**buffer pool**

我们将构建一个面向磁盘的数据库系统，这样的一个数据库系统的主要存储位置是磁盘。首先我们要在存储管理系统中实现一个缓冲池(**buffer pool**),缓冲池负责物理页面和主存之间数据的来回移动。缓冲池允许数据库管理系统处理大于系统可用内存的数据。缓冲池的操作对数据库的其他部分是透明的。例如，系统利用唯一的标识符**page\_id\_t**向缓冲池请求一个页面，系统不知道这个页面是已经在内存中了还是需要从磁盘中将该页面取出。

我们的实现将是线程安全(**thread-safe**)的。由于多个线程可能同时访问内部数据结构，因此我们需要确保它们的关键部分受到**锁存器(latches)**的保护(在操作系统中被称为锁，即lock)。

在本部分，我们将实现如下功能：

* **LRU换页策略**
* **缓冲池管理器实例**
* **并行的缓冲池管理器**

**一．LRU换页策略**

这一组件用于追踪缓冲池中每一页的使用情况。我们将在**src/include/buffer/lru\_replacer.h**以及与其相关的实现文件**src/buffer/lru\_replacer.cpp**中实现类**LRUReplacer**。**LRUReplacer**继承自抽象类**Replacer(src/include/buffer/replacer.h)**，该文件包含各函数的功能说明。

**LRUReplacer**的最大页数与缓冲池的大小相同，这是因为**LRUReplacer**包含**BufferPoolManager**中所有帧的占位符。然而，在任意时刻，并非所有帧一定都在**LRUReplacer**中。初始化**LRUReplacer**中没有任何帧。只有新的取消占用(占用=固定)(**unpinned**)的帧将被考虑放入**LRUReplacer**中。

我们将实现如下方法：

* **Victim(frame\_id\_t\*)** : 在**Replacer**追踪的元素中，删除最近最少访问的对象，在输出参数中存储该对象的内容并返回**True**。如果**Replacer**是空的，返回**False**。
* **Pin(frame\_id\_t)** :在将一个页面固定到**BufferPoolManager**的帧后该方法应该被调用。该方法将从**LRUReplacer**中移除该被固定到**BufferPoolManager**的帧。
* **Unpin(frame\_id\_t)** : 当一个页面的**pin\_count**变为0后该方法应该被调用。该方法将包含取消固定的页面的帧移入**LRUReplacer**中。
* **Size()** : 该方法返回目前在**LRUReplacer**中帧的数量。

**实现细节注意**：

**①std::scoped\_lock lock{mutex\_}**;用于申请单个互斥锁。

c++中为了避免new运算符申请的内存块因为“人为”错误忘记释放而带来**内存泄露**，在新标准中引入了**智能指针**。智能指针名称上有指针两个字，但是它其实是一个模板类定义的对象，只是这个对象有指针的一些行为。用智能指针对代码中所使用的实际对象类型进行了包装，也就是在创建智能指针的时候传入了新申请的实际对象。智能指针在其构造函数中初始化，在析构函数中进行释放。正因为是对象，所以**在对象作用域结束的时候，析构函数会自动调用，这样就避免了内存泄露**。类似的，因为使用锁的话需要上锁，需要解锁，如果对锁**忘记解锁**的话，就会很麻烦。所以**智能锁**就出现了。智能锁就是**scoped\_lock**。智能锁的另一好处：可以避免死锁。下面介绍这方面的内容。

本节中，我们将会故意的创造一个**死锁**的情况。然后，在相同资源的情况下，如何创造出一个死锁的情形。再使用**C++17**中，STL的std::scoped\_lock如何避免死锁的发生。

**How to do it…**

本节中有两对函数要在并发的线程中执行，并且有两个互斥量。其中一对制造死锁，另一对解决死锁。主函数中，我们将使用两个互斥量：

1.**包含必要的头文件，并声明所使用的命名空间**：

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

using namespace chrono\_literals;

2.**实例化两个互斥量对象，制造死锁**：

mutex mut\_a;

mutex mut\_b;

3.为了使用两个互斥量制造死锁，我们需要有两个函数。其中一个函数试图对互斥量A进行上锁，然后对互斥量B进行上锁，而另一个函数则试图使用相反的方式运行。让两个函数在等待锁时进行休眠，则这段代码永远处于一个死锁的状态。(这就达到了我们演示的目的。当我们重复运行程序，那么程序在没有任何代码一直休眠时，**可能会有成功运行的情况**。)需要注意的是，这里我们没有使用\n字符作为换行符，我们使用的是endl。endl会输出一个换行符，同时也会对cout的流缓冲区进行刷新，所以我们**可以确保打印信息不会有延迟或同时出现**：

static void deadlock\_func\_1()

{

cout << "bad f1 acquiring mutex A..." << endl;

lock\_guard<mutex> la {mut\_a};

this\_thread::sleep\_for(100ms);

cout << "bad f1 acquiring mutex B..." << endl;

lock\_guard<mutex> lb {mut\_b};

cout << "bad f1 got both mutexes." << endl;

}

4.deadlock\_func\_2和deadlock\_func\_1看起来一样，就是A和B的顺序相反：

static void deadlock\_func\_2()

{

cout << "bad f2 acquiring mutex B..." << endl;

lock\_guard<mutex> lb {mut\_b};

this\_thread::sleep\_for(100ms);

cout << "bad f2 acquiring mutex A..." << endl;

lock\_guard<mutex> la {mut\_a};

cout << "bad f2 got both mutexes." << endl;

}

5.现在我们看两个无死锁版本的函数。它们使用了scoped\_lock，其会将作为构造函数参数的所有互斥量进行上锁。其析构函数会进行解锁操作。锁定这些互斥量时，其内部应用了**避免死锁**的策略。这里需要注意的是，两个函数还是对A和B互斥量进行操作，并且顺序相反：

static void sane\_func\_1()

{

scoped\_lock l {mut\_a, mut\_b};

cout << "sane f1 got both mutexes." << endl;

}

static void sane\_func\_2() {

scoped\_lock l {mut\_b, mut\_a};

cout << "sane f2 got both mutexes." << endl;

}

6.主函数中观察这两种情况。首先，我们使用不会死锁的函数：

int main()

{

{

thread t1 {sane\_func\_1};

thread t2 {sane\_func\_2};

t1.join();

t2.join();

}

7.然后，调用制造死锁的函数：

{

thread t1 {deadlock\_func\_1};

thread t2 {deadlock\_func\_2};

t1.join();

t2.join();

}

}

8.编译并运行程序，就能得到如下的输出。前两行为无死锁情况下，两个函数的打印结果。接下来的两个函数则产生死锁。因为我们能看到f1函数始终是在等待互斥量B，而f2则在等待互斥量A。这两个函数都没有成功的对两个互斥量上锁。我们可以让这个程序持续运行，不管时间是多久，结果都不会变化。程序只能从外部杀死，这里我们使用Ctrl + C的组合键，将程序终止：

$ ./avoid\_deadlock

sane f1 got both mutexes

sane f2 got both mutexes

bad f2 acquiring mutex B...

bad f1 acquiring mutex A...

bad f1 acquiring mutex B...

bad f2 acquiring mutex A...

**How it works…**

例子中，我们故意制造了死锁，我们也了解了这样一种情况发生的有多快。在一个很大的项目中，多线程开发者在编写代码的时候，都会共享一些互斥量用于保护资源，所有开发者都需要遵循同一种加锁和解锁的顺序。这种策略或规则是很容易遵守的，不过也是很容易遗忘的。

scoped\_lock对于这种情况很有帮助。其实在C++17中添加，其工作原理与lock\_guard和unique\_lock一样：其构造函数会进行上锁操作，并且析构函数会对互斥量进行解锁操作。scoped\_lock特别之处是，**可以指定多个互斥量**。

scoped\_lock使用std::lock函数，其会调用一个特殊的算法对所提供的互斥量调用try\_lock函数，这是为了避免死锁。因此，在加锁与解锁的顺序相同的情况下，使用scoped\_lock或对同一组锁调用std::lock都是非常安全的。

②**unordered\_map用法**

**1.find(key)**：查找以 key为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（即end()方法返回的迭代器）。

2.可以调用unordered\_map的成员函数erase()来移除元素。参数可以是标识元素的**一个键**或是指向它的一个**迭代器**。当参数是键时，erase() 会返回一个整数，它是移除元素的个数，所以0表示没有找到匹配的元素。当参数是迭代器时，返回的迭代器指向被移除元素后的元素。下面是一些示例：

auto n = people.erase ("Jim");// Returns 0 if key not found

auto iter = people.find ("May") ; // Returns end iterator if key not found

if(iter != people.end())

iter = people.erase (iter) ;// Returns iterator for element after "May"

也可以移除指定的一个元素序列。例如：

//Remove all except 1st and last

auto iter = people.erase(++std:rbegin(people),--std:rend(people));

③list用法

erase() 成员函数有以下 2 种语法格式：

iterator erase (iterator position);

iterator erase (iterator first, iterator last);

利用第一种语法格式，可实现删除list容器中position迭代器所指位置处的元素，例如：

#include <iostream>

#include <list>

using namespace std;

int main()

{

list<int>values{ 1,2,3,4,5 };

//指向元素 1 的迭代器

auto del = values.begin();

//迭代器右移，改为指向元素 2

++del;

values.erase(del); //{1,3,4,5}

for (auto begin = values.begin(); begin != values.end(); ++begin)

{

cout << \*begin << " ";

}

return 0;

}

运行结果为：

1 3 4 5

利用第二种语法格式，可实现删除list容器中first迭代器和last迭代器限定区域内的所有元素（包括first指向的元素，但不包括last指向的元素）。例如：

#include <iostream>

#include <list>

using namespace std;

int main()

{

list<int>values{ 1,2,3,4,5 };

//指定删除区域的左边界

auto first = values.begin();

++first;//指向元素 2

//指向删除区域的右边界

auto last = values.end();

--last;//指向元素 5

//删除 2、3 和 4

values.erase(first, last);

for (auto begin = values.begin(); begin != values.end(); ++begin)

{

cout << \*begin << " ";

}

return 0;

}

运行结果为：

1 5

erase()成员函数是按照被删除元素所在的位置来执行删除操作，如果想根据元素的值来执行删除操作，可以使用remove()成员函数。例如：

#include <iostream>

#include <list>

using namespace std;

int main()

{

list<char>values{'a','b','c','d'};

values.remove('c');

for (auto begin = values.begin(); begin != values.end(); ++begin)

{

cout << \*begin << " ";

}

return 0;

}

运行结果为：

a b d

**二．缓冲池管理实例**

接下来，我们实现实现缓冲池管理功能(**BufferPoolManagerInstance**)。**BufferPoolManagerInstance**负责从**DiskManager**中获取数据库页面，并将其存储在主存中。在明确指示或者需要换出旧页面为新页面腾出空间时，**BufferPoolManagerInstance**可以将脏页写入磁盘。注意，**DiskManager**负责将数据从磁盘中读出以及将内存中数据写入磁盘。

数据库系统中所有在主存中的页面都用**Page**对象表示。**BufferPoolManagerInstance**不需要理解这些页的内容。但是作为系统开发者，我们需要明确，**Page**对象就是缓冲池中存储内容的容器，因此它不和某个特定页面相关联。也就是说，每个**Page**对象都包含一块内存，**DiskManager**将它从磁盘中读取的物理页面的内容复制到该位置。当**BufferPoolManagerInstance**从磁盘中读写数据时，它将重用相同的**Page**对象来存储数据。这意味着在系统的生命周期内，相同的**Page**对象可能包含不同的物理页面。**Page**对象的标识符(**page\_id**)跟踪它包含的物理页面。如果一个**Page**对象不包含任何物理页面，那么它的**page\_id**必须设定为**INVALID\_PAGE\_ID**。

每个**Page**对象还维护一个计数器，用于统计固定了该页面的线程数。**BufferPoolManagerInstance**不允许释放一个被固定住的页面。每个**Page**对象还时刻追踪它是否是脏页，我们需要记录在一个页面被取消固定前它是否被修改。在**Page**对象可被重用前，**BufferPoolManagerInstance**必须将脏**Page**的内容写回磁盘。

**BufferPoolManagerInstance**将使用之前实现的**LRUReplacer**类。它将使用**LRUReplacer**来跟踪**Page**对象的访问时间，这样当**BufferPoolManagerInstance**需要为磁盘中的一个新物理页面腾出一个空闲帧时，它可以确定应该将哪个帧移出。

我们将实现头文件src/include/buffer/buffer\_pool\_manager\_instance.h以及源文件src/buffer/buffer\_pool\_manager\_instance.cpp中的如下函数：

* **FetchPgImp(page\_id)**
* **UnpinPgImp(page\_id, is\_dirty)**
* **FlushPgImp(page\_id)**
* **NewPgImp(page\_id)**
* **DeletePgImp(page\_id)**
* **FlushAllPagesImpl()**

如果在空闲列表中没有可用的页面并且所有的页面都被占用，**FetchPgImp**应该返回NULL。不论页面的状态如何(pin或unpin)，**FlushPgImp**都应该刷新页面。

对于**UnpinPgImp**，参数**is\_dirty**用于页面被占用时追踪该页是否被修改过。

注意：**Pin**和**Unpin**在**LRUReplacer**和**BufferPoolManagerInstance**中有着相反的含义，在**LRUReplacer**中，一个页面被pin意味着我们不应该删除该页面,因为该页面正在被使用。这意味着我们应该将该页面从**LRUReplacer**中移除。另一方面，在**BufferPoolManagerInstance**中一个页面被pin意味着我们想使用该页面，那么我们不应将该页从缓冲池中移除。

**实现细节：**

**1.<atomic>：**原子类型是封装了一个值的类型，它的访问保证不会导致数据的竞争，并且可以用于在不同的线程之间**同步内存访问**。

**2.push\_back()和emplace\_back()**

emplace\_back()函数是 C++ 11 新增加的，其功能和push\_back()相同，都是在vector容器的尾部添加一个元素。emplace\_back()和push\_back()的区别，就在于底层实现的机制不同。push\_back()向容器尾部添加元素时，首先会创建这个元素，然后再将这个元素拷贝或者移动到容器中（如果是拷贝的话，事后会自行销毁先前创建的这个元素）；而 emplace\_back()在实现时，则是直接在容器尾部创建这个元素，省去了拷贝或移动元素的过程。因此，在实际使用时，建议优先选用emplace\_back()。

**3.成员函数AllocatePage和ValidatePageId的含义：**

这两个函数主要是为了应对有并行缓冲池时，各缓冲池应该如何分配页号的问题。距离来说，如果有三个缓冲池，那么0号缓冲池应该分配的页号是0，3，6等，1号缓冲池分配1，4，7等，2号缓冲池分配2，5，8等。

**4.先看NewPgImp函数**

首先应该理解好unordered\_map<page\_id\_t, frame\_id\_t> page\_table\_;的含义，其实它就相当于一个缓冲池实例的页表。其中page\_id\_t是以磁盘文件db\_file为中心的，该变量表示如果一个Page被写入磁盘，则它在数据库磁盘文件db\_file中的偏移量应该是多少；而frame\_id\_t是以一个缓冲池实例为中心的，它表示该Page在对应缓存池实例的pages\_数组中的下标。

然后再梳理一下数据库是如何将内存中的数据写入磁盘中的，这与文件src\storage\disk\disk\_manager.cpp有关，首先构造函数打开磁盘中的数据库文件：

db\_io\_.open(db\_file, std::ios::binary | std::ios::in | std::ios::out);

然后在WritePage(page\_id\_t page\_id, const char \*page\_data)函数中，先根据page\_id得到page\_data应该写到db\_file的哪个位置，然后向db\_file中写入page\_data

**三．并行的缓冲池管理**

之前实现的单例缓冲池管理实例为了实现线程安全需要锁存器。但当若干个线程与缓冲池交互争夺一个锁存器时，这会导致性能问题。一个潜在的解决方案是为数据库系统分配多个缓冲池，每个缓冲池都有自己的锁存器。

**ParallelBufferPoolManager**是有着多个**BufferPoolManagerInstance**的类。对于每个操作，**ParallelBufferPoolManager**选取一个**BufferPoolManagerInstance**实例并调用该实例。

我们依据给定的page id来决定使用哪个**BufferPoolManagerInstance**。如果我们有**num\_instances**个**BufferPoolManagerInstance**，那么我们要用某种方式将给定的page id映射到[0,num\_instances)之间，在这里我们使用模运算**page\_id mod num\_instances**将给定的page\_id映射到该区间。

当**ParallelBufferPoolManager**初始化的时候，他的起始索引应该是0。此后每创建一个新的页面，都应从起始索引开始尝试每个**BufferPoolManagerInstance**直到某个成功。然后将起始索引加1。

创建**BufferPoolManagerInstances**时，构造函数应使用uint32\_t num\_instances和uint32\_t instance\_index，来保证页面id被正确地创建。

我们将实现头文件src/include/buffer/parallel\_buffer\_pool\_manager.h和源文件src/buffer/parallel\_buffer\_pool\_manager.cpp中的如下函数：

* **ParallelBufferPoolManager(num\_instances, pool\_size, disk\_manager, log\_manager)**
* **~ParallelBufferPoolManager()**
* **GetPoolSize()**
* **GetBufferPoolManager(page\_id)**
* **FetchPgImp(page\_id)**
* **UnpinPgImp(page\_id, is\_dirty)**
* **FlushPgImp(page\_id)**
* **NewPgImp(page\_id)**
* **DeletePgImp(page\_id)**
* **FlushAllPagesImpl()**