哈尔滨工业大学

<<数据库系统>>

实验报告二

(2022 年度春季学期)

姓名：	冯开来
学号：	1190201215
学院：	计算机学院
教师：	程思瑶

实验二（project 2）

一、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

该实验在虚拟机的 Linux 系统下实现。版本为：Linux version 5.11.0-49-generic (builddalcy02-and64-054)(gcc (Ubuntu 10.3.0-1ubuntu1) 10.3.0, GNU ld (GNU Binutils for ubuntu)2.36.1)

三、实验过程及结果

在整个实验所给的源代码中，我们只需要完成其中的 buffer.cpp，实现缓冲区管理器类就可以了。同时也可以在主函数 main.cpp 中加入更多的测试用例。buffer.cpp 的具体实现过程如下：

其中的 BufMgr(const int bufs)和~BufMgr()已经给出，他们的作用是。为缓冲池分配一个包含 bufs 个页面的数组，并为缓冲池的 BufDesc 表分配内存。当缓冲池的内存被分配后，缓冲池中所有页框的状态被置为初始状态。接下来，将记录缓冲池中 当前存储的页面的哈希表被初始化为空。以及。将缓冲池中所有脏页写回磁盘，然后释放缓冲池、BufDesc 表和哈希表占用的内存。

void advanceClock()的实现：

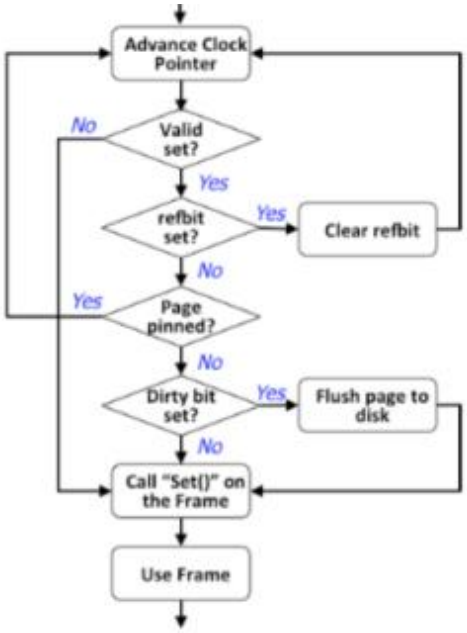
由于该函数的作用是顺时针旋转时钟算法中的表针，将其指向缓冲池中下一个页框。因此只需要对页框序号 clockHand 不断进行加一操作，并对总页框数进行取模操作即可。

void allocBuf(FrameId& frame)的实现：

首先判断该页框的 valid 值，若 valid=false，则说明该页框为空，可以使用，便将页框序号返回即可跳出循环，结束查找。如果 valid=1，继续判断该页框的 refbit 值，若该值为 1，说明该页框最近有被读取，因此将其 refbit 值改为 0 后，进入下一次循环（将页框序号加一后继续重复该判断过程）。如果其 refbit 值为 0，则继续判断其是否被 pinned。如果其已经被 pinned，则将记录页框被 pinned 的变量值+1。若该变量值达到页框总数，则表示所有的页框都处于 pinned 状态，此时报错。如果该页框没有被 pinned，则判断该页框有没有处于被重写过但没写入磁盘的状态（也就是 dirty 是否为 1）。若 dirty 值为 1，则说明该页框中页面是脏的，要将其先写回磁盘后将 dirty 值更改为 0.若 dirty 值为零则可以将该页框作为要写入的页框（即使该页框非空）。当然，由于该页

框含有有效页面，所以则必须将该页面从哈希表中删除。最后，分配的页框的编号通过参数 frame 返回。

其实这个函数的具体实现思路，就是报告中的那个程序框图。只需要将它看懂并且按照程序框图将代码写出来就好了。具体步骤如图：




```
void BufMgr::allocBuf(FrameId &frame)
{
    bool flag = true;
    for (uint32_t i = 0; i < numBufs || !flag; ++i)
    {
        advanceClock();
        if (!bufDescTable[clockHand].valid)
        {
            frame = clockHand;
            return ;
        }
        if (bufDescTable[clockHand].pinCnt)
            continue;
        flag = false;
        if (bufDescTable[clockHand].refbit)
        {
            bufDescTable[clockHand].refbit = false;
            continue;
        }
        if (bufDescTable[clockHand].dirty)
        {
            bufDescTable[clockHand].file->writePage(bufPool[clockHand]);
            bufDescTable[clockHand].dirty = false;
        }
        hashTable->remove(bufDescTable[clockHand].file,
            bufDescTable[clockHand].pageNo);
        bufDescTable[clockHand].Clear();
        frame = clockHand;
        return ;
    }
    throw BufferExceededException();
}
```

void readPage(File* file, const PageId PageNo, Page*& page) 的实现：

根据实验指导书提示，首先调用哈希表的 lookup()方法检查待读取的页面(file, PageNo)是否已经在缓冲池中。如果页面在缓冲池中。在这种情况下，将页框的 refbit 置为 true，并将 pinCnt 加 1。最后，通过参数 page 返回指向该页框的指针。如果页面不在缓冲池中。在这种情况下，调用 allocBuf()方法分配一个空闲的页框。然后， 调用 file->readPage()方法将页面从磁盘读入刚刚分配的空闲页框。接下来，将该页面插入到 哈希表中，并调用 Set()方法正确设置页框的状态，Set()会将页面的 pinCnt 置为 1。最后，通过参数 page 返回指向该页框的指针。具体实现方法如下图：

```
void BufMgr::readPage(File *file, const PageId pageNo, Page *&page)
{
    FrameId frame;
    try
    {
        hashTable->lookup(file, pageNo, frame);
        bufDescTable[frame].refbit = true;
        bufDescTable[frame].pinCnt++;
    }
    catch(HashNotFoundException&)  // 页面不在缓冲池
    {
        allocBuf(frame);  // 分配一个新的空闲页框
        bufPool[frame] = file->readPage(pageNo);  // 从磁盘读入到这个页框
        hashTable->insert(file, pageNo, frame);  // 该页面插入哈希表
        bufDescTable[frame].Set(file, pageNo);  // 设置页框状态
    }
    page = bufPool + frame;  // 通过pege返回指向该页框的指针
}
```

void unPinPage(File* file, const PageId PageNo, const bool dirty)实现：

首先通过判断调用哈希表的 lookup()方法检查待读取的页面是否在缓冲池中，如果在，就将该页面所在页框的 pinCnt 值减 1。如果参数 dirty 等于 true，则 将页框的 dirty 位置为 false。如果 pinCnt 值已经是 0，则抛出 PAGENOTPINNED 异常。

void allocPage(File* file, PageId& PageNo, Page*& page)实现：

如指导书所说，首先调用 file->allocatePage()方法在 file 文件中分配一个空闲页面，file->allocatePage()返回 这个新分配的页面。然后，调用 allocBuf()方法在缓冲区中分配一个空闲的页框。接下来，在哈希表中插入一条项目，并调用 Set()方法正确设置页框的状态。该方法既通过 pageNo 参数返回新分配的页面的页号，还通过 page 参数返回指向缓冲池中包含该页面的页框的指针。该指针为 bufPool + frame。

```
void BufMgr::allocPage(File *file, PageId &pageNo, Page *&page)
{
    FrameId frame;
    allocBuf(frame);  // 分配一个新的页框
    bufPool[frame] = file->allocatePage(); // 返回一个空闲页面
    pageNo = bufPool[frame].page_number(); //
    hashTable->insert(file, pageNo, frame); // 哈希表中插入该页面
    bufDescTable[frame].Set(file, pageNo);  // 设置页框状态
    page = bufPool + frame;  // 通过page参数返回指向缓冲池中包含该页面的页框的指针
}
```

void disposePage(File* file, const PageId pageNo)实现：

判断页号为 pageNo 的页面是否在缓冲池中，如果在，则将该页面所在的页框清空并从哈希表中删除该页面。

void flushFile(File* file)实现：

扫描 bufTable，检索缓冲区中所有属于文件 file 的页面。首先判断该页面是否有效以及该页面是否被固定住。如果页面无效或者被固定住，则抛出出 BadBufferException 异常。之后判断该页面 dirty 是否为 1，为 1 则将该页面内容写回磁盘后将 dirty 值变为 false。最后将页面从哈希表中删除后，调用 BufDesc 类的 Clear()方法将页框的状态进行重置。

实验结果：

我们以 test4 为例。

```
void test4()
{

    bufMgr->allocPage(file4ptr, i, page);
    bufMgr->unPinPage(file4ptr, i, true);
    try
    {
        bufMgr->unPinPage(file4ptr, i, false);
        PRINT_ERROR("ERROR :: Page is already unpinned. Exception should have been thrown before execution reaches this point.");
    }
    catch(PageNotPinnedException e)
    {
    }

    std::cout << "Test 4 passed" << "\n";

}
```

当执行执行 unPinPage()这个函数时，收 i 西安判单 file4ptr 是否在对应的页框中，如果不在，直接抛出异常。如果在的话继续判断他的 pinCnt 值，如果 pinCnt 大于 0，说明此时有应用正在使用，结束使用的时候 unPin 使 pinCnt 减一，如果 pinCnt 小于等于 0，则不能进行 unPin 操作，抛出异常。在这个 test 中，file4ptr 的 pinCnt 等于 0，不能进行 unPin，所以会抛出异常。

最后，所有的测试样例：

四、实验心得

试验做起来还算比较轻松。实验要求很清晰且思路也很明确。指导书的讲解很详细也很透彻。通过这次试验我对缓冲区管理这方面的知识有了更加深刻的理解，对知识点也有了更加牢固地掌握。相当于是在实验中不断复习学过的知识了。