**IFileSytem（文件系统）**

* **概述(2017.7.15)**

这个类主要就是把一个磁盘文件当作一个**虚拟磁盘(VD,Virtual Disk)，**并且配备相应的文件管理操作（在这个大二进制块里面进行逻辑文件的创建、路径枚举、打开关闭修改逻辑文件），从而实现一个模拟的文件管理系统（在）。因为从宿主操作系统的角度来看，这个虚拟磁盘就是一个文件，也就是一个二进制流。所以二进制流“分割”为逻辑文件和各种路径的信息需要储存在虚拟磁盘的头部，这样子才能模拟出一个文件管理系统。

* **I/O时间**

因为磁盘的读取时间实在是有点久，常见游戏都是用一个大文件来装一堆资源，然后在游戏关卡初始化的时候一次过读取一整块数据进入内存，然后再在内存里进行文件的逻辑分割。机械硬盘一次寻道是毫秒级的，5-15ms左右吧，不考虑SSD。磁盘I/O的速率在几十M/s到几百M/s不等。所以其实就产生了一个问题：

究竟是一次把整个“Virtual Disk”的数据加载进入内存再进行逻辑文件的分割，还是每次在Virtual Disk上Open File的时候把单个逻辑文件读取进入内存呢？

暂时决定的解决方案是：**在每次OpenFile的时候把整个文件的二进制块读取进入内存里；如果对文件进行过write操作，那么在CloseFile的时候把文件缓冲区的内容写到磁盘上，然后关闭std::fstream。（2017.7.15）**

* **虚拟硬盘的分块**

因为在CreateFile的时候要分配空间给文件，那么就需要知道哪些空间是available的。有两种解决方案，一个是用链表（每个node记录下起始地址和文件大小），一种是用bitmap（详情见os课本，其中每个bit的两个状态表示对应的“磁盘分块”是否已被占用，一般每个块就几~几十KB）。

暂定的解决方案是：Bitmap来keep track of the available blocks，然后每次分配的时候就扫一遍bitmap。

因为我决定要让文件在一堆连续的block里面，所以的话要找到足够大的空位可能要一定的开销。这就需要一些对空间和时间权衡的考虑了（大文件与小文件造成的overhead）。

假设我们的Virtual Disk是1GB容量，忽略头部信息，如果一个block是4KB，那么就需要256KB的bitMap。

* **虚拟硬盘的逻辑结构**

这个文件管理系统是用到了index-node table来记录各种文件的信息的，所以**磁盘的头部必须要定长（Magic Number, Version Number,i-node table，User List等）**，其他能让用户比较动态地操作的东西、**不定长**的信息（例如directory，用户文件）都统一**以文件的形式储存。**

具体的逻辑构造看下表

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 偏移(offset:byte) | 空间(单位byte) | 备注 |
| Magic Number | 0 | 4 |  |
| Version Number | 4 | 4 |  |
| Disk Capacity | 8 | 4 | 文件空间的大小，不包括**硬盘头部信息** |
| Disk Header Length | 12 | 4 | 包括inode table的头部定长部分的byte数，值为20+M |
| I-node count | 16 | 4 |  |
| I-node table | 20 |  | Start of i-node table |
| I-node 0 ~ (I-node count) | 20 | M=Sizeof(i-node) \*count(i-node) | 以512MB的虚拟磁盘来计，共有32k = 32768个i-node，也就是最多就这么多文件+文件夹了（不想有太多磁盘头的overhead） |
| 文件块1~n | 20+M |  | 文件：路径文件或者用户文件 |

为了i-node数量好算一点，**虚拟硬盘的大小还是只限于某几个吧（128MB,256MB,512MB,1GB）。**

* **目录(directory)文件**

之前说了目录和用户文件统一为文件，所以某一个文件夹下的东西，只需要到**i-node指向的文件里面读取i-node号的列表。**

磁盘初始化时手里就抓着**第0个i-node，也就是根目录 / 的i-node**。

要访问某个文件夹下的文件，首先手里要有这目录的i-node，然后访问i-node指向的目录文件，然后读取到目录文件的“pairs个数”+ “文件名-inode号”的pairs之后就可以得到目标文件的i-node，然后就可以跳到文件的二进制区段进行读取了

* **虚拟磁盘的初始化**

加载头部信息，加载i-node table，计算出加上inode table的硬盘定长头部offset，然后**遍历整个i-node table，计算已用文件的地址区段，**计算剩余可用的空间

* **用户与帐号信息**

既然是一个文件管理系统，如果把用户信息储存在一个虚拟硬盘上，那实在感觉太别扭了（哪有一个Virtual Disk对应一套用户系统的），所以暂时的解决方案是预置几个用户，hard-coded在代码里。

预置用户信息：

|  |  |  |
| --- | --- | --- |
| 帐号 | 密码 | userID |
| (无) | (无) | 0 |
| ROOT | ROOT666666 | 1 |
| GUEST | GUEST666666 | 2 |
|  |  |  |

* **用户文件地址空间(user file address space)**

磁盘头部是定长的信息，磁盘总体信息和i-node table，剩余空间=capacity-头部。这些剩余的给文件（用户文件与目录文件）的空间就用zero-offset的地址（最小地址为0）来寻址，所以这个**用户文件地址**转换到**虚拟硬盘文件流实际地址**(fstream的seekg/seekp的streampos)的时候，要加上VD头部信息的大小（对于某个虚拟硬盘，这是一个常量offset）

（2017.7.22）考虑一个解决方案，加载虚拟硬盘的时候把整个硬盘都加载到内存里。

* **地址空间分配器(Allocator)**

(2017.7.18)磁盘的地址空间分配和回收由一个专门的类CAllocator来管理，调用allocate(start,size)来初始化allocator（跟它说明哪些地址段已经被占用了，在文件系统里特指被文件占用的地址空间），然后调用allocate(size)来分配新的地址段给新建的文件。详情见Allocator的文档CAllocator.docx。

* **目录文件（Directory File）**

(2017.7.21)理论上这种文件应该是对用户隐藏的。这种文件就是用来储存“文件夹名-目录文件inode号”和“文件名-用户文件inode号”的pairs（也就相当于一层的indirection）。OS书上说linux访问一个文件就是从根节点的i-node开始，然后访问这个i-node指向的“目录文件”的内容，查看里面的内容（一堆mapping）之后根据目录文件的内容来获取下一级的i-node，然后重复这个过程，直到获取到目标文件所在文件夹（folder）的i-node，我们就获取到了working directory的i-node，然后在working dir的i-node指向的文件的内容里查询出目标文件的i-node，然后访问文件内容。

所以才会有如下例子：

访问\aaa\bbb\d.txt需要经历一系列怎样的disk access？（假设i-node for root directory已经在内存里）

Directory (file) for \

I-node for \aaa\

Directory (file) for \aaa\

I-node for \aaa\bbb\

Directory for \aaa\bbb\

I-node for \aaa\bbb\d.txt

然后就可以访问文件内容了。

现在规定目录文件的文件结构：

|  |  |  |  |
| --- | --- | --- | --- |
| 内容 | 偏移 | 字节 | 备注 |
| 当前路径下**文件夹(folder)**数 | 0 | 4 | 也就是接下来有多少条关于folder的record，值为subFolderCount |
| 当前路径下**文件(files)**数 | 4 | 4 | 有多少条关于文件的record，值为subFileCount |
| FolderName - IndexNode号的pairs | 8 | (124+4)\*subFolderCount | 记录当前目录下的“目录文件”的i-node号 |
| FileName - IndexNode号的pairs | 8+128\*subFolderCount | (124+4)\*subFileCount | 记录当前目录下的文件的i-node号 |

* **路径规范**

一个合法的路径要满足如下条件：

1，路径string不能为空

2，路径string的开头必须为\或者/

3，路径string可以用\或者/来做分隔符

* **数据一致性(data consistency)**

虚拟硬盘的数据是把硬盘(hard disk)当成稳定储存的介质non-volatile storage。然后在程序里面Install的时候，整个Virtual Disk的数据就加载进入到内存里，这段内存成为虚拟硬盘的**内存映像(image)**，然后可以修改这个内存映像，等到程序Un-install 虚拟硬盘的时候，数据就会被重新更新到hard disk中。所以如果在**un-install完成之前，硬盘上的数据是不会变的**。