
Practicum in Database Systems

Instructor: Hao-Hua Chu

Fall Semester, 2005

Practicum Assignment 1: Heap File Page Structure & Buffer Manager

Deadline: 09:00 in class, Nov. 21 (Monday), 2005

Cheating Policy: If you are caught cheating, your grade is 0.

Late Policy: You may hand in your late assignment at Teacher's office hour on Tuesday (11/22/2005) for 80% of original grade, or at TA's office hour on Thursday (11/24/2005) for 70%. We will not accept any assignment submissions after Thursday (11/24/2005).

Introduction

This assignment is divided into 2 parts. One is Heap File Page Structure and the other is Buffer Manager. For the ease of programming and testing, these two parts are completely separated. Each has its own directory, source codes, and Makefile.

You need to write your code on a Linux operating system, because your code needs to be linked with several pre-compiled Linux libraries. We strongly recommend you to use a previous version GNU C compiler, not the version 3.0 or later. (If you could compile the program with these newer compilers, please tell us your solution.)

A. Heap File Page Structure

In this part, you will implement the page structure for the Heap File layer. You will be given some source code and some driver routines to test the code.

A.1 Preliminary Work

Begin by reading the description of Heap Files in section 9.5.1, and the description of page formats in section 9.6.2. A HeapFile is seen as a collection of records. Internally, records are stored on a collection of HFPages.

You will be implementing just the HFPAGE class, and not all of the HeapFile code. Read the description in the text of how variable length records can be stored on a slotted page, and follow this page organization.

A.2 Compiling Your Code and Running the Tests

Copy *Makefile* in *HFPAGE/src* to your working directory and type “*make setup*” in the command line. This will copy the necessary files to your working directory. If you *make* the project, it will create an executable named *hfpager*. Right now, it does not work; you will need to fill in the bodies of the HFPAGE class methods. The methods are defined (empty) in file *hfpager.C*.

Sample output of a correct implementation is available in *sample_output*.

A.3 Design Overview and Implementation Details

Have a look at the file *hfpager.h* in **HFPAGE/include**. It contains the interfaces for the HFPAGE class. This class implements a "heap-file page" object. Note that the protected data members of the page are given to you. All you need to do is to implement the public member functions. You should put all your code into the file *hfpager.C*.

A note on the slot directory: In the description of the text, the slot directory is located at the end of the page, and grows toward the beginning of the page. This does mean, however, that you will need to write the code so the records themselves are placed beginning at the start of the page. Be very careful with your pointer arithmetic.

Also note that in order to add a record to a page, there has to be a room for the record itself in the data area, and also room for a new slot in the data area (unless there happens to be a pre-allocated slot that's empty).

Please follow the [Minibase Error Protocol](#). An example file illustrating the use of the error protocol is available in **HFPAGE/src/ErrProc.sample**. It covers much of what you need to know about the protocol. You can look at *new_error.h* for more details. It is in **HFPAGE/include**.

A.4 The Methods to be Implemented

void HFPAGE::init(PageId pageNo):

This member function is used to initialize a new heap file page with page number *pageNo*. It should set the following data members to reasonable defaults: *nextPage*, *PrevPage*, *slotCnt*, *curPage*, *freePtr*, *freeSpace*. You will find the definitions of these data members in *hfpager.h*. The *nextPage* and *prevPage* data members are used for keeping track of pages in a

HeapFile. A good default unknown value for a PageId is INVALID_PAGE, as defined in page.h. Note that freePtr is an offset into the data array, not a pointer.

PageId HFPage::getNextPage():

This member function should return the page id stored in the nextPage data member.

PageId HFPage::getPrevPage():

This member function should return the page id stored in the prevPage data member.

void HFPage::setNextPage(PageId pageNo):

This member function sets the nextPage data member.

void HFPage::setPrevPage(PageId pageNo):

This member function sets the prevPage data member.

Status HFPage::insertRecord(char* recPtr, int reclen, RID& rid):

This member function should add a new record to the page. It returns OK if everything went OK, and DONE if sufficient space does not exist on the page for the new record. If it returns OK, it should set rid to be the RID of the new record (otherwise it can leave rid untouched.) Please note in the parameter list **recPtr** is a char pointer and **RID&** denotes passed by reference. The Status enumerated type is defined in new_error.h if you're curious about it. You may want to look that file over and handle errors in a more informative manner than suggested here. The RID struct is defined to be:

Struct RID {

PageID pageNo;

int slotNo;

int operator == (const RID rid) const

{ return (pageNo == rid.pageNo) && (slotNo == rid.slotNo); }

int operator != (const RID rid) const

{ return (pageNo != rid.pageNo) || (slotNo != rid.slotNo); }

friend ostream& operator << (ostream& out, const struct RID rid); }

In C++, **struct** are aggregate data types built using elements of other types. The pageNo

identifies a physical page number (something that the buffer manager and the DB layers understand) in the file. The slotNo specifies an entry in the slot array on the page.

Status HFPAGE::deleteRecord(const RID& rid):

This member function deletes the record with RID rid from the page and compact the hole created from the deleted record. Compacting the hole, in turn, requires that all the offsets (in the slot array) of all records after the hole be adjusted by the size of the hole, because you are moving these records to "fill" the hole. You should leave a "hole" in the slot array for the slot which pointed to the deleted record, if necessary, to make sure that the rids of the remaining records do not change. The slot array can be compacted only if the record corresponding to the last slot is being deleted. It returns OK if everything goes OK, or FAIL otherwise. (what could go wrong here?)

Status HFPAGE::firstRecord(RID& firstRid):

This routine should set firstRid to be the rid of the "first" record on the page. The order in which you return records from a page is entirely up to you. If you find a first record, return OK, else return DONE.

Status HFPAGE::nextRecord(RID curRid, RID& nextRid):

Given a *valid* current RID, curRid, this member function stores the next RID on the page in the nextRid variable. Again, the order of your return records is up to you, but do make sure you return each record exactly once if someone calls nextRecord! Don't worry about changes to the page between successive calls (e.g. records inserted to or deleted from the page). If you find a next RID, return OK, else return DONE. In case of an error, return FAIL.

Status HFPAGE::getRecord(RID rid, char * recPtr, int& recLen):

Given a rid, this routine copies the associated record into the memory address *recPtr. You may assume that the memory pointed by *recPtr has been allocated by the caller. RecLen is set to the number of bytes that the record occupies. If all goes well, return OK, else return FAIL.

Status HFPAGE::returnRecord(RID rid, char*& recPtr, int& recLen):

This routine is very similar to HFPAGE::getRecord, except in this case you do not copy the record into a caller-provided pointer, but instead you set the caller's recPtr to point directly to the record on the page. Again, return either OK or FAIL.
DONE is a special code for non-errors that are nonetheless not "OK": it generally means "finished" or "not found." FAIL is for errors that happen outside the bounds of a subsystem.

int HFPage::available_space(void):

This routine should return the amount of space available for a new record that is left on the page. For instance, if all slots are full and there are 100 bytes of free space on the page, this method should return (100 - sizeof(slot_t)) bytes. This accounts for the fact that sizeof(slot_t) bytes must be reserved for a new slot and cannot be used by a new record.

bool HFPage::empty(void): Returns true if the page has no records in it, and false otherwise.

B. Buffer Manager

In this part, you will have to implement a simple buffer management layer (without support for concurrency control and recovery) for the Minibase database system. The code for the underlying Disk Space Manager will be given. HTML documentation for Minibase is available on the web (<http://www.cs.wisc.edu/coral/minibase/project.html>). In particular, you should read the description of the DB class, which you will use extensively in this assignment.

Note: The Minibase buffer manager layer differs from what you have to implement in that it contains methods to support concurrency control and recovery.

B.1 Design Overview and Implementation Details

The *buffer pool* is collection of *frames* (page-sized sequence of main memory bytes) that is managed by the Buffer Manager. It should be stored as an array **bufPool[numbuf]** of Page objects. In addition, you should maintain an array **bufDescr[numbuf]** of *descriptors*, one per frame. Each descriptor is a record with the following fields:

page number, pin_count, dirtybit

The *pin_count* field is an integer, *page number* is a **PageId** object, and *dirtybit* is a boolean. This describes the page that is stored in the corresponding frame. A page is identified by a *page number* that is generated by the DB class when the page is allocated, and is unique over all pages in the database. The **PageId** type is defined as an integer type in minirel.h.

A simple *hash table* should be used to figure out what frame a given disk page occupies. The hash table should be implemented (entirely in main memory) by using an array of pointers to lists of <*page number, frame number*> pairs. The array is called the *directory* and each list of pairs is called a *bucket*. Given a *page number*, you should apply a *hash function* to find the directory entry pointing to the bucket that contains the frame number for this page, if the page is in the buffer pool. If you search the bucket and don't find a pair containing this page

number, the page is not in the pool. If you find such a pair, it will tell you the frame in which the page is in,

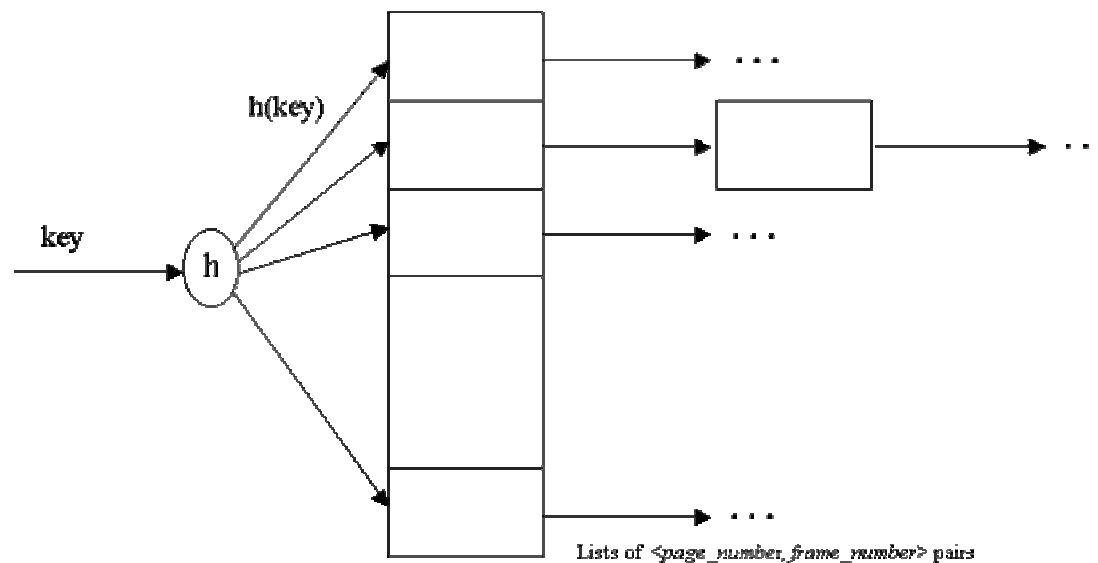


Figure 1: Hash Table

The hash function must distribute values in the domain of the search field uniformly over the collection of buckets. If we have HTSIZE buckets, numbered 0 through M-1, a hash function h of the form

$$h(\text{value}) = (a \times \text{value} + b) \bmod \text{HTSIZE}$$

works well in practice. HTSIZE should be chosen to be a prime number. When a page is requested the buffer manager should do the following: Check the buffer pool (by using the hash table) to see if it contains the requested page. If the page is not in the pool, it should be brought in as follows:

- Choose a frame for replacement, using the LOVE/HATE replacement policy.
- If the frame chosen for replacement is dirty, *flush* it (i.e., write out the page that is contains to disk, using the appropriate DB class method).
- Read the requested page (again, by calling the DB class) into the frame chosen for replacement; the *pin_count* and *dirtybit* for the frame should be initialized to 0 and FALSE, respectively.
- Delete the entry for the old page from the Buffer Manager's hash table and insert an entry for the new page. Also, update the entry for this frame in the **bufDescr** array to reflect these changes.

- *Pin* the requested page by incrementing the *pin_count* in the descriptor for this frame and return a pointer to the page to the requester.

B.2 The Love/Hate Replacement Policy

Theoretically, the best candidate page for replacement is the page that will be last requested in the future. Since implementing such policy requires a future predicting oracle, all buffer replacement policies try to approximate it one way or another. The LRU policy, for example, uses the past access pattern as an indication for the future. However, sequential flooding can ruin this scheme and MRU becomes more appropriate in this particular situation. In this assignment you are supposed to implement the *love/hate* replacement policy. The policy tries to enhance prediction of the future by relying on a hint from the upper levels about the page. The upper level user hints the buffer manager that the page is *loved* if it is more likely that the page will be needed in the future, and *hated* if it is more likely that the page will not be needed. The policy is supposed to maintain an MRU list for the hated pages and an LRU list for the loved pages. If a page is needed for replacement, the buffer manager selects from the list of hated pages first and then from the loved pages if no hated ones exist.

A situation may arise when a page is both loved and hated at the same time. This can happen if the page was pinned by two different users and then was unpinned by the first one as a hated page and by the other as a loved page. In this case, assume that “love conquers hate”, meaning that once a page is indicated as loved it should remain loved.

B.3 The Buffer Manager Interface

The simplified buffer manager interface that you will implement allows a higher level program to allocate and deallocate pages on disk, to bring a disk page to the buffer pool and pin it, and to unpin a page in the buffer pool.

The methods that you have to implement are described below:

```
class BufMgr {
public:
    // This is made public just because we need it in your driver_test.C.
    // It could be private for real use.
    page* bufPool;
    // The physical buffer pool of pages.

public:
    BufMgr(int numbuf, Replacer *replacer = 0);
```

// Allocate "numbuf" pages (frames) for the pool in main memory.

~BufMgr();

// Should flush all dirty pages in the pool to disk before shutting down

// and deallocate the buffer pool in main memory.

Status pinPage(PageId PageId_in_a_DB, Page*& page, int emptyPage=0);

// Check if this page is in buffer pool. If it is, increment the pin_count

// and return a pointer to this page. If the pin_count was 0 before the

// call, the page was a replacement candidate, but is no longer a candidate.

// If the page is not in the pool, choose a frame (from the set of replacement

// candidates) to hold this page, read the page (using the appropriate DB

// class method) and pin it.

// Also, must write out the old page in chosen frame if it is dirty before

// reading new page. (You can assume that emptyPage == 0 for this assignment.)

Status unpinPage(PageId globalPageId_in_a_DB, int dirty, int hate);

// hate should be TRUE if type page is "hated" and FALSE otherwise.

// Should be called with dirty == TRUE if the client has modified the page.

// If so, this call should set the dirty bit for this frame. Further, if pin_count > 0

// should decrement it.

// If pin_count = 0 before this call, return error.

Status newPage(PageId& firstPageId, Page*& firstpage, int howmany=1);

// Find a frame in the buffer pool for the first page.

// If a frame exists, call DB object to allocate a run of new pages and pin it.

// (This call allows a client of the Buffer Manager to allocate pages on disk.)

// If buffer is full, i.e., you can't find a frame for the first page, return error.

Status freePage(PageId globalPageId);

// This method should be called to delete a page that is on disk.

// This routine must call the DB class to deallocate the page.

Status flushPage(int pageId);

// Used to flush a particular page of the buffer pool to disk

// Should call the write_page method of the DB class.

Status flushAllPages();

// Flush all pages of the buffer pool to disk.

};

B.4 Compiling Your Code and Running the Tests

Copy *Makefile* in *BufMgr/src* to your working directory and type “*make setup*” in the command line. This will copy the necessary files to your working directory. If you *make* the project, it will create an executable named *buftest*. Right now, it does not work; you will need to modified *buf.h* to add your own structures of hash table, LRU and MRU lists. Furthermore, you should implement all these method bodies in file *buf.C*.

Sample output of a correct implementation is available in *sample_output*.

How to hand-in

Compress the files “*hfpage.C*”, “*buf.h*” and “*buf.C*” and Email to r93922001@ntu.edu.tw with the title “DBMS practicum 01” before deadline.