实验1：缓冲池管理器实现

⼀、实验⽬的

1. 掌握Rucbase缓冲池⻚⾯替换策略的实现⽅法。

2. 掌握Rucbase缓冲池管理器的实现⽅法。

二、相关知识

1. 缓冲池的组成

2. 缓冲池的⻚⾯替换策略

3. 缓冲池访问请求的处理⽅法

三、实验内容

本实验包括2项任务。

**任务1：缓冲池⻚⾯替换策略实现**

补全 LRUReplacer 类，实现最近最少使⽤（Least Recently Used, LRU）⻚⾯替换策略。 LRUReplacer 类继承了 Replacer 类。当缓冲池没有空闲⻚⾯时，缓冲池管理器需要使⽤ Replacer 类实现的⻚⾯替换策略选择⼀个⻚⾯进⾏淘汰。

**1.1理解 LRUReplacer 类的设计，并回答下列问题：**

1. LRUlist\_ 的作⽤是什么？

LRUlist\_ 是一个双向链表（std::list<frame\_id\_t>），用于按照LRU（最近最少使用）的顺序存储所有未被固定（unpinned）的帧编号。

具体作用：

• 链表头部存储最近被访问（最近被unpin）的帧

• 链表尾部存储最久未被访问（最早被unpin）的帧

• 当需要选择victim页面时，从尾部取出最久未被访问的帧进行淘汰

• 当页面被unpin时，将其加入到链表头部，表示最近被访问

1. LRUhash\_ 的作⽤是什么？

LRUhash\_ 是一个哈希表（std::unordered\_map<frame\_id\_t, std::list<frame\_id\_t>::iterator>），用于建立帧编号到其在LRU链表中位置的快速映射。

具体作用：

• 键：帧编号（frame\_id\_t）

• 值：该帧在LRUlist\_链表中的迭代器位置

• 提供O(1)时间复杂度的查找和删除操作

• 当需要将某个帧从LRU链表中移除时，可以快速定位其在链表中的位置

3. LRUlist\_ 和 LRUhash\_ 的关系是什么？

两者配合实现了高效的LRU（最近最少使用）替换策略：

互补关系：

• LRUlist\_ 维护帧的访问顺序，支持顺序遍历和头尾操作

• LRUhash\_ 提供快速的随机访问能力

协同工作：

• 插入操作：当帧被unpin时，同时在LRUlist头部插入帧编号，在LRUhash中记录该帧在链表中的位置

• 删除操作：当帧被pin时，通过LRUhash快速找到帧在链表中的位置，然后从LRUlist中删除

• victim选择：直接从LRUlist\_尾部取出最久未被访问的帧

时间复杂度优化：

• 如果只有链表，删除中间元素需要O(n)时间

• 如果只有哈希表，无法维护访问顺序

• 两者结合实现了所有操作的O(1)时间复杂度

**1.2在这一部分，你需要列举并介绍你实现的全部方法。你需要按照方法所在的类进行组织。在介绍每个方法的具体实现时，需要包含以下内容：**

1. 方法的声明。给出方法的声明（注意：是方法声明，不是方法定义）。如果这个方法是你自己声明的，请说明它的功能以及为何要声明这个方法，何时调用这个方法。
2. 方法实现思路。根据方法实现的难度，可以采用不同的介绍形式。对于简单的方法，简要介绍方法的实现思路即可。对于复杂的方法，如果执行过程非常复杂，可以借助流程图或伪代码进行介绍。
3. 方法实现难点。如果你在实现这个方法的过程中遇到了较大的困难，不妨介绍一下你遇到的是什么困难，你最终的解决办法是什么。

Eg:

1.2.1：

void DiskManager::write\_page(int fd, page\_id\_t page\_no, const char \*offset, int num\_bytes);

**1.方法声明：**

方法名：write\_page

返回类型：void

功能：在对应文件的对应页面写入规定数量的数据

参数列表：

|  | **fd** | **page\_no** | **\*offset** | **num\_bytes** |
| --- | --- | --- | --- | --- |
| 类型 | int | page\_id\_t ：int32\_t | const char | int |
| 含义 | 文件描述符 | 页号 | 存放数据 | 数据的bit数 |

**2.方法实现思路：**

lseek()定位到文件头，通过(fd,page\_no)可以定位指定页面及其在磁盘文件中的偏移量

调用write()函数

如果写入到数据数量和num\_bytes不等，注意处理异常

**3.遇到困难….**

bool LRUReplacer::victim(frame\_id\_t\* frame\_id);

**1.方法声明：**

方法名：victim

返回类型：bool

功能：使用LRU策略删除一个victim frame，并返回该frame的id

参数列表：

|  | **frame\_id** |
| --- | --- |
| 类型 | frame\_id\_t\* |
| 含义 | 被移除的frame的id，如果没有frame被移除返回nullptr |

**2.方法实现思路：**

使用std::scoped\_lock对latch\_加锁，保证线程安全

检查LRU链表LRUlist\_是否为空，如果为空则没有可淘汰的页面，返回false

选择链表尾部的frame作为victim（最久未被访问的页面）

从链表中移除该元素：LRUlist\_.pop\_back()

从哈希表中移除该元素：LRUhash\_.erase(victim\_frame)

将victim frame id赋值给输出参数\*frame\_id

返回true表示成功淘汰了一个页面

**3.实现难点：**

主要难点在于理解LRU策略的数据结构设计：链表头部存放最近访问的页面，尾部存放最久未访问的页面。因此淘汰时选择尾部元素。

void LRUReplacer::pin(frame\_id\_t frame\_id)

**1.方法声明：**

方法名：pin

返回类型：void

功能：固定指定的frame，即该页面无法被淘汰

参数列表：

|  | frame\_id |
| --- | --- |
| 类型 | frame\_id\_t |
| 含义 | 需要固定的frame的id |

**2.方法实现思路：**

使用std::scoped\_lock对latch\_加锁，保证线程安全

在哈希表LRUhash\_中查找指定的frame\_id

如果找到该frame（即该frame当前是unpinned状态）：

使用哈希表中存储的迭代器从LRU链表中删除该frame

从哈希表中删除该frame的记录

如果没找到，说明该frame已经是pinned状态或不存在，无需操作

**3.实现难点：**

需要理解pin操作的含义：将页面从可淘汰状态变为不可淘汰状态，因此需要从LRU数据结构中移除。

void LRUReplacer::unpin(frame\_id\_t frame\_id)

**1.方法声明：**

方法名：unpin

返回类型：void

功能：取消固定一个frame，代表该页面可以被淘汰

参数列表：

|  | frame\_id |
| --- | --- |
| 类型 | frame\_id\_t |
| 含义 | 取消固定的frame的id |

**2.方法实现思路：**

使用std::scoped\_lock对latch\_加锁，保证线程安全

检查该frame是否已经在LRU链表中，如果已存在则直接返回（避免重复添加）

检查当前链表大小是否已达到最大容量max\_size\_，如果是则返回

将frame\_id添加到链表头部（表示最近访问的位置）：LRUlist\_.push\_front(frame\_id)

在哈希表中记录该frame在链表中的迭代器：LRUhash\_[frame\_id] = LRUlist\_.begin()

**3.实现难点：**

需要理解unpin操作将页面添加到链表头部而不是尾部，因为新unpin的页面被认为是最近访问的。同时需要注意检查重复添加和容量限制。

**任务2：缓冲池管理器实现**

补全 BufferPoolManager 类，实现Rucbase缓冲池管理.

**2.1理解 Page 和 BufferPoolManager 类的设计，并回答下列问题：**

1. Page::is\_dirty\_ 的作⽤是什么？

Page::is\_dirty\_ 是一个布尔值，用于标记页面是否为脏页。

作用：

* 写优化：只有脏页在被淘汰或刷写时才需要写入磁盘，避免不必要的磁盘写操作
* 数据一致性：确保修改过的页面能够正确写回磁盘，保持内存和磁盘数据的一致性
* 性能提升：减少不必要的磁盘I/O操作，提高系统性能

使用场景：

* 当页面内容被修改时，设置为true
* 当页面被写入磁盘后，设置为false
* 在页面淘汰时，只有脏页需要写回磁盘

2. Page::pin\_count\_ 的作⽤是什么？

Page::pin\_count\_ 是一个整数，表示当前有多少个线程或操作正在使用该页面。

作用：

* 引用计数：记录页面的使用情况，防止正在使用的页面被误删除或替换
* 并发控制：在多线程环境下保护页面不被意外淘汰
* 资源管理：确保页面在使用期间保持在内存中

状态转换：

* pin\_count\_ > 0：页面正在被使用，不能被淘汰
* pin\_count\_ = 0：页面可以被加入LRU替换候选列表
* 每次fetch\_page时pin\_count\_递增，每次unpin\_page时递减

3.BufferPoolManager::page\_table\_ 的作⽤是什么？

page\_table\_ 是一个哈希表（std::unordered\_map<PageId, frame\_id\_t, PageIdHash>），维护页面ID到帧编号的映射关系。

作用：

* 快速查找：给定PageId，能够O(1)时间内找到该页面在缓冲池中的帧位置
* 存在性检查：判断某个页面是否已经在缓冲池中
* 地址转换：将逻辑页面标识转换为物理帧位置

键值关系：

* 键：PageId（包含fd和page\_no的页面唯一标识）
* 值：frame\_id\_t（该页面在缓冲池中的帧编号）

4. BufferPoolManager::free\_list\_ 的作⽤是什么？

free\_list\_ 是一个链表（std::list<frame\_id\_t>），维护当前空闲的帧编号列表。

作用：

* 空间管理：跟踪缓冲池中哪些帧当前没有存储有效页面
* 快速分配：当需要新的帧时，可以直接从free\_list\_中取出，无需使用复杂的替换策略
* 初始化：系统启动时，所有帧都在free\_list\_中
* 回收：当页面被删除时，对应的帧重新加入free\_list\_

使用优先级：

* 优先使用free\_list\_中的空闲帧
* 只有当free\_list\_为空时，才使用LRU替换策略选择victim帧

生命周期：

* 系统初始化：所有帧编号都在free\_list\_中
* 页面分配：从free\_list\_中取出帧，用于存储新页面
* 页面删除：释放的帧重新加入free\_list\_

**2.2:**

bool BufferPoolManager::find\_victim\_page(frame\_id\_t\* frame\_id)

1.方法声明：

方法名：find\_victim\_page

返回类型：bool

功能：从free\_list或replacer中得到可淘汰帧页的frame\_id

参数列表：

|  | frame\_id |
| --- | --- |
| 类型 | frame\_id\_t\* |
| 含义 | 帧页id指针，返回成功找到的可替换帧id |

2.方法实现思路：

首先检查缓冲池是否有空闲帧（检查free\_list\_是否为空）

如果有空闲帧：从free\_list\_头部获取一个空闲frame\_id，并从链表中移除

如果没有空闲帧：调用replacer\_->victim(frame\_id)使用LRU策略选择一个victim页面

返回查找结果（true表示成功，false表示失败）

3.实现难点：

需要理解缓冲池管理的两个阶段：空闲帧管理和页面替换。优先使用空闲帧，只有在缓冲池满时才进行页面替换。

Page\* BufferPoolManager::fetch\_page(PageId page\_id);

1.方法声明：

方法名：fetch\_page

返回类型：Page\*

功能：从buffer pool获取需要的页。如果页表中存在page\_id则直接返回并增加pin\_count，否则从磁盘读取页面到缓冲池

参数列表：

|  | page\_id |
| --- | --- |
| 类型 | PageId |
| 含义 | 需要获取的页的PageId |

2.方法实现思路：

使用std::scoped\_lock加锁保证线程安全

在page\_table\_中查找目标页是否已在缓冲池中

如果页面已在缓冲池：增加pin\_count\_，调用replacer\_->pin(frame\_id)固定页面，返回页面指针

如果页面不在缓冲池：

调用find\_victim\_page获取可用frame

如果victim页面是脏页，先写回磁盘

从磁盘读取目标页到frame

更新页面元数据（id、pin\_count、is\_dirty）

更新page\_table映射关系

固定页面并返回

3.实现难点：

需要处理页面置换时的脏页写回，确保数据一致性。同时要正确维护page\_table的映射关系和页面的固定状态。

bool BufferPoolManager::unpin\_page(PageId page\_id, bool is\_dirty);

1.方法声明：

方法名：unpin\_page

返回类型：bool

功能：取消固定pin\_count>0的在缓冲池中的page

参数列表：

|  | page\_id | is\_dirty |
| --- | --- | --- |
| 类型 | PageId | bool |
| 含义 | 目标page的page\_id | 若目标page应该被标记为dirty则为true |

2.方法实现思路：

使用std::scoped\_lock加锁保证线程安全

在page\_table\_中查找目标页，如果不存在则返回false

检查页面的pin\_count\_，如果已经为0则返回false

将pin\_count\_减1

如果减1后pin\_count\_为0，调用replacer\_->unpin(frame\_id)将页面加入可替换列表

根据is\_dirty参数更新页面的脏标记

返回true

3.实现难点：

需要正确理解pin/unpin机制：pin\_count为0的页面才能被替换，unpin操作需要与replacer同步。

bool BufferPoolManager::flush\_page(PageId page\_id);

1.方法声明：

方法名：flush\_page

返回类型：bool

功能：将目标页写回磁盘，不考虑当前页面是否正在被使用

参数列表：

|  | page\_id |
| --- | --- |
| 类型 | PageId |
| 含义 | 目标页的page\_id，不能为INVALID\_PAGE\_ID |

2.方法实现思路：

使用std::scoped\_lock加锁保证线程安全

在page\_table\_中查找目标页，如果不存在则返回false

调用disk\_manager\_->write\_page将页面数据写回磁盘

将页面的is\_dirty\_标记设为false

返回true

3.实现难点：

实现相对简单，主要是确保强制将页面写回磁盘，无论页面是否为脏页。

Page\* BufferPoolManager::new\_page(PageId\* page\_id)

1.方法声明：

方法名：new\_page

返回类型：Page\*

功能：创建一个新的page，即从磁盘中移动一个新建的空page到缓冲池某个位置

参数列表：

|  | page\_id |
| --- | --- |
| 类型 | PageId\* |
| 含义 | 当成功创建一个新的page时存储其page\_id |

2.方法实现思路：

使用std::scoped\_lock加锁保证线程安全

调用find\_victim\_page获得一个可用的frame

如果victim页面是脏页，先写回磁盘

调用disk\_manager\_->allocate\_page分配新的page\_id

初始化新页面的元数据（id、pin\_count=1、is\_dirty=false）

清空页面数据（reset\_memory()）

更新page\_table\_映射关系

调用replacer\_->pin固定页面

返回新创建的页面指针

3.实现难点：

需要协调磁盘空间分配、缓冲池管理和页面初始化，确保新页面正确地被创建和固定。

bool BufferPoolManager::delete\_page(PageId page\_id);

1.方法声明：

方法名：delete\_page

返回类型：bool

功能：从buffer\_pool删除目标页

参数列表：

|  | page\_id |
| --- | --- |
| 类型 | PageId |
| 含义 | 目标页 |

2.方法实现思路：

使用std::scoped\_lock加锁保证线程安全

在page\_table\_中查找目标页，如果不存在则返回true

检查目标页的pin\_count\_，如果不为0则返回false（页面正在被使用）

如果是脏页，先写回磁盘

从page\_table\_中删除映射关系

重置页面元数据（page\_id设为INVALID\_PAGE\_ID，pin\_count=0，is\_dirty=false）

清空页面数据

将frame加入free\_list\_以供重用

返回true

3.实现难点：

需要确保只有未被固定的页面才能被删除，并正确回收frame资源。

void BufferPoolManager::flush\_all\_pages(int fd);

1.方法声明：

方法名：flush\_all\_pages

返回类型：void

功能：将buffer\_pool中的所有页写回到磁盘

参数列表：

|  | fd |
| --- | --- |
| 类型 | int |
| 含义 | 文件句柄 |

2.方法实现思路：

使用std::scoped\_lock加锁保证线程安全

遍历缓冲池中的所有页面（pages\_[0]到pages\_[pool\_size\_-1]）

对于每个页面，检查其是否属于指定文件（page->get\_page\_id().fd == fd）且有效（page\_no != INVALID\_PAGE\_ID）

如果条件满足，调用disk\_manager\_->write\_page将页面写回磁盘

将页面的is\_dirty\_标记设为false

3.实现难点：

需要遍历整个缓冲池并正确识别属于指定文件的页面，确保批量写回操作的正确性。

四、实验总结

技术收获

深入理解缓冲池原理: 掌握了数据库系统中内存管理的核心机制，包括页面缓存、替换策略、脏页管理等

并发编程实践: 学会使用互斥锁保证多线程环境下数据结构的安全访问

系统设计思维: 理解了高效数据结构设计的重要性，如LRU的O(1)实现

资源管理: 掌握了系统资源（内存、磁盘）的有效调度和管理策略

实验意义

通过本次实验，全面理解了数据库系统缓冲池管理的完整流程，为后续学习数据库内核技术奠定了坚实基础。实验中涉及的内存管理、并发控制、I/O优化等技术，在实际的数据库系统开发中具有重要的应用价值。