

哈尔滨工业大学

<<数据库系统>>

实验报告三

(2023 年度秋季学期)

姓名:	张智雄
学号:	2021112845
学院:	计算学部
教师:	程思瑶

实验三 缓冲区管理器实现

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

Vmware Workstation 17 + Ubuntu22.04 LTS 操作系统、gcc 11.4.0 编程环境。

三、实验过程及结果

本实验拟基于提供的 BadgerDB 数据库管理系统实现一个简单的缓冲区管理器，实验过程及结果如下：

3.1 实验准备

首先简单学习一下存储管理器和缓冲区管理器的原理：

3.1.1 BadgerDB 的 I/O 层

BadgerDB 数据库管理系统的最底层是 I/O 层，它为系统上层提供了创建/删除文件、分配/释放文件页面、读/写文件页面等功能。I/O 层由两个 C++类实现，分别是文件类(File)和页面类(Page)。这两个类使用 C++异常来处理系统运行过程中发生的异常事件。

3.1.2 BadgerDB 的缓冲区管理器

数据库缓冲池(buffer pool)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数据库页面(page，也称作磁盘块)。

磁盘上的数据库通常比缓冲池大，因此任何时候只有一部分数据库页面可以被读入缓冲区。缓冲区管理器(buffer manager)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；
- 否则，则缓冲区管理器会释放一个页框(如果页框中的页面被修改过，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。

3.1.3 BadgerDB 的缓冲区页面替换策略

当需要从缓冲池获取一个空闲页框时，有很多方法来确定替换掉缓冲池中哪个页面。尽管 LRU 是最常用的策略之一，但它的开销大，因此本次实验中使用时钟算法(the clock algorithm)来近似 LRU 算法，其优点是执行速度非常快。

图 1（左）给出了概念上的缓冲池布局，其中每个正方形表示一个页框。假设缓冲池中包含 numBuf 个页框，编号从 0 到 numBufs-1。所有页框在概念上被组织成一个环形列表。

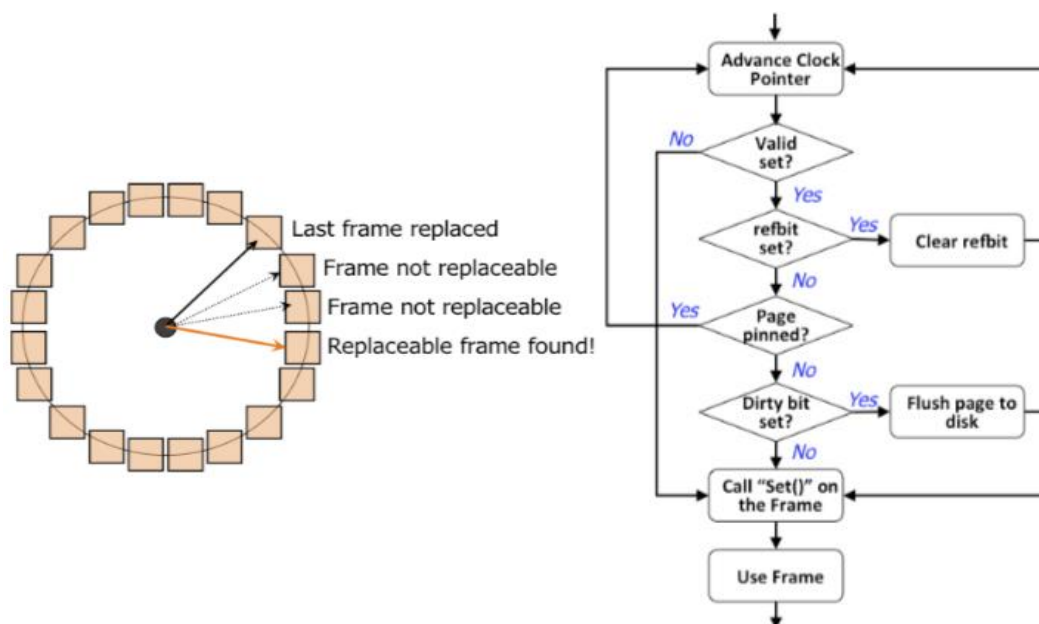


图 1 Buffer Manager 的结构与算法（左为页框环形列表，右为时钟算法的流程）

而时钟算法的具体流程如图 1（右）所示：

- ◆ 每个页框带有 1 位引用位，称作 refbit。每当缓冲池中一个页面被访问了(通过调用缓冲区管理器的 readPage()函数)，则该页框的 refbit 被置为 1。
- ◆ 表针指向缓冲池中的一个页框, 用一个 0 到 numBufs-1 的整数来记录被表针指向的页框。
- ◆ 表针顺时针转动，其实现方法是将表针指向的页框号变量加 1，再对 numBufs 取模。每当表针指向一个页框时，算法检查该页框的 refbit 的值，并将 refbit 置为 0。
 - a) 如果 refbit 此前为 1，则该页框页面“最近”被访问过，不替换该页框中的页面；
 - b) 如果 refbit 此前为 0，则选择该页框页面进行替换(假设该页面没有被固定)。

3.2 实验过程

BadgerDB 的缓冲区管理器由 3 个类(BufMgr、BufDesc 和 BufHashTbl)实现。BufMgr 类只有一个实例。这个类的主要构成要素是缓冲池，即 numBufs 个页框构成的数组，每个页框的大小为一个页面的大小。除缓冲区数组外，BufMgr 类的实例还包含 numBufs 个 BufDesc 类的实例，用于描述缓冲池中各页框的状态。

BufMgr 类的实例使用一个**哈希表**来记录当前存储在缓冲池中的页面。该哈希表由 BufHashTbl 类的实例实现，同时该实例是 BufMgr 类的私有成员变量。

本次实验中主要补充完善 BufMgr 类中的方法，实现一个基于时钟算法的缓冲池管理器。

3.1.1 构造方法 BufMgr(const int bufs)

这是 BufMgr 类的构造函数，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，缓冲池中所有的页框的状态被置为初始状态。而后初始化缓冲池中当前存储的页面的哈希表为空。

```
BufMgr::BufMgr(std::uint32_t bufs)
: numBufs(bufs) {
    bufDescTable = new BufDesc[bufs];    // 申请 bufs 个 BufDesc 对象的数组 bufDescTable

    for (FrameId i = 0; i < bufs; i++){
        bufDescTable[i].frameNo = i;      // 为每个帧初始化 frameNo 为循环索引 i
        bufDescTable[i].valid = false;    // 为每个帧初始化 valid 为 false，表示这些帧初始时
        都是无效的
    }

    bufPool = new Page[bufs];             // 实际的缓冲池,包含 bufs 个 Page 对象的数组
    int htsize = (((int) (bufs * 1.2)) * 2) / 2 + 1;    // 计算哈希表的大小 htsize
    hashTable = new BufHashTbl (htsize);    // allocate the buffer hash table

    clockHand = bufs - 1;    // 初始化时钟算法的 clockHand 为缓冲池中的最后一个帧的索引
}
```

3.1.2 析构方法 ~BufMgr()

这是 BufMgr 类的析构函数，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。

```
BufMgr::~BufMgr() {
    for (FrameId i = 0; i < numBufs; i++) {
        // 将缓冲池中的所有脏页写回磁盘
        if (bufDescTable[i].dirty==1)
            bufDescTable[i].file->writePage(bufPool[i]);
    }
    delete[] bufPool;    //释放缓冲池
    delete[] bufDescTable;    //释放 bufDescTable
    delete hashTable;    //释放哈希表
}
```

3.1.3 时钟旋转方法 advanceClock()

顺时针旋转时钟算法中的表针，将其指向缓冲池中的下一个页框。将 $clock+1$ 然后对 $numBufs$ 取模得到下一个页框号。

```
void BufMgr::advanceClock(){
    clockHand = (clockHand + 1) % numBufs;    //顺时针旋转表针，指向下一个页框
}
```

3.1.4 空闲页框分配方法 allocBuf(FrameId& frame)

使用时钟算法分配一个空闲页框。如果页框中的页面是脏的，则需要将脏页先写回磁盘。如果缓冲池中所有页框都被固定(pinned)，则需要抛出异常 `BufferExceededException`。如果被分配的页框中包含一个有效页面，则必须将该页面从哈希表中删除。最后，分配的页框的编号通过参数 `frame` 返回。

```
void BufMgr::allocBuf(FrameId & frame)
{
    std::uint32_t pincount = 0;
    while(pincount<numBufs){
        advanceClock();
        // 此页框无效，通过 frame 返回该页框号
        if (!bufDescTable[clockHand].valid){
            frame = clockHand;
            return ;
        }
        // 若页框有效
        else {
            //若引用位为 1, refbit 置零
            if(bufDescTable[clockHand].refbit){
                bufDescTable[clockHand].refbit=false;
                continue;
            }
            else {
                //若引用位为 0, pinCnt>0, 则被固定页面数+1
                if(bufDescTable[clockHand].pinCnt>0){
                    pincount++;
                    continue;
                }
                //若引用位为 0, pinCnt=0 且页脏，则将该页写回磁盘，并将 dirty 位置 0
                else if(bufDescTable[clockHand].dirty){
                    bufDescTable[clockHand].dirty = false;
                    bufDescTable[clockHand].file->writePage(bufPool[clockHand]);
                }

                try{
                    // 从哈希表中移除当前缓冲帧对应的页

```

```

hashTable->remove(bufDescTable[clockHand].file,bufDescTable[clockHand].pageNo);
                bufDescTable[clockHand].Clear();    // 初始化当前缓冲帧
                frame = clockHand;    // 将当前帧号通过 frame 参数返回
            }

            // 处理哈希表未找到异常，如果有需要的话
            catch(HashNotFoundException e){

            }

            return;
        }

    }

    // 如果 pincount 超过了 numBufs，说明所有的页框都被固定住了，抛出缓冲区超出限制异常
    if(pincount >= numBufs)
        throw BufferExceededException();
}

```

3.1.5 页访问方法 readPage(File* file, const PageId pageNo, Page*& page)

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。如果该页面已经在缓冲池中，则通过参数 page 返回指向该页面所在的页框的指针；如果该页面不在缓冲池中，则哈希表的 lookup()方法会抛出 HashNotFoundException 异常。

具体而言，根据 lookup()的返回结果，处理以下两种情况：

a) 页面在缓冲池中：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。

b) 页面不在缓冲池中：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。

```

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++;
    }
    // 页面不在缓冲池
    catch(HashNotFoundException&){
        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 page 返回指向该页框的指针
}

```

3.1.6 文件访问清除方法 unPinPage(File* file, const PageId pageNo, const bool dirty)

将缓冲区中包含(file, pageNo)表示的页面所在的页框的 pinCnt 值减 1。如果参数 dirty 等于 true，则将页框的 dirty 位置为 true。如果 pinCnt 值已经是 0，则抛出 PAGENOTPINNED 异常。如果该页面不在哈希表中，则什么都不用做。

```

void BufMgr::unPinPage(File* file, const PageId pageNo, const bool dirty)
{
    FrameId frame;
    try{
        hashTable->lookup(file, pageNo, frame);    // 找到(file,PageNo)所在页框
        if (!bufDescTable[frame].pinCnt)
            throw PageNotPinnedException(file->filename(),pageNo, frame);
        bufDescTable[frame].pinCnt--;    // 表示页面所在的页框的 pinCnt 减一
        if (dirty)
            bufDescTable[frame].dirty = dirty;    // 将页框的 dirty 设置 true
    }
    catch(HashNotFoundException&){
    }
}

```

3.1.7 文件页面分配方法 allocPage(File* file, PageId& pageNo, Page*& page)

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。

```

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
}

```

3.1.8 页面删除方法 disposePage(File* file, const PageId pageNo)

该方法从文件 file 中删除页号为 pageNo 的页面。在删除之前，如果该页面在缓冲池中，需要将该页面所在的页框清空并从哈希表中删除该页面。

```
void BufMgr::disposePage(File* file, const PageId pageNo){
    FrameId frame;
    try {
        hashTable->lookup(file, pageNo, frame); // 如果页面在缓冲池中
        hashTable->remove(file, pageNo);        // 将页框清空并从哈希表中删除页面
        bufDescTable[frame].Clear();
    }
    catch(HashNotFoundException){
    }
    file->deletePage(pageNo);                    // 删除页号为 pageNo 的页面
}
```

3.1.9 文件删除方法 flushFile(File* file)

扫描 bufTable，检索缓冲区中所有属于文件 file 的页面。对每个检索到的页面，进行如下操作：

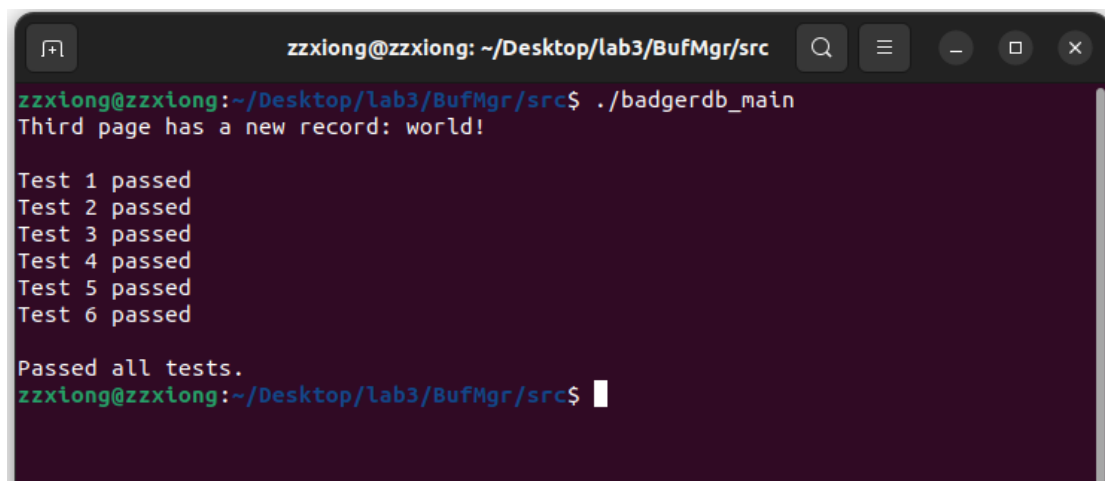
- 如果页面是脏的，则调用 file->writePage()将页面写回磁盘，并将 dirty 位置为 false;
- 将页面从哈希表中删除;
- 调用 BufDesc 类的 clear()方法将页框的状态进行重置。

如果文件 file 的某些页面被固定住(pinned)，则抛出 BadBufferException 异常。如果检索到文件 file 的某个无效页，则抛出 BadBufferException 异常。

```
void BufMgr::flushFile(const File* file) {
    for (uint32_t i = 0; i < numBufs; ++i){
        if (bufDescTable[i].file == file) {
            if (!bufDescTable[i].valid) // 无效页
                throw BadBufferException(i, bufDescTable[i].dirty, bufDescTable[i].valid, bufDescTable[i].refbit);
            if (bufDescTable[i].pinCnt) // 被固定
                throw PagePinnedException(file->filename(), bufDescTable[i].pageNo, i);
            // 如果页面是脏的，则写回磁盘
            if (bufDescTable[i].dirty) {
                bufDescTable[i].file->writePage(bufPool[i]);
                bufDescTable[i].dirty = false;
            }
            hashTable->remove(file, bufDescTable[i].pageNo);
            bufDescTable[i].Clear();
        }
    }
}
```


3.3 实验结果

进入 BufMgr 目录下打开终端进行编译\$make all 后在 src 目录下运行可执行文件 badgerdb_main 结果如下：



```
zzxiong@zzxiong: ~/Desktop/lab3/BufMgr/src
zzxiong@zzxiong:~/Desktop/lab3/BufMgr/src$ ./badgerdb_main
Third page has a new record: world!

Test 1 passed
Test 2 passed
Test 3 passed
Test 4 passed
Test 5 passed
Test 6 passed

Passed all tests.
zzxiong@zzxiong:~/Desktop/lab3/BufMgr/src$
```

图 2 实验结果

可以看到，6 个测试全部通过。

四、实验心得

4.1 问题解决

4.1.1 数组访问越界问题

在最初运行 badgerdb_main 的时候出现 Segmentation fault(core dumped)错误，经过一番 bug 排查发现是 while 循环的控制表达式“pincount <= numBufs”出现了问题，应该将其改成“pincount < numBufs”，不然就会导致内存数组的越界访问错误。

4.2 实验收获

经过本次实验，对课上中所学的数据库管理系统的各部分功能以及时钟算法有了更深的理解，并能自己动手编程实现。了解了数据库管理系统的存储管理器的工作原理。进一步熟悉了 C++面向对象程序设计方法。