

THE PAGED FILE (PF) LAYER

To simplify the project, we will give you code for this layer of the system. The Paged File layer provides facilities to allow client layers to do file I/O in terms of pages. Interface functions are 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 and delete pages of a given file. In order to use this interface, simply say `#include "pf.h"` near the top of the .h file for your client layer, and then be sure to link the Paged File layer code with your code when creating an executable program that makes use of the Paged File layer routines.

The interfaces for the Paged File layer routines follow. (The name of each routine begins with the prefix PF to indicate the layer that they implement.) Most of the routines return an integer value, with a negative value meaning that an error has occurred. There are several possible PF error codes that can be returned; they are described later. In addition, 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) its data there. While a page is in memory and available for manipulation, it is said to be "fixed" in the buffer pool; pages are fixed as a side effect of reading. Such a page must be explicitly "unfixed" 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. Pages are identified by their page numbers, which are essentially their physical locations within the file on disk. (Note: This means that the page numbering scheme is not necessarily sequential or even monotonic in nature.) Finally, the number of bytes available for data storage on each page is specified by the constant PF_PAGE_SIZE (defined in the PF layer).

Paged File Layer Interface Routines

(1) `PF_Init()`

This routine initializes the PF layer. It must be the first function called in order to use the PF layer. It produces no return value.

(2) `errVal = PF_CreateFile(fileName)`
`char *fileName; /* name of file */`

This routine creates a paged file called fileName. The file should not already exist. It returns PFE_OK if the new file is successfully created, and a PF error code otherwise.

(3) `errVal = PF_DestroyFile(fileName)`
`char *fileName; /* name of file */`

This routine destroys the paged file whose name is fileName. The file should exist, and should not be already open. This routine returns PFE_OK if the file is successfully destroyed, and a PF error code otherwise.

(4) `fileDesc = PF_OpenFile(fileName)`
`char *fileName; /* name of file */`

This routine opens the paged file whose name is `fileName`. It is possible to open a file more than once if desired, getting a different file descriptor (handle) for each open instance of the file. Warning: Opening a file more than once for write operations is not prevented, but doing so can easily corrupt the file structure and thereby crash the Paged File functions. On the other hand, opening a file more than once for reading is OK. This routine returns a file descriptor (which is non-negative) if the file is successfully opened, and a PF error code otherwise.

```
(5)  errVal = PF_CloseFile(fileDesc)
int fileDesc; /* file descriptor */
```

This routine closes the open file instance associated with file descriptor `fileDesc`. The file instance should have been opened with `PF_OpenFile()`. Note that it is an error to close a file if any of its pages are still fixed in the buffer pool. This routine returns `PFE_OK` if the file is successfully closed, and a PF error code otherwise.

```
(6)  errVal = PF_GetFirstPage(fileDesc,pageNum,pageBuf)
int fileDesc; /* file descriptor */
int *pageNum; /* page number of first page */
char **pageBuf; /* indirect pointer to buffer */
```

This routine reads the first page of the file into memory and then sets `*pageBuf` to point to it. It also sets `*pageNum` to be the page number of the page read. The page read is fixed in the buffer pool until it is explicitly unfixd with `PF_UnfixPage()`. It returns `PFE_OK` if the first page is successfully read, `PFE_EOF` if end-of-file is reached (meaning there is no first page), and a PF error code otherwise.

```
(7)  errVal = PF_GetNextPage(fileDesc,pageNum,pageBuf)
int fileDesc; /* file descriptor */
int *pageNum; /* page number of first/next page */
char **pageBuf; /* indirect pointer to buffer */
```

This routine reads the next valid page after `*pageNum`, which is the current page number, setting `*pageBuf` to point to the page data. It also sets `*pageNum` to be the new page number on completion. The new page is fixed in the buffer pool until `PF_UnfixPage()` is called. This routine returns `PFE_OK` if no error occurs, `PFE_EOF` if end-of-file is reached (meaning there there is no next page), `PFE_INVALIDPAGE` if the incoming page number is invalid, and another PF error code otherwise.

```
(8)  errVal = PF_GetThisPage(fileDesc,pageNum,pageBuf)
int fileDesc; /* file descriptor */
int pageNum; /* page number of desired page */
char **pageBuf; /* indirect pointer to buffer */
```

This routine reads the page specified by `pageNum`, setting `*pageBuf` to point to the page's data. The page number must be a valid one. The page that is read is fixed in the buffer pool until it is unfixd via `PF_UnfixPage()`. This routine returns `PFE_OK` if no error occurs, `PFE_INVALIDPAGE` if the incoming page number is invalid, and another PF error code otherwise.

```
(9)  errVal = PF_AllocPage(fileDesc,pageNum,pageBuf)
int fileDesc; /* file descriptor */
int *pageNum; /* page number of new page */
char **pageBuf; /* indirect pointer to buffer */
```

This routine allocates a new, empty page in the file associated with the file descriptor fileDesc. It sets *pageNum to the new page's page number, and sets *pageBuf to point to the buffer for that page. This routine returns PFE_OK if a new page is successfully added to the file, and a PF error code otherwise.

```
(10)  errVal = PF_DisposePage(fileDesc,pageNum)
int fileDesc; /* file descriptor */
int pageNum; /* page number of old page */
```

This routine disposes the page numbered pageNum in the file associated with the file descriptor fileDesc. Only a page that is not currently fixed in the buffer pool can be disposed of. This routine returns PFE_OK if the page is successfully disposed of, and a PF error code otherwise.

```
(11)  errVal = PF_UnfixPage(fileDesc,pageNum,dirty)
int fileDesc; /* file descriptor */
int pageNum; /* page number of page */
int dirty; /* true if page is dirty */
```

This routine tells the Paged File layer that page pageNum of the file associated with fileDesc is no longer needed in memory. The parameter dirty must be set to TRUE if the page has been modified at all, as otherwise the change will not be written back to the page on disk. This routine returns PFE_OK if the page is successfully unfixed, and a PF error code otherwise.

```
(12)  PF_PrintError(errString)
char *errString; /* string to write */
```

This routine writes the string errString onto stderr, and then writes the last error message produced by the PF layer onto stderr as well. It has no return value.

Paged File Layer Error Handling

Error handling is done in the Unix style, with a global variable PFerrno keeping track of the last error. PF_PrintError() can be called to print out the last error message. If PFerrno is equal to PFE_UNIX, which means that a Unix error occurred, the Unix function perror() is called by PF_PrintError() to print the error message. Thus, a given error can cause as many as three messages to be produced in sequence: the message given in the errString argument to PF_PrintError(), the message produced by the PF layer itself, and the message produced by Unix in the case of a Unix error. Only in very few cases, all involving unexpected internal errors, does the PF layer actually exit the program by itself. It is suggested that the caller also implement error handling in this hierarchical manner. A list of the PF error codes follows:

```
#define PFE_OK 0 /* OK */
#define PFE_NOMEM -1 /* no memory */
#define PFE_NOBUF -2 /* no buffer space */
#define PFE_PAGEFIXED -3 /* page already fixed in buffer */
#define PFE_PAGENOTINBUF -4 /* page to be unfixed is not in the buffer */
#define PFE_UNIX -5 /* unix error */
#define PFE_INCOMPLETEREAD -6 /* incomplete read of page from file */
```

```
#define PFE_INCOMPLETEWRITE -7 /* incomplete write of page to file */
#define PFE_HDRREAD -8 /* incomplete read of header from file */
#define PFE_HDRWRITE -9 /* incomplete write of header to file */
#define PFE_INVALIDPAGE -10 /* invalid page number */
#define PFE_FILEOPEN -11 /* file already open */
#define PFE_FTABFULL -12 /* file table is full */
#define PFE_FD -13 /* invalid file descriptor */
#define PFE_EOF -14 /* end of file */
#define PFE_PAGEFREE -15 /* page already free */
#define PFE_PAGEUNFIXED -16 /* page already unfixed */

/* Internal error: please report this to me */
#define PFE_PAGEINBUF -17 /* new page to be allocated already in buffer */
#define PFE_HASHNOTFOUND -18 /* hash table entry not found */
#define PFE_HASHPAGEEXIST -19 /* page already exists in hash table */
```