Buffer Manager

熊郁文 3130000829

实验需求

Buffer Manager负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换
3. 记录缓冲区中各页的状态，如是否被修改过等
4. 提供缓冲区页的pin功能，及锁定缓冲区的页，不允许替换出去 为提高磁盘I/O操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍，一般可定为4KB或8KB。

实现原理

我们针对Buffer Manager模块（区别于只操作Buffer的Buffer Manager，在代码中我们将整个模块称之为Paged File(PF) Manager），定义了可供上层模块使用的三个类，Record Manager与Index Manager以及Catalog Manager可利用这三个类来完成对文件部分的操作（关于Buffer部分的操作将被隐藏在此模块中，需要实现公共接口的使用的私有函数也未包含在此处）：

```C++
class PF_Manager {
public:
    PF_Manager();
    ~PF_Manager();
    RC CreateFile(const char *fileName);
    RC DestroyFile(const char *fileName);
    RC OpenFile(const char *fileName, PF_FileHandle &fileHandle);
    RC CloseFile(PF_FileHandle &fileHandle);
};

class PF_FileHandle {
public:
    PF_FileHandle();
    ~PF_FileHandle();
    PF_FileHandle (const PF_FileHandle &fileHandle);
    PF_FileHandle& operator= (const PF_FileHandle &fileHandle);

    RC GetThisPage(PageNum pageNum, PF_PageHandle &pageHandle) const;
    RC AllocatePage(PF_PageHandle &pageHandle);
    RC DisposePage(PageNum pageNum);
    RC MarkDirty(PageNum pageNum) const;
    RC UnpinPage(PageNum pageNum) const;
    RC ForcePages(PageNum pageNum = ALL_PAGES);
    RC FlushPages();
};

class PF_PageHandle {
    PF_PageHandle();
    ~PF_PageHandle();
    PF_PageHandle(const PF_PageHandle &pageHandle);
    PF_PageHandle& operator= (const PF_PageHandle &pageHandle);
    RC GetData(char* &data) const;
    RC GetPageNum(PageNum &pageNum) const;
};
```

其中 PF_Manager类对上层模块提供文件的创建、删除、打开和关闭接口，上层模块可调用这四个接口来对文件进行操，OpenFile与CloseFile需要传入PF_FileHandle类的一个实例，来获得文件句柄。

PF_FileHandle类可提供get page, allocate page, dispose page, mark dirty page, unpin page(在get page时将会自动pin page), force page等功能。Get page与allocate page需要一个PF_PageHandle的实例作为参数来获得页的句柄。

PF_PageHandle类提供对一个page的操作，提供了对page num与data的获取接口，其中get data部分调用者将会获得实际指向page那一块内存的指针，可以直接修改其中的数据再mark dirty即可，为了简化实现因此未提供set data接口。

以上三个类已可提供完备的基于page的文件操作，但为了提高性能，必须使用buffer。我们在PF模块中还定义了内部所使用的Buffer Manager：

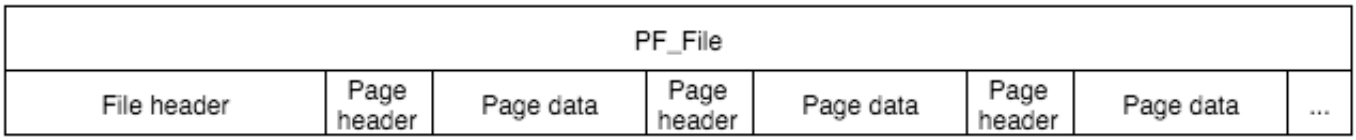
```C++
class PF_BufferManager {
public:
    PF_BufferManager(int numPages);
    ~PF_BufferManager();
    RC GetPage(FILE* fd, PageNum pageNum, char* &buffer, bool multiplePins = true);
    RC AllocatePage(FILE* fd, PageNum pageNum, char* &buffer);
    RC MarkDirty(FILE* fd, PageNum pageNum);
    RC UnpinPage(FILE* fd, PageNum pageNum);
    RC ForcePages(FILE* fd, PageNum pageNum);
    RC FlushPages(FILE* fd);

private:
    RC InsertFree(int slot);
    RC MakeMRU(int slot);
    RC LinkHead(int slot);
    RC Unlink(int slot);
    RC InternalAlloc(int &slot);

    RC ReadPage(FILE* fd, PageNum pageNum, char *dest);
    RC WritePage(FILE* fd, PageNum pageNum, char *source);

    RC InitPageDesc(FILE* fd, PageNum pageNum, int slot);

    std::vector<PF_BufferPageDesc> bufTable;
    PF_HashTable hashTable;
    int numPages;
    size_t pageSize;
    std::list<int> free;
    std::list<int> used;
};
```

每一个PF_Manager实例将对应一个PF_BufferManager实例，可以看出PF_BufferManager的公共接口与PF_FileHandle并无区别，PF_FileHandle类在处理完文件头之后将会调用这些函数，需要传入一个文件指针是为了处理一个PF_Manager打开多个文件生成多个File Handle的情况。

PF_BufferManager内部有一个HashTable来维护(FILE*, pageNum)->buffer中的slot的对应关系，free, used两个链表来维护当前可用与已使用的slot, used 链表中的顺序即为使用顺序（即需要替换页时将会从链表尾部开始寻找unpinned page），通过MakeMRU来维护正确的链表顺序。

实现细节

PF_Manager创建的文件结构如下图所示：



其中文件开始的4096 bytes记录了file header，包括page的数量以及文件中第一个free page的位置。

file header每4096 bytes一个page，page开头的4 byte为page header，如果当前page为free page，page header指向下一个free page，否则page header的值为-1。