的时间成本问题，我决定继续使用当前这个解决方案。 1.3.2. 变长文件的扇区分配 在Lab4中我们实现了文件的变长功能。Lab5中我们将保留这个功能。我的做法是修改一下更新文件头所控制的数据扇区的函数的逻辑：   |  | | --- | | void  FileHeader::updateFileLength(int newFileLength,int secNo) {  DEBUG('d',"update file length,new file length:%d\n",newFileLength);  //calculate current used sectors  int curSectors = divRoundUp(newFileLength,SectorSize);  int oldSectors = divRoundUp(numBytes,SectorSize);  if(curSectors>oldSectors){  int add = curSectors-oldSectors;  DEBUG('d',"allocate new sectors for current file header\n");  // find the last file header  FileHeader\* fHdr = new FileHeader;  int lastFHdrSecNo = secNo;  fHdr->FetchFrom(secNo);  DEBUG('f',"Next file header index: %d\n",fHdr->getIndirect());  while(fHdr->getIndirect()!=IllegalIndirectSectorNo){  lastFHdrSecNo = fHdr->getIndirect();  fHdr->FetchFrom(fHdr->getIndirect()); // 5  DEBUG('f',"Next file header index: %d\n",fHdr->getIndirect());  curSectors -= NumDirect;  }  //DEBUG('f',"last file header's length: %d, needed sectors: %d\n",fHdr->FileLength(),fHdr->calculateNumSectors());  // where to start append  int startNo = fHdr->calculateNumSectors();  int bitmapSecNo = 0;  // fetch the bit map from disk  OpenFile\* bitmap = new OpenFile(bitmapSecNo); // 0  // load the bitmap to mem  BitMap\* bm = new BitMap(NumSectors);  bm->FetchFrom(bitmap); // 2  // for the last file header,make changes to the bit map  DEBUG('f',"Ready to allocate sectors for last file header on sector: %d\n",lastFHdrSecNo);  AllocateEachFHdr(bm,startNo,add,lastFHdrSecNo);  FetchFrom(secNo);  bm->WriteBack(bitmap); // 2  delete bm;  delete bitmap;  delete fHdr;  }  this->numBytes = newFileLength;  updateLastFileHeaderLength(secNo);  } |   流程如下：   1. 首先先查看是否需要分配新的数据扇区，可能上一次分配的某数据扇区中的内碎片还没有使用完，如果没使用完就继续使用。简单更新文件大小即可。  |  | | --- | | int curSectors = divRoundUp(newFileLength,SectorSize);  int oldSectors = divRoundUp(numBytes,SectorSize); |  1. 如果需要分配新的数据扇区，那么先找到当前文件的最后一个文件头，因为新增的长度是附加在已有数据的最后的。怎么找最后一个文件头之前已经提及，这里不再赘述。  |  | | --- | | if(curSectors>oldSectors){  int add = curSectors-oldSectors;  // find the last file header  FileHeader\* fHdr = new FileHeader;  int lastFHdrSecNo = secNo;  fHdr->FetchFrom(secNo);  DEBUG('f',"Next file header index: %d\n",fHdr->getIndirect());  while(fHdr->getIndirect()!=IllegalIndirectSectorNo){  lastFHdrSecNo = fHdr->getIndirect();  fHdr->FetchFrom(fHdr->getIndirect()); // 5  DEBUG('f',"Next file header index: %d\n",fHdr->getIndirect());  curSectors -= NumDirect;  } |   这里要注意的是我们要记录最后一个文件头的扇区号，还记得之前的递归函数的入参吗？要求传入当前更新的文件头的扇区号（因为要把更新结果写回）。   |  | | --- | | // where to start append  int startNo = fHdr->calculateNumSectors(); |  1. 查看当前文件头已经分配的扇区数，从第一个没有分配的数据扇区开始分配  |  | | --- | | // where to start append  int startNo = fHdr->calculateNumSectors(); |  1. 由于FileSystem实例中的位图文件是个私有字段，所以我们这里要把位图从磁盘拉出来。  |  | | --- | | int bitmapSecNo = 0;  // fetch the bit map from disk  OpenFile\* bitmap = new OpenFile(bitmapSecNo); // 0  // load the bitmap to mem  BitMap\* bm = new BitMap(NumSectors);  bm->FetchFrom(bitmap); // 2 |  1. 递归地从最后一个文件头开始分配数据扇区，如果需要额外地文件头，则递归分配，这里直接复用之前初始化的递归函数。  |  | | --- | | AllocateEachFHdr(bm,startNo,add,lastFHdrSecNo); |  1. 热更新内存中的文件头信息，原因之前初始化方面已经详述过。  |  | | --- | | FetchFrom(secNo); |  1. 写回位图文件。  |  | | --- | | bm->WriteBack(bitmap); |   h.更新文件长度。以及热更新最后一个文件覆盖的数据长度！原因1.3.1节已经提及。   |  | | --- | | this->numBytes = newFileLength;  updateLastFileHeaderLength(secNo); |   至此Lab5兼容了Lab4中文件变长的功能。 1.4. DFS算法回收磁盘块 Nachos既然支持了创建文件的功能，也势必支持删除文件的功能。在删除文件时，我们要释放该文件控制的磁盘扇区，以供其他文件使用。  既然分配可以用DFS，那么回收也当然可以：   |  | | --- | | void  FileHeader::Deallocate(BitMap \*freeMap,int curSectorNo)  {  if(curSectorNo==IllegalIndirectSectorNo){  // the end of the external file headers  return;  }  FileHeader\* fHdr = new FileHeader;  // get current file header  fHdr->FetchFrom(curSectorNo);  int numSectors = fHdr->calculateNumSectors();  int\* ds = fHdr->getDataSectors();  for (int i = 0; i < numSectors; i++) {  ASSERT(freeMap->Test((int) ds[i])); // ought to be marked!  freeMap->Clear((int) ds[i]);  }  ASSERT(freeMap->Test((int)curSectorNo));  freeMap->Clear((int)curSectorNo);  Deallocate(freeMap,fHdr->getIndirect());  } |   流程也很简单：   1. 递归基：若递归到-1（最后一个文件头的下一个文件头扇区号，是一个宏定义的无效值），那么代表回收全部完成，结束。 2. 从磁盘拉出当前文件头。将位图上对应的扇区释放。 3. 回收下一个文件头。  1.5. 无限大的Nachos文件 这里只是为了展示一下我扩展的文件系统下的文件到底可以多大。截图为证。  Untitled 3  这里我选择了测试用例中最大的Unix文件，2432字节，然后-ap选项拼接了很多次，最后得到一个这样的文件：58368个字节。注意，这并不是这个系统的极限，只是没必要继续下去了，这已经证明了我的扩展下的Nachos文件的理论大小是无限大，实际上只受磁盘大小的限制了。  而一般的二级索引，最多只能支持(29+30)\*128 = 7552字节。huge文件随便多append几次，一般的扩展逻辑就会错误。 统计选项-DI 这个需求其实对文件系统内核代码没有任何改动，只是计算了一些数据打印出来查看而已。由于打印的是整个文件系统的信息，所以我就在FileSystem类中实现了这个功能，实现如下：   |  | | --- | | void  FileSystem::Statistic() {  // Fetch the bitmap and dir  Directory\* dir;  dir = new Directory(NumDirEntries);  dir->FetchFrom(directoryFile);  BitMap\* bm;  bm = new BitMap(NumSectors);  bm->FetchFrom(freeMapFile);  FileHeader\* fHdr = new FileHeader;  printf("Disk size: %d sectors, %d bytes.\n",NumSectors,NumSectors\*SectorSize);  int used = NumSectors-bm->NumClear();  printf("Used: %d sectors, %d bytes.\n",used,used\*SectorSize);  int idle = NumSectors-used;  printf("Free: %d sectors, %d bytes.\n",idle,idle\*SectorSize);  DirectoryEntry\* dirEntry = dir->getTable();  int\* fSizes = new int[NumDirEntries];  int dirSize = 0;  for(int i=0;i<NumDirEntries&&dirEntry[i].inUse;i++){  fHdr->FetchFrom(dirEntry[i].sector);  fSizes[i] = fHdr->FileLength();  dirSize++;  }  int fOccupy = 0;  int rfSize = 0;  for(int i=0;i<dirSize;i++){  fOccupy += divRoundUp(fSizes[i],SectorSize);  rfSize += fSizes[i];  }  printf("%d bytes in %d files, occupy %d bytes(%d sectors).\n",rfSize,dirSize,fOccupy\*SectorSize,fOccupy);  int fragments = 0;  int fragSectors = 0;  for(int i=0;i<dirSize;i++){  fragments += SectorSize-(fSizes[i]%SectorSize);  if(fSizes[i]%SectorSize!=0){  fragSectors += 1;  }  }  printf("%d bytes of internal fragmentation in %d sectors.\n",fragments,fragSectors);  delete dir;  delete bm;  delete fHdr;  } |   流程如下：   1. 由于统计信息需要用到位图和目录文件内容，所以我们先从磁盘把这些内容拉出来。 2. 磁盘扇区数即（盘片数platter每个盘片的磁轨数track每个磁轨的扇区数Sectors），由于我们就一个盘片，所以就是track\*Sectors，而字节数就是再乘上每个扇区的字节数。  |  | | --- | | printf("Disk size: %d sectors, %d bytes.\n",NumSectors,NumSectors\*SectorSize); |  1. 已经使用的扇区数就是总的扇区数减去位图中未标记的扇区数。空闲的扇区数就是位图中未标记的扇区数。  |  | | --- | | int used = NumSectors-bm->NumClear();  printf("Used: %d sectors, %d bytes.\n",used,used\*SectorSize);  int idle = NumSectors-used;  printf("Free: %d sectors, %d bytes.\n",idle,idle\*SectorSize); |  1. 计算文件总大小需要从目录中找到每个文件的扇区号，然后从磁盘读取对应扇区得到根文件头，根文件头中存储了整个文件按的大小，累加即可得到总的文件大小。总计分配的扇区数就是每个文件分配的扇区数之和。一个文件分配的扇区数即为该文件的字节数除以一个扇区的字节数再向上取整。  |  | | --- | | DirectoryEntry\* dirEntry = dir->getTable();  int\* fSizes = new int[NumDirEntries];  int dirSize = 0;  for(int i=0;i<NumDirEntries&&dirEntry[i].inUse;i++){  fHdr->FetchFrom(dirEntry[i].sector);  fSizes[i] = fHdr->FileLength();  dirSize++;  }  int fOccupy = 0;  int rfSize = 0;  for(int i=0;i<dirSize;i++){  fOccupy += divRoundUp(fSizes[i],SectorSize);  rfSize += fSizes[i];  }  printf("%d bytes in %d files, occupy %d bytes(%d sectors).\n",rfSize,dirSize,fOccupy\*SectorSize,fOccupy); |  1. 总的内碎片数即为每个文件的内碎片数累加之和。每个文件的内碎片数相当于最后一个文件头的内碎片数（因为之前的文件头全部分配满了，或者是最后一个文件头就是根文件头，也就是这个文件的大小小于等于3712字节），一个文件头的内碎片数相当于一个扇区的字节数与这个文件头所覆盖的字节数取余一个扇区的字节数之差。   而总的存在内碎片的扇区数相当于每个文件的存在内碎片的扇区数，也就是每个文件最后一个文件头的存在内碎片的扇区数，如果最后一个文件覆盖的字节数不是128的整数倍，那么这个文件头包括了一个含有内碎片的扇区。   |  | | --- | | int fragments = 0;  int fragSectors = 0;  for(int i=0;i<dirSize;i++){  fragments += SectorSize-(fSizes[i]%SectorSize);  if(fSizes[i]%SectorSize!=0){  fragSectors += 1;  }  }  printf("%d bytes of internal fragmentation in %d sectors.\n",fragments,fragSectors); |   至此，我们实现了统计选项-DI。 3. Nachos文件的rwx权限 这里我基于上学期操作系统理论课学习过的ACL（Access Control List，权限控制列表）来实现Nachos文件的权限功能。 3.1. 什么是ACL？ ACL is short for Access Control List。每一个文件的目录项中都存放了一个ACL，标识了哪些用户可以访问这个文件，以及访问时的操作权限。如下图：  Untitled 4  一个ACL里存放三个一维数组，分别代表   1. 可读的用户id 2. 可写的用户id 3. 可执行的用户id   这里我们规定，所有有效的用户id全部都是非负数。所以-1是一个无效值，用来填充数组的。 3.2. 数据结构的更新3.2.1. ACL的数据结构 3.1节中提到过，ACL本质上就是三个一维的数组拼起来的二维数组。因此成员字段为：   |  | | --- | | #define MAXUSER 10  class AccessControlList{  public:  int rwx[3][MAXUSER];  }; |   这里我们假设最大用户数量为10。那么ACL就是3\*10=30的一个矩阵。 3.2.2. ACL作为目录表项的成员字段 3.1节中提到，每个文件对应的目录表项中都有一个ACL（每个文件的rwx权限对不同的用户也不同），因此成员字段修改为：   |  | | --- | | class DirectoryEntry {  public:  bool inUse; // Is this directory entry in use?  int sector; // Location on disk to find the  // FileHeader for this file  char name[FileNameMaxLen + 1]; // Text name for file, with +1 for the trailing '\0'  AccessControlList acl;  }; |  3.3. 为文件注册ACL 那么我们在目录中添加一个文件时，同时也要注册这个文件的ACL。   |  | | --- | | bool  Directory::Add(char \*name, int newSector,AccessControlList\* acl)  {  if (FindIndex(name) != -1)  return FALSE;  for (int i = 0; i < tableSize; i++)  if (!table[i].inUse) {  table[i].inUse = TRUE;  strncpy(table[i].name, name, FileNameMaxLen);  table[i].sector = newSector;  table[i].acl = acl;  return TRUE;  }  return FALSE; // no space. Fix when we have extensible files.  } |   这里假设ACL对象acl已经在调用Add前初始化好了，Add只为目录表项注入一个已经初始化完成的acl。 3.4. 用户与鉴权 ACL的好处在于，对于不同的用户，可以标识出不同的权限。试想一下，如果我们直接在目录表象中标识r,w,x的值，那么就相当于这个文件的权限对于所有用户一视同仁。但在真实的操作系统中，用户的权限是有区别的，比如root/Administrator用户的权限就比一般用户的权限高，可以执行一些不同的操作，使用ACL就可以区分这一点。  那么既然都有用户的概念了，也必然有对应的类。 3.4.1. 用户类 我认为用户只需要一个用来标识的id和用来验证的密码即可。但除此之外还仿照DirectoryEntry类中添加了一个use字段。这在概念上是相符的，目录作为文件的注册中心；那么用户鉴权模块（3.4.2节中提到）就是用户的注册中心。  成员字段如下：   |  | | --- | | #define pwdLen 9  class User{  public:  int uid;  char password[pwdLen+1];  bool use;  }; |  3.4.2. 用户鉴权表 也就是3.4.1节中提到的用户鉴权模块，相当于目录之于文件一样，用户鉴权表之于用户，起到了一个注册中心的作用。  数据结构如下：   |  | | --- | | class Authentication{  //...  bool Add(int uid,char\* pwd);  bool Auth(int uid,char\* pwd);  private:  int tableSize;  User \*users;  }; |   这里我只标出了比较重要的成员函数，也即:   1. Add:用户注册函数 2. Auth:用户鉴权函数   这两个函数的实现也比较简单，先看Add函数：   |  | | --- | | bool  Authentication::Add(int uid, char \*pwd) {  for(int i=0;i<tableSize;i++){  if(!users[i].use){  users[i].use = true;  users[i].password = pwd;  user[i].uid = uid;  return true;  }  }  return false;  } |  1. 查找用户鉴权表，找到一个没有在使用的表项，注册用户，返回true代表注册成功。 2. 如果不存在没有使用的表项，返回false代表注册失败。   然后是Auth函数：   |  | | --- | | bool  Authentication::Auth(int uid, char \*pwd) {  for(int i=0;i<tableSize;i++){  if(users[i].use&&users[i].uid==uid){  if(!strcmp(users[i].password,pwd)){  return false;  }  return true;  }  }  return false;  } |  1. 找到对应用户uid的表项，检查密码是否相同，相同返回true，不相同返回false。 2. 如果没有这个用户，返回false。   最后记得要在文件系统中注册并及时写回这个用户鉴权表，我们需要将他持久化到磁盘。   |  | | --- | | FileSystem::FileSystem(bool format)  {  if (format) {  BitMap \*freeMap = new BitMap(NumSectors);  Directory \*directory = new Directory(NumDirEntries);  Authentication \*auth = new Authentication(MAXUSER);  FileHeader \*mapHdr = new FileHeader;  FileHeader \*dirHdr = new FileHeader;  FileHeader \*authHdr = new FileHeader;  freeMap->Mark(FreeMapSector);  freeMap->Mark(DirectorySector);  freeMap->Mark(AuthenticationSector);  ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize,FreeMapSector));  ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize,DirectorySector));  ASSERT(authHdr->Allocate(freeMap, AuthenticationFileSize,AuthenticationSector));  mapHdr->WriteBack(FreeMapSector);  dirHdr->WriteBack(DirectorySector);  authHdr->WriteBack(AuthenticationSector);  freeMapFile = new OpenFile(FreeMapSector);  directoryFile = new OpenFile(DirectorySector);  authFile = new OpenFile(AuthenticationSector);  freeMap->WriteBack(freeMapFile); // flush changes to disk  directory->WriteBack(directoryFile);  auth->WriteBack(authFile); |   这里我把用户鉴权表作为和位图以及目录文件一样的特殊文件存放在Nachos虚拟磁盘中。 3.4.3. 用户注册 这个接口提供给更上层的应用模块。   |  | | --- | | bool register(int uid,char\* pwd){  //...  // suppose we have a correct Authentication instance now,its name is auth  auth->Add(uid,pwd);  //...  //write the auth instance back to the disk  } |   假设一个模块专门负责用户注册，那么用户提交完密码和id后，然后调用鉴权模块的Add函数，在鉴权表中添加一个用户，即为注册。  当然也有登录和注销操作，这里不赘述了。 3.5. 用户访问文件 这个接口提供给更上层的应用。  假设一个模块专门负责用户访问文件，那么用户提交密码、id和文件名后，先检查是否存在该用户，且密码正确。然后检查指定文件名的ACL对应的操作权限中是否有这个用户。   |  | | --- | | /\*\*  \* Read a nachos virtual file  \* @param uid user id  \* @param pwd password  \* @param fName file name  \* @param op operation index  \* @return null if there's no file with fName or Authentication fail  \*/  OpenFile\* AccessNachosFile(int uid,char\* pwd,char\* fName,int op){  // suppose we have a correct Authentication instance now,its name is auth  if(!auth->Auth(uid,pwd)){  return NULL; // Authentication fail  }  // suppose we have a correct Directory instance now,its name is dir  if(!dir->VerifyUid(uid,fName,op)){  return NULL; // Authentication fail  }  int secNo = dir->Find(fName);  if(secNo==-1){  return NULL; // there's no file with name fName  }  OpenFile\* ret = new OpenFile(secNo);  return ret;  } |  3.5.1. 用户权限验证 通过用户鉴权表中的Auth函数，检查用户是否存在且密码是否正确。有一项不对就返回空指针。   |  | | --- | | OpenFile\* AccessNachosFile(int uid,char\* pwd,char\* fName,int op){  // suppose we have a correct Authentication instance now,its name is auth  if(!auth->Auth(uid,pwd)){  return NULL; // Authentication fail  } |  3.5.2. 文件权限验证 检查指定文件的ACL中是否允许当前用户做指定操作。如果没有该文件或不允许该用户做该操作则返回空指针。   |  | | --- | | // suppose we have a correct Directory instance now,its name is dir  if(!dir->VerifyUid(uid,fName,op)){  return NULL; // Authentication fail  }  int secNo = dir->Find(fName); |   那么我们就要检查目录表项的ACL了，实现为：   |  | | --- | | //directory.h  bool VerifyUid(int uid,char\* fName,int op);  //directory.cc  bool  Directory::VerifyUid(int uid,char\* fName,int op) {  for(int i=0;i<tableSize;i++){  // find the specific file  if(table[i].inUse&&!strcmp(table[i].name,fName)){  // check it's valid uid for specific operation  int\* opUid = table[i].acl.rwx[op];  for(int i=0;i<MAXUSER;i++){  if(opUid[i]==uid){}  return true;  }  }  }  return false; // no file with name fName or authentication failed  } |   思路也很简单：   1. 查找指定文件 2. 若找到，检查该文件的ACL是否允许该用户id执行对应操作。允许返回true。 3. 若没找到或不允许，返回false。   流程图如下：  Untitled 5 | | | |
| 结论分析与体会：  Lab5是对Lab4的扩展，我觉得这个Lab是前五个Lab中最有意思，自由度最高的Lab。  实验要求中要求我们实现二级索引。我觉得代码写死了很不优雅，因此我决定实现任意级索引。Nachos的磁盘空间索引结构是一种Linked+Indexed结合的结构，文件头中既存放了该文件头控制的数据扇区的扇区号，又记录了下一个文件头（如果有的话）的磁盘扇区号。这着实好好帮我复习了一遍上学期理论课的内容。  为了实现任意级索引，我选择使用Linked+DFS的方式递归地分配/释放磁盘空间。为了兼容Nachos原有的各个接口，我对FileHeader类中的Allocate\Deallocate类做了很大一番改造，或者可以说是直接重写了。最终，我实现的多级索引结构可以在理论上支持无限大的Nachos文件（只要磁盘无限大，文件就可以无限大）。  对于统计选项，我先手推了各个统计量的计算公式，然后在Stats类中进行了打印，观察了总字节数、扇区数、内碎片数等各个统计量的内容。也思考了诸如Compaction等压缩内碎片的解决方案。  最后，我尝试使用上学期理论课中的ACL（Access Control List）来实现Nachos文件的rwx权限。由于rwx要和用户交互，我还写了个模拟鉴权的Authentication类来作为用户账密检查的模块。随后我修改了DirectionEntry类，使得它能够支持记录不同用户r/w/x的权限。当用户操作文件的请求到来时，检查该目录表项，查看是否有对应权限。  Lab5的内容比较简单，但是通过自己编码的方式，我更进一步了解和认识了操作系统对于磁盘空间的控制和调度方式，并切实地把上学期理论中提到的各个概念使用到了自己地系统中。由此加深了自己对os的理解。 | | | |