?
§1 Introduction §3 The Paged File (PF) Layer

§ 2. The Buffer Pool (BF) Layer

Access to data on a page of a file involves reading the page into the buffer pool in main memory and then manipulating (*e.g.*, reading or updating) it there. While a page is in memory and available for manipulation, it is said to be *pinned* in the buffer pool. Such a page must be explicitly *unpinned* when the client is done manipulating it in order to make room for future pages and to ensure that updates are reflected back to the appropriate page on disk. *A page is identified by a file it belongs to and its page number*, which is effectively its relative location within the file.

Most of the interface routines return an integer value, with a negative value meaning that an error has occurred. There are a number of *possible* BF error codes that can be returned; these are described in this section and bf.h file in the public directory $MINIREL_HOME/h.

§ 2.1. Data Structures

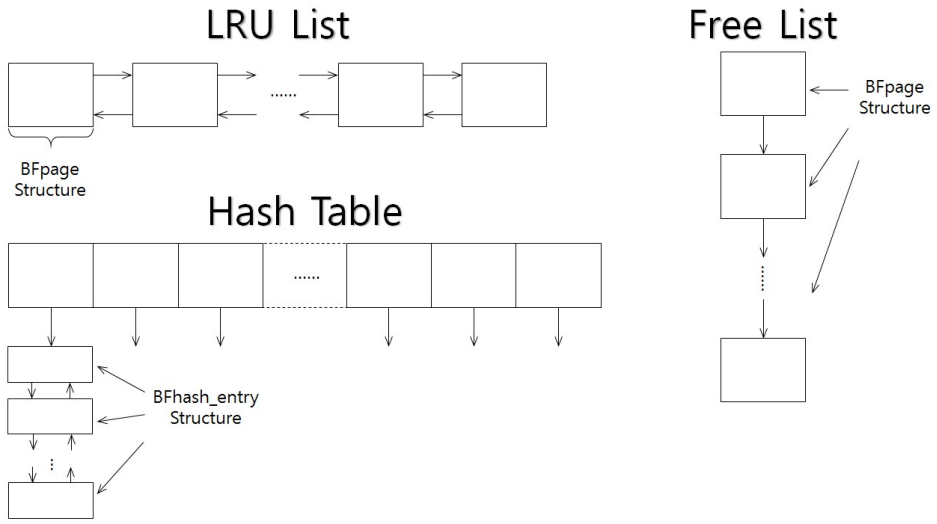

Figure 2. Data Structures for Buffer Pool Management

The buffer pool is a doubly linked *LRU* list of buffer page structures, each one corresponding to a PF file page, as shown in the top left of Figure 2. As pages are requested from a client, they are brought to the buffer pool. If a page is already in the buffer then the request is satisfied immediately without reading the file. If a page needs to be read from disk and the maximum number of pages has already been allocated in the buffer (the maximum number is determined by BF_MAX_BUFS), then a page must be chosen as victim from the LRU list. Its content has to be written back to disk (in case it is *dirty*) and then the buffer entry would be reused for the page to be read. The victim page is chosen by going backward from the last (*i.e.*, least recently used) buffer page. The allocated page is put at the beginning of the buffer. If all the pages in the LRU list are pinned and the list has the maximum number of pages allowed, then an error condition must be returned. The structure of a buffer entry may be defined as follows.

```
typedef struct BFpage {
  PFpage         fpage;       /* page data from the file                */
  struct BFpage  *nextpage;   /* next in the linked list of buffer pages */
  struct BFpage  *prevpage;   /* prev in the linked list of buffer pages */
  bool_t         dirty;       /* TRUE if page is dirty                  */
  short          count;       /* pin count associated with the page     */
  int            pageNum;     /* page number of this page               */
  int            fd;          /* PF file descriptor of this page        */
  int            unixfd;      /* Unix file descriptor of this page      */
} BFpage;
```

A free list of BFpage structures is also maintained by the buffer manager. BFpage entries are returned to this list in situations like when the PF pages of a file are released from the buffer.

A *hash table* is commonly used in order to locate the buffer entry in the buffer pool corresponding to a PF file page. The value used to hash is the fd and pageNum of the PF file page of interest. The fd value is not the Unix file descriptor but the PF file descriptor (index in the PF file table). The hash table has a size of BF_HASH_TBL_SIZE and each bucket entry is a pointer to a hash node, whose structure may be defined as follows. See the bottom of Figure 2 for illustration.

```
typedef struct BFhash_entry {
  struct BFhash_entry *nextentry;     /* next hash table element or NULL */
  struct BFhash_entry *preventry;     /* prev hash table element or NULL */
  int fd;                             /* file descriptor                 */
  int pageNum;                        /* page number                     */
  struct BFpage *bpage;               /* ptr to buffer holding this page */
} BFhash_entry;
```

Note that the structure definitions for the buffer pages and the hash entries are provided merely as suggestions. You can modify them as you wish or you can use your own designs.

§ 2.2. BF Interface Routines

The functions associated with buffer pool management are described below. You are expected to use these functions to design the PF layer, which sits right on top of the BF layer in the MiniRel architecture. Most of the BF functions communicate with the PF layer via a buffer request control block BFreq, which is shown below and defined in minirel.h in the public directory $MINIREL_HOME/h.

```
typedef struct _buffer_request_control {
    int         fd;                  /* PF file descriptor */
    int         unixfd;              /* Unix file descriptor */
    int         pagenum;             /* Page number in the file */
    bool_t      dirty;               /* TRUE if page is dirty */
} BFreq;
```

- void BF_Init(void)
  This function initializes the BF layer. Specifically, it creates all the buffer entries and add them to the free list. It also intializes the hash table. This must be the first function to call in order to use th BF layer. No value returns.

- int BF_GetBuf(BFreq bq, PFpage **fpage)

```
BFreq   bq;             /* buffer control block */
PFpage  **fpage;        /* used to return the page itself */
```

This function returns a PF page in a memory block pointed to by *fpage. This memory block stores a file page associated with a PF file descriptor and a page number passed over in the buffer control block bq. If the buffer page is already in the buffer pool, the pin count of the corresponding buffer entry is incremented by one. Otherwise, a new buffer page should be allocated (by page replacement if there is no free page in the buffer pool). Then, the file page is read into the buffer page, its pin count is set to one, and its dirty flag is set to FALSE, and other appropriate fields of the buffer entry are set accordingly. In both the cases, the page becomes the most recently used. This function returns BFE_OK if the operation is successful, an error condition otherwise.

- int BF_AllocBuf(BFreq bq, PFpage **fpage)

```
BFreq   bq;             /* buffer control block */
PFgpage **fpage;        /* points to the buffer data */
```

Unlike BF_GetBuf(), this function is used in situations where a *new* page is added to a file. Thus, if there already exists a buffer page in the pool associated with a PF file descriptor and a page number passed over in the buffer control block bq, a PF error code must be returned. Otherwise, a new buffer page should be allocated (by page replacement if there is no free page in the buffer pool). Then, its pin count is set to one, its dirty flag is set to FALSE, other appropriate fields of the BFpage structure are set accordingly, and the page becomes the most recently used. Note that it is not necessary to read anything from the file into the buffer page, because a brand new page is being appended to the file. This function returns BFE_OK if the operation is successful, an error condition otherwise.

- int BF_UnpinBuf(BFreq bq)

```
BFreq   bq;             /* buffer control block */
```

This function unpins the page whose identification is passed over in the buffer control block bq. This page must have been pinned in the buffer already. Unpinning a page is carried out by decrementing the pin count by one. This function returns BFE_OK if the operation is successful, an error condition otherwise.

- int BF_TouchBuf(BFreq bq)

```
BFreq   bq;             /* buffer control block */
```

This function marks the page identified by the buffer control block bq as dirty. The page must have been pinned in the buffer already. The page is also made the most recently used by moving it to the head of the LRU list. This function returns BFE_OK if the operation is successful, an error condition otherwise.

- int BF_FlushBuf(int fd)

```
int     fd;             /* PF file descriptor */
```

This function releases all the buffer pages belonging to a file with PF file descriptor fd from the buffer pool by writing any dirty pages to disk and returning all of those buffer pages into the free list. This function returns BFE_OK if the operation is successful, an error condition otherwise. If any page of the file fd is pinned, an error condition must be returned.

- int BF_ShowBuf(void)
  This is a service function that displays the status of the LRU list. It may print such information as the number of pages in the LRU list, the identification and the dirty flag of each individual buffer page, and so on.

§ 2.3. Implementation Notes

§ 2.3.1. Pinning buffer pages

Note that unpinning a page does not necessarily cause the page to be removed from the buffer; an unpinned page is kept in memory as long as its space in the buffer pool is not needed. If a client needs to read a new page into the memory and there is no free space in the buffer pool, the BF layer will choose an unpinned page to remove from the pool and will reuse its space.

§ 2.3.2. Error handling

In each function, incoming parameters should be checked for validity before doing anything. If an error is detected, an error code should be returned to the calling function. Ideally, it should not be possible for a caller to crash the BF layer (or any lower layer) by passing incorrect arguments to an BF layer function. In addition to validating parameters on entry, you should check all return codes to verify that no errors have occurred in a lower layer. If an internal error (an error in a certain layer or one caused by it but detected elsewhere) is found, then you should return right away with an error code.

It is a good idea to have a different error code for each different kind of error. A list of presumptive error codes is provided for each layer in the public directory $MINIREL_HOME/h. Below are a few example error codes defined for the BF layer.

```
#define BFE_OK                0  /* BF function successful */
#define BFE_NOMEM            -1  /* No memory can be allocated */
#define BFE_NOBUF            -2  /* No buffer unit available */
...
```

Obviously this list of error codes is not meant to be comprehensive. Except for a few documented clearly in the function specifications, you can choose to use or not to use any of the error codes. Besides, it will be also a good idea to have a function (like BF_PrintError(int errcode)) that decodes an error code and prints a human-readable error message for convenience.