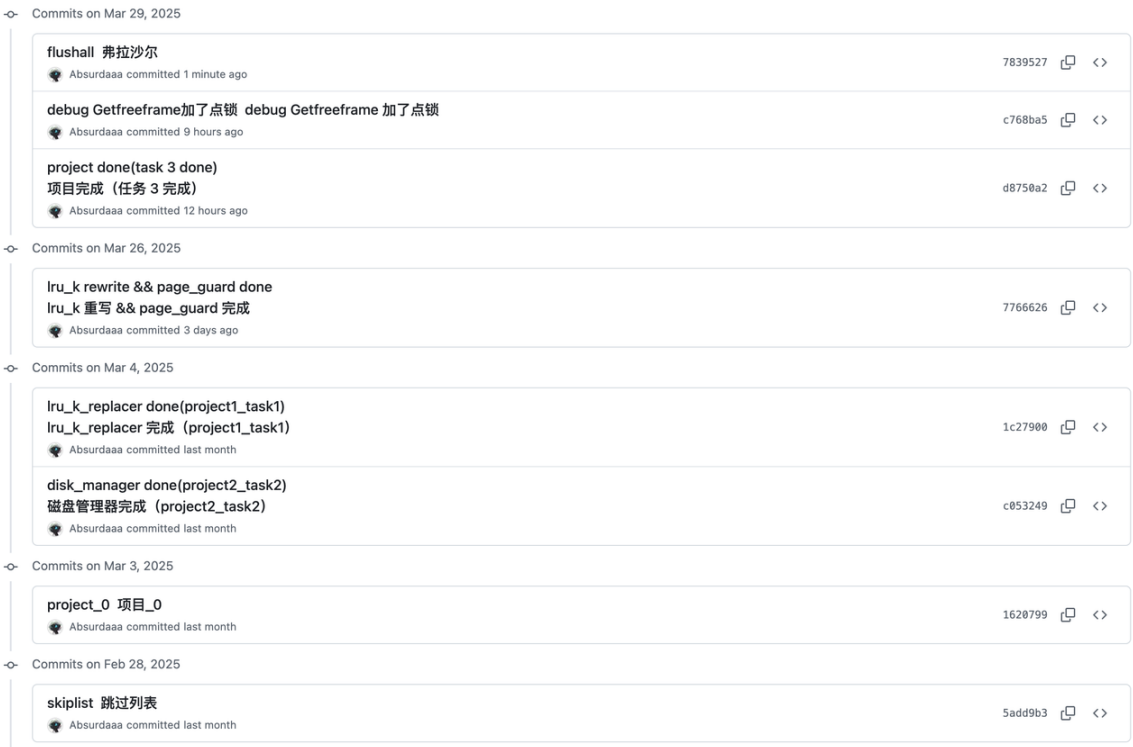
数据库第一次实验报告

2312966 林晖鹏

仓库地址：https://github.com/Absurdaaa/Database.git

本报告的飞书文档地址：数据库第一次实验报告



commit提交记录

一、Project0

1. Task1

本任务主要实现 `skiplist` 类以及 `skipNode` 类，主要实现以下函数：

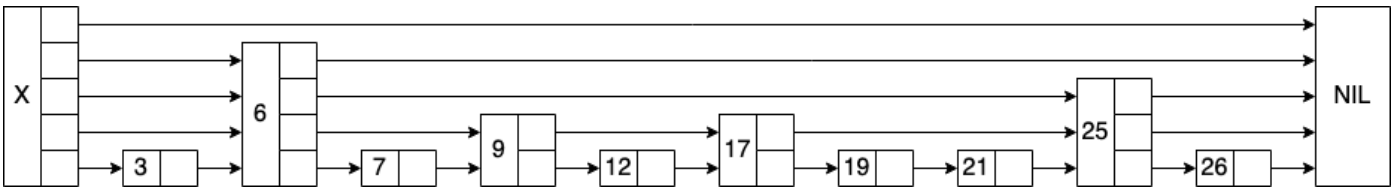
skiplist类

- `SkipNode(size_t height, K key = K{})`：构造函数，初始化变量。
- `Empty()`：返回跳表是否为空，这里在判断之前要上锁，以防并行的时候同时往跳表加了元素。
- `Size()`：返回跳表的元素个数，这里同理也要上锁。
- `Drop()`：实现对跳表析构。
- `Clear()`：实现对跳表的清空。

- `Insert(const K &key)`：插入元素，如果失败就返回false。
- `Erase(const K &key)`：删除元素，如果不存在返回false。
- `Contains(const K &key)`：查找元素，返回是否存在此元素。

skipNode类

- `Next(size_t level)`：返回level层指向的下一个节点。
- `Height()`：返回该节点的高度。
- `SetNext(size_t level, const std::shared_ptr<SkipNode> &node)`：设置level层的下一个节点。
- `Key()`：返回节点值。



1.1 设计思路

本task没有过多的设计空间，该有的成员变量都写好了，只需根据LRU-K算法进行实现即可。在对跳表进行读写操作的时候（应当包含对跳表属性的读取操作），要注意读写锁的获取。

1.2 重难点及实现

1.2.1 Contains查找函数

这个查找函数应该是增删函数的基础。

这里要先理解函数指针compare的用法，详情可以看原代码注释。

> // 这里的compare_（a,b）是默认a小于b的时候返回true

这里主要是要理解算法的内容。从节点的高位开始遍历跳表，直到这层找到或者指向 nullptr ，考虑返回结果或者降高度继遍历。代码如下：

```
SKIPLIST_TEMPLATE_ARGUMENTS auto SkipList<K, Compare, MaxHeight,
Seed>::Contains(const K &key) -> bool {
  auto readLock = Read();
  auto curr = Header();
  for (size_t i = height_ - 1; i >= 0 && i != SIZE_MAX; i--) {
    while (curr->Next(i) != nullptr && compare_(curr->Next(i)->Key(), key)) {
        curr = curr->Next(i);
```

```
      }//直到这一层没有元素或者找到一个比key大的元素，就停止，往下一层走
      if(curr->Next(i) != nullptr &&
       !compare_(curr->Next(i)->Key(), key)&&
       !compare_(key, curr->Next(i)->Key())){
        return true;
      }//如果下一个就等于key，返回true
    }
    return false;
  }
```

1.2.2 Insert插入函数

先初始化一个skipNode，然后按照查找的思路去寻找插入的位置，如果已存在返回false，这里要记录查找到时候的前一个节点（高度必定高于或等于这里将要插入的节点）。最后从插入节点的最高层开始向下走，一方面前面的节点指向这个新节点，另一方面，这个新节点的指针要指向前节点原本指向的对象。代码如下：

```
KIPLIST_TEMPLATE_ARGUMENTS auto SkipList<K, Compare, MaxHeight,
Seed>::Insert(const K &key) -> bool {
  auto writeLock = Write();
  // 预先设置高度，用来后面记录要修改指针的节点
  size_t nodeHeight = RandomHeight();
  // 保存要修改的节点
  std::shared_ptr<SkipNode> update[MaxHeight];
  //初始化update数组
  for (size_t i = 0; i < MaxHeight; i++) update[i] = Header();
  auto curr = Header();
  for (size_t i = MaxHeight - 1; i >= 0 && i !=SIZE_MAX; i--) {
    while (curr->Next(i) != nullptr && compare_(curr->Next(i)->Key(), key)) {
      curr = curr->Next(i);
    }  // 直到这一层没有元素或者找到一个比key大的元素，就停止，往下一层走
    if (curr->Next(i) != nullptr && !compare_(curr->Next(i)->Key(), key) &&
!compare_(key, curr->Next(i)->Key())) {
      return false;
    }  // 如果下一个就等于key，返回false
    update[i]=curr;
  }
  // 当找不到key时，插入节点。此时的curr指向前一个节点
  auto node = std::shared_ptr<SkipNode>(new SkipNode(nodeHeight, key));
  for (size_t i = nodeHeight - 1; i >= 0 && i != SIZE_MAX; i--) {
    node->SetNext(i,update[i]->Next(i));
    update[i]->SetNext(i,node);
  }
```

```
  size_++;
  if(node->Height() > height_)height_ = node->Height();
  return true;
}
```

1.2.3 Erase删除函数

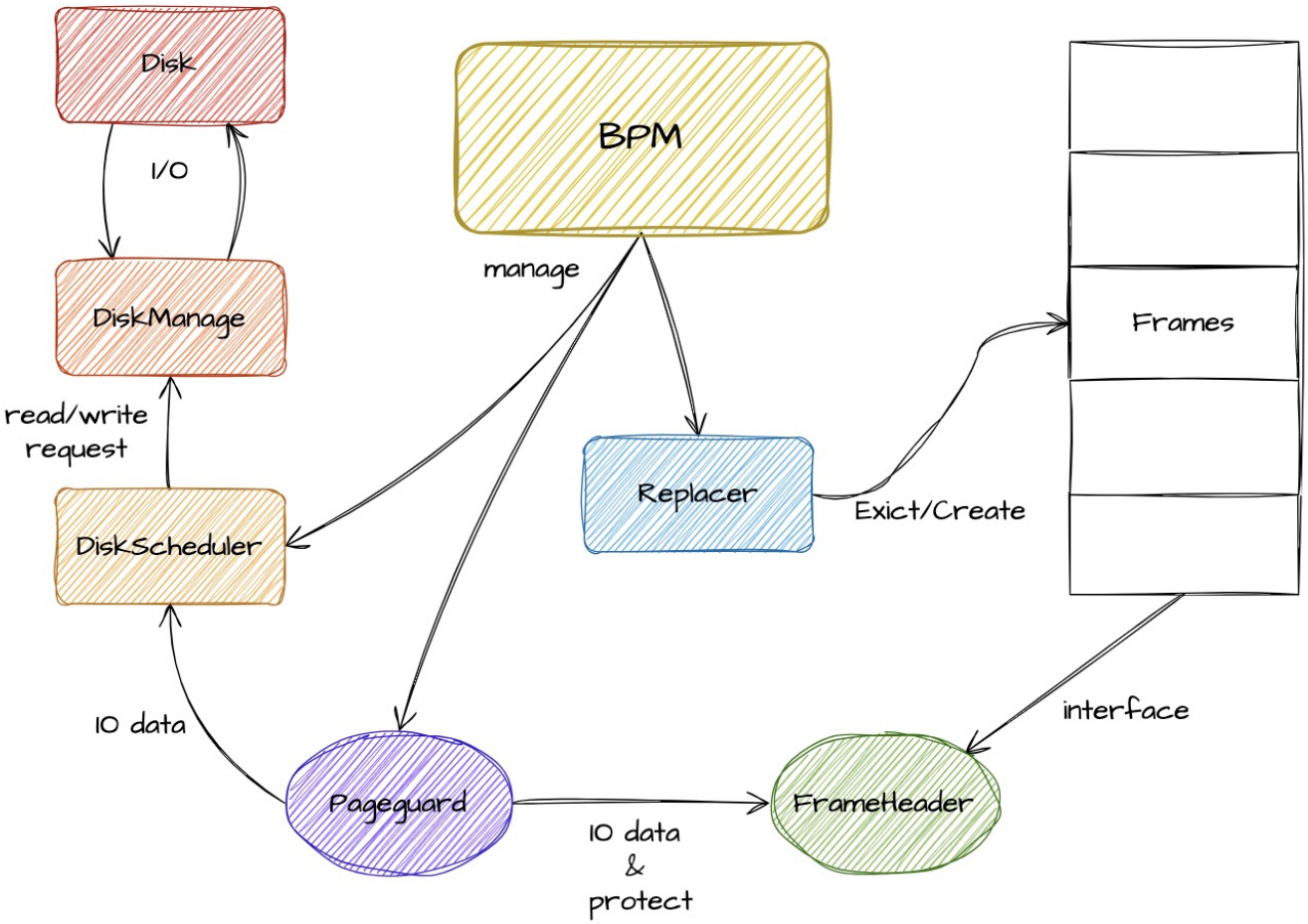
和插入函数类似，先查找，找不到返回false。找到时候记录前一个节点位置，此时操作和从链表里面删除一个节点类似，修改指针指向，然后析构节点对象。代码如下：

```
SKIPLIST_TEMPLATE_ARGUMENTS auto SkipList<K, Compare, MaxHeight,
Seed>::Erase(const K &key) -> bool {
  auto writeLock = Write();
  // 保存要修改的节点
  std::shared_ptr<SkipNode> update[MaxHeight];
  std::shared_ptr<SkipNode> target = nullptr;
  // 初始化update数组
  for (size_t i = 0; i < MaxHeight; i++) update[i] = Header();
  auto curr = Header();
  for (size_t i = height_ - 1; i >= 0 && i != SIZE_MAX; i--) {
    while (curr->Next(i) != nullptr && compare_(curr->Next(i)->Key(), key)){
      curr = curr->Next(i);
    }
    if (curr->Next(i) != nullptr && !compare_(curr->Next(i)->Key(), key) &&
!compare_(key, curr->Next(i)->Key())) {
      if (target == nullptr) target = curr->Next(i);
      update[i] = curr;
    }
  }
  if(target == nullptr)return false;
  for (size_t i = target->Height() - 1; i >= 0 && i != SIZE_MAX; i--) {
    update[i]->SetNext(i,target->Next(i));
  }
  size_--;
  // 释放target节点
  target.reset();
  return true;
  //UNIMPLEMENTED("TODO(P0): Add implementation.");
}
```

2. Task2

本任务只是在需要对跳表进行读写的时候上锁保证程序能够并发进行，比较简单，不做赘婿。

二、Project1



个人理解

1. Task1

本任务要求实现LRU-K算法，LRU-K算法是一种缓存管理机制，缓存帧Frame都规定一个k-distance，当某个缓存帧被访问k次以上时，k-distance为过往第k次的访问时间与现在时间的差值，当缓存帧的历史访问次数没有达到k次时，它的k-distance算为无穷大。当缓存区已满但需要请求缓存空间时，LRU-K算法优先清理k-distance比较大的缓存帧（都为正无穷时，取最近一次的访问时间比较老的帧删除）。

要求设计 `LRUKNode` 类和 `LRUKReplacer` 类。replacer是对缓存区里面的缓存帧Frame进行管理的类，对缓存区进行增删查改操作。

主要实现以下函数，对于简单的属性访问这里不作赘述：

LRUKReplacer类

- `Evict()`：按照LRU-K规则清理一个缓存帧，如果没有可以清理的返回'空值'。
- `RecordAccess(frame_id_t frame_id)`：对某个缓存帧增加访问记录。
- `SetEvictable(frame_id_t frame_id, bool set_evictable)`：对某个缓存帧设置状态，这个可否驱逐状态决定这个缓存帧是否可以被驱逐。
- `Remove(frame_id_t frame_id)`：删除某个缓存帧。

1.1 设计思路

在上面这些主要函数中，主要涉及的一个问题是，在驱逐的时候，如何找到应该要驱逐的缓存帧。对于两种缓存帧，没有k次记录的缓存帧的特征值是最新一次访问时间，而有k次访问记录对缓存帧对特征值是第k次访问时间，我们只需要比较这些特征值即可。一般要从没有k次访问记录的帧优先驱逐，然后再在有k次访问记录的帧中驱逐。我们这里主要添加了三个成员变量在LRUKReplacer类中：unk_list_、k_list_、timestamp_store_。

- `std::unordered_map<size_t,frame_id_t> timestamp_store_`：用一个map去存储一个键值对<特征值的访问时间戳, frame_id>，用于根据特征值寻找要修改的缓存帧。
- `std::set<size_t> unk_list_`:用set储存一个有序的时间戳序列，这里存的是没有k次记录的缓存帧的特征值，用于快速寻找要驱逐的帧，然后利用timestamp_store_寻找对应的i
- `std::set<size_t> k_list_`：和unk_list_同理，用来储存有k次访问记录的缓存帧的特征值。

然后还添加了对这三成员变量进行维护的函数`update_list(frame_id_t frame_id, bool is_new,size_t old_time)`，剩下的都是添加一些对这些成员变量进行属性描述的属性，不作赘述。

1.2 重难点及实现

1.2.1 Evict驱逐函数

先从unk_list_中寻找可以被驱逐的帧，如果没有就在k_list_中寻找，由于这两个都是有序的，这里只需要从开头开始遍历。如果都不可驱逐，就返回‘空值’。注意这个是改操作，需要上写锁。

```
1   auto LRUKReplacer::Evict() -> std::optional<frame_id_t> {
2     latch_.lock();
3
4     std::optional<frame_id_t> frame_id = std::nullopt;
5     //寻找最远的可驱逐的frame,先从unk_list找:
6     if(!unk_list_.empty()){
7       for (size_t timestamp : unk_list_)
8         {
9           auto it_frame = timestamp_store_.find(timestamp);
10          if (it_frame == timestamp_store_.end()) continue;
```

```
11
12          auto it_Node = node_store_.find(it_frame->second);
13          if (it_Node == node_store_.end()) continue;
14
15          if (it_Node->second.get_is_evictable()) {
16            frame_id = it_frame->second;
17            unk_list_.erase(timestamp);  // ✔ OK: 你后续不再访问 `it`，立即 break
18            break;
19          }
20        }
21    }
22    else if(!k_list_.empty()){//再从k_list找
23      for (size_t timestamp : k_list_) {
24        auto it_frame = timestamp_store_.find(timestamp);
25        if (it_frame == timestamp_store_.end()) continue;
26        auto it_Node = node_store_.find(it_frame->second);
27        if (it_Node == node_store_.end()) continue;
28        if (it_Node->second.get_is_evictable()) {
29          frame_id = it_frame->second;
30          k_list_.erase(timestamp);
31          break;
32        }
33      }
34    }
35  if (frame_id.has_value()) {
36      Remove(frame_id.value());
37    }
38    latch_.unlock();
39    return frame_id;
40  }
41
```

1.2.2 update_list有序序列维护函数

当对某个缓存帧增加访问次数的时候，它可能到达了访问次数k，这时候要把它从unklist拿出并放到klist中。

```
1   void LRUKReplacer::update_list(frame_id_t frame_id, bool is_new,size_t
    old_time){
2     auto it = node_store_.find(frame_id);
3     if(it == node_store_.end()){return;}
4
5     if(it->second.get_history_size() < k_){//如果小于k_
6       if(is_new){//如果新放进去的
7         unk_list_.insert(it->second.get_k_distance());
8       }else{//如果原本就在里面
```

```
      //删掉之前记录
      // auto it_list = std::lower_bound(unk_list_.begin(), unk_list_.end(), old_time);
      unk_list_.erase(old_time);

      unk_list_.insert(it->second.get_k_distance());
      }
    }
    else{ //如果等于k_
      if (!it->second.get_in_k_list()){//原本不在k里面
        // 删掉之前记录
        // auto it_list = std::lower_bound(unk_list_.begin(), unk_list_.end(), old_time);
        unk_list_.erase(old_time);

        k_list_.insert(it->second.get_k_distance());
        it->second.set_in_k_list(1);
      }
      else{
        // auto it_list = std::lower_bound(k_list_.begin(), k_list_.end(), old_time);
        k_list_.erase(old_time);

        k_list_.insert(it->second.get_k_distance());
      }
    }
    return;
  }
  }  // namespace bustub

```

`set_in_k_list` 是LRUKNode中新加的函数，用来更新这个缓存帧的特征值，因为是否达到k次访问会影响节点特征值的计算。

```
size_t update_history(size_t timestamp) {//增加新的
    if(history_size_ < k_){
      history_.push_front(timestamp);
      history_size_++;
    }
    else{
      history_.pop_back();
      history_.push_front(timestamp);
    }

    if(history_size_ == k_){
```

```
        //更新k距离
        k_distance_ = history_.back();
      }
      else{
        k_distance_ = history_.front();
      }
      return k_distance_;
    }
```

1.2.3 RecordAccess增加访问记录函数

每次增加一个记录，都要递增replacer中的时间戳。在缓冲区中找有无这个frameid，如果没有，尝试去申请一个新的节点。如果有，则更新有序序列list以及节点的访问信息。

```
void LRUKReplacer::RecordAccess(frame_id_t frame_id, [[maybe_unused]] AccessType access_type){
  latch_.lock();

  current_timestamp_++;//增加时间戳
  auto it = node_store_.find(frame_id);
  if(it == node_store_.end()) {
    if (node_store_size_ == replacer_size_) {
      latch_.unlock();//Evict里面又上了一次互斥锁
      Evict();
      latch_.lock();
    }
    //创建新的node放进去
    LRUKNode new_node(frame_id, k_);

timestamp_store_.insert({new_node.update_history(current_timestamp_),frame_id});

    node_store_.insert({frame_id, std::move(new_node)});
    node_store_size_++;
    update_list(frame_id ,1,SIZE_T_MAX);
    // curr_size_++;
  }
  else{//找到这个id了
    size_t old_time = node_store_.find(frame_id)->second.get_k_distance();
    timestamp_store_.erase(timestamp_store_.find(old_time));
    timestamp_store_.insert({(it->second).update_history(current_timestamp_), frame_id});
    update_list(frame_id, 0, old_time);
  }
  latch_.unlock();
}
```

```
29
```

2. Task2

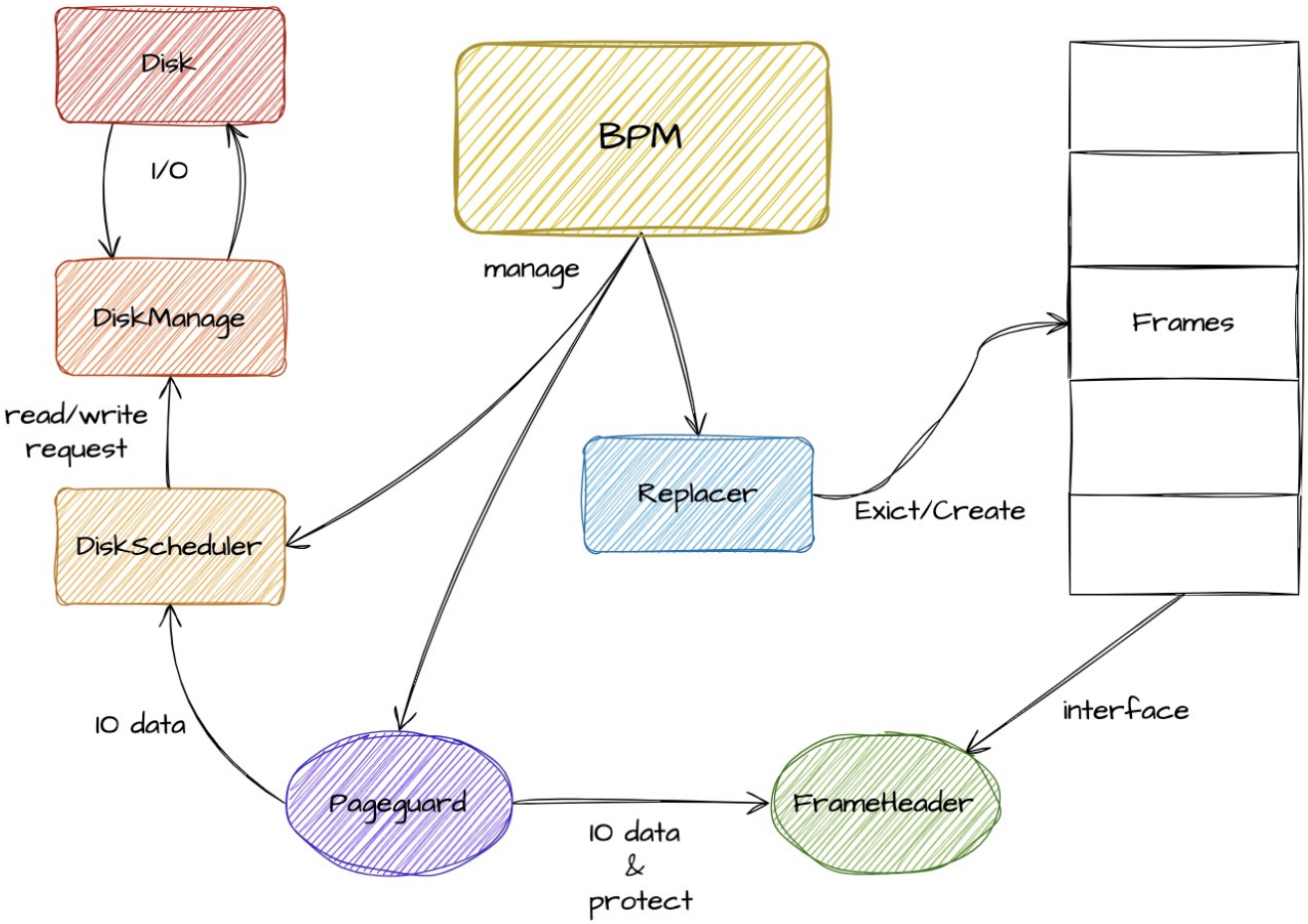
本任务要求我们实现一个Disk Scheduler类，用来对读写磁盘请求进行调度和处理。这个scheduler其实就是在构建一个处理进程，当有读写磁盘需求的时候，调用DiskManage来进行数据的读写。

本节没有过多的要设计的地方，主要是理解这个scheduler的工作方式即可。当这个scheduller创建的时候，会进行一个死循环，一直处理加入的读写请求进程。当析构的时候，放入一个空进程，这个死循环就会终止。

主要函数实现：

```
1   void DiskScheduler::StartWorkerThread() {
2       while(true){
3           auto request = request_queue_.Get();
4           // 如果r为nullopt，就退出循环
5           if(!request.has_value()){
6               break;
7           }
8           if(request->is_write_){
9               disk_manager_->WritePage(request->page_id_, request->data_);
10          }else{
11              disk_manager_->ReadPage(request->page_id_, request->data_);
12          }
13          //回调函数
14          request->callback_.set_value(true);
15      }
16  }
```

3. Task3



个人理解

3.1 设计思路

本节主要是结合前两个task的内容，设计bpm来对整体进行管理，主要涉及设计ReadPageGuard类和WritePageGuard以及BufferPoolManager类。难点主要是要理解这个框架。

下面我们来简单介绍一下每个类是什么功能：

- BPM：协同每个类进行工作，对数据读写、缓存区进行管理的入口。
- Replacer：管理缓存区的帧
- FrameHeader：是缓存帧的接口，只能通过这个类来对缓存帧进行读写操作。
- Pageguard：相当于对某个缓存帧的读写请求，可以理解为用来获取读锁和写锁。并且同时处理缓存区与disk中数据的交互。

3.2 Pageguard重难点及实现

在ReadPageGuard类和WritePageGuard类中，要实现的函数其实大同小异，只是读写锁的区别，这里只对ReadPageGuard中的主要函数进行分析：

- `operator=(ReadPageGuard &&that)`：移动赋值函数，这里涉及移动构造，就是将变量的生命周期用move赋值给另一个对象，相当于把东西全部搬到另一个对象中。

- `ReadPageGuard(ReadPageGuard &&that)`：移动构造，和移动赋值函数类似。
- `ReadPageGuard`：构造函数。
- `is_valid_`：这个变量代表着给pageguard是否掌握着有效资源，如果为false，就不需要析构，而且这个对象的成员变量都为空。
- `Flush()`：刷新disk中的数据，如果缓存帧里面的数据被修改，则用这个函数将修改之后的数据写回到disk磁盘中进行数据更新。

3.2.1 构造函数

这里除了初始化变量之外，要对frameheader进行加pin以及将这个访问的frame通过frameheader设置不可驱逐，这样意味着此时有一个对这个frame进行读写访问的pageguard，防止缓存被清理。

```
1   ReadPageGuard::ReadPageGuard(page_id_t page_id, std::shared_ptr<FrameHeader>
    frame,
2                                std::shared_ptr<LRUKReplacer> replacer,
    std::shared_ptr<std::mutex> bpm_latch,
3                                std::shared_ptr<DiskScheduler> disk_scheduler)
4       : page_id_(page_id),
5         frame_(std::move(frame)),
6         replacer_(std::move(replacer)),
7         bpm_latch_(std::move(bpm_latch)),
8         disk_scheduler_(std::move(disk_scheduler)) {
9     //上锁
10    frame_->rwlatch_.lock_shared();
11    //bpm_latch_->lock();//这里需要上锁吗？？？
12    frame_->pin_count_.fetch_add(1);
13
14    this->is_valid_ = true;
15
16    // 这里把frame设置成不可驱逐
17    replacer_->SetEvictable(frame_->frame_id_, false);
18  }
```

3.2.2 移动构造

移动构造函数比较简单，这里直接移动完之后对被移动的对象进行析构即可。

移动赋值函数要考虑多一点。如果被移动的和本身一致（别名），就不进行移动，防止自己被释放。如然后移动之前要把自己先释放一下资源。

```
1   auto ReadPageGuard::operator=(ReadPageGuard &&that) noexcept -> ReadPageGuard
    & {
```

```
  if (this == &that) {  // 当我移动到自己的引用的时候会删掉自己
    return *this;
  }
  // if (!that.is_valid_) {
  //     return that;
  //   }
    if (this->is_valid_) {this->Drop();}//如果this持有资源就释放

    // 这里为什么不能move
    this->page_id_ = that.page_id_;
    this->frame_ = std::move(that.frame_);
    this->replacer_ = std::move(that.replacer_);
    this->bpm_latch_ = std::move(that.bpm_latch_);
    this->disk_scheduler_ = std::move(that.disk_scheduler_);
    this->is_valid_ = that.is_valid_;

    that.is_valid_ = false;
    that.page_id_ = INVALID_PAGE_ID;
    that.frame_ = nullptr;
    that.replacer_ = nullptr;
    that.bpm_latch_ = nullptr;
    that.disk_scheduler_ = nullptr;

    return *this;
}
```

3.2.3 Flush刷新函数

没有太难点，主要是这里设置一个future来实现异步操作，等待读写进程完成之后才继续进行代码，确保读写操作完成。

```
void ReadPageGuard::Flush() {
  // 如果没有被修改,就不需要送回磁盘修改了
  if (!IsDirty()) {
    return;
  }
  auto promise = disk_scheduler_->CreatePromise();
  auto future = promise.get_future();
  disk_scheduler_->Schedule({true, frame_->GetDataMut(), frame_->page_id_,
std::move(promise)});
  future.get();
  }
```

3.3 BufferPoolManager重难点及实现

BPM主要是实现对数据读写以及缓存区的管理。下面是主要实现函数的分析：

- `NewPage()`：新建一个Page，并申请一个frame来存储数据，如果没有空闲的frame，则新建失败。这里要注意的是，新建的要被设置成不可驱逐。
- `DeletePage(page_id_t page_id)`：从磁盘里面删除某一页，并删除缓存区里面的内容。
- `CheckedWritePage`：用于建立pageguard。
- `GetPinCount(page_id_t page_id)`：获取这个页面目前的访问数目，用于判断这个缓存区的页面能否被清除。
- `GetFreeFrame()`：获取空闲的帧，利用replacer申请一个新帧，用来存新页面或者将要访问页面的数据。

3.3.1 CheckedReadPage函数

本task这个是比较主要的函数。想访问某一个页面。如果在缓冲区里面没有找到这个页面，则在缓存区里面创建这个页面的新缓存，初始化操作和前面讲的一致。如果有这个页面，就获取这个页面的帧。最后增加对该缓存帧的访问次数以及返回初始化pageguard对象。

这里有个疑点就是需不需要上全局锁，假设我们有两个进程同时对同一个page创建pageguard，但是这个pageguard在这个缓存区里面没有，就会导致在缓存区里面出现两同一页面的缓存，导致数据不一致，以及数据覆盖的隐患。

当我尝试在这里上全局锁的时候，导致了死锁，但是我并没有找到原因。

```
1   auto BufferPoolManager::CheckedWritePage(page_id_t page_id, AccessType
    access_type) -> std::optional<WritePageGuard> {
2     frame_id_t frame_id = INVALID_FRAME_ID;
3     if (page_id == INVALID_PAGE_ID) {
4       return std::nullopt;
5     }
6     auto it = page_table_.find(page_id);
7     if(it == page_table_.end()){//找不到这个页面
8
9       frame_id = GetFreeFrame();
10      if(frame_id == INVALID_FRAME_ID){//如果没有多余的帧了
11        return std::nullopt;
12      }
13      frames_[frame_id]->page_id_ = page_id;
14
15      // 读取页面
16      auto promise = disk_scheduler_->CreatePromise();
17      auto future = promise.get_future();
18      disk_scheduler_->Schedule({false, frames_[frame_id]->GetDataMut(),
    page_id, std::move(promise)});
```

```
      future.get();

      // 加入关系对
      page_table_.insert({page_id, frame_id});
    }
    else{
      frame_id = it->second;
    }
    // 增加一次访问
    replacer_->RecordAccess(frame_id);
    //每次都设置一次不可驱逐
    replacer_->SetEvictable(frame_id,false);

    WritePageGuard guard(page_id, frames_[frame_id], replacer_, bpm_latch_,
disk_scheduler_);

    return std::optional<WritePageGuard>(
      std::move(guard)
      );
```