实验 1：缓冲池管理器实现

一、实验目的

1. 掌握 Rucbase 缓冲池页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. 缓冲池的页面替换策略
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：缓冲池页面替换策略实现

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

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

2. 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**
类型	int	page_id_t ： int32_t
含义	文件描述符	页号

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。