数据库系统设计-MiniSQL

buffer模块详细设计报告

黄昭阳 3130000696 - 11/6/15

buffer模块总体设计

一、实验要求：

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、实验完成情况：

完美实现实验要求功能，其中buffer主要实现算法有：

1. buffer模块每一页的大小为4K，每次读、写文件均以4K为单位，且每个文件的大小必然为4K的整数倍。
2. 使用孙建伶老师上课时介绍的clock算法作为缓冲区的替换策略。
3. 为了实现替换策略，buffer模块记录了每个模块的多个状态，如countRef，timeStamp等等。
4. 当countRef不等于零（说明正有模块持有此页，并对其进行读写等操作），该页将被pin在缓冲区内不被替换出去。

模块整体详细分析

一、buffer整体结构分析和代码介绍：

```
#define MAXBLOCKNUMBER 100000
  //bufferManager总计10^5页，缓存区大小为400MB
#define BLOCKSIZE 4096
  //4K一页

class bufferManager
{
private:
    static unsigned char* buffer[MAXBLOCKNUMBER];
  //共10^5个指针，每个指针指向一块4K大小的内存空间

    static int index;
  //用来为clock算法记录当前在第几快内存页处。

    static int countRef[MAXBLOCKNUMBER];
  //用来记录每一页被多少个指针引用了

    static int timeStamp[MAXBLOCKNUMBER];
  //用来实现clock算法，timeStamp=1时表示当前不可被置换，timeStamp=0时表示当前可以
被置换

    static FILE* bufferToFile[MAXBLOCKNUMBER];
  //记录缓存区中每一页分别对应的哪一个文件

    static int bufferToPageNumber[MAXBLOCKNUMBER];
  //记录缓存区中每一页分别是对应文件的第几页

    static map<pair<FILE*,int>,int> filePageNumberTobuffer;
  //记录一个二元组<FILE*,int>(表示一个文件和该文件的第几页）在缓存区中的第几页

    static map<void*,int> pointerToPageNumber;
  //记录一个指针对应缓存区中的第几页

    static vector<FILE*> files;
  //记录当前缓存区中所有相关联的文件指针

    static vector<string> fileNames;
  //记录当前缓存区中所有相关联的文件名

    static int getIndexFromFileAndPage(FILE* fp,int pageNumber);
  //传入一个文件指针，和该文件的第几页，返回该二元组对应的在缓存区中的索引
```

```
    static int findAnAvailableBlock();
  //使用clock算法在缓存区中找到一块可以用的页，返回该页的索引

    static int findABlockToReplace(FILE* fp,int pageNumber);
  //使用clock算法在缓存区中找到一页，用该页的内容来替换，载入fp文件中的第pageNumber
页，返回该页的索引

    static int loadIntoMemory(FILE* fp,int pageNumber,int index);
  //将fp文件的第pageNumber页载入缓存区的第index页中

    static int flushIntoDisk(FILE* fp,int pageNumber,int index);
  //将缓存区的第index页存入fp文件的第pageNumber页中

    FILE* fp;
  //注意! 该私有变量非静态，表示此类实例化的对象所绑定的文件
    string* fileName;
  //注意! 该私有变量非静态，表示此类实例化的对象所绑定的文件的文件名
public:
    ~bufferManager();
    bufferManager(string* fileName);
  //使用文件名实例化一个bufferManager类，将该对象与文件名对应的文件进行绑定

    static int initialMemory();
  //对整个bufferManager类进行初始化
    static void quitBufferManager();
  //结束对bufferManager类的所有使用

    void* startUsePage(int pageNumber);
  //通过实例化的buffer调用，传出一个指针，该指针对应buffer所绑定的文件的第
pageNumber页的内容，可以通过该指针对该内容进行修改

    void finishUsePage(int pageNumber);
  //通过实例化的buffer调用，buffer所绑定的文件的第pageNumber页的countRef将减一
（少一个指针引用），当countRef为0时，bufferManager将不再确保该页被pin在缓存区内。

    void* startUseBuffer(int offset);
  //通过实例化的buffer调用，传出一个指针，该指针对应buffer所绑定的文件的offset偏移位
置所在的页的内容，可以通过该指针对该内容进行修改

    void finishUseBuffer(int offset);
  //通过实例化的buffer调用，buffer所绑定的文件的偏移位置offset所对应的页得出
countRef将减一（少一个指针引用），当countRef为0时，bufferManager将不再确保该页被
pin在缓存区内。

    int getFreshPage();
  //通过实例化的buffer调用，buffer将从绑定的文件中找出一页可以用的空白页以供使用，返
回该空白页在文件中所对应的页号

    int pageNumber; //表示绑定的文件共有多少页，即文件大小为pageNumber*BLOCKSIZE
};
```

二、bufferManager使用方法：

1. 当决定启用bufferManager的时候应当调用initialMemory()函数，该函数将向系统申请一块缓存区，并且来对整个缓存区进行初始化；决定停用bufferManager的时候应当调用quitBufferManager()，该函数将将所有仍在缓存区的页全部写入对应的文件中，并注销掉整个缓存区。

2. bufferManager提供一个构造函数，通过文件名来构建一个对象buffer，bufferManager会自动将该对象buffer与传入的文件名进行绑定，使用这个对象的任何操作都将视为对该文件的特定操作。删除该对象之后bufferManager将解除绑定。

注意！可以看到bufferManager中大部分是静态函数，当创建了两个绑定不同文件的对象buffer1和buffer2时，他们使用的是一块共同的缓存区，即替换算法会对该两个对象所申请的页一视同仁的进行操作。

3. 使用示例：

```
bufferManager::initialMemory();
  //初始化整个bufferManager

bufferManager* buffer1 = new bufferManager(new string("/Users/
drinkingcoder/Documents/university/Database Design/indexManagerTest/
test.txt"));
  //创造一个绑定test.txt的对象buffer1

void pointer = buffer1->startUsePage(1);
  //bufferManager会将test.txt的第一页(4K大小)载入缓存区，并将它pin在缓存区内，不允许被替换出去，并返回该缓存内容对应的一个指针

buffer1->finishUsePage(1);
  //通知bufferManager对该页的使用已经结束，它将不被pin在缓存区内，可以被替换策略替换出去

delete buffer;
  //接触buffer的绑定

bufferManager::quitBufferManager();
  //将所有缓存区内容写进对应的文件内，注销掉整块缓存区
```

模块关键函数详细介绍与分析

1. bufferManager构造函数：

```
bufferManager::bufferManager(string* fileName)
   //使用文件名构造一个自己的buffer管理器，将该文件与实例化的buffer进行绑定
{
    fp = NULL;
    for(int i=0; i<files.size();i++) //检查这个文件指针是否已经被记录
        if(fileNames[i] == *fileName)
        {
            fp = files[i];  //已被记录则直接使用该文件指针即可
            break;
        }
    if(fp == NULL)  //该文件为新出现的文件，需要创建新的文件指针
    {
        fp = fopen(fileName->c_str(),"rb+");
        if(fp == NULL) //文件不存在，需要新建立一个文件
        {
            fp = fopen(fileName->c_str(),"a");
            fclose(fp);
            fp = fopen(fileName->c_str(),"rb+");
            if(fp == NULL){  //文件创建失败
                cout << "Failed To open File: " << *fileName << endl;
                delete fileName;
                return;
            }
        }
        fileNames.push_back(*fileName); //将新的文件放入容器
        files.push_back(fp);
    }
    this->fileName = fileName;
    fseek(fp, 0, SEEK_END);
    pageNumber = ftell(fp)/BLOCKSIZE; //计算这个文件共有多少页
}
```

分析：将一个实例化buffer与一个文件直接进行绑定，方便使用者对自己的文件进行想要的操作，同时也防止了使用者对无关的文件进行非法操作。

2. bufferManager的整个缓存区的初始化函数initialMemory：

```
int bufferManager::initialMemory() //完成所有需要的初始化
{
    index = 0;
    quitBufferManager();
    memset(countRef,0,sizeof(countRef));
    memset(timeStamp,0,sizeof(timeStamp));
    memset(bufferToFile,0,sizeof(bufferToFile));
    memset(bufferToPageNumber,0,sizeof(bufferToPageNumber));
    filePageNumberTobuffer.clear();
    files.clear();
    fileNames.clear();  //对所有内容进行初始化
    for(int tmp = 0; tmp<MAXBLOCKNUMBER ; tmp++)  //分别申请缓存区空间
        buffer[tmp] = (unsigned char*)malloc(BLOCKSIZE);
    return 1;
}
```

分析：对所有内容进行初始化，以便之后的操作。

3. bufferManager的使用结束函数quitBufferManager：

```
void bufferManager::quitBufferManager()
  //将所有内存页中的内容写入文件并销毁所有内容
{
    for ( map<pair<FILE*,int>,int>::iterator itr =
filePageNumberTobuffer.begin(); itr!=filePageNumberTobuffer.end();itr++)
        flushIntoDisk(itr->first.first, itr->first.second, itr->second);
  //将所有在内存块中的内容写入磁盘

    filePageNumberTobuffer.clear();
    for(vector<FILE*>::iterator itr = files.begin() ; itr!=files.end() ;
itr++)
        fclose(*itr);
  //关闭所有打开的文件

    for(int tmp = 0; tmp<MAXBLOCKNUMBER ; tmp++)
        free(buffer[tmp]);
  //释放申请的内存块

}
```

分析：将所有残留在缓存区中的内容都写入对应的文件，注销掉整个缓存区。

4. 执行替换算法的函数findAnAvailableBlock()

```
int bufferManager::findAnAvailableBlock()
   //采用LRU原理（clock算法）找到一个可以使用的块
{
    int tmp = 0;
    while(tmp<MAXBLOCKNUMBER*2)
                //设置搜索次数上限
    {
        tmp++;
        if(countRef[index] == 0)
                //当countRef>0时说明该内存块正被引用，被pin在内存内，无法被替换
        {
            if(timeStamp[index] == 0) return 1;
                //当timeStamp = 0 时代表可用
            timeStamp[index] = 0;
                //否则将timeStamp置为0，这样在下一圈就可以被搜索到
        }
        index++;
        if(index == MAXBLOCKNUMBER) index = 0;
    }
    return 0;
                //当指针循环两圈仍然找不到可以放的块，说明所有块都被pin在内存内
}
```

分析：使用孙建伶老师上课介绍的clock算法来执行替换策略，对于每一个刚刚被使用的模块都将timeStamp置为1，每次使用index寻找一页进行替换时，若当前块可替换（countRef=0）且timeStamp=0则使用该页进行替换，否则timeStamp=1 则将timeStamp置为0，这样下一回到达此处时将选用此页进行替换。

5. 通过bufferManager对文件内容进行操作函数startUsePage：

```
void* bufferManager::startUsePage(int pageNumber) //使用这一页内容
{
    map<pair<FILE*,int>,int>::iterator itr =
filePageNumberTobuffer.find(pair<FILE*,int>(fp,pageNumber));
    if(itr != filePageNumberTobuffer.end())
     //如果这一页原本就在内存中，则直接使用该页
    {
        countRef[itr->second]++;
        timeStamp[itr->second] = 1;
        index = itr->second;
        return (void*)(buffer[index]);
    } else {
        int index = findABlockToReplace(fp,pageNumber);
      //否则就找一页来进行替换
        if (index == -1) return  NULL;
        return (void*)(buffer[index]);
    }
}
```

分析：利用缓存区，可以有效的减少从硬盘中读取数据的次数，从而加快数据库的运行速度。

6. 结束对文件内容的操作finishUsePage：

```
void bufferManager::finishUsePage(int pageNumber)
 //这一页内容使用完毕，释放指针
{
    int index = getIndexFromFileAndPage(fp,pageNumber);
 //取得这一个文件页所在的内存指针
    countRef[index]-;
  //销毁引用
}
```

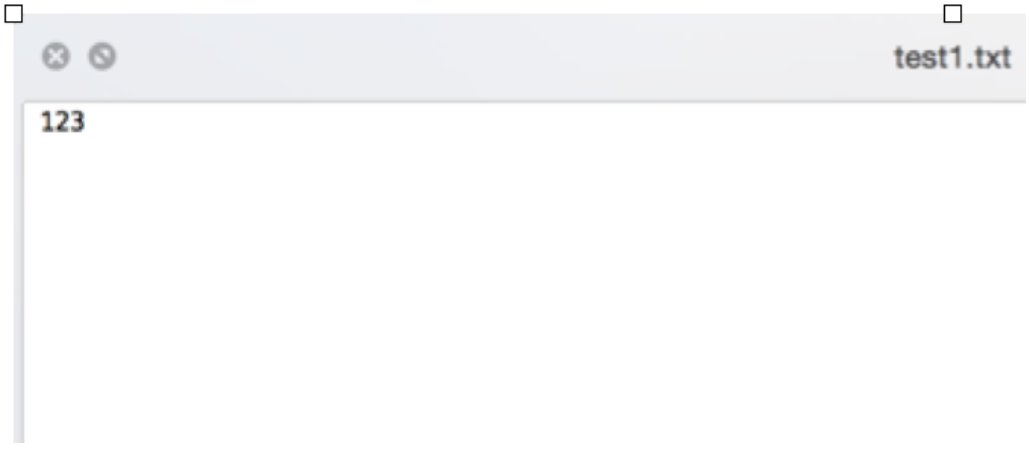
buffer模块测试

一、测试简单读写

1. 测试代码：

```
#include <iostream>
#include "bufferManager.hpp"

int main(int argc, const char * argv[]) {
    char filename[] =
"/Users/drinkingcoder/Documents/university/Database
Design/bufferManagerTest/test1.txt";
    bufferManager::initialMemory();
    bufferManager* buffer1 = new bufferManager(new
string(filename));
    char* str = (char*)(buffer1->startUsePage(0));
    str[0] = '1';
    str[1] = '2';
    str[3] = '3';
    bufferManager::quitBufferManager();
    return 0;
}
```

2. 测试结果：



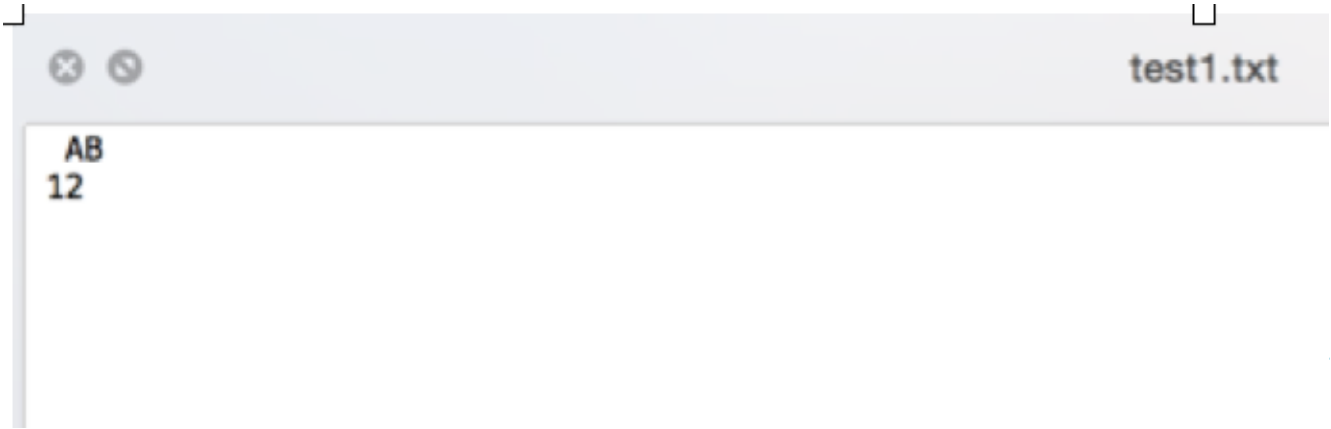
3. 分析：运行结果逻辑正确

二、测试getFreshPage函数

1. 测试代码：

```
#include <iostream>
#include "bufferManager.hpp"
using namespace std;

int main(int argc, const char * argv[]) {
    char filename[] =
"/Users/drinkingcoder/Documents/university/Database
Design/bufferManagerTest/test1.txt";
    bufferManager::initialMemory();
    bufferManager* buffer1 = new bufferManager(new
string(filename));
    int pN = buffer1->getFreshPage();
    char* str1 = (char*)(buffer1->startUsePage(pN));
    for(int tmp=0; tmp<BLOCKSIZE ; tmp++)
        str1[tmp] = ' ';
    str1[1] = 'A';
    str1[2] = 'B';
    char* str2 = (char*)(buffer1->getFreshPage());
    for(int tmp=0; tmp<BLOCKSIZE ; tmp++)
        str2[tmp] = ' ';
    str2[1] = '1';
    str2[2] = '2';
    bufferManager::quitBufferManager();
    return 0;
}
```

2. 测试结果：



3. 分析：运行结果逻辑正确，将AB和12分别写入两页中

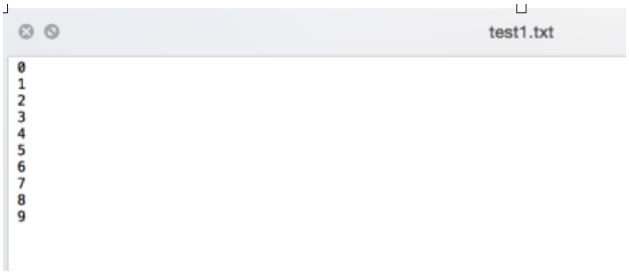
三、测试文件页替换策略

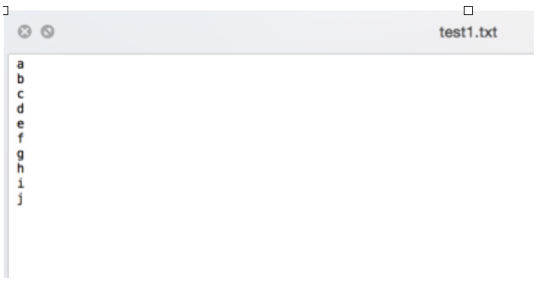
1. 测试代码：

```
#include <iostream>
#include "bufferManager.hpp"
using namespace std;

int main(int argc, const char * argv[]) {
    char filename[] =
"/Users/drinkingcoder/Documents/university/Database
Design/bufferManagerTest/test1.txt";
    bufferManager::initialMemory();
    bufferManager* buffer1 = new bufferManager(new
string(filename));
    char* str[10];
    for(int tmp = 0;tmp<10;tmp++)
    {
        if(tmp == buffer1->pageNumber) str[tmp] =
(char*)(buffer1->getFreshPage());
        else str[tmp] = (char*)(buffer1->startUsePage(tmp));
        for(int i = 0; i<BLOCKSIZE ;i++)
            str[tmp][i] = ' ';
        str[tmp][0] = '0' + tmp;
        buffer1->finishUsePage(tmp);
    }
    bufferManager::quitBufferManager();
    return 0;
}
```

```
#include <iostream>
#include "bufferManager.hpp"
using namespace std;

int main(int argc, const char * argv[]) {
    char filename[] =
"/Users/drinkingcoder/Documents/university/Database
Design/bufferManagerTest/test1.txt";
    bufferManager::initialMemory();
    bufferManager* buffer1 = new bufferManager(new
string(filename));
    char* str[10];
    for(int tmp = 0;tmp<10;tmp++)
    {
        if(tmp == buffer1->pageNumber) str[tmp] =
(char*)(buffer1->getFreshPage());
        else str[tmp] = (char*)(buffer1->startUsePage(tmp));
        for(int i = 0; i<BLOCKSIZE ;i++)
            str[tmp][i] = ' ';
        str[tmp][0] = 'a' + tmp;
        buffer1->finishUsePage(tmp);
    }
    bufferManager::quitBufferManager();
    return 0;
}
```

2. 测试结果:





3. 分析：如上所示，总共使用了两段代码，第一段代码往文件中前10页分别写入0~9数字，第二段代码在第一段代码运行之后将文件中前10页数字修改为字母，运行结果逻辑正确。

buffer模块总结

1. 通过对buffer模块的精心设计和细致实现，使我完全理解了数据库管理软件中低层的buffer是如何实现的，这也使得我在数据库系统设计课程设计的学习中对于两阶段锁协议，日志文件的作用等的理解更为深刻和透彻。
2. buffer模块使用了静态函数和实例结合的结构，使得我对于面向对象编程思想的理解有了更好的训练和更深入的理解
3. 为了与其他模块能够良好结合，buffer模块的接口设计完全独立，与外部其他模块的耦合性十分低，能够很好地结合和使用。