Buffer Manager说明文档

作者：2016级软件工程专业

傅诤哲 程浩然 叶慕祈 石磊 钱程

一、Buffer Manager负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件；
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换；
3. 记录缓冲区中各页的状态，如是否被修改过等；
4. 提供缓冲区页的pin功能，及锁定缓冲区的页，不允许替换出去。

   为提高磁盘I/O操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍，一般可定为4KB或8KB。

二、模块总体设计思路

记录管理模块（Record Manager）和索引管理模块（Index Manager）向缓冲区管理申请所要的数据，缓冲区管理器首先在文件池中查看数据是否存在，若存在，而且已经把数据加载到缓冲区则直接返回，否则，从磁盘中将数据加入文件池再读入缓冲区，然后返回。

最近最少使用(LRU)算法：记录所有缓冲区块的近期使用次数，每当在缓冲区块中文件被访问都会使，文件增加它的使用次数，而每当缓冲区中有块被替换时，所有的块的访问次数递减，同时讲替换的块的访问次数设为1。

改进：考虑增加脏块的设置。

三、具体实现

1、声明了宏

```
#define MAX_FILE_NUM 100                    //最大文件数
#define MAX_BLOCK_NUM 10                    //最大block数
#define BLOCK_SIZE_FILE 1                   //每个block中的文件数
#define FILE_SIZE 8096                      //文件大小
#define INITIALNUM 0                        //初始值
#define ROOT_PATH "***"                     //decided by use
```

2、声明结构体

1）文件头结构体(struct fileNode)：

包含文件的基本信息：文件名、第几页、节点是否被使用的标志、节点是否在block中的标志、指向下一个文件的指针、指向上一个文件的指针、该文件节点在block中的节点号。

2）块信息结构体(struct blockNode)：

包含块的基本信息：存放信息的字符型数组、近期引用次数（用于LRU算法）、锁、指向文件的指针。

```
struct fileNode{                                    //node in file poor
    string fileName;                                //文件名
    int pagenum;                                    //第几页
    fileNode* prefile;                              //之前页
    fileNode* nextfile;                             //之后页
    bool blocksign;                                 //该文件节点是否在block中
    int file_block;                                 //该文件节点在block中的节点号
    int written;                                    //该文件节点是否被使用
};
struct blockNode{                                   //node in block poor
    char* block;                                    //指向内存中block的字符指针
    bool pined;                                     //该block是否被锁定
    fileNode* block_file;                           //该block节点在文件池中的节点号
    int reference;                                  //近期引用次数
};
```

3、BufferManager类参数声明：

```
fileNode file_pool[MAX_FILE_NUM];
blockNode block_pool[MAX_BLOCK_NUM];
int total_block; // the number of block that have been used, which means
the block is in the list.
int total_file; // the number of file that have been used, which means the
file is in the list.
int indexoffreefile;
int badblock;
```

1. file_pool 文件池，用于存放所有文件节点(fileNode)。
2. block_pool 缓冲块池，用于存放所有缓冲区块的信息(blockNode)。
3. total_file 记录在池中的文件数。
4. total_block记录在池中的块数。
5. indexoffreefile 记录下一个空闲的文件块。
6. badblock记录当前最少使用的块，并准备用于替换。

4、主要函数及其功能描述如下：

1. `char* readfromfile(string file,int offest)`

   file为文件名，offest为页偏移量。函数实现从文件系统中读取文件加入文件池和缓存块。

   该函数先判断offset是否小于0来组织非法输入。在认为输入是合法的后，这个函数调用find函数来在在文件池和文件系统中寻找希望读到的文件。由于在具体使用中往往出现文件未生成或是已存在却没有在文件池中保留的情况，所以find函数在文件池中查询失败会去文件系统中找文件。如果文件不存在就生成文件并加入文件池。如果文件存在于文件系统就直接加入文件池。当文件处于文件池后，就可以把该文件置于块中或者文件已经存在于内存块中，直接从块中读取。

在做完find操作后，如果文件不再内存块中，readfromfile函数先调用了findthebadblock函数来使badblock的值刚好是当前最差的block的索引。然后将被替换的块指向的文件节点的引用设置失效。并启用新插入块的文件节点的引用，设置这个文件节点的引用为该块，最后将这个最差块指向新插入块的文件节点。

做完这些结构设置上的工作，就该把文件打开并取出文件大小长度的数据，置于内存block中，最后返回该block的数据指向来给函数调用者。

在函数收尾的过程中，如果发生了块的替换，所有的块的访问次数递减，同时讲替换的块的访问次数设为1。

2. `bool findthebadblock()`

   找到缓存块中最坏的一个块，并存到BufferManager中的badblock。设置初始索引位置为-1以及最小引用数为99999，然后遍历所有的block找到最少使用而且没有锁定的block，将该block设为最坏的block，把这个block的索引值交给类中私有成员badblock。在调用完findthebadblock后，就可以通过类的私有元素badblock调整block池中的block值以及修改文件池中对block的引用。

3. `void init_block(blockNode & block)`

   `void init_file(fileNode & file)`

   初始化块和初始化文件。init_block为block申请内存中的空间，并设置所有块都是没有加锁状态以及赋一些属性的初值，init_file将初始文件名命名为false，并标记该文件未使用，未映射到block中，并赋一些属性的初值。

4. `fileNode* find(string file,int offest)`

   在文件池和文件系统中寻找希望读到的文件。find函数首先在文件池中查看文件是否存在，若存在，而且已经把文件加载到缓冲区则直接返回缓冲区的位置，否则，从磁盘中将数据加入文件池再读入缓冲区，然后返回该缓冲区。find通过遍历文件池，寻找文件名一致的文件节点，在找到文件名一致的文件节点后，以该节点为基础寻找页偏移量一致的文件页节点，如果其中有任何一部分找不到都会新建一个节点，并打开文件系统通过验证`fstream f0(path,ios::in);`运行后的 f0是否成果打开来获得文件的存在状况并在文件不存在的情况下，增添一个新的文件。最后确保文件节点在文件池中存在并返回指向该节点的指针。

5. `bool writetofile(string file,int offest,int index,char* data,int length)`

   函数writetofile把长为length的位于data的数据从第offest页中的第index位写穿到磁盘中的file文件。该函数先判断offset是否小于0来组织非法输入。在认为输入是合法的后，这个函数调用find函数来在在文件池和文件系统中寻找希望读到的文件。由于在具体使用中往往出现文件未生成或是已存在却没有在文件池中保留的情况，所以find函数在文件池中查询失败会去文件系统中找文件。如果文件不存在就生成文件并加入文件池。如果文件存在于文件系统就直接加入文件池。当文件处于文件池后，就可以把该文件置于块中或者文件已经存在于内存块中，直接从块中读取。这部分是和readfromfile函数一致的。

   在做完find操作后，如果文件不再内存块中，writetofile函数先调用了findthebadblock函数来使badblock的值刚好是当前最差的block的索引。然后将被替换的块指向的文件节点的引用设置失效。并启用新插入块的文件节点的引用，设置这个文件节点的引用为该块，最后将这个最差块指向新插入块的文件节点。

   做完这些结构设置上的工作，就该把文件打开并取出文件大小长度的数据，置于内存block中，使用memcpy函数将一段定长的字符串写入内存并立即写会到磁盘。如果成果就返回true值。

   在函数收尾的过程中，如果发生了块的替换，所有的块的访问次数递减，同时讲替换的块的访问次数设为1。

6. `char* read(string file,int index)`

从文件池和文件系统读取file文件包含index的那一页。将index按文件大小分解成页号和和当前页的索引号并调用readfromfile函数。

7. `bool write(string file,int index,char* data,int length)`

   `bool write(string file,int index)`

   第一个write是从内存写穿到磁盘，第二个write为使用者修改好block，调用函数存入磁盘，必须保证文件在块中。第一个write函数将index按文件大小分解成页号和和当前页的索引号并调用writetofile函数。第二个write直接遍历查询block池，找到文件名和页数一致的块，并把整个块写回到磁盘。

8. `bool freefile(string file)`

   用于删除文件名为file的所有文件。找到文件首节点，从首节点遍历到尾节点删除对应文件以及释放节点空间。

9. `bool setpin(string file,int index)`

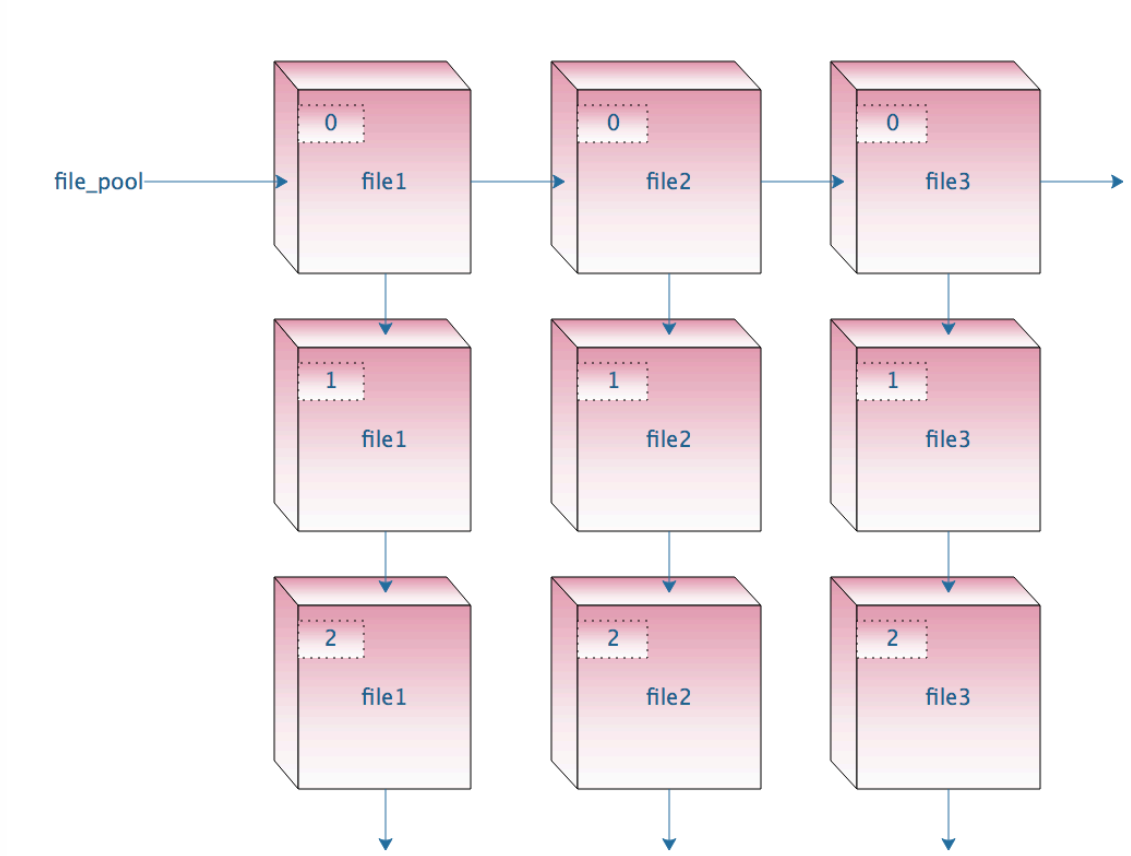
   设置file文件包含index的那一页锁定，文件必须在块中。setpin直接遍历查询block池，找到文件名和页数一致的块，并设置加锁。

10. `bool freepin(string file,int index)`

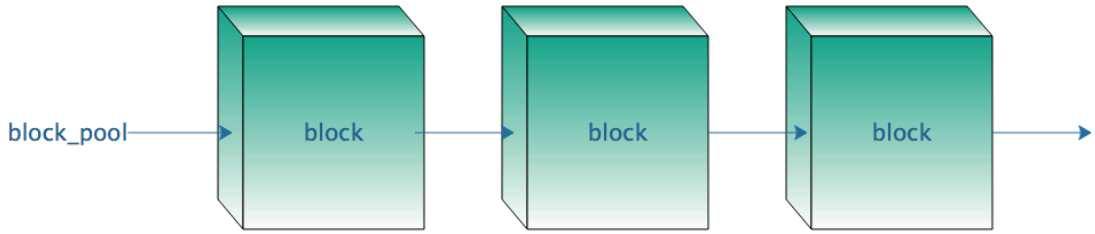
    释放file文件包含index的那一页锁定，文件必须在块中。freepin直接遍历查询block池，找到文件名和页数一致的块，并设置解锁。

5、内存中维护的结构

1）file_pool维护的结构



2）block_pool维护的结构


block_pool
block
block
block