模块一实验报告

负责人：常潍麟　日本语言文学　3210100056
电子邮箱：xishanqiming@gmail.com
开发时间：2023 年 5 月 15 日 — 31日

一．实验概述

在 MiniSQL 的设计中，Disk Manager 和 Buffer Pool Manager 模块位于架构的最底层。Disk Manager 主要负责数据库文件中数据页的分配和回收，以及数据页中数据的读取和写入。其中，数据页的分配和回收通过位图（Bitmap）实现，位图中每个 bit 对应一个数据页的分配情况，用于标记该数据页是否空闲（0 表示空闲，1 表示已分配）。

当 Buffer Pool Manager 需要向 Disk Manager 请求某个数据页时，Disk Manager 会通过某种映射关系，找到该数据页在磁盘文件中的物理位置，将其读取到内存中返还给 Buffer Pool Manager。而 Buffer Pool Manager 主要负责将磁盘中的数据页从内存中来回移动到磁盘，这使得我们设计的数据库管理系统能够支持那些占用空间超过设备允许最大内存空间的数据库。

Buffer Pool Manager 中的操作对数据库系统中其他模块是透明的。例如，在系统的其它模块中，可以使用数据页唯一标识符 page_id 向 Buffer Pool Manager 请求对应的数据页。但实际上，这些模块并不知道该数据页是否已经在内存中还是需要从磁盘中读取。同样地，Disk Manager 中的数据页读写操作对 Buffer Pool Manager 模块也是透明的，即 Buffer Pool Manager 使用逻辑页号 logical_page_id 向 Disk Manager 发起数据页的读写请求，但 Buffer Pool Manager 并不知道读取的数据页实际上位于磁盘文件中的哪个物理页（对应页号 physical_page_id）。

为了简化工程，目前只考虑单线程下的设计，但在程序中仍然保留了并发控制相关的锁，未来可以增加实现事务、并发控制、故障恢复相关功能。

在模块一中，主要包括了四个部分，下面将对四个部分的实现进行分别阐述：

1. 位图的实现
2. 磁盘管理的实现
3. 缓冲池替换策略的实现
    - LRU 替换策略

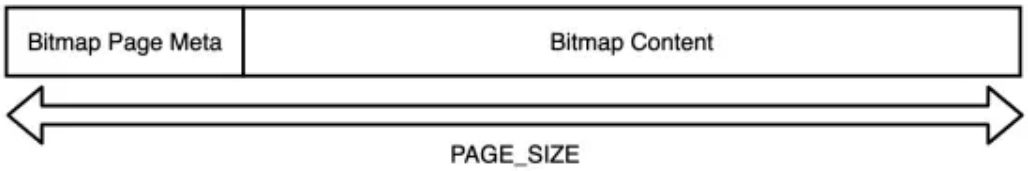
- Clock 替换策略

4. 缓冲池的实现

二. 位图页的实现

实现一个简单的位图页（Bitmap Page），位图页是 Disk Manager 模块中的一部分，是实现磁盘页分配与回收工作的必要功能组件。位图页与数据页一样，占用 PAGE_SIZE (4KB) 的空间，标记一段连续页的分配情况。

Bitmap Page 由两部分组成，一部分是用于加速 Bitmap 内部查找的元信息（Bitmap Page Meta），包含当前已经分配的页的数量（page_allocated_）以及下一个空闲的数据页(next_free_page_)。除去元信息外，页中剩余的部分就是 Bitmap 存储的具体数据，其大小 可以通过 PAGE_SIZE – BITMAP_PAGE_META_SIZE 来计算，Bitmap Page 能够支持最多纪录 BITMAP_CONTENT_SIZE * 8 个连续页的分配情况。



位图页的功能主要在 page/bitmap_page.cpp 中被实现，定义了 BitmapPage 类，这是一个通用的位图页面管理模块，能够处理页面的分配和回收，以及检查页面是否是空闲状态。这个类以模板的形式实现，可通过更改模板参数以支持不同的页面大小。

1. AllocatePage 函数

```
template <size_t PageSize>
bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset);
```

- 功能：分配一个空闲页
- 输入：page_offset，引用类型的无符号整数，表示所需分配的空闲页在段中的下标
- 输出：返回一个布尔值，表示是否成功分配。如果分配成功，返回 true，并将所分配的空闲页的位图位（page_offset）设置为 1；如果请求的页面已经被分配，或者超出了最大允许的页面数量，返回 false。

2. DeAllocatePage 函数

```
template <size_t PageSize>
bool BitmapPage<PageSize>::DeAllocatePage(uint32_t page_offset);
```

- 功能：回收已经被分配的页
- 输入：page_offset，引用类型的无符号整数，表示需要回收的页在段中的下标
- 输出：返回一个布尔值，表示是否成功回收。如果成功回收，返回 true，并将所回收的页的位图位（page_offset）设置为 0，同时更新下一个可用页面索引并将分配的页面数减少 1；如果请求的页面未被分配，返回 false。

3. IsPageFree 函数

```
template <size_t PageSize>
bool BitmapPage<PageSize>::IsPageFree(uint32_t page_offset) const;
```

- 功能：判断给定的页是否是空闲（未分配）的
- 输入：page_offset，引用类型的无符号整数，表示需要判断的页在段中的下标
- 输出：返回一个布尔值，表示给定的页是否是空闲的。如果是空闲的，返回 true；否则返回 false。

4. 类的设计和实现细节

BitmapPage 类的实现主要依赖于 bitset 数据结构，允许对单个 bit 进行操作。通过管理位图来跟踪页面的状态（是否被分配）。函数 AllocatePage 和 DeAllocatePage 分别用于分配和回收页，而 IsPageFree 则用于检查指定的页是否被分配。

在底部，BitmapPage 类被实例化为各种不同的页面大小，如 64、128、256、512、1024、2048 和 4096。这意味着这个类可以灵活地用于不同大小的页的管理。

Bitmap Page 相关代码在 bitmap_page.h 和 bitmap_page.cpp 中，实现了以下函数：

- BitmapPage::AllocatePage(&page_offset)：分配一个空闲页，并通过 page_offset 返回所分配的空闲页位于该段中的下标（从 0 开始）；
- BitmapPage::DeAllocatePage(page_offset)：回收已经被分配的页；
- BitmapPage::IsPageFree(page_offset)：判断给定的页是否是空闲（未分配）的。

此外，与该模块相关的测试代码位于 test/storage/disk_manager_test.cpp 中。

5. 可改进的地方

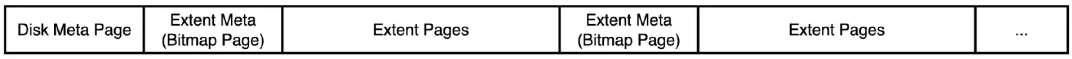
目前代码中 IsPageFreeLow 函数没有实现，如果需要对低级别的位进行精确管理，可以进一步实现此函数。（不过目前的精细度对于一般的事务处理都可以进行）

三. 磁盘数据页管理

在实现了基本的位图页后，就可通过位图页加上一段连续的数据页（数据页的数量取决于位图页最大能够支持的比特数）对磁盘文件（DB File）中数据页进行分配和回收。

但实际上还存在着问题，假设数据页的大小为 4KB，一个位图页中的每个字节都用于记录，那么这个位图页最多能够管理 32768 个数据页，也就是说，这个文件最多只能存储 4K * 8 * 4KB = 128MB 的数据，很容易发生数据溢出的情况。

为了应对上述问题，解决思路是把一个位图页加一段连续的数据页看成数据库文件中的一个分区（Extent），再通过一个额外的元信息页来记录这些分区的信息。通过这种方式，来使磁盘文件能够维护更多的数据页信息。其主要结构如下图所示：



Disk Meta Page 是数据库文件中的第 0 个数据页，维护分区相关的信息，如分区数量、每个分区中已分配页的数量等。每一个分区还包含了一个位图页和一段连续的数据页。

在这样的设计下，假设 Disk Meta Page 能够记录 4K/4=1K 个分区的信息，那么整个数据库能够维护的数据页的数量以及能够存储的数据数量扩大了 1000 倍。

不过，这样的设计还存在着一个问题。由于元数据所占用的数据页实际上是不存储数据库数据的，而它们实际上又占据了数据库文件中的数据页，因此实际上真正存储数据的数据页是不连续的。

物理页号	0	1	2	3	4	5	6
职责	磁盘元数据	位图页	数据页	数据页	数据页	位图页	数据页
逻辑页号	/	/	0	1	2		3

但实际上，对于上层的 Buffer Pool Manager 来说，希望连续分配得到的页号是连续的，为此，在 Disk Manager 中，需要将页号映射成逻辑页号，这样才能使得上层的 Buffer Pool Manager 对于 Disk Manager 中的页分配是无感知的。

因此在这个模块中实现了以下函数，与之相关的代码位于 disk_manager.h 和 disk_manager.cpp。

- DiskManager::AllocatePage()：从磁盘分配一个空闲页，并返回其逻辑页号；
- DiskManager::DeAllocatePage(logical_page_id)：释放逻辑页号对应的物理页。
- DiskManager::IsPageFree(logical_page_id)：判断逻辑页号对应的数据页是否空。
- DiskManager::MapPageId(logical_page_id)：可根据需要实现。在 DiskManager 类的私有成员中，该函数可以用于将逻辑页号转换成物理页号。

DiskManager 类是一个用于磁盘文件操作的管理类，负责数据库中物理文件的打开、关闭、读取、写入、页面分配、释放等操作。它提供了从逻辑页面 ID 到物理页面 ID 的映射，并通过元数据管理文件中的空闲页面。

另外，DiskManager 类中的 meta_data_成员实际上是 MetaPage 在内存中的缓存。使用时只需通过 reinterpret_cast 将 meta_data_转换成 MetaPage 类型的对象即可。

1. ReadPage 函数

```
ReadPage(page_id_t logical_page_id, char *page_data)
```

- 功能：该方法用于从磁盘中读取指定逻辑页号的页面数据。
- 输入：logical_page_id 为需要读取的逻辑页面 ID，page_data 是一个字符数组指针，用于存储读取的页面数据。

2. WritePage 函数

```
WritePage(page_id_t logical_page_id, const char *page_data)
```

- 功能：该方法用于将页面数据写入磁盘的指定逻辑页号的页面中。
- 输入：logical_page_id 是需要写入的逻辑页面 ID，page_data 是一个字符数组指针，包含需要写入的页面数据。

3. AllocatePage 函数

```
AllocatePage()
```

- 功能：该方法用于在磁盘中分配一个新的页面，并返回分配的逻辑页面号。

4. DeAllocatePage 函数

```
DeAllocatePage(page_id_t logical_page_id)
```

- 功能：该方法用于在磁盘中释放指定逻辑页号的页面。

- 输入：需要释放的逻辑页面 ID。

5. IsPageFree 函数

```
IsPageFree(page_id_t logical_page_id)
```

- 功能：该方法用于判断磁盘中指定逻辑页号的页面是否为空闲。
- 输入：需要判断的逻辑页面 ID。
- 输出：如果指定的页面是空闲的，返回 true，否则返回 false。

6. MapPageId 函数

```
MapPageId(page_id_t logical_page_id)
```

- 功能：该方法用于将逻辑页面号转换成物理页面号。
- 输入：需要转换的逻辑页面 ID。
- 输出：对应的物理页面号。

7. GetFileSize 函数

```
GetFileSize(const std::string &file_name)
```

- 功能：该方法用于获取指定文件的大小。
- 输入：需要获取大小的文件名。
- 输出：文件的大小，单位为字节。

8. ReadPhysicalPage 函数

```
ReadPhysicalPage(page_id_t physical_page_id, char *page_data)
```

- 功能：该方法用于读取指定物理页号的页面数据。
- 输入：physical_page_id 是需要读取的物理页面 ID，page_data 是一个字符数组指针，用于存储读取的页面数据。

9. WritePhysicalPage 函数

```
 WritePhysicalPage(page_id_t physical_page_id, const char *page_data)
```

- 功能：该方法用于将页面数据写入磁盘的指定物理页号的页面中。
- 输入：physical_page_id 是需要写入的物理页面 ID，page_data 是一个字符数组指针，包含需要写入的页面数据。

10. 类的设计和实现细节

DiskManager 类通过文件 I/O 实现磁盘页面的读写操作。当需要对磁盘进行读写操作时，会使用一个逻辑页面 ID，该类将逻辑页面 ID 映射为物理页面 ID，然后进行读写操作。此外，它还管理一个元数据页面，用于跟踪文件中的空闲页面和已分配的页面。

11. 可改进的地方

在实现磁盘管理时，应考虑到文件 I/O 操作的开销，可能需要优化磁盘 I/O 操作，例如通过缓存或预读技术减少磁盘访问次数。同时，在处理大量页面请求时，可能需要实现更复杂的页面分配策略，以提高存储空间的利用率和性能。

四．LRU 缓存替换策略

（在本节中，额外实现了一个 bonus 目标，即：设计了一个 CLOCK 缓存替换策略）

Buffer Pool Replacer 负责跟踪 Buffer Pool 中数据页使用情况，并在没有空闲页时决定替换哪一个数据页。实现一个基于 LRU 替换算法的 LRUReplacer，LRUReplacer 类在 lru_replacer.h 中被定义，其扩展了抽象类 Replacer。LRUReplacer 的大小默认与 Buffer Pool 的大小相同。

因此，在这个模块中，需要实现以下函数，与之相关的代码位于 lru_replacer.cpp 中。

- LRUReplacer::Victim(*frame_id)：替换（即删除）与所有被跟踪的页相比最近最少被访问的页，将其页帧号（即数据页在 Buffer Pool 的 Page 数组中的下标）存储在输出参数 frame_id 中输出并返回 true，如果当前没有可以替换的元素则返回 false；
- LRUReplacer::Pin(frame_id)：将数据页固定使之不能被 Replacer 替换，即从 lru_list_中移除该数据页对应的页帧。Pin 函数应当在一个数据页被 Buffer Pool Manager 固定时被调用；
- LRUReplacer::Unpin(frame_id)：将数据页解除固定，放入 lru_list_中，使之可以在必要时被 Replacer 替换掉。Unpin 函数应当在一个数据页的引用计数变为 0 时被 Buffer Pool Manager 调用，使页帧对应的数据页能够在必要时被替换；
- LRUReplacer::Size()：返回当前 LRUReplacer 中能够被替换的数据页的数量。

在 lru_replacer 中，定义了一个名为 LRUReplacer 的类，这个类实现了一个简单的最近最少使用 (LRU) 策略的页替换器。在数据库中，内存缓冲池的大小是固定的，当需要加载新的数据页时，如果缓冲池已满，需要替换掉其中的某个数据页，选择牺牲页的过程由页面替换器决定。LRU 策略选择最近最少使用的页面进行替换。

在该类中有五个主要的方法，分别是构造函数 LRUReplacer(size_t num_pages)，析构函数 ~LRUReplacer()，以及三个主要功能方法 Victim(frame_id_t *outputFrameId)，Pin(frame_id_t frame_id) 和 Unpin(frame_id_t frameId)。

1. LRUReplacer 构造函数

```
LRUReplacer::LRUReplacer(size_t num_pages)
    :capacity(num_pages) {}
```

- 功能：初始化 LRUReplacer 类，设置其容量为 num_pages。
- 输入：num_pages，表示内存中能够存储的数据页的最大数量，即 Buffer Pool 的大小。

2. ~LRUReplacer 析构函数

```
bool LRUReplacer::Victim(frame_id_t *outputFrameId)
```

- 功能：默认析构函数，销毁 LRUReplacer 类的实例，不接收任何参数。

3. Victim 函数

```
bool LRUReplacer::Victim(frame_id_t *outputFrameId)
```

- 功能：根据 LRU 策略选取一个可替换的页帧，即从所有被跟踪的页中选取最近最少被访问的页（数据页在 Buffer Pool 的 Page 数组中的下标）。如果当前没有可以替换的数据页，则返回 false。
- 输出：outputFrameId，将被替换掉的数据页在 Buffer Pool 中的位置（即下标）存储在这里。

4. Pin 函数

```
void LRUReplacer::Pin(frame_id_t frame_id)
```

- 功能：该方法接收一个帧 ID，表示将该帧固定在内存中，即不会被替换出去。如果帧 ID 在 LRUReplacer 中的映射表 mapping 中存在，则将其从 LRU 列表 frameList 中删除，并从 mapping 中删除该映射。这样就可以保证该帧不会在后续的替换过程中被选中，从而被替换出去。该函数应在一个数据页被 Buffer Pool Manager 固定时被调用。
- 输入：frame_id，表示要被固定的数据页的 id。

5. Unpin 函数

```
void LRUReplacer::Unpin(frame_id_t frameId)
```

- 功能：解除对数据页的固定，使其可以在必要时被替换掉。该函数应当在一个数据页的引用计数变为 0 时被 Buffer Pool Manager 调用。
- 输入：frameId，表示要被解除固定的数据页的 id。

6. Size 函数

```
LRUReplacer::Size()
```

- 功能：返回当前 LRUReplacer 中能够被替换的数据页的数量。
- 输出：返回值为当前能够被替换的数据页的数量（直接返回 mapping 的 size ）。

7. 类的设计和实现细节

LRUReplacer 类主要通过双向链表和哈希映射两种数据结构实现。双向链表存储了数据页的访问历史，链表的头部是最近最少被访问的数据页，尾部是最近被访问的数据页。

哈希映射存储了数据页的 id 到其在双向链表中位置的映射，这样可以在 O(1)的时间内找到任何一个数据页在双向链表中的位置。

8. 可改进的地方

可以进一步优化数据结构以减少内存使用、添加更多的函数以提供更多的功能（如：获取某个数据页的访问记录、或对访问记录进行排序）。

五. （Bonus）基于 CLOCK 的缓存替换策略

在 clock_replacer.cpp 中实现了一个”CLOCKReplacer”类，它是用于操作系统页面替换策略中的一种 CLOCK 算法（也称为最近未使用 LRU 的近似算法）的实现。
CLOCK 算法的主要目标是在内存中替换最近未使用的页面，优化系统的内存管理。

在 CLOCK 算法中，系统为每个页面设置一个访问位，并将页面置于一个环形队列中，当需要替换页面时，操作系统会选择访问位为 0 的页面进行替换。

CLOCKReplacer 的主要功能包括添加和移除页面（通过 Unpin 和 Pin 函数），检查页面替换器的大小（通过 Size 函数），以及寻找和返回 victim 页面。

1. CLOCKReplacer 构造函数

```
CLOCKReplacer::CLOCKReplacer(size_t num_pages)
    :second_chance(num_pages,State::EMPTY),
      pointer(0),
```

```
    capacity(num_pages) {}
```

- 功能：初始化一个具有指定页面数量的 CLOCKReplacer，并为每个页面都赋予一个 second_chance（二次机会）标记，second_chance 数组的长度就是 num_pages，并且初始化状态全部为 EMPTY 并设置初始指针指向第一个页面。pointer 指针和 capacity 都被设置为 num_pages。
- 输入：num_pages：页面数量。

2. ~CLOCKReplacer 析构函数

```
CLOCKReplacer::~CLOCKReplacer() {}
```

- 功能：CLOCKReplacer 的析构函数。因为 CLOCKReplacer 对象使用的是 std::vector 管理资源，所以析构函数无需进行任何操作，std::vector 会自动释放资源。

3. Victim 函数

```
bool CLOCKReplacer::Victim(frame_id_t *frame_id)
```

- 功能：使用 Clock 算法来找到 victim frame id。首先遍历整个 second_chance 数组，如果某个 frame 的状态为 ACCESSED，则给予第二次机会，设为 UNUSED；如果状态为 UNUSED 且没有找到 victim，则设置当前 frame 为 victim。如果所有的 frame 都已被访问，则指定第一个 UNUSED 的 frame 为 victim。如果找不到 victim，则返回 false；否则，将找到的 victim frame 的状态设为 EMPTY，更新 pointer，并返回 true。
- 输出：frame_id，即返回的 victim frame id 的指针。如果找不到 victim frame id，则该值会被设置为 0。

4. Pin 函数

```
void CLOCKReplacer::Pin(frame_id_t frame_id) {
  second_chance[frame_id % capacity] = State::EMPTY;
}
```

- 功能：通过将指定 frame 的状态设置为 EMPTY，从而实现将其从 replacer 中移除。
- 输入：frame_id，即需要被移除的 frame 的 id。

5. UnPin 函数

```
void CLOCKReplacer::Unpin(frame_id_t frame_id) {
  second_chance[frame_id % capacity] = State::ACCESSED;
```

```
}
```

- 功能：通过将指定 frame 的状态设置为 ACCESSED，从而实现将其添加进 replacer。
- 输入：frame_id，即需要被添加的 frame 的 id。

6. Size 函数

```
size_t CLOCKReplacer::Size() {
  return count_if(second_chance.begin(), second_chance.end(), IsEmpty);
}
```

- 功能：该函数通过 count_if 函数计算 second_chance 数组中状态不等于 EMPTY 的元素数量，从而得到 replacer 的大小。
- 输出：返回 replacer 当前的大小。

7. IsEmpty 函数

```
bool CLOCKReplacer::IsEmpty(CLOCKReplacer::State& item) {
  return item != State::EMPTY;
}
```

- 功能：通过比较 item 和 EMPTY，检查 frame 的 State 是否不等于 EMPTY。
- 输入：item，即需要被检查的 State。
- 输出：如果 State 不等于 EMPTY，返回 true；否则，返回 false。

8. 类的设计和实现细节

在设计和实现 CLOCK 的替换策略上，主要是使用 second_chance 数组存储每个页面的状态，可以是 EMPTY、UNUSED 或 ACCESSED。数组长度由构造函数设置为 num_pages，初始状态为 EMPTY。该类还维护一个指针 pointer 指向当前的页面，以及一个 capacity 变量表示页面数量。

另外，与基于 LRU 的替换策略相似，也使用了 Victim 函数寻找并返回一个 victim frame id。该函数遍历整个 second_chance 数组，找到一个 UNUSED 的页面，如果所有页面都被访问过，会重新扫描并选取第一个 UNUSED 的页面。这种设计实现了 Clock 替换策略，避免了频繁使用的页面被替换。

而 Pin 函数和 Unpin 函数分别用于将指定的 frame 从 replacer 中移除或添加。它们通过改变 second_chance 数组中指定 frame 的状态实现功能。

最后，使用 Size 函数返回状态不等于 EMPTY 的 frame 数量。该函数通过 count_if 函数统计 second_chance 数组中状态不等于 EMPTY 的元素数量。使用 IsEmpty 函数检查给定的 State 是否不等于 EMPTY，用于 Size 函数的条件判断。

9. 可改进的地方

可以进一步优化的地方主要是在替换策略的检测性能上。Victim 函数的性能可以进一步提升。当前实现可能需要两次遍历 second_chance 数组才能找到 victim frame id，如果能够优化该过程，会进一步提升性能。

另外，Pin 和 Unpin 函数没有检查输入的 frame_id 是否有效。如果 frame_id 大于 capacity，这将会导致越界访问，可以增加对 frame_id 有效性的检查。同时当前 Size 函数的需要遍历整个 second_chance 数组来计算大小，会影响性能。如果能够在 Pin 和 Unpin 函数中动态维护当前大小，可以提高 Size 函数的性能。

六. 缓冲池管理

在实现缓冲池替换策略的算法后，需要实现缓冲池管理器，负责从磁盘管理器中获取数据页并将它们存储在内存中，并在必要时将脏页转存到磁盘中（为新页面腾出空间）。

在数据库中，所有内存页面都由 Page 对象表示，每个 Page 对象都包含了一段连续的内存空间 data_和与该页相关的信息（如是否是脏页，页的引用计数等等）。

但 Page 对象只是一个用于存放从磁盘中读取的数据页的容器，同一个 Page 在系统的生命周期内，可能会对应很多不同的物理页。Page 对象的唯一标识符 page_id_用于跟踪它所包含的物理页，如果 Page 对象不包含物理页，那么 page_id_必须被设置为 INVALID_PAGE_ID。

同时，每个 Page 对象还对应着一个计数器，用于记录固定(Pin)该页面的线程数。缓冲池管理器不允许释放已经被固定的 Page。每个 Page 对象还将记录它是否脏页，在复用 Page 对象之前必须将脏的内容转储到磁盘中。

在 BufferPoolManager 的实现中，通过使用已经实现的 LRUReplacer 跟踪 Page 对象何时被访问，以便决定在缓冲池没有空闲页时决定替换哪个数据页。在该模块中实现了以下函数：

- BufferPoolManager::FetchPage(page_id)：根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取；

- BufferPoolManager::NewPage(&page_id)：分配一个新的数据页，并将逻辑页号于 page_id 中返回；
- BufferPoolManager::UnpinPage(page_id, is_dirty)：取消固定一个数据页；
- BufferPoolManager::FlushPage(page_id)：将数据页转储到磁盘中；
- BufferPoolManager::DeletePage(page_id)：释放一个数据页；
- BufferPoolManager::FlushAllPages()：将所有的页面都转储到磁盘中。

对于 FetchPage 操作，如果空闲页列表（free_list_）中没有可用的页面并且没有可以被替换的数据页，则应返回 nullptr。FlushPage 操作应该将页面内容转储到磁盘中，无论其是否被固定。

1. BufferPoolManager 构造函数

```
BufferPoolManager::BufferPoolManager(size_t pool_size, DiskManager *disk_manager)
```

- 功能：初始化 Buffer Pool Manager。
- 输入：pool_size，表示缓冲池的大小，即可以容纳的页面数量；disk_manager，磁盘管理器的指针，用于从磁盘读取或写入页面。

2. ~BufferPoolManager 析构函数

```
BufferPoolManager::~BufferPoolManager()
```

- 功能：销毁 Buffer Pool Manager，包括将所有页面（脏页）刷新到磁盘，并释放所有已经分配的资源（页面数组和 LRU 替换器）。

3. FetchPage 函数

```
Page *BufferPoolManager::FetchPage(page_id_t page_id)
```

- 功能：根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取。
- 输入：page_id，要获取的数据页的逻辑页号。
- 输出：获取到的数据页的指针。

4. NewPage 函数

```
Page *BufferPoolManager::NewPage(page_id_t &page_id)
```

- 功能：分配一个新的数据页，并将逻辑页号于 page_id 中返回。
- 输入：page_id 的引用，用于返回新分配的数据页的逻辑页号。

- 输出：新分配的数据页的指针。

5. DeletePage 函数

```
bool BufferPoolManager::DeletePage(page_id_t page_id)
```

- 功能：释放一个数据页。
- 输入：page_id：要释放的数据页的逻辑页号。
- 输出：如果成功释放，则返回 true。如果指定的页不存在，或者该页正在被使用（引用计数大于 0），返回 false。

6. UnpinPage 函数

```
bool BufferPoolManager::UnpinPage(page_id_t page_id, bool is_dirty)
```

- 功能：取消对数据页的引用，使其可以在需要时被替换掉。
- 输入：page_id，需要取消引用的数据页的 id；is_dirty，表示是否需要将该数据页标记为脏页，如果是 true，在替换这个页之前需要先将其写回磁盘。
- 输出：如果成功取消引用返回 true，否则返回 false。

7. FlushPage 函数

```
FlushPage(page_id_t page_id)
```

- 功能：将指定的数据页写回磁盘。
- 输入：page_id，需要写回磁盘的数据页的 id。
- 输出：如果成功写回返回 true，否则返回 false。

8. 类的设计和实现细节

BufferPoolManager 类的设计是为了从磁盘管理器中获取数据页面并将这些页面存储在内存中。同时，当需要为新页面腾出空间时，它负责将内存中的脏页面（已经被修改的页面）转储到磁盘中。它的设计基于缓冲池和 LRU 替换策略，以提高数据的访问性能和使用效率。

因为该类结构较为复杂，故在此对类中定义的数据成员进行分析：

pool_size_：存储缓冲池的大小。

disk_manager_：磁盘管理器的指针，用于与磁盘进行交互。

pages_：存储所有页面的数组。

replacer_：LRU 替换器的指针，负责在缓冲池中选择被替换的页面。

free_list_：存储空闲页面 ID 的列表，也就是当前没有使用的页面的列表。

page_table_：存储页面 ID 和对应的帧 ID（frame ID）的映射，帧 ID 就是该页面在 pages_：数组中的索引。

9. 可改进的地方

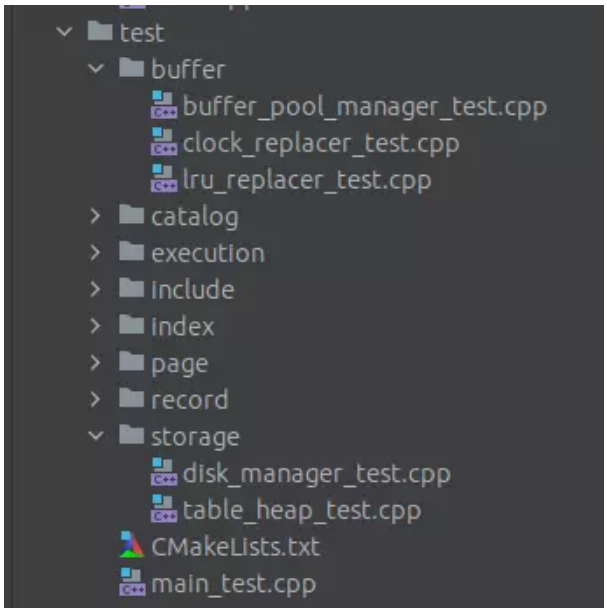
首先在构造函数中，使用了 new 关键字动态分配数组内存，但在析构函数中，只释放了这些数组内存，并没有检查数组元素内的内容是否有需要释放的内存。对于 Page 类型的元素，如果其内部也有动态分配的内存，那么就可能会出现内存泄露。解决这个问题的一种方法是为 Page 添加一个析构函数，来确保其内部的资源得到释放。

另外当检测到页面没有被全部 unpin 的时候，代码中使用了 LOG(ERROR)进行错误日志的打印。一般来说，除了错误日志，还可以增加一些关键路径的信息日志，例如页面从磁盘加载到内存，或者页面被替换出去等事件，这样在出现问题时，可以更好地进行调试和问题定位。同时，日志等级也需要灵活设置，可以控制日志的输出等级。

七. 单元测试

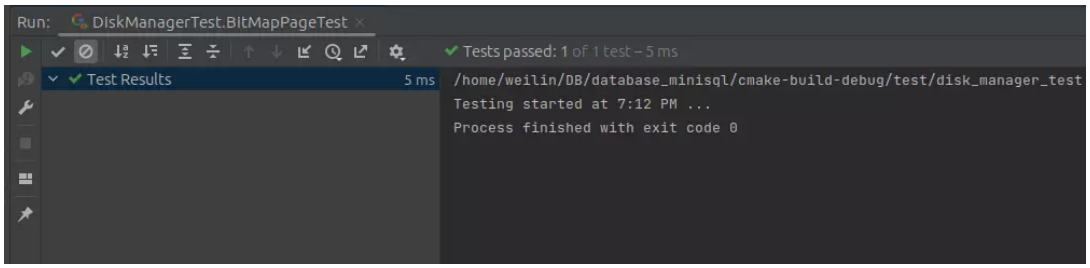
本模块需要进行四项测试，其中包含一个自己编写的 CLOCK 的缓存替换策略的测试。
disk_manager_test.cpp
lru_replacer_test.cpp
clock_replacer_test.cpp
buffer_pool_manager_test.cpp

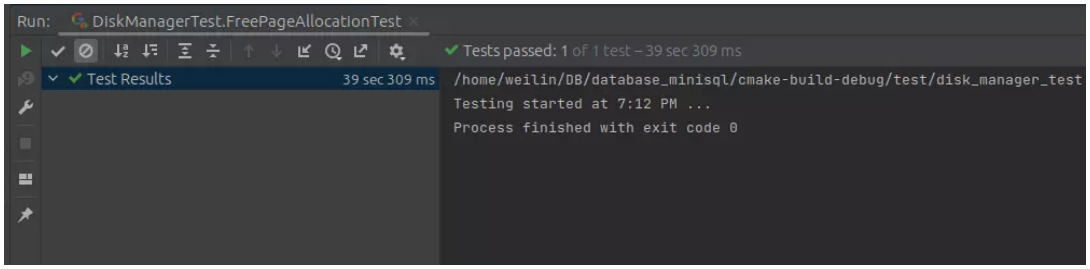
其中在”DiskManagerTest, BitMapPageTest”测试中，新增了 ofs 的初始化和更新。在原代码中，ofs 的初始值没有明确定义。在新代码中，ofs 被初始化为 0，这避免了可能的未初始化变量使用问题。　在新代码中，每次循环，ofs 都会自增 1，这样 ofs 将在每次迭代中使用不同的值。同时在”DiskManagerTest, FreePageAllocationTest”测试中，新增了一段对变量 a 的操作代码。



1. disk_manager_test.cpp

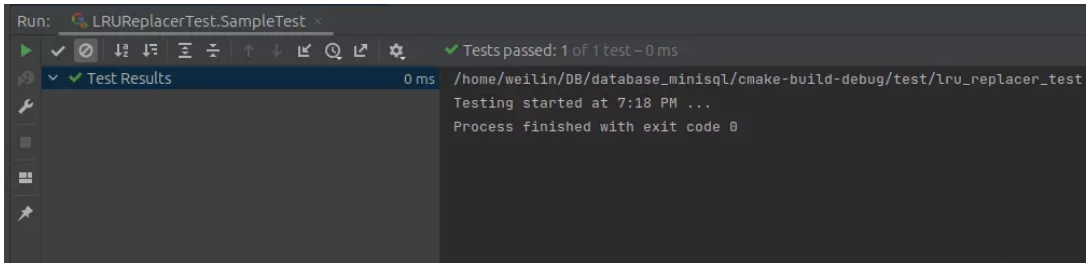
该测试文件包含两个子测试，分别对位图和空页图的分配进行测试。





2. lru_replacer_test.cpp

该测试使用一个简单的样例测试基于 LRU 的缓存替换策略。

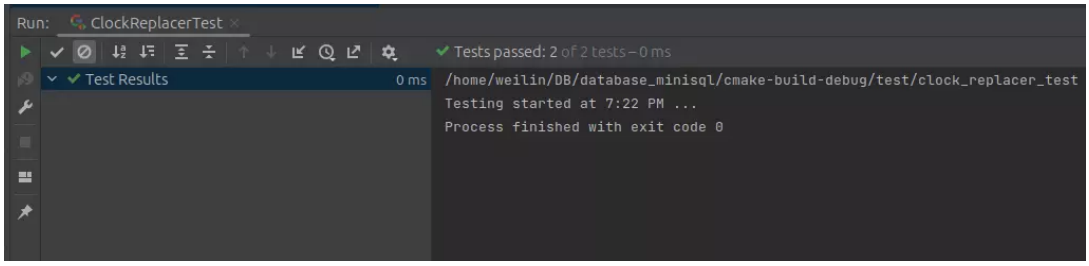


3. clock_replacer_test.cpp

该测试是基于 CLOCK 替换策略，自己编写的测试，包含两个子测试。在 CLOCK 算法中，系统为每个页面设置一个访问位，并将页面置于一个环形队列中，当需要替换页面时，操作系统会选择访问位为 0 的页面进行替换。

其中 SampleTest 首先将 6 个页面标记为不可访问（Unpin），并检查是否这 6 个页面被正确地标记为不可访问。然后选择 3 个页面进行替换（Victim），检查这些页面的值是否符合预期。接着将页面 3 和 4 标记为可访问（Pin），并检查不可访问的页面数量是否被正确地更新为 2。最后，将页面 4 标记为不可访问，然后再次选择 3 个页面进行替换，检查这些页面的值是否符合预期。

CornerCaseTest 首先将页面 3 和 2 标记为不可访问，并检查不可访问的页面数量是否正确。然后选择一个页面进行替换，检查该页面的值是否符合预期。接着将页面 1 标记为不可访问，并检查不可访问的页面数量是否正确。然后再选择两个页面进行替换，检查这些页面的值是否符合预期。最后尝试在没有可用页面进行替换的情况下选择一个页面进行替换，检查返回值是否为 false，并检查不可访问页面数量是否正确更新为 0。



4. buffer_pool_manager_test.cpp

该测试检验缓冲池管理器是否能够正确从磁盘管理器中获取数据页并将它们存储在内存中，并在必要时将脏页转存到磁盘中。