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：Lab4 | | | |
| 实验题目：扩展文件系统 | | | |
| 实验学时：4 | | 实验日期：2023.11.4 | |
| 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)。 | | | |
| 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. 源码阅读与分析 2](#_Toc12729)  [1.1. nachos虚拟磁盘 2](#_Toc19283)  [1.1.1. 磁盘在哪？ 2](#_Toc14290)  [1.1.1.1. 架构 2](#_Toc16288)  [1.1.1.2. 软硬件模拟磁盘源码 3](#_Toc5318)  [1.1.1.3. 磁盘有多大？ 5](#_Toc4689)  [1.1.2. 基于I/O中断的文件系统 7](#_Toc6826)  [1.1.2.1. 注入中断回调函数 7](#_Toc22246)  [1.2.文件系统的实现 14](#_Toc22989)  [1.2.1.磁盘空间管理 14](#_Toc17400)  [1.2.1.1.位示图 14](#_Toc8134)  [1.2.2.单文件空间管理 15](#_Toc9372)  [1.2.2.1.文件头(filehdr) 15](#_Toc22474)  [1.2.3.目录结构 22](#_Toc6622)  [1.2.3.1.文件头注册中心 22](#_Toc7864)  [1.2.3.2.目录 22](#_Toc21790)  [1.2.4.文件系统 23](#_Toc12413)  [2. 实现 25](#_Toc31321)  [2.1. 扩展文件大小 25](#_Toc16842)  [2.1.1. 不可扩展大小的文件 26](#_Toc4508)  [2.1.1.1.什么是不可扩展大小 26](#_Toc12326)  [2.1.2. 实现方式与细节 27](#_Toc26935)  [2.1.2.1.文件头长度字段调整 27](#_Toc997)  [2.1.1.2.动态更新位图 31](#_Toc5741)  [2.2. 为文件增加时间戳 38](#_Toc23182)  [2.2.1. 详细需求 38](#_Toc28847)  [2.2.2. 实现方式与细节 39](#_Toc17841)  [2.2.2.1.一个扇区的兼容方案 39](#_Toc934)  [2.2.2.2 文件头更新写回的时机 39](#_Toc18758) 源码阅读与分析1.1. nachos虚拟磁盘1.1.1. 磁盘在哪？1.1.1.1. 架构 Nachos选择以一种软硬件协作的方式来模拟Nachos虚拟机下的磁盘。Nachos使用宿主机上的磁盘空间作为自己的磁盘空间，而这个磁盘空间是一个单文件。  Untitled  如果你想在Nachos下外挂更多的虚拟磁盘，那么就用更多的宿主机下的单文件来模拟这些磁盘即可。 1.1.1.2. 软硬件模拟磁盘源码 首先看一下模拟磁盘的类disk的数据结构   |  | | --- | | // disk.h  private:  int fileno; // UNIX file number for simulated disk  VoidFunctionPtr handler; // Interrupt handler, to be invoked  // when any disk request finishes  \_int handlerArg; // Argument to interrupt handler  bool active; // Is a disk operation in progress?  int lastSector; // The previous disk request  int bufferInit; // When the track buffer started  // being loaded |   最重要的：fileno ，代表了linux/unix下真实文件的标识符，可以简单理解成文件的指针，指向了宿主机上的一个真实存在的文件。  那么怎么初始化一个实例来模拟一个磁盘？   |  | | --- | | // disk.cc  //----------------------------------------------------------------------  // Disk::Disk()  // Initialize a simulated disk. Open the UNIX file (creating it  // if it doesn't exist), and check the magic number to make sure it's  // ok to treat it as Nachos disk storage.  //  // "name" -- text name of the file simulating the Nachos disk  // "callWhenDone" -- interrupt handler to be called when disk read/write  // request completes  // "callArg" -- argument to pass the interrupt handler  //----------------------------------------------------------------------  Disk::Disk(const char\* name, VoidFunctionPtr callWhenDone, \_int callArg)  {  int magicNum;  int tmp = 0;  DEBUG('d', "Initializing the disk, 0x%x 0x%x\n", callWhenDone, callArg);  handler = callWhenDone;  handlerArg = callArg;  lastSector = 0;  bufferInit = 0;    fileno = OpenForReadWrite((char\*)name, FALSE);  if (fileno >= 0) { // file exists, check magic number  Read(fileno, (char \*) &magicNum, MagicSize);  ASSERT(magicNum == MagicNumber);  } else { // file doesn't exist, create it  fileno = OpenForWrite((char\*)name);  magicNum = MagicNumber;  WriteFile(fileno, (char \*) &magicNum, MagicSize); // write magic number  // need to write at end of file, so that reads will not return EOF  Lseek(fileno, DiskSize - sizeof(int), 0);  WriteFile(fileno, (char \*)&tmp, sizeof(int));  }  active = FALSE;  } |   先别管磁盘操作的中断回调函数，主要关注fileno相关的操作。（在sysdep.cc里封装的各种轮子这里我就不详细介绍了，大多是系统调用，也不难）   1. 首先尝试按照磁盘名name打开对应的宿主机单文件，如果打开成功了，除了检查确保当前打开的文件确实是nachos的虚拟磁盘，不做其他事情。   这里官方用了一个小trick，它向nachos虚拟磁盘对应的单文件头部写入了一个硬编码值来标识当前的文件是否是Nachos虚拟磁盘：   |  | | --- | | // We put this at the front of the UNIX file representing the  // disk, to make it less likely we will accidentally treat a useful file  // as a disk (which would probably trash the file's contents).  #define MagicNumber 0x456789ab  #define MagicSize sizeof(int) |   也就是nachos虚拟磁盘的前四个字节一定就是0x456789ab，只要我们打开的文件的前四个字节是这个MagicNumber ，就代表当前文件已经被Nachos虚拟机“征用”。   1. 如果打开失败，这说明该磁盘还不存在，我们要建立对应的宿主机文件，因此它创建了一个文件：  |  | | --- | | fileno = OpenForWrite((char\*)name); |   然后向头部写入了上面提到的标识符MagicNumber 。   |  | | --- | | magicNum = MagicNumber;  WriteFile(fileno, (char \*) &magicNum, MagicSize); // write magic number |  1. 最后还没有结束，官方很细节地认为读磁盘时不能返回一个EOF（磁盘哪来的End Of File)，虽然其实也无伤大雅，但总给用户一种“露馅”了的感觉：“原来我用的是虚拟磁盘啊？”   于是官方选择在最后加上四个字节，表示该磁盘容量的最大范围：   |  | | --- | | // need to write at end of file, so that reads will not return EOF  Lseek(fileno, DiskSize - sizeof(int), 0);  WriteFile(fileno, (char \*)&tmp, sizeof(int)); |   这也是为什么Nachos磁盘的真实大小要多加上四个字节的原因：   |  | | --- | | #define DiskSize (MagicSize + (NumSectors \* SectorSize)) |   至此，我们采用软硬件结合的方式模拟出来了Nachos的虚拟磁盘。 1.1.1.3. 磁盘有多大？ 首先先回顾一下磁盘的物理结构：  Untitled 1   1. 盘片platter：一个磁盘可能有多个盘片 2. 磁轨track：一个盘片有多个磁轨（同心圆） 3. 扇区sector：一个磁轨有多个扇区 4. 柱面cylinder：不同盘片的同一位置的磁轨组成一个柱面   那么怎么找到磁盘上存储内容的位置呢？其实有点像多级索引：   1. 确定在哪个盘片 2. 确定在哪个磁轨 3. 确定在哪个扇区 4. 确定在哪个字节   一般是按字节寻址，不会精确到bit。  那么Nachos虚拟机的磁盘有多大呢？首先我们不知道Nachos可能申请多少张盘片，因此这个是无法估计的，那么我们就来看一张盘片有多大：   |  | | --- | | #define SectorSize 128 // number of bytes per disk sector  #define SectorsPerTrack 32 // number of sectors per disk track  #define NumTracks 32 // number of tracks per disk  #define NumSectors (SectorsPerTrack \* NumTracks) |   一个扇区128字节，一个磁轨有32各扇区，一张盘片有32道磁轨，那么很显然共有：  128B \* 32 \* 32 = 128KB  但别忘了我们在磁盘尾部有个界定符tmp，因此共有128KB + 4B大小。 1.1.2. 基于I/O中断的文件系统 我们知道有 3 种 I/O 方式可以控制 I/O 设备和设备驱动来完成 I/O:   1. 轮询 I/O Polling I/O 2. 中断 I/O Interupt-Driver I/O 3. 直接内存访问 DMA I/O   Nachos采用第二种方式实现I/O操作。 1.1.2.1. 注入中断回调函数 之前提到过，在虚拟磁盘初始化时，注入过中断回调：   |  | | --- | | // disk.cc :: Disk  //...  handler = callWhenDone;  handlerArg = callArg;  //... |   后面我们会看到，callArg本质上就是Disk实例的内存地址，或者你可以认为handlerArg本质上就是一张虚拟磁盘，而handler是磁盘操作结束后要调用的中断回调函数。  我们知道中断分为好多种，之前在线程管理的部分遇到了TimerInt（时钟中断），现在我们将遇到DiskInt（磁盘中断），而所有中断都呆在Pending List中，这里面的元素均为PendingInterrupt的实例，不妨再回顾一下：   |  | | --- | | // interrupt.h  class PendingInterrupt {  public:  PendingInterrupt(VoidFunctionPtr func, \_int param, int time, IntType kind);  // initialize an interrupt that will  // occur in the future  VoidFunctionPtr handler; // The function (in the hardware device  // emulator) to call when the interrupt occurs  \_int arg; // The argument to the function.  int when; // When the interrupt is supposed to fire  IntType type; // for debugging  }; |  1. handler：中断处理函数，依赖于arg对应的实例的成员函数 2. arg：模块实例，可能是时钟，可能是磁盘，可能是控制台等等…这里是磁盘 3. when：触发时间 4. type：中断类型   还记得时钟中断时我们注入的PendingInterrup实例吗？   1. handler：TimerHandler 2. arg：Timer实例timer 3. when：计算属性 4. type：TimerInt   其中，handler调用TimerExpired，后者加入一个新的时钟中断，并调用TimerInterruptHandler说明当前线程时间片过期，应当切换。  磁盘中断的结构同样需要结合一个实际使用的场合来讲，这个实际应用场合就是带有同步机制的虚拟磁盘类SynchDisk。 1.1.2.1.1. Disk的聚合类:SynchDisk1.1.2.1.1.1. 泛化观点下的SynchDisk 为什么叫SynchDisk呢？因为磁盘本质上是个临界资源，因此我们需要同步管理访问磁盘的线程。所以就诞生了SynchDisk。而SynchDisk本质上不是通过继承Disk而来的，而是通过聚合Disk的实例得到的，因此称它为Disk的聚合类：   |  | | --- | | //synchdisk.h  private:  Disk \*disk; // Raw disk device  Semaphore \*semaphore; // To synchronize requesting thread  // with the interrupt handler  Lock \*lock; // Only one read/write request  // can be sent to the disk at a time |   互斥锁管理访问磁盘的线程，信号量同步对于磁盘的操作。  其实把SynchDisk的解析放在这里是由于它是Disk派生出来的一个实际得到应用的类：   |  | | --- | | //synchdisk.cc  SynchDisk::SynchDisk(const char\* name)  {  semaphore = new Semaphore("synch disk", 0);  lock = new Lock("synch disk lock");  disk = new Disk(name, DiskRequestDone, (\_int) this);  } |   这里我们终于看到了一个Disk的构造函数，传递的回调函数是DiskRequestDone，arg是当前的SyncDisk实例。  先来看下DiskRequestDone干了什么：   |  | | --- | | static void  DiskRequestDone (\_int arg)  {  SynchDisk\* dsk = (SynchDisk \*)arg; // disk -> dsk  dsk->RequestDone(); // disk -> dsk  } |   这里的arg实际上就是成员变量disk，也就是除了同步部分以外的磁盘实例，调用了RequestDone。   |  | | --- | | void  SynchDisk::RequestDone()  {  semaphore->V();  } |   RequestDone只做了一件事，释放磁盘资源，唤醒阻塞在信号量队列中的线程。  我们可以来看一下为Disk实例注入DiskRequestDone回调函数后会发生什么：   1. handler：DiskRequestDone 2. handlerArg：SynchDisk实例   比如我们想读磁盘，这个时候会调用SynchDisk中的ReadSector 函数：   |  | | --- | | void  SynchDisk::ReadSector(int sectorNumber, char\* data)  {  lock->Acquire(); // only one disk I/O at a time  disk->ReadRequest(sectorNumber, data);  semaphore->P(); // wait for interrupt  lock->Release();  } |  1. 确保获取互斥锁 2. 调用disk实例中已经封装好的成员函数ReadRequest ：  |  | | --- | | void  Disk::ReadRequest(int sectorNumber, char\* data)  {  int ticks = ComputeLatency(sectorNumber, FALSE);  ASSERT(!active); // only one request at a time  ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));    DEBUG('d', "Reading from sector %d\n", sectorNumber);  Lseek(fileno, SectorSize \* sectorNumber + MagicSize, 0);  Read(fileno, data, SectorSize);  if (DebugIsEnabled('d'))  PrintSector(FALSE, sectorNumber, data);    active = TRUE;  UpdateLast(sectorNumber);  stats->numDiskReads++;  interrupt->Schedule(DiskDone, (\_int) this, ticks, DiskInt);  } |   读取文件的代码这里不详细解释了，也不难。重点在于读完了之后要做中断回调，因此要给Pending List中注入一个中断事件:   |  | | --- | | interrupt->Schedule(DiskDone, (\_int) this, ticks, DiskInt); |   现在我们来看看这个PendingInterrupt的实例里面有什么：   1. handler：DiskDone 2. arg：Disk实例，也就是SynchDisk实例中的disk成员变量   when和type这里不讲了，不是重点。  那么在系统时钟推进，到达这个中断事件发生的时刻后，CheckIfDue函数将执行中断事件中的handler。参数是arg。也就是这里的**DiskDone((\_int)disk)**，那么会发生什么？   |  | | --- | | //disk.cc  static void DiskDone(\_int arg) { ((Disk \*)arg)->HandleInterrupt(); }  void  Disk::HandleInterrupt ()  {  active = FALSE;  (\*handler)(handlerArg);  }  // handler: DiskRequestDone  // handlerArg：SynchDisk实例  static void  DiskRequestDone (\_int arg)  {  SynchDisk\* dsk = (SynchDisk \*)arg; // disk -> dsk  dsk->RequestDone(); // disk -> dsk  }  void  SynchDisk::RequestDone()  {  semaphore->V();  } |   看起来有点混乱，但是我们可以从目的的角度来理清这个东西：  Q1:Disk实例的中断回调函数想要干什么？  A：什么都不想干！线程读或者写或者执行一个文件，那你自己弄去好了，弄完之后该干什么干什么，为什么要把内核喊起来中断处理，压根就没有要处理的东西。  Q2:SynchDisk实例的中断回调函数想要干什么？  A：想要释放磁盘临界资源。因为访问磁盘的操作是需要同步进行的（防止类似于脏读幻读之类的数据不一致），而在访问之前我是从信号量队列里出来的，访问之后我要告诉后面的操作说：“我访问完了，你来吧。”所以SynchDisk实例的中断回调函数想要释放临界资源。  现在最关键的来了，SynchDisk作为Disk的一个容器（注意我没有说子类，只是前者聚合了后者），它本身关注的点是同步机制，而不是其他基本功能，而中断回调属于任何一个磁盘的基本功能，也就是你这个磁盘即便没有同步机制，它也是可以处理中断回调的（即便如同我在Q1中说的，它可能完全不需要回调，但需不需要和能不能做是两码事）。  所以即便Disk实例不需要中断回调，它也应该为它的聚合类提供一个中断回调的机制，也就是至少为SyncDisk类提供这个机制（可能还有带有其他功能的很多聚合类，但现在只有SyncDisk这个聚合类）。  这就是Nachos源码在做的事情，它在更泛化的Disk类中，尝试为它的聚合类提供中断回调机制。所以这也是为什么Disk实例初始化的时候必定要传入一个中断回调函数。然后包装到HandleInterrupt中，再包装到DiskDone中传给Pending List。  **Why this bother?**  说实话，两个用来包装回调函数和调用实例的函数其实根本没用，为什么不直接传HandleInterrupt和RequestDone呢？就像这样：   |  | | --- | | void  Disk::HandleInterrupt ()  {  active = FALSE;  (\*handler)();  }  //这个还是要的，因为handler可能会出现需求变更，这样做解耦合  void  SynchDisk::RequestDone()  {  semaphore->V();  }  //在向Pending List中传中断事件时，传函数句柄，连注入实例都不用  interrupt->Schedule(HandleInterrupt,when,type) |   因为C++特性：不允许类成员函数作为函数的参数被传递。  那只能外面套一层静态函数了… 1.1.2.1.1.2. 同步机制实现：为什么要有信号量 其实我一开始没有搞懂为什么要加一个信号量，按理说只需要互斥锁就可以了，后来我发现信号量其实是为了模拟硬件方面磁盘寻道的，这里举个例子：   |  | | --- | | void  SynchDisk::ReadSector(int sectorNumber, char\* data)  {  lock->Acquire(); // only one disk I/O at a time  disk->ReadRequest(sectorNumber, data);  semaphore->P(); // wait for interrupt  lock->Release();  } |   现在我们假想自己是个线程，跟着这个函数走一遍：   1. 首先磁盘是唯一资源，想访问先获取互斥锁 2. 随后调用读磁盘的函数，读取某个扇区内容   这里要特别注意，Nachos本质上是个虚拟机，ReadRequest实际调用了C++标准库中提供的系统调用，系统调用看似轻而易举，但背后和硬件的交互是需要时间的。也就是磁盘寻道的时间，Latency Time。  因此ReadRequest中先计算了这个时间，然后往Pending List中加了个读磁盘完成的中断，提醒os内核I/O已经结束，可以继续调度线程执行了。   |  | | --- | | // disk.cc :: ReadRequest  // ...  int ticks = ComputeLatency(sectorNumber, FALSE);  //...  interrupt->Schedule(DiskDone, (\_int) this, ticks, DiskInt);  //... |  1. 信号量P操作，注意信号量实例在刚开始注入时初始值是0：  |  | | --- | | // synchdisk.cc :: SynchDisk  //...  semaphore = new Semaphore("synch disk", 0); |   那么第一个调用ReadSector的线程肯定是被阻塞的。随后切换到其他线程。   1. 现在不会立即执行释放锁的操作，因为现在这个线程还在被阻塞着呢。那什么时候重新回到ready List呢？3中切换到别的线程，会推进系统时钟。   到了b中放入的DiskInt到期时，会触发Pending List中对应的中断事件里的回调函数，也就是1.1.1小节中讲的四个函数一层包一层封装出的信号量V操作的回调函数，这时V操作，才会把信号量下的c中阻塞的线程取出来，然后放到ready List去。   1. 这样等到d中放到ready List的线程得到调度时，就会释放锁，磁盘资源也就可以重新访问了（对于其他线程来说）   在这个过程中，P操作实际上就是在等待I/O结束的中断事件，因为读磁盘是要时间的，因此os内核可以把CPU赋予其他线程使用，而因为要做I/O而被阻塞的线程只要在I/O事件结束后被唤醒即可。  流程图如下：  流程图 1.2.文件系统的实现1.2.1.磁盘空间管理1.2.1.1.位示图 bitmap用来表示Nachos虚拟磁盘每个扇区是否被占用，这属于一个工具类，后面会聚合到其他的类中，比如filesys等模块。   |  | | --- | | class BitMap {  public:  BitMap(int nitems); // Initialize a bitmap, with "nitems" bits  // initially, all bits are cleared.  ~BitMap(); // De-allocate bitmap    void Mark(int which); // Set the "nth" bit  void Clear(int which); // Clear the "nth" bit  bool Test(int which); // Is the "nth" bit set?  int Find(); // Return the # of a clear bit, and as a side  // effect, set the bit.  // If no bits are clear, return -1.  int NumClear(); // Return the number of clear bits  void Print(); // Print contents of bitmap    // These aren't needed until FILESYS, when we will need to read and  // write the bitmap to a file  void FetchFrom(OpenFile \*file); // fetch contents from disk  void WriteBack(OpenFile \*file); // write contents to disk  private:  int numBits; // number of bits in the bitmap  int numWords; // number of words of bitmap storage  // (rounded up if numBits is not a  // multiple of the number of bits in  // a word)  unsigned int \*map; // bit storage  }; |   先来看成员变量：   1. numBits：由于一位对应于一个扇区，那么Nachos虚拟磁盘上有多少个扇区，这里就有多少位 2. numWords：一个bitmap需要占据的磁盘空间大小。比如一个字长是4B，那么numBits位就有ceil(numBits/32)个字 3. map：位示图本体   注意函数 find()的作用，它返回找到的第一个空闲位的索引同时将该位置”1”。因为内存是易失性的，所以对应硬盘的位示图需要作为一个文件保存到磁盘上。它作为内核一个特殊文件被管理。函 数 FetchFrom(OpenFile \*file)和 WriteBack(OpenFile \*file)用于完成这一目的。  剩下的成员函数，实现逻辑就如同他们的函数名一样清晰，大部分都是一些位运算的trick，由于这只是一个工具类，因此不再详述。 1.2.2.单文件空间管理1.2.2.1.文件头(filehdr) 文件头描述了一个文件的数据被存储在了磁盘上的哪个位置，还有一些其他的信息（比如长度、属主等）  文件头不能独立于文件存在，否则将毫无意义。  数据结构：   |  | | --- | | class FileHeader {  public:  bool Allocate(BitMap \*bitMap, int fileSize);// Initialize a file header,  // including allocating space  // on disk for the file data  void Deallocate(BitMap \*bitMap); // De-allocate this file's  // data blocks  void FetchFrom(int sectorNumber); // Initialize file header from disk  void WriteBack(int sectorNumber); // Write modifications to file header  // back to disk  int ByteToSector(int offset); // Convert a byte offset into the file  // to the disk sector containing  // the byte  int FileLength(); // Return the length of the file  // in bytes  void Print(); // Print the contents of the file.  private:  int numBytes; // Number of bytes in the file  int numSectors; // Number of data sectors in the file  int dataSectors[NumDirect]; // Disk sector numbers for each data  // block in the file  }; |   成员变量：   1. numBytes：一个文件中的字节数 2. numSectors：一个文件中的扇区数 3. dataSectors数组：具体占用了哪些扇区   一般来说一个文件头要连续存放，而Nachos规定一个文件头最大只能占到一个扇区的长度，因此：   |  | | --- | | #define SectorSize 128 // number of bytes per disk sector  #define NumDirect (int)((SectorSize - 2 \* sizeof(int)) / sizeof(int)) |   一个扇区128B，由于numBytes和numSectors已经占用了两个4B了（int一般就是4B），那么就只剩下120B了，也就是30个字长。  就算我们不要numBytes和numSectors了，一个文件头的空间128B全部分配给dataSectors数组，那这个文件相当于最多占用128B/4B=32个扇区，也就是32\*128=4KB的磁盘空间。  这里只支持一级索引，后面会扩展二级索引的，大大增加一个文件的容量。  当我们要将磁盘空间分配给一个文件时，需要先查看位示图上是否有足够的空间（剩余扇区数是否足够）才能分配：   |  | | --- | | //filehdr.cc  bool  FileHeader::Allocate(BitMap \*freeMap, int fileSize)  {  numBytes = fileSize;  numSectors = divRoundUp(fileSize, SectorSize);  if (freeMap->NumClear() < numSectors)  return FALSE; // not enough space  for (int i = 0; i < numSectors; i++)  dataSectors[i] = freeMap->Find();  return TRUE;  }  //bitmap.cc  int  BitMap::NumClear()  {  int count = 0;  for (int i = 0; i < numBits; i++)  if (!Test(i)) count++;  return count;  }  int  BitMap::Find()  {  for (int i = 0; i < numBits; i++)  if (!Test(i)) {  Mark(i);  return i;  }  return -1;  } |   这里先调用NumClear查看扇区数是否足够，如果足够，每次找到第一个空闲的扇区标记为1，并返回扇区号置于dataSectors数组中，这样以后想访问这个文件的数据只需要按照dataSectors中的扇区号去对应扇区找就可以了。  随后比较重要的是FetchFrom和WriteBack，但这俩本质上是一样的，只是一个是从磁盘读，一个是往磁盘写，这里就以FetchFrom为例：   |  | | --- | | void  FileHeader::FetchFrom(int sector)  {  synchDisk->ReadSector(sector, (char \*)this);  } |   之前分析disk类的时候提到过，ReadSector需要你传递一个需要被填充的字符指针，由于整个文件头实际上只占用一个扇区，所以从磁盘读一个扇区出来，里面的内容就是这个文件头的内容，两个长度标记和占用扇区号的数组。  **1.2.2.2. 打开文件**  这个类名看起来像是一个动作，实际上是一个名称，OpenFile的翻译应该是“**打开的**文件”，或者是File has been opened。但用后者作为类名实在太蠢了，正常程序员都不会这么干的，所以简写为OpenFile。  数据结构：   |  | | --- | | class OpenFile {  public:  OpenFile(int sector); // Open a file whose header is located  // at "sector" on the disk  ~OpenFile(); // Close the file  void Seek(int position); // Set the position from which to  // start reading/writing -- UNIX lseek  int Read(char \*into, int numBytes); // Read/write bytes from the file,  // starting at the implicit position.  // Return the # actually read/written,  // and increment position in file.  int Write(char \*from, int numBytes);  int ReadAt(char \*into, int numBytes, int position);  // Read/write bytes from the file,  // bypassing the implicit position.  int WriteAt(char \*from, int numBytes, int position);  int Length(); // Return the number of bytes in the  // file (this interface is simpler  // than the UNIX idiom -- lseek to  // end of file, tell, lseek back    private:  FileHeader \*hdr; // Header for this file  int seekPosition; // Current position within the file  }; |   很明显的是，每一个文件都要有一个文件头hdr，以及当前文件打开的偏移量，这证明在Nachos中我们采用的是Random Access。  先来看看怎么打开一个文件，打开一个文件需要向内存中加载它占用磁盘的扇区号，因此我们需要它的文件头，也就是打开一个文件我们得知道它的文件头在哪才行:   |  | | --- | | OpenFile::OpenFile(int sector)  {  hdr = new FileHeader;  hdr->FetchFrom(sector);  seekPosition = 0;  } |   我们将指定扇区号sector中对应的数据加载到文件头中，把偏移量设置为0，就打开了一个文件。当然nachos是没有GUI的，没法像成熟的操作系统那样，双击一个文件图标，然后跳出来一个编辑界面，上面的光标指在第一个字符上。  接下来看一个细节比较多的函数，ReadAt（指定字节偏移量以及长度读取文件数据），这个函数和WriteAt很像，所以就不分析后者了：   |  | | --- | | int  OpenFile::ReadAt(char \*into, int numBytes, int position)  {  int fileLength = hdr->FileLength();  int i, firstSector, lastSector, numSectors;  char \*buf;  if ((numBytes <= 0) || (position >= fileLength))  return 0; // check request  if ((position + numBytes) > fileLength)  numBytes = fileLength - position;  DEBUG('f', "Reading %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;  // read in all the full and partial sectors that we need  buf = new char[numSectors \* SectorSize];  for (i = firstSector; i <= lastSector; i++)  synchDisk->ReadSector(hdr->ByteToSector(i \* SectorSize),  &buf[(i - firstSector) \* SectorSize]);  // copy the part we want  bcopy(&buf[position - (firstSector \* SectorSize)], into, numBytes);  delete [] buf;  return numBytes;  } |   首先用户可能乱给长度和偏移量，比如文件长度才10，偏移量给50，或者文件长度10，偏移量给1，但是要求读取的长度50，这种用户的逻辑错误我们要进行检查，并对应处理。对于前者，Nachos选择忽视这次读取请求；对于后者，Nachos选择从postion开始，读取剩下的所有文件数据。  之后是主要逻辑，我来画个图说明这段代码在干什么：  Untitled   1. 首先最重要的一个假设：我们读取文件数据时不可能细化到字节级别，至少是扇区级别的。所以如果用户指定的position并不是某个扇区的头部字节，我们就应该向下取整，去计算这个偏移量对应的那个扇区是什么，即便用户不想要position之前的数据，我们也应该把这个扇区的所有数据取出来，最后截取他想要的部分传递。 2. 对于position+numBytes-1这个偏移量来说也是如此。但为什么结尾的字节索引是这个呢？注意，position到position+numBytes-1恰好numBytes个字节，而position到position+numBytes已经有numBytes+1个字节了。 3. 根据1、2的逻辑，我们计算出第一个扇区偏移量和最后一个扇区偏移量。正如我图中加粗标红的OFFSET一样，这些扇区号只是偏移量而已，也就是说这三个扇区号是针对于文件头对应的扇区号的，而不是磁盘中的绝对扇区号。 4. 因此我们要检查文件头中的dataSectors，查看想要查询的扇区对应的磁盘扇区号，比如dataSectors[1] = 10,也就是相对于文件头的第1个扇区在磁盘上的扇区号是10。我们遍历所有想要查询的扇区号，然后把它装载到一个缓冲区buf中，从0索引处开始缓冲。 5. 最后，position-firstSector\*SectorSize就是buf中用户想要数据的开始部分，之前的都不需要传给用户，也就是从p’开始传，而numBytes个字节之后的部分也不需要传。相当于我们把每个完整的扇区都取出来了，但是只截取了用户想要的部分给他。  1.2.3.目录结构1.2.3.1.文件头注册中心 之前提到过，打开一个文件的必要条件是知道文件头所在扇区号。那么用户不可能知道这些扇区号在哪里，因此我们需要一个文件头注册中心来记录文件头的位置，而文件头注册中心的每一项由DirectoryEntry类的实例表示：   |  | | --- | | 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'  }; |  1. inUse：当前表项是否为一个文件头征用 2. sector：如果征用，文件头的扇区号 3. name：文件名  1.2.3.2.目录 数据结构如下：  需要注意的是，目录本身也作为一个Nachos文件存放在磁盘上，因此我们访问Nachos目录时，需要把它从磁盘装载到内存。而目录文件中记录了每个文件头的表项，就有了：   |  | | --- | | class Directory {  public:  Directory(int size); // Initialize an empty directory  // with space for "size" files  ~Directory(); // De-allocate the directory  void FetchFrom(OpenFile \*file); // Init directory contents from disk  void WriteBack(OpenFile \*file); // Write modifications to  // directory contents back to disk  int Find(char \*name); // Find the sector number of the  // FileHeader for file: "name"  bool Add(char \*name, int newSector); // Add a file name into the directory  bool Remove(char \*name); // Remove a file from the directory  void List(); // Print the names of all the files  // in the directory  void Print(); // Verbose print of the contents  // of the directory -- all the file  // names and their contents.  private:  int tableSize; // Number of directory entries  DirectoryEntry \*table; // Table of pairs:  // <file name, file header location>  int FindIndex(char \*name); // Find the index into the directory  // table corresponding to "name"  }; |   剩下的部分比较简单，这里不再赘述了。 1.2.4.文件系统 数据结构：   |  | | --- | | void  Directory::FetchFrom(OpenFile \*file)  {  (void) file->ReadAt((char \*)table, tableSize \* sizeof(DirectoryEntry), 0);  } |  1. freeMapFile:文件系统需要有一个位图来表示哪些块已经被占用了，哪些块没有被重用，这个位图存放在一个文件里。 2. directoryFile:1.2.3-2中提到的目录文件。   来看看一个文件系统是如何初始化的，在nachos内核启动前，你可以指定是否需要文件系统，因此初始化的操作根据你的选项也有所不同：   |  | | --- | | class FileSystem {  public:  FileSystem(bool format); // Initialize the file system.  // Must be called \*after\* "synchDisk"  // has been initialized.  // If "format", there is nothing on  // the disk, so initialize the directory  // and the bitmap of free blocks.  bool Create(char \*name, int initialSize);  // Create a file (UNIX creat)  OpenFile\* Open(char \*name); // Open a file (UNIX open)  bool Remove(char \*name); // Delete a file (UNIX unlink)  void List(); // List all the files in the file system  void Print(); // List all the files and their contents  private:  OpenFile\* freeMapFile; // Bit map of free disk blocks,  // represented as a file  OpenFile\* directoryFile; // "Root" directory -- list of  // file names, represented as a file  }; |   由于不需要文件系统比较简单，这里仅详述需要文件系统的时候是怎样初始化的：   1. 首先初始化位图以及目录对象，之后我们会把目录对象的各表项写到磁盘。 2. 初始化位图和目录文件的文件头。由于位图和目录都有一个文件头，因此位图中有两个位已经是被占用了。 3. 随后为位图和目录的文件头各自分配一个扇区，注意，如果分配失败，代表磁盘初始化失败了！ 4. 将位图和目录的文件头写回磁盘。 5. 创建位图和目录对应的文件。 6. 将位图和目录对应的文件写到磁盘上。 7. 此时我们已经保存了位图和目录文件，之前为了创建并将文件写回磁盘而申请的内存都可以不要了，释放。   文件系统结构图如下：  Untitled 1  其余函数（如创建、打开、删除等文件操作）比较简单，这里不再赘述。  而文件头以及文件占用扇区的组织结构如下：  Untitled 2 实现扩展文件大小不可扩展大小的文件2.1.1.1.什么是不可扩展大小 不可扩展大小指的是Nachos的虚拟文件的实际大小不可扩展，而不是最大大小不可扩展（当然后者也确实是不可扩展的，但那已经是Lab5的内容了，不在这次Lab的讨论范围内）。  什么叫文件的实际大小？Nachos文件头的数据结构中：   |  | | --- | | private:  int numBytes; // Number of bytes in the file |   numBytes字段就是文件的字节数，也即文件长度。不可扩展指的是这个字段无法改写（回磁盘），而我之前理解的不可扩展是：   |  | | --- | | int dataSectors[NumDirect]; // Disk sector numbers for each data  // block in the file |   一个文件可以控制的扇区数（32-1-1=30个扇区，共30\*128=3840B）不可扩展，这个是靠多级索引去扩展的，不是这次Lab的内容。  **2.1.1.2.为什么Nachos 虚拟文件不可扩展大小**  现在来看看为什么numBytes这个字段一旦初始化赋值之后就无法再改写，首先在一个文件头分配磁盘空间时：   |  | | --- | | bool  FileHeader::Allocate(BitMap \*freeMap, int fileSize)  {  numBytes = fileSize;  numSectors = divRoundUp(fileSize, SectorSize);  if (freeMap->NumClear() < numSectors)  return FALSE; // not enough space  for (int i = 0; i < numSectors; i++)  dataSectors[i] = freeMap->Find();  return TRUE;  } |   可以看到调用者决定了numBytes的赋值，也就是调用Allocate为一个文件分配磁盘空间时，需要预设这个文件的大小并把这个大小传给这个函数，即便这个文件的大小比最大文件大小要小，都没有任何办法再在这个文件中添加内容了。  注意，Allocate只能重新分配文件所占据的磁盘空间，而不能附加！因为每次Allocate都会冲刷numBytes和dataSectors中的值。  可以发现的是，Nachos源码中没有提供除此之外任何修改numBytes的方式，一旦numBytes得到初始化，这个文件的实际大小就限制在了numBytes字节，此时用户只能修改这个numBytes个字节，这一点在writeAt(打开的文件修改内容并写回磁盘的函数）中有所体现：   |  | | --- | | int  OpenFile::WriteAt(char \*from, int numBytes, int position)  {  //...  if ((position + numBytes) > fileLength)  numBytes = fileLength - position;  //... |   position是相对于这个文件开始字节的偏移量，numBytes是希望写入的字节数。可以看到，如果前两个值相加大于文件之前限制的大小，那么修改的最大字节数只能到这个文件的末字节了。  这两个函数就确定了Nachos虚拟文件无法扩展大小的格局。 实现方式与细节2.1.2.1.文件头长度字段调整 首先我们要明确的一点是，文件头的numBytes字段在更新后是要写回磁盘的，不然计算机一关机，内存中的FileHeader对象就没了，下次开机重新从磁盘加载文件头到内存时，这次的更新就消失了。因此文件头长度字段调整的方式就是将文件头重新写回磁盘。  这一点实验指导的readme中提示了四个方案：   1. 在OpenFile类中，增加WriteBack方法，手动调用此方法写回文件头。 2. 在OpenFile类析构函数~OpenFile中自动写回文件头(不判断文件长度或文件头是否已改变)。 3. 在OpenFile类的析构函数~OpenFile中，判断文件长度改变时自动写回文件头。 4. 在FileHeader类的析构函数~FileHeader中，判断文件长度改变时自动写回文件头。   我们来一个一个分析：   1. 对于一号方案来说，手动调用是没问题的，但就是有时候不方便开发者。因为有时候在完成自己封装的一个函数后，可能会忘记调用这个WriteBack函数，造成bug。   但一号方案也有必要性，比如，我们想通过类似于Ctrl+S的方式来手动保存文件。那么在实现时就可以监听键盘中断信号，然后调用这个函数写回文件头。   1. 二号方案是一种万无一失的方案，无论怎样，都会在已打开的文件对象析构（也就是文件关闭）时将文件头写回磁盘，缺点是很多时候可能用户就是打开文件看一眼，实际上没有做任何操作，这样做其实在浪费计算机资源。 2. 三号方案是二号方案的优化版本。但这样的解决方案其实不够人性化，因为在很多现代计算机中，都提示用户是否选择在文件关闭时保存文件内容。而方案三的结果就是忽略用户的意愿，强行帮用户保存。 3. 四号方案是粒度更小的二号方案，但这种实现方案不切实际。因为FileHeader写回磁盘的函数中要求提供文件头的扇区号，而这个扇区号是调用者提供的，不是FileHeader类的实例中自带的字段。  |  | | --- | | void  FileHeader::WriteBack(int sector)  {  synchDisk->WriteSector(sector, (char \*)this);  } |   而我们知道析构函数是不能有参数的。这就导致我们必须把文件头对应的扇区号要保存在这个对象的字段中，但是Nachos文件头一定要保存在一个扇区中，也就是FileHeader类的实例的内存大小是确定的，128字节，因此我们只能减少dataSectors数组的大小（numSectors之后将被替换为时间戳，所以不能用扇区号替代这个字段）。  因此这种解决方案就有结构性的矛盾，当我们扩展功能后，原有功能的表现居然没有以前好？（文件最大大小改变了）这对于我来说是无法接受的，因此我决定放弃这种解决方案。  不过前三种方案我认为都是可以选用的，因为三号方案无效操作相对于二号方案较少。因此我决定同时采用一号和三号方案。先来看一号方案如何实现。  写回文件头的方法比较好实现，因为已打开的文件已经维护了一个文件头对象的指针，直接调用文件头对象的写回函数即可：   |  | | --- | | void  OpenFile::WriteHeaderBack(int sectorNo) {  //hdr->setLastUpdateTime((int)time(NULL));  //this means the caller should pass the correct sector number of the file header on disk  this->hdr->WriteBack(sectorNo);  } |   但问题是，这要求调用者知晓这个文件头的扇区号。但已打开的文件对象是没有维护这个字段的。不过，内存中已有一个OpenFile对象的前提，是调用者知道这个OpenFile对应的函数名，打开文件的函数如下：   |  | | --- | | OpenFile \*  FileSystem::Open(char \*name)  {  Directory \*directory = new Directory(NumDirEntries);  OpenFile \*openFile = NULL;  int sector;  DEBUG('f', "Opening file %s\n", name);  directory->FetchFrom(directoryFile);  sector = directory->Find(name);  if (sector >= 0)  openFile = new OpenFile(sector); // name was found in directory  delete directory;  return openFile; // return NULL if not found  }  int  Directory::Find(char \*name)  {  int i = FindIndex(name);  if (i != -1)  return table[i].sector;  return -1;  }  OpenFile::OpenFile(int sector)  {  hdr = new FileHeader;  hdr->FetchFrom(sector); |   我们要先传递文件名，随后去文件注册中心（目录）中去遍历表项，找到该文件的文件头的扇区号，然后才能将对应文件头从磁盘读取出来。  这个函数提示我们：如果你想知道一个文件的文件头的扇区号，只要根据文件名去目录对象中去找就好了。  没有目录怎么办？目录文件的文件头的扇区号我们是知道的，这是一个商定好的数字，只要根据这个数字去加载磁盘上对应的目录文件的文件头，然后根据目录文件头中的占用的扇区号从对应磁盘上去上加载数据即可。如下所示：   |  | | --- | | //filesys.cc  #define DirectorySector 1  //...  directoryFile = new OpenFile(DirectorySector);  //...  //openfile.cc  OpenFile::OpenFile(int sector)  {  hdr = new FileHeader;  hdr->FetchFrom(sector);  } |   因此我实现了一个FileSystem类的实例暴露的根据文件名查询对应文件头扇区号的函数：   |  | | --- | | //filesys.cc  int  FileSystem::querySectorNoByFileName(char \*fileName) {  Directory\* dir;  dir= new Directory(NumDirEntries);  dir->FetchFrom(directoryFile);  int ans = dir->Find(fileName);  delete dir;  return ans;  } |   这样，调用WriteHeaderBack的一方，只要再通过这个函数去查询扇区号即可。  对于方案二，由于析构函数无法传递参数，因此只能在OpenFile类中维护一个私有字段来标识这个已打开文件对应文件头的扇区号。并且由于要判断文件长度是否改变，还需要记录一个初始值用于比较。   |  | | --- | | //openfile.h  //...  int fileHeaderSectorNo;  int latestLength;  //... |   随后在构造函数时用这个字段记录对应扇区号，析构时判断文件长度是否改变根据这个字段写回文件头：   |  | | --- | | OpenFile::OpenFile(int sector)  {  hdr = new FileHeader;  hdr->FetchFrom(sector);  /\*\*  \* Lab4:filesys extension  \* if we want to write the file header back to the disk automatically  \* when the file is closed,namely,the ~OpenFile() destructor has been invoked  \* we should know which sector on the disk should the file header be written into  \*/  fileHeaderSectorNo = sector;  //...  OpenFile::~OpenFile()  {  if(latestLength!=hdr->FileLength()){  hdr->WriteBack(fileHeaderSectorNo);  }  delete hdr;  } |   这样就完成了文件头的更新。但仅仅如此会出现一个非常严重的Segment Error。放在下一小节讲。 2.1.1.2.动态更新位图 这个问题是我遇到了BUG之后才发觉的，先贴一张BUG的图：  Untitled  可以看到的是，bit map file（也就是位图文件）中的内容遭到了极大的破坏。本来位图的值应该是一串长度仅为1024的0-1串（1024是因为32个track，一个track32个扇区，所以总计32\*32个扇区，一个扇区对应一个bit)，而图中已经完全不是一个0-1串了，携带了许多乱七八糟的信息。这是为什么呢？  在这篇文档中，我想先从一个程序员平时DEBUG的角度来看这个问题，然后再从原理层面上详述一下这种现象发生的原因，以及对应的解决方案。  作为一个程序员，遇到BUG第一反应肯定是检查当前出问题的模块，可是位图的代码都是源码自带的，根本没有做过改动，那么就可以确定不是位图模块的问题。那么就只能是上游的问题了。  根据自己平时DEBUG的经验，我注意到bit map的文件头的扇区号是一个很特殊的数字：   |  | | --- | | #define FreeMapSector 0 |   而恰好，之前实验指导书让我增加文件头的一个构造函数，在内存中new一个文件头时让该文件占用的扇区号先暂时全部为0。   |  | | --- | | FileHeader::FileHeader(){  memset(dataSectors,0,sizeof (dataSectors));  } |   也就是说每个文件头中的dataSectors的元素一开始都是0，恰好是bit map的文件头的扇区号。  而系统中有大量类似于   |  | | --- | | //openfile.cc  synchDisk->WriteSector(hdr->ByteToSector(i \* SectorSize),  &buf[(i - firstSector) \* SectorSize]);  //filehdr.cc  int  FileHeader::ByteToSector(int offset)  {  return(dataSectors[offset / SectorSize]);  } |   的调用，也就是向dataSectors所指示的扇区号中写入数据。  这就提示我，是否是因为dataSectors中的信息没有及时更新，导致元素都是0，在写入时全部都写入了0号扇区的位置，也就是bit map文件头的扇区？  事实上确实是这样的，我们来看一下这个bug要怎么复现，在复现的过程中讲清楚原理。  在控制台输入以下指令：   |  | | --- | | rm -f DISK  ./nachos -f  ./nachos -ap test/big big  ./nachos -D |   -f是创建Nachos虚拟磁盘的，这个过程在之前详述过，不再赘述。  随后是最重要的一环（-D只是把错误显示出来而已，造成错误的命令是-ap），./nachos -ap test/big big。这个命令会复制一个位于test文件夹下的名称为big的Unix文件中的内容到Nachos下名为big的虚拟文件中，如果不存在这个虚拟文件，则创建该文件：   |  | | --- | | void  Append(char \*from, char \*to, int half)  {  FILE \*fp;  OpenFile\* openFile;  int amountRead, fileLength;  char \*buffer;  // start position for appending  int start;  // Open UNIX file  if ((fp = fopen(from, "r")) == NULL) {  printf("Copy: couldn't open input file %s\n", from);  return;  }  // Figure out length of UNIX file  fseek(fp, 0, 2);  fileLength = ftell(fp);  fseek(fp, 0, 0);  if (fileLength == 0)  {  printf("Append: nothing to append from file %s\n", from);  return;  }    if ( (openFile = fileSystem->Open(to)) == NULL)  {  // file "to" does not exits, then create one  flag = false;  if (!fileSystem->Create(to, 0))  {  printf("Append: couldn't create the file %s to append\n", to);  fclose(fp);  return;  }  openFile = fileSystem->Open(to);  }  ASSERT(openFile != NULL);  // append from position "start"  start = openFile->Length();  if (half) start = start / 2;  openFile->Seek(start);    // Append the data in TransferSize chunks  buffer = new char[TransferSize];  while ((amountRead = fread(buffer, sizeof(char), TransferSize, fp)) > 0)  {  int result;  // printf("start value: %d, amountRead %d, ", start, amountRead);  // result = openFile->WriteAt(buffer, amountRead, start);  result = openFile->Write(buffer, amountRead);  // printf("result of write: %d\n", result);  ASSERT(result == amountRead);  // start += amountRead;  // ASSERT(start == openFile->Length());  }  delete [] buffer;  // Write the inode back to the disk, because we have changed it  ASSERT(fileSystem->querySectorNoByFileName(to)!=-1);  openFile->WriteHeaderBack(fileSystem->querySectorNoByFileName(to));  printf("inodes have been written back\n");    // Close the UNIX and the Nachos files  delete openFile;  fclose(fp);  } |   实现有点长，但前面一半都是打开Unix文件的部分，这里不是重点先略去。重点在于：   |  | | --- | | if (!fileSystem->Create(to, 0))  {  printf("Append: couldn't create the file %s to append\n", to);  fclose(fp);  return;  } |   一开始我们的虚拟磁盘上没有big这个文件，所以会走这个if语句创建该文件。   |  | | --- | | bool  FileSystem::Create(char \*name, int initialSize)  {  Directory \*directory;  BitMap \*freeMap;  FileHeader \*hdr;  int sector;  bool success;  DEBUG('f', "Creating file %s, size %d\n", name, initialSize);  directory = new Directory(NumDirEntries);  directory->FetchFrom(directoryFile);  if (directory->Find(name) != -1)  success = FALSE; // file is already in directory  else {  freeMap = new BitMap(NumSectors);  freeMap->FetchFrom(freeMapFile);  sector = freeMap->Find(); // find a sector to hold the file header  if (sector == -1)  success = FALSE; // no free block for file header  else if (!directory->Add(name, sector))  success = FALSE; // no space in directory  else {  hdr = new FileHeader;  if (!hdr->Allocate(freeMap, initialSize))  success = FALSE; // no space on disk for data  else {  success = TRUE;  // everthing worked, flush all changes back to disk  hdr->WriteBack(sector);  directory->WriteBack(directoryFile);  freeMap->WriteBack(freeMapFile);  }  delete hdr;  }  delete freeMap;  }  delete directory;  return success;  } |   而创建文件的函数会根据起初传进来的值分配磁盘空间，也就是：   |  | | --- | | if (!hdr->Allocate(freeMap, initialSize)) |   之前就提到过Allocate函数了，这个函数会初始化文件头中的numBytes值，也就是该文件的长度，而Create的调用者传来的fileSize=0，因此这个文件的初始长度就为0。又因为我们添加了一个memset的函数，因此此时该文件头中的dataSectors的元素全部为0。  Untitled 1  随后文件创建成功后，我们还把它写回磁盘了，也就是这个文件所征用的扇区为0号扇区（其实这是错误的，因为0号扇区是bit map file的文件头存储位置）。  这里我们要注意的是，在Create之前，磁盘上已经有5个扇区被征用了，它们是0-4号扇区，而创建big文件时需要创建它的文件头，文件头按顺序查找下一个空闲的磁盘块，也就是5号块，则0-5号扇区对应内容如下表所示：   | **扇区号** | **内容** | | --- | --- | | 0 | bit map 文件头 | | 1 | 目录文件头 | | 2 | bit map文件数据 | | 3 | 目录文件数据 | | 4 | 目录文件数据 | | 5 | big文件头 |   创建完big文件的文件头后，我们就要把源Unix文件中的内容搬过去了，而在写入时会调用WriteAt函数：   |  | | --- | | // fstest.cc  //...  result = openFile->Write(buffer, amountRead);  //...  // openfile.cc  int  OpenFile::Write(char \*into, int numBytes)  {  int result = WriteAt(into, numBytes, seekPosition);  seekPosition += result;  return result;  } |   这个文件在后来我修改正确的版本中显示为608字节。  Untitled 2  而写入时它会查找这个文件下所属的扇区，然后逐个写入。  那么现在问题就来了，此时我们并没有更新过这些扇区的编号，例如608B需要5个扇区，但此时我们的dataSectors中全是0，其实根本没有可以分配的扇区。  但执行程序根本不会管应不应该，它只按编写的逻辑运行，因此便向磁盘0号扇区中写入了big文件的内容。如下图。  Untitled 3  这样我们的写入操作就把位图的文件头给损坏了，这样我们再也找不到位图的数据块了（也即2号扇区）。  随后我们调用了./nachos -D，尝试打印文件信息。这就要把bit map的文件头调出来然后查看这个有什么扇区被这个文件征用了。结果好巧不巧的是，之前我们非法写入的前四个字节是个很大的数：164629073，这就导致程序认为bit map文件有164629073个字节，所以它后面打印了超长一段数字，并引发了段错误（Segment Error）。  现在我们知道这个bug产生的原因了，就是因为在动态改变文件大小的同时，没有为它分配新的磁盘空间，所以我们要做的就是在写入数据前，查看位图，把其中空闲的块分配给这个文件，直到分配到的块的大小之和大于等于这个文件的实际长度为止：   |  | | --- | | void  FileHeader::updateFileLength(int newFileLength) {  //calculate current used sectors  int curSectors = divRoundUp(newFileLength,SectorSize);  int oldSectors = calculateNumSectors();  if(curSectors>oldSectors){  int bitmapSecNo = 0;  // fetch the bit map from disk  OpenFile\* bitmap = new OpenFile(bitmapSecNo);  // load the bitmap to mem  BitMap\* bm = new BitMap(NumSectors);  bm->FetchFrom(bitmap);  // update the bit map  for(int i=oldSectors;i<curSectors;i++){  dataSectors[i] = bm->Find();  }  bm->WriteBack(bitmap);  delete bm;  delete bitmap;  }  this->numBytes = newFileLength;  //this->numSectors = divRoundUp(newFileLength, SectorSize);  } |   这段代码的逻辑其实不复杂，首先我们把位图从磁盘读取出来，然后为当前文件的文件头下的dataSectors赋值，直到数量足够覆盖这个文件的真实长度为止。  如此修改之后就不会再出现之前的bug了，下面贴一张正确的图：  Untitled 5 为文件增加时间戳详细需求 文档中给的需求其实不够详细，有些具体内容我是看到了readme才了解到的，现在我来总结一下有关文件时间戳的具体需求：   1. . 当用户**创建一个文件**时，这个文件的时间戳就是当前系统时间。 2. 当用户**手动修改一个文件**时，这个文件的时间戳就是当前系统时间。 3. 当用户**复制一个文件成功**时，目标文件的时间戳与源文件的时间戳一致。 4. 当用户**将一个文件A中的内容附加到另一个文件B**时 5. **若B不存在**，则创建B，且时间戳与A的时间戳相同 6. **若B存在**，则时间戳为当前系统时间  实现方式与细节2.2.2.1.一个扇区的兼容方案 既然要持久化这个时间戳，那么这个字段一定是添加在FileHeader类中的。因为OpenFile类的实例是不会写回到磁盘的。  但问题时，FileHeader的成员变量是不能随意添加的，我们一定要保证FileHeader的成员变量大小等同于一个扇区的大小（128字节），现在已经占满了，分别是：   1. 文件长度numBytes（单位：字节） → 4B 2. 文件长度numSectors（单位：扇区） → 4B 3. 文件征用的扇区dataSectors → 120B（相当于一个文件最多占用120B/4B=30个扇区）   我不可能为了加一个时间戳就直接把dataSectors占用的长度改为116B甚至更小，这点我已经说过了，**不可能为了增加一个新功能而使得旧功能的性能变差，若如此则说明这个实现方案不好**。因此我们要另想办法。  注意到我们可以用numBytes计算出numSectors，也就是说numSectors是一个**计算属性**，它依附于已有的成员变量或其他常量存在。在这里：   |  | | --- | | numSectors = ceil(numBytes/SectorSize) // ceil代表向上取整 |   所以我们可以把numSectors字段给去掉，换成时间戳字段，这也就导致时间戳字段只有4个字节，其实不利于扩展，但是我们可以通过修改单位的方式来缩小这个时间戳，使得4个字节可以存放得下时间戳的值，像这里的单位就是秒。以后再不够用了，就用分钟、小时、天、周、月、年、世纪……总有能兼容的单位。 2.2.2.2 文件头更新写回的时机 其实剩下的问题就比较简单了，按照需求在不同时机写回不同的时间戳即可。  先来看写回当前系统时间，经过查阅c++的标准库，我使用了这个函数：   |  | | --- | | time\_t time(&ts); //c++标准库time |   这个函数接受一个time\_t类型的指针ts，返回当前系统时间，并将ts指向的值变为当前系统时间。如果ts为NULL/nullptr，那么简单地返回当前系统时间。  比如我们想要在修改完一个文件后更新它的时间戳，只需要：   |  | | --- | | time\_t ts;  time(&ts);  openFile->getFileHeader()->setLastUpdateTime((int) ts); |   当然setLastUpdateTime是我自己编写的函数，不是源码里的，但实在太简单了，就是个setter，具体实现不赘述了。  还有一个问题是怎样获取源文件的时间戳，这一点对于Nachos虚拟文件来说很简单，有setter肯定也可以有getter嘛。但对于Unix文件，怎么获得它的时间戳呢？  经过查阅c++标准库，我使用如下方案：   |  | | --- | | struct stat buf;  stat(from, &buf);  const long int time = buf.st\_mtime; |  1. 这行代码定义了一个stat结构体的变量buf。stat结构体通常用于存储文件的状态信息，包括文件的修改时间(st\_mtime)，访问时间(st\_atime)，更改时间(st\_ctime)等。 2. 这行代码调用了stat函数，第一个参数是from，它应该是一个字符串，表示要获取状态信息的文件的路径。第二个参数是&buf，这是buf结构体的地址，stat函数会把结果存储在这个结构体中。   这样我们就能拿到Unix文件的时间戳了，然后设置到Nachos虚拟文件的文件头中保存起来即可。  这里放一张运行截图：  Untitled 5  这里可以看到，我从test文件夹下复制了一个文件到Nachos虚拟磁盘上，则目标文件的时间戳就是这个源文件的时间戳。 | | | |
| 结论分析与体会：  Lab4可以看作是一个简单的教学式Lab，虽然简单，但是为后续的Lab5奠定了重要的基础结构。  通过阅读源码和实验指导书，我初步了解了Nachos文件系统的架构。并对OpenFile类做了一番研究，OpenFile类中只保留了对应文件的文件头。但没有立刻把这个文件头控制的所有文件数据拿到内存中，我感觉这其实和Lab7中的虚存有一点像，他们的核心思想都是：既然我已经有了找到磁盘上对应扇区的手段（文件头/页表），那么我就只在需要用到某些数据时再把该数据拿到内存。这样的做法可以极大地节省内存空间。  起初我对于文件大小不可扩展产生了误解，以为是最大长度不可扩展，后来发现这是Lab5的内容。Lab4中的不可扩展指的是一旦用户指定了某个文件的初始长度，那么这个长度就不能再改变了，即便这个长度没有到达最大限度。  在明确了不可扩展的含义之后，我意识到在修改文件（比如向文件尾部添加数据）时需要额外动态地为当前文件头分配扇区，以及更新文件长度。稍微增加了一些字段和函数后，我便完成了第一项内容：扩展文件大小。  而对于为文件增加时间戳的需求，我首先观察了readme中各命令下打印的时间戳的结果。最终明确了何时保留源文件的时间戳、何时修改的需求。为了节省文件头占用的磁盘空间在一个扇区内，我通过观察发现，numSectors可以通过numBytes得到，本身是一个计算属性。因此我去掉了该字段，改为了时间戳字段，并编码完成了时间戳更改的功能。  Lab4让我初步了解了Nachos文件系统，将理论课的知识运用到了实际编码中。比如FileHeader类就使用了Indexed Allocation分配磁盘空闲空间。在Lab5中我们将把这种方式扩展为Linked-Indexed Allocationg的结合体，使得每个文件头的大小都在一个扇区内，而文件的大小可以做到无限大。 | | | |