?
§2 The Buffer Pool (BF) Layer §4 The Heap File (HF) Layer

§ 3. The Paged File (PF) Layer

The Paged File (PF) layer provides facilities to allow client layers to do file I/O in terms of pages. Interface functions should be provided to create, open and close files; to scan through a given file; to read a specific page of a given file; and to add new pages to a given file. Currently, the PF layer does not support deleting pages from a given file.

§ 3.1. Data Structures

As files are opened, the PF layer needs to keep track of each one of them. This is done through the *PF file table*. The PF file table has a maximum of PF_FTAB_SIZE entries. When a file is open, an entry in the table is assigned to it. This index in the table is called the *PF file descriptor*. Note that this descriptor should not be confused with the UNIX file descriptor associated with a file. The structure of a file table entry may look like the following.

```
typedef struct PFftab_ele {
    bool_t    valid;       /* set to TRUE when a file is open. */
    ino_t     inode;       /* inode number of the file         */
    char      *fname;      /* file name                        */
    int       unixfd;      /* Unix file descriptor             */
    PFhdr_str hdr;         /* file header                      */
    short     hdrchanged;  /* TRUE if file header has changed  */
} PFftab_ele;
```

Each PF file is implemented by using a UNIX file, and organized as a *file header* and a series of data pages. The structure definition of the file header and data pages can be as simple as the following.

```
typedef struct PFhdr_str {
    int     numpages;      /* number of pages in the file */
} PFhdr_str;

typedef struct PFpage {
    char pagebuf[PAGE_SIZE];
} PFpage;
```

The constant PAGE_SIZE specifies the number of bytes available to store data on each page. It is typically a multiple of a disk sector size.

§ 3.2. PF Interface Routines

The functions associated with paged file management are described in this section. These are the functions you are expected to use in the design of the HF and AM layers.

- void PF_Init(void)

This function initializes the PF layer. It invokes `BF_Init()` and initializes the file table. This must be the first function to call in order to use th PF layer. There is no return value.

- int PF_CreateFile (char *filename)

```
char      *filename;     /* name of file to be created */
```

This function creates a file named `filename`. This file should not have already existed. The system call `open()` is used to create the file, the PF file header is initialized and written to the file, and the file is closed using the `close()` system call. This function returns `PFE_OK` if the operation is successful, an error condition otherwise.

- int PF_DestroyFile (char *filename)

```
char      *filename;     /* name of file to be destroyed */
```

This function destroys the file `filename`. The file should have existed, and should not be already open. This function returns `PFE_OK` if the operation is successful, an error condition otherwise.

- int PF_OpenFile (char *filename)

```
char      *filename;     /* name of the file to be opened */
```

This function opens the file named `filename` using the system call `open()`, and reads in the file header. Then, the fields in the file table entry are filled accordingly and the index of the file table entry is returned. *This is the PF file descriptor.* This function returns a PF file descriptor if the operation is successful, an error condition otherwise.

- int PF_CloseFile (int fd)

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

This function closes the file associated with PF file descriptor `fd`. This entailes releasing all the buffer pages belonging to the file from the LRU list to the free list. Meanwhile, dirty pages must be written back to the file if any. All the buffer pages of a file must have been unpinned in order for the file to be closed successfully. If the file header has changed, it is written back to the file. The file is finally closed by using the system call `close()`. The file table entry corresponding to the file is freed. This function returns `PFE_OK` if the operation is successful, an error condition otherwise.

- int PF_AllocPage (int fd, int *pageNum, char **pagebuf)

```
int       fd;           /* PF file descriptor */
int       *pageNum;     /* return the number of the page allocated */
char      **pagebuf;    /* return a pointer to the page content */
```

This function allocates a page in the file associated with a file descriptor `fd`. This new page is appended to the end of the file. This function also allocates a buffer entry corresponding to the new page. The value of `pageNum` for the page being allocated must be determined from the information stored in the file header. The page allocated by this function is pinned and marked dirty so that it will be written to the file eventually. Upon a successful page allocation, the file header must be updated accordingly. This function returns PFE_OK if the operation is successful, an error condition otherwise.

- int PF_GetNextPage (int fd, int *pageNum, char **pagebuf)

```
int     fd;            /* PF file descriptor */
int     *pageNum;      /* return the number of the next page */
char    **pagebuf;     /* return a pointer to the page content */
```

This function gets the page right after the one referred to by pageNum in the file associated with file descriptor fd. The pageNum of the next page will be just one more than the current page unless something unusual has happened. The pagebuf argument points to the content of the PF data page. This function returns PFE_OK if the operation is successful, PFE_EOF if the end of file is reached without finding any valid page, a PF error code otherwise.

- int PF_GetFirstPage(int fd, int *pageNum, char **pagebuf)

```
int     fd;            /* PF file descriptor */
int     *pageNum;      /* return the number of the first page */
char    **pagebuf;     /* return a pointer to the page content */
```

This function gets the first valid page in the file associated with file descriptor fd. This function is implemented using PF_GetNextPage() by passing -1 for the pageNum argument. It returns PFE_OK if the operation is successful, PFE_EOF if the end of file is reached without finding any valid page, a PF error code otherwise.

- int PF_GetThisPage (int fd, int pageNum, char **pagebuf)

```
int     fd;            /* PF file descriptor */
int     pageNum;       /* number of page to retrieve */
char    **pagebuf;     /* return the content of the page data */
```

This function reads a *valid* page specified by pageNum from the file associated with file descriptor fd, and sets *pagebuf to point to the page data. If the page specified by pageNum is not valid (*e.g.*, pageNum no smaller than the total number of pages in the file), an error code PFE_INVALIDPAGE is returned. It returns PFE_OK if the operation is successful, and a PF error code otherwise.

- int PF_DirtyPage(int fd, int pageNum)

```
int     fd;            /* PF file descriptor */
int     pageNum;       /* number of page to be marked dirty */
```

After checking the validity of the fd and pageNum values, this function marks the page associated with pageNum and fd dirty. It returns PFE_OK if the operation is successful, an error condition otherwise.

- int PF_UnpinPage(int fd, int pageNum, int dirty)

```
int     fd;            /* PF file descriptor */
int     pageNum;       /* number of page to be unpinned */
int     dirty;         /* dirty indication */
```

After checking the validity of the `fd` and `pageNum` values, this function unpins the page numbered `pageNum` of the file with file descriptor `fd`. Besides, if the argument `dirty` is TRUE, the page is marked dirty. This function returns `PFE_OK` if the operation is successful, an error condition otherwise.

§ 3.3. Implementation Notes

The PF layer implements a paged file manager that allows you to access pages of a file in random or sequential order, to pin pages in the buffer while they are being used, and to unpin them when they are not needed. The functions `PF_AllocPage`, `PF_GetThisPage`, `PF_GetFirstPage`, and `PF_GetNextPage` get a page and pin it in the buffer. The `PF_UnpinPage` function can be used to unpin a page. It is allowed to pin the same page more than once (without unpinning it). The first call to a `PF_GetXxxxPage` function will pin a page in the buffer pool, and set a pin counter for that page to one. Subsequent calls to get the same page will simply increment the pin counter. A call to `PF_UnpinPage` will decrement the counter, and the page will actually be unpinned only when the count drops to zero.

Thus, it is important that *each time you get a page, you do not forget to unpin it*. If you fail to unpin the page, the buffer pool will slowly fill up, and the performance of the system will get worse and worse, until you can no longer get more pages at all (at which point the PF layer will return an error code for any `PF_GetXxxxPage` function). The `PF_CloseFile` call will fail if there are any pages of the file pinned in the buffer, so it is possible to determine whether this is the case or not.