Nachos文件管理

一、概述

在操作系统中，文件管理系统是负责管理和存取文件信息的子系统，它是 操作系统中与用户关系最为密切的部分。文件系统为用户提供了一种简便、统一的存取和管 理信息的方法。用户可以通过文件名，简单直观地操作存取所需要的信息，而不必关心文件是如何在物理存储介质上存放的以及一些硬件输入输出的细节。

1.1 文件

1.1.1 文件的逻辑结构

比较常见的文件逻辑结构有以下三种：

- 无结构的字节流：在这种结构下，操作系统并不知道也不关心文件的内容是什么，文件 的涵义由使用该文件的应用程序来解释，这样的文件结构比较通用。
- 定长记录结构：在这种结构下，文件的组成是一个个定长记录。记录的涵义由具体的应 用程序解释。操作系统以记录为单位读取、写入文件。
- 变长记录树状结构：在这种结构下，文件由许多不等长的记录组成。在记录的固定位置 是该记录的关键值。通过关键值将文件组织成分类树。当需要对某个记录进行 操作时，可以快速地找到该记录。目前的大型机仍然沿用该逻辑结构。

1.1.2 文件的物理结构

文件的物理结构是文件在物理介质上的存储结构。一般有连续结构、链接结构和索引结构。

- 连续结构 一个逻辑文件的信息存放在物理介质中连续编号的物理块中，这样的文件是连续结构的 文件。文件在磁带上的存放是连续结构的。
- 链接结构 这是一种非连续的存储结构。存放文件的每一个盘块中有一个指针字，指向下一个盘块 。在相应文件目录项中则包含指向文件第一个物理盘块的指针以及文件长 度的信息。DOS 文件系统的 FAT 结构即源于这种结构，所不 同的是 FAT 表将各个盘块中的链接指针组织在一起，这样可以比较方便地实现随机存储。
- 索引结构 这种结构是为每个文件建立逻辑块号到物理块号的对照表，这张表称为该文件的索引 表，索引表的索引项按文件的逻辑块号顺序排列。

1.1.3 文件访问

对文件的访问形式分为顺序访问和随机访问两类。所谓**顺序访问**，就是进程必须以顺序的方 式对文件进行字节或记录的读取或写入。比如进程需要读取第 1000 字节位置的内容，它首 先需要读取前 999 个字节。

随机访问则可以从文件的任意指定位置开始读、写操作，于是也就消除了顺序存取这个限制。 例如进程需要从第1000字节位置开始读写，可以直接将它的文件读写指针移至第1000字节， 然后读取。随机访问可以用在对磁盘文件的访问上，当然磁盘文件也可以用顺序访问方式进 行存取。

1.2 目录

用户一般按名字访问文件，文件系统要按名字对文件进行管理，这都要求实现从文件名到到 文件物理实体的映照，实施这种功能的重要结构是文件目录。每个文件目录是由目录项组成， 每个目录项对应一个文件。

具体结构略

1.3 UNIX文件系统的实现

略

二、Nachos文件管理系统

Nachos 是在其模拟磁盘上实现了文件系统。它包括一般文件系统的所有的特性，可以：

- 按照用户的要求创建文件和删除文件
- 按照用户要求对文件进行读写操作
- 对存放文件的存储空间进行管理，为各个文件自动分配必要的物理存储空间，并为文件的逻辑结构以及它在存储空间中的物理位置建立映照关系。
- 用户只需要通过文件名就可以对文件进行存放，文件的物理组织对用户是透明的。

但是应该强调的是，Nachos 只是提供了一个现代操作系统的实验平台和框架。目前实现的文件系统部分同其它部分一样，比较简单。但是 Nachos 为读者提供了进一步改进和发展的基础。它和 UNIX 操作系统课程中的文件系统有较大的区别。主要表现在：

1. 在磁盘的组织结构方面

   Nachos 中文件同样有其 inode 和一般存储磁盘块，即 Nachos 物理磁盘的扇区。但是它不象UNIX，将文件系统中所有文件的inode放在一起，即 inode 区中。每个Nachos 文件的 inode 占用一个单独的扇区，分散在物理磁盘的任何地方，同一般存储扇区用同样的方式进行申请和回收。它没有文件系统管理块，是通过位图来管理整个磁盘上的空闲块。

2. 在文件系统空闲磁盘数据块和 inode 块管理方面

   Nachos 中有一个特殊的文件，即位图文件。该文件存放的是整个文件系统的扇区使用情况的位图。如果一个扇区为空闲，则它在位图文件相应的位为 0，否则为 1。Nachos 中没有专门对 inode 扇区进行管理。当需要申请一个扇区时，根据位图文件寻找一个空闲的扇区，并将其相应的位置为 1。 当释放一个扇区时，将位图中相应的位置为 0。位图文件是一个临界资源，应该**互斥访问。**现有的文件系统没有实现互斥访问，所以每次只允许一个线程访问文件系统。**位图文件的 inode 占据 0 号扇区。**

3. 在文件系统的目录管理方面

   一般的文件系统都采用树状目录结构，有的 UNIX 文件系统还有目录之间的勾连，形成图状文件系统结构。Nachos 则比较简单，**只有一级目录**，也就是只有根目录，所有的文件都在根目录下。而且根目录中可以存放的文件数是有限的。Nachos 文件系统的根目录也是通过文件方式存放的，**它的 inode 占据了 1 号扇区。**

4. 文件的索引结构上

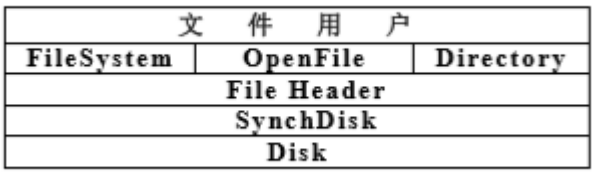
   Nachos 同一般的 UNIX 一样，采用索引表进行物理地址和逻辑地址之间的转换，索引表存放在文件的 inode 中。但是目前 Nachos 采用的索引都是**直接索引**，所以 Nachos 的 最大文件长度不能大于4K。

5. Nachos 文件系统除了在以上几点上有所不足外，还有以下一些不完善的地方：

- 必须在文件生成时创建索引表。所以 Nachos 在创建一个文件时，必须给出文件的大小；而且当文件生成后，就**不能改变文件的大小。**
- 目前该文件系统没有 Cache 机制
- 目前文件系统的健壮性不够强。当正在使用文件系统时，如果突然系统中断，文件系统中的内容可能不保证正确。

三、Nachos文件系统的实现

Nachos 的文件系统是建立在 Nachos 的模拟物理磁盘上的，文件系统实现的结构如图所示：



在 Nachos 文件系统中，许多数据结构既可存放在宿主机内存里，又可存放在磁盘上。为了一致起见，文件系统中 Synchdisk 以上的类都有一个 **FetchFrom 成员方法，它把数据结构从磁盘读到内存**；**WriteBack 成员方法，与 FetchFrom 相反，它把数据结构从内存写回磁盘。**在内存中的数据结构与磁盘上的完全一致，这给管理带来了不少方便。

3.1 同步磁盘分析（文件synchdisk.cc 、synchdisk.h）

和其它设备一样，Nachos 模拟的磁盘是异步设备。**当发出访问磁盘的请求后立刻返回，当 从磁盘读出或写入数据结束后，发出磁盘中断，说明一次磁盘访问真正结束。** Nachos 是一个多线程的系统，如果多个线程同时对磁盘进行访问，会引起系统的混乱。所以必须作出这样的限制：

- 同时只能有一个线程访问磁盘
- 当发出磁盘访问请求后，必须等待访问的真正结束。

```
class SynchDisk {
  public:
    SynchDisk(char* name);                    //  生成一个同步磁盘
    ~SynchDisk();                             //  析构方法
    void ReadSector(int sectorNumber, char* data);  //  同步读写磁盘，只有当真正读写完毕
    void WriteSector(int sectorNumber, char* data); //  后返回
    void RequestDone();                       //  磁盘中断处理时调用
  private:
    Disk *disk;                               //  物理异步磁盘设备
    Semaphore *semaphore;                     //  控制读写磁盘返回的信号量
    Lock *lock;                               //  控制只有一个线程访问的锁
};
```

3.2 位图模块分析（文件bitmap.cc、bitmap.h）

在 Nachos 的文件系统中，是通过位图来管理空闲块的。Nachos 的物理磁盘是以扇区为访问 单位的，将扇区从 0 开始编号。所谓位图管理，就是将这些编号填入一张表，表中为 0 的地方说明该扇区没有被占用，而非 0 位置说明该扇区已被占用。这部分内容是用 BitMap 类实现的。

```
class BitMap {
  public:
    BitMap(int nitems);             //  初始化方法，给出位图的大小，将所有位标明未用
    ~BitMap();                  //  析构方法
    void Mark(int which);           //  标志第 which 位被占用
    void Clear(int which);          //  清除第 which 位
    bool Test(int which);           //  测试第 which 位是否被占用，若是，返回 TRUE
    int Find();                     //  找到第一个未被占用的位，标志其被占用；
                                    //  若没有找到，返回-1
    int NumClear();                 //  返回多少位没有被占用
    void Print();                   //  打印出整个位图（调试用）
    void FetchFrom(OpenFile *file); //  从一个文件中读出位图
    void WriteBack(OpenFile *file); //  将位图内容写入文件
  private:
    ...                             //  内部实现属性
};
```

3.3 文件系统模块分析（文件filesys.cc、filesys.h）

读者在增强了线程管理的功能后，可以同时开展文件系统部分功能的增强或实现虚拟内存两 部分工作。在 Nachos 中，实现了两套文件系统，它们对外接口是完全一样的：一套称作为 FILESYS_STUB，它是建立在 UNIX 文件系统之上的，而不使用 Nachos 的模拟磁盘，它主要用于读者先实现了用户程序和虚拟内存，然后再着手增强文件系统的功能；另一套是 Nachos 的文件系统，它是实现在 Nachos 的虚拟磁盘上的。当整个系统完成之后，只能使用第二套文件系统的实现。

```
class FileSystem {
    public:
        FileSystem(bool format);                    //    生成方法
        bool Create(char *name, int initialSize);   //    生成一个文件
        OpenFile* Open(char *name);                 //    打开一个文件
        bool Remove(char *name);                    //    删除一个文件
        void List();                                //    列出文件系统中所有的文件
                                                    //    （实际上就是根目录中所有的文件）
        void Print();                               //    列出文件系统中所有的文件和它们的内容
    private:
        OpenFile* freeMapFile;                      //    位图文件打开文件结构
```

6 :


```
        OpenFile* directoryFile;                    //    根目录打开文件结构
};
```

虽然说FileSystem中的Create、Open以及Remove方法类似于UNIX操作系统中的creat、open 和unlink系统调用。但是在Nachos中，打开和创建文件没有给出打开和创建方式。这是因为 在目前的Nachos文件系统中，没有用户分类的概念，也就没有同组用户和其它用户的概念。 一个线程打开文件以后，就获得了对该文件操作所有的权利。大多数实用文件系统都提供对 文件存取进行保护的功能，保护的一般方法是将用户分类、以及对不同类的用户规定不同的存取权。

3.3.1 生成方法

语法： FileSystem (bool format)

参数： format: 是否进行格式化的标志

功能： 在同步磁盘的基础上建立一个文件系统。当 format 标志设置时，建立一个新的文件系统；否则使用原来文件系统中的内容。

实现：

1. 如果 format 标志没有设置，则使用原有的文件系统，打开位图文件和目录文件，返回
2. 如果 format 标志设置，则生成一个新的文件系统

- 生成新的位图和空白的根目录
- 生成位图 FileHeader 和目录 FileHeader
- 在位图中标识 0 和 1 号扇区被占用（虽然此时还没有占用）
- 为位图文件和目录文件申请必要的空间，如果申请不到，系统出错返回
- 将位图 FileHeader 和目录 FileHeader 写回 0 和 1 号扇区
- 打开位图文件和目录文件
- 将当前的位图和目录写入相应的文件中（位置确定，内容终于可以写入）而且这两个文件保持打开状态

3.3.2 Create 方法

语法： bool Create (char *name, int initialSize)

参数： name: 需要创建的文件名 initialSize: 需要创建的文件的初始大小

功能： 在当前的文件系统中创建一个固定大小的文件

实现： 在根目录下搜寻该文件名

1. 如果搜索到，出错返回
2. 如果没有搜索到，
   2.1. 申请文件 FileHeader 所需的空间，如果申请不到，出错返回
   2.2. 将文件加入到目录文件中，如果失败，出错返回
   2.3. 为新文件申请 FileHeader
   2.4. 根据新文件的大小申请相应块数的扇区，如果申请不到，出错返回
   2.5. 将所有有变化的数据结构写入磁盘

在 Nachos 的文件系统中，对目录对象和位图对象的操作应该是临界区操作。因为如果两个线程同时需要向同一个目录中写入一个文件，可能会出现两个线程同时申请到同一个目录项；在空闲块分配时，也会出现相类似的情况。**但是目前 Nachos 没有对此进行处理。**

3.3.3 Open 方法

语法： OpenFile * Open (char *name)

参数： name: 需要打开的文件名

功能： 在当前的文件系统中打开一个已有的文件

实现： 在根目录下搜寻该文件名 1. 如果没有搜索到，返回 NULL 2. 如果搜索到，打开该文件并返回打开文件结构

返回： 打开文件结构

3.3.4 Remove 方法

语法： bool Remove (char *name)

参数： name: 需要删除的文件名

功能： 在当前的文件系统中删除一个已有的文件

实现： 在根目录下搜寻该文件名

1. 如果没有搜索到，返回 FALSE
2. 如果搜索到，打开该文件并返回打开文件控制块
   2.1. 将该文件从目录中删除
   2.2. 释放 FileHeader 所占用的空间
   2.3. 释放文件数据块占用的空间
   2.4. 将对位图和目录的修改写回磁盘

3.4 文件头模块分析（文件filehdr.cc、filehdr.h）

文件头实际上就是 UNIX 文件系统中所说的 inode 结构，它给出一个文件除了文件名之外的所有属性，包括文件长度、地址索引表等等（文件名属性在目录中给出）。所谓索引表，就是文件的逻辑地址和实际的物理地址的对应关系。Nachos 的文件头可以存放在磁盘上，也可以存放在宿主机内存中。**在磁盘上存放时一个文件头占用一个独立的扇区。**Nachos 文件头的索引表**只有直接索引。**

```
class FileHeader {
  public:
    bool Allocate(BitMap *bitMap, int fileSize);      //  通过文件大小初始化文件头
                                                      //  根据文件大小申请磁盘空间
    void Deallocate(BitMap *bitMap);                  //  将一个文件占用的数据空间释放
                                                      //  （没有释放文件头占用的空间）
    void FetchFrom(int sectorNumber);                 //  从磁盘扇区中取出文件头
    void WriteBack(int sectorNumber);                 //  将文件头写入磁盘扇区
    int ByteToSector(int offset);                     //  文件逻辑地址向物理地址的转换
    int FileLength();                                 //  返回文件长度
    void Print();                                     //  打印文件头信息（调试用）
  private:
    int numBytes;                                     //  文件长度（字节数）
    int numSectors;                                   //  文件占用的扇区数
```



```
    int dataSectors[NumDirect];                       //  文件索引表
};
```

在 Nachos 中，每个扇区的大小为 128 个字节。每个 inode 占用一个扇区，共有 30 个直接索 引。所以 Nachos 中最大的文件大小不能超过 3840 个字节。

3.5 打开文件结构分析（文件openfile.cc、openfile.h）

该模块定义了一个打开文件控制结构。当用户打开了一个文件时，系统即为其产生一个打开 文件控制结构，以后用户对该文件的访问都可以通过该结构。打开文件控制结构中的对文件 操作的方法同 UNIX 操作系统中的系统调用。

针对 FileSystem 结构中的两套实现，这里的打开文件控制结构同样有两套实现。这里分析建 立在 Nachos 上的一套实现：

```
class OpenFile {
  public:
    OpenFile(int sector);                    //  打开一个文件，该文件的文件头在 sector 扇区
    ~OpenFile();                             //  关闭一个文件
    void Seek(int position);                 //  移动文件位置指针（从文件头开始）
    int Read(char *into, int numBytes);      //  从文件中读出 numByte 到 into 缓冲
                                             //  同时移动文件位置指针（通过 ReadAt 实现）
    int Write(char *from, int numBytes);     //  将 from 缓冲中写入 numBytes 个字节到文件中
                                             //  同时移动文件位置指针（通过 WriteAt 实现）
    int ReadAt(char *into, int numBytes, int position);
                                             //  将从 position 开始的 numBytes 读入 into 缓冲
    int WriteAt(char *from, int numBytes, int position);
                                             //  将 from 缓冲中 numBytes 写入从 position 开始的区域
    int Length();                            //  返回文件的长度
  private:
    FileHeader *hdr;                         //  该文件对应的文件头（建立关系）
    int seekPosition;                        //  当前文件位置指针
};
```

3.5.1 ReadAt 方法

语法： int ReadAt (char *into, int numBytes, int position)

参数： into: 读出内容存放的缓冲 numBytes: 需要读出的字节数 position: 需读出内容的开始位置

功能： 将从 position 开始的 numBytes 读入 into 缓冲

实现：

1. 计算实际需要读出的字节数
2. 计算出需要读出内容的扇区起始地址
3. 将这些扇区的内容读入一个内部缓冲
4. 将所需要的内容从缓冲中读出到 into 中

返回： 实际读出的字节数

3.5.2 WriteAt 方法

语法： int WriteAt (char *from, int numBytes, int position)

参数： from: 存放需写入内容的缓冲 numBytes: 需写入的字节数 position: 需写入内容的开始位置

功能： 将 from缓冲中的 numberBytes 字节从 position 开始的位置写入文件

实现：

1. 计算实际需要读出的字节数
2. 计算出需要读出内容的扇区起始地址
3. 申请一个内部缓冲
4. 将首尾扇区中不能修改内容先读入内部缓冲适当位置
5. 将需要写入文件的内容写入内部缓冲适当位置
6. 内部缓冲中内容写入磁盘文件

返回： 实际写入的字节数

实际上，对文件的一次写操作应该是原子操作，否则会出现两个线程交叉写的状况。比如 A 线程和 B 线程都需要对扇区 a 和 b 进行修改。工作过程如下：

A（写 a）-> 线程切换 -> B（写 a） -> B（写 b）-> 线程切换 -> A（写 b）

这样扇区 b 中的内容是 A 线程写的，而扇区 a 的内容是 B 线程写的。这样，数据的一致性 不能得到保证。**但是目前 Nachos 没有对此进行处理。**

3.6 目录模块分析（文件directory.cc directory.h）

目录在文件系统中是一个很重要的部分，它实际上是一张表，将字符形式的文件名与实际文 件的文件头相对应。这样用户就能方便地通过文件名来访问文件。

Nachos 中的目录结构非常简单，它只有一级目录，也就是只有根目录；而且根目录的大小是固定的，整个文件系统中只能存放有限个文件。

```
class DirectoryEntry {                  //    目录项结构
  public:
    bool inUse;                         //    该目录项是否在使用标志
    int sector;                         //    对应文件的文件头位置
    char name[FileNameMaxLen + 1];      //    对应文件的文件名
};

class Directory {
  public:
    Directory(int size);                //    初始化方法，size 规定了目录中可以放多少文件
    ~Directory();                       //    析构方法
    void FetchFrom(OpenFile *file);     //    从目录文件中读入目录结构
    void WriteBack(OpenFile *file);     //    将该目录结构写回目录文件
    int Find(char *name);               //    在目录中寻找文件名，返回文件头的物理位置
    bool Add(char *name, int newSector);  //  将一个文件加入到目录中
    bool Remove(char *name);            //    将一个文件从目录中删除
    void List();                        //    列出目录中所有的文件
    void Print();                       //    打印出目录中所有的文件和内容（调试用）
  private:
    int tableSize;                      //    目录项数目
    DirectoryEntry *table;              //    目录项表
    int FindIndex(char *name);          //    根据文件名找出该文件在目录项表中的表项序号
```