实验题目	缓冲区管理器实现			实验日期	2022.4.3
班级	1903103	学号	1190200208	姓名	李旻翀

CS33503 数据库系统实验

实验检查记录

实验结果的正确性(60%)		表达能力(10%)	
实验过程的规范性(10%)		实验报告(20%)	
加分(5%)		总成绩(100%)	

实验报告

一、实验目的（介绍实验目的）

1. 掌握数据库管理系统的存储管理器的工作原理。
2. 掌握数据库管理系统的缓冲区管理器的工作原理。
3. 使用 C++面向对象程序设计方法实现缓冲区管理器。

二、实验环境（介绍实验使用的硬件设备、软件系统、开发工具等）

系统：Ubuntu 18.04
调试工具：gdb，valgrind 等

三、实验过程（介绍实验过程、设计方案、实现方法、实验结果等）

1. buffer.cpp 中的主要变量

因为本次实验中主要的修改集中于buffer.cpp中，所以以下主要围绕着buffer.cpp中涉及的变量来讲解缓冲器的整体结构。

根据 BufMgr 类的构造函数，可以得到缓冲器的三个主要组成部分：

- bufDescTable：包含 numBufs 个页框（BufDesc）的数组，可以通过每一个 BufDesc 页框的属性来描述页框的实际状态。
- bufPool：包含 numBufs 个 page 的数组。代表实际的缓冲池。
- hashTable：将(File，Page)映射到页框中的哈希表。哈希函数如下：

```
int htsize = ((((int)(bufs * 1.2)) * 2) / 2) + 1;
```

缓冲区管理器的原理如下：

bufPool 是由一组固定大小的内存缓冲区(buffer)构成的数组，用于存放从磁盘读入内存的 page。缓冲池中每个固定大小的内存缓冲区称作页框。当磁盘上的页面被首次读入缓冲池时，缓冲池中的页面和磁盘上对应页面一样。一旦 DBMS 修改了缓冲池中该页面的内容，则缓冲池中的页面与它在磁盘上对应的页面就不再相同了。我们将缓冲池中被修改过的页面称为“脏”页面（对应 dirty 位为 true）。缓冲区管理器(BufMgr)用

于控制哪些页面驻留在缓冲池中。每当缓冲区管理器收到了一个页面访问请求，它会首先检查被请求的页面是否已经存在于缓冲池的某个页框中。如果存在，则返回指向该页框的指针；如果不存在，则缓冲区管理器会释放一个页框(如果页框中的页面是脏的，则需要将该页面先写回磁盘)，并将被请求的页面从磁盘读入刚刚释放的页框。

本实验中，缓冲区页面的替换策略选用时钟算法。

2. 对 buffer.cpp 的修改

下面介绍 buffer.cpp 中自行编写的具体函数。

1) 析构函数~BufMgr()：

实现先将脏页写回磁盘，然后清除占用内存的功能。注意在删除时为避免空指针出现，最好按指针的指向顺序删除。

```
// 将所有脏页写回磁盘，然后释放缓冲池、BufDesc表和哈希表占用的内存
BufMgr::~BufMgr()
{
    // 脏页全部写回磁盘
    for (FrameId i = 0; i < numBufs; i++)
    {
        if (bufDescTable[i].dirty && bufDescTable[i].valid)
        {
            bufDescTable[i].file->writePage(bufPool[i]);
            bufDescTable[i].dirty = false;
        }
    }

    // 按指向顺序删除，避免产生空指针
    delete hashTable;       // 删除页表
    delete[] bufPool;       // 删除 Buffer Pool
    delete[] bufDescTable; // 删除每一个页框的描述
}
```

2) void advanceClock()

通过参数 clockHand 在 0 ~ numBufs-1 之间循环自增，实现时钟算法中的表针转动的功能。每调用一次 advanceClock()，clockHand 在 0 ~ numBufs-1 之间自增 1。

3) void allocBuf(FrameId& frame)

使用时钟算法分配一个空闲页框。函数逻辑如实验指导书所示。按照实验指导书设计函数即可。由于代码过长，此处不作展示，仅说明一些要点：

- 页框中包含有效页面的删除操作：

```
// 如果被分配的页框中包含一个有效页面，则必须将该页面从页表中删除
if (bufDescTable[clockHand].valid)
{
    try
    {
        hashTable->remove(bufDescTable[clockHand].file, bufDescTable[clockHand].pageNo);
    }
    catch (HashNotFoundException &)
    {
    }
}
```

- 该函数不需要返回值，因为 frame 参数以指针形式传入，在函数内对其进行的

修改可以作用于全局。

4) void readPage(File* file, const PageId PageNo, Page*& page)
分页面在缓冲池中和页面不在缓冲池中两种情况。用 try catch 语句实现其功能。

```
// 上层读取页面
void BufMgr::readPage(File *file, const PageId pageNo, Page *&page)
{
    FrameId frame_num;
    // 页面在缓冲池中
    try
    {
        hashTable->lookup(file, pageNo, frame_num);
        bufDescTable[frame_num].refbit = true;
        bufDescTable[frame_num].pinCnt++;
        page = frame_num + bufPool; // 通过参数page返回指向该页框的指针
    }
    // 页面不在缓冲池中
    catch (HashNotFoundException &)
    {
        allocBuf(frame_num);                          // 分配一个空闲的页框
        bufPool[frame_num] = file->readPage(pageNo); // 将页面从磁盘读入刚刚分配的空闲页框
        hashTable->insert(file, pageNo, frame_num);  // 将该页面插入哈希表
        bufDescTable[frame_num].Set(file, pageNo);   // 调用Set()方法正确设置页框的状态
        page = frame_num + bufPool;                   // 通过参数page返回指向该页框的指针
    }
}
```

在 try 语句中，若 lookup 函数捕捉到 HashNotFoundException，则转入 catch 语句块中的异常处理程序，实现了两种情况的 readPage 功能。

5) void unPinPage(File* file, const PageId PageNo, const bool dirty)
实现将缓冲区中包含(file, PageNo)表示的页面所在的页框的 pinCnt 值减 1 的功能。仍然通过 lookup 函数实现查找特定页面的功能。该函数设计并无难点，按照实验指导书的程序逻辑设计即可。

6) void allocPage(File* file, PageId& PageNo, Page*& page)
按照实验指导书编写程序即可。如图所示，展示每一步实现的功能。

```
// 分配页面
void BufMgr::allocPage(File *file, PageId &pageNo, Page *&page)
{
    FrameId frame_num;

    Page new_page = file->allocatePage(); // 在file文件中分配一个空闲页面
    allocBuf(frame_num);                  // 在缓冲区中分配一个空闲的页框
    bufPool[frame_num] = new_page;

    pageNo = new_page.page_number(); // 通过pageNo参数返回新分配的页面的页号
    page = frame_num + bufPool;      // 通过page参数返回指向缓冲池中包含该页面的页框的指针

    hashTable->insert(file, pageNo, frame_num); // 在哈希表中插入一条项目
    bufDescTable[frame_num].Set(file, pageNo);  // 调用Set()方法正确设置页框的状态
}
```

7) void disposePage(File* file, const PageId pageNo)
通过 deletePage 从 file 中删除该页面。在此之前，需要根据页面是否在缓冲池中进行不同的操作。具体实现通过 try catch 语句完成。

```
// 从文件file中删除页号为pageNo的页面
void BufMgr::disposePage(File *file, const PageId PageNo)
{
    FrameId frame_num;
    // 若该页面在缓冲池中
    try
    {
        hashTable->lookup(file, PageNo, frame_num); // 寻找该页面
        hashTable->remove(file, PageNo);            // 若找到，则从哈希表中删除该页面
        bufDescTable[frame_num].Clear();            // 将该页面所在的页框清空
    }
    catch (HashNotFoundException &) // 若该页面不在缓冲池中，则不做操作
    {
    }
    file->deletePage(PageNo); // 从file中删除该页面
}
```

8) void flushFile(File* file)

通过 for 循环实现对缓冲区页框的遍历。通过 bufDescTable[i].file == file 条件判断当前页框的页是否属于某个文件。另外需要注意，对脏页面/无效页/页面被固定三种特殊情况的处理需要在调用 remove 和 clear 函数对哈希表和页框进行重置与删除操作前完成。

具体代码如下：

```
// 扫描页面
void BufMgr::flushFile(const File *file)
{
    // 遍历，检索缓冲区中所有属于文件file的页面
    for (FrameId i = 0; i < numBufs; i++)
    {
        if (bufDescTable[i].file == file)
        {
            // 检索到文件file的某个无效页或文件file的某些页面被固定住(pinned)，抛出BadBufferException异常
            if (!bufDescTable[i].valid || bufDescTable[i].pinCnt > 0)
                throw BadBufferException(i, bufDescTable[i].dirty, bufDescTable[i].valid, bufDescTable[i].refbit);
            // 如果页面是脏的，则调用file->writePage()将页面写回磁盘，并将dirty位置为false
            if (bufDescTable[i].dirty)
            {
                bufDescTable[i].file->writePage(bufPool[i]);
                bufDescTable[i].dirty = false;
            }
            // 将页面从哈希表中删除
            hashTable->remove(file, bufDescTable[i].pageNo);
            // 调用BufDesc类的Clear()方法将页框的状态进行重置
            bufDescTable[i].Clear();
        }
    }
}
```

3. 新增测试用例

在此次实验给定的 6 个测试用例以外，我新增了 1 组测试用例 test7，用于测试 pageNo 过大，造成 InvalidPageException 的情况。具体代码如下：

```
// pageNo过大，造成InvalidPageException
void test7()
{
    try
    {
        bufMgr->readPage(file1ptr, 200, page);
        PRINT_ERROR(
"ERROR :: PageNo out of range. Exception should have been thrown before execution reaches this point.");
    }
    catch (InvalidPageException e)
    {
    }

    std::cout << "Test 7 passed"
              << "\n";
}
```

4. 编译与结果

若要通过编译，首先需要在 main.cpp 中将该部分代码注释掉，否则会报错（目前原因未知）：

```
        // Iterate through all records on the page.
        // for (PageIterator page_iter = (*iter).begin();
        //   page_iter != (*iter).end();
        //   ++page_iter)
        // {
        //  std::cout << "Found record: " << *page_iter
        //            << " on page " << (*iter).page_number() << "\n";
        // }
```

然后，在 BufMgr 文件夹下打开终端，输入 make 执行 Makefile，再执行 src 文件夹下的 badgerdb_main 文件查看测试结果。对于总共 7 个测试用例，运行结果如下：

```
mincoolee@mincoolee:~/桌面/SharedFile/BufMgr$ make
cd src;\
g++ -std=c++0x *.cpp exceptions/*.cpp -I. -Wall -o badgerdb_main
mincoolee@mincoolee:~/桌面/SharedFile/BufMgr$ ./src/badgerdb_main
Third page has a new record: world!


start tests.
Test 1 passed
Test 2 passed
Test 3 passed
Test 4 passed
Test 5 passed
Test 6 passed
Test 7 passed

Passed all tests.
mincoolee@mincoolee:~/桌面/SharedFile/BufMgr$
```

为方便起见，我在 BufMgr 文件夹下新建了 run.sh 脚本，执行该脚本即可完成 make 与执行 badgerdb_main 的功能。

实验题目	缓冲区管理器实现			实验日期	2022.4.3
班级	1903103	学号	1190200208	姓名	李旻翀

四、实验结论（总结实验发现及结论）

通过此次实验，熟悉了数据库管理系统的存储管理器和缓冲区管理器的工作原理，了解了缓冲池的具体实现方法，同时，使用 C++编程的能力得到了不小的提高。