Report Project 2

Use the two layers to implement two external sort algorithms (one using the merge sort, and the other based on B+ tree scan), and compare performances for the sample data

Members:

1. Kunal Kumar
2. Shalaj Yadev
3. Sandeep

Introduction:

The Paged file implementation is divided into 3 parts: The external interface, the buffer manager, and the hash table routines. The external interface, found in the file pf.c, keeps track of the opened file table, and communicates with the buffer manager to read/write pages of the file. The buffer manager, in the file buf.c, uses the hash table entries to store and retrieve memory buffer addresses given file descriptors and page numbers. The hash table functions can be found in the file hash.c

<u>The external Interface:</u>

The file header contains the following information:

```
typedef struct PFhdr_str {
int     firstfree;     /* first free page in the linked list of
                          free pages, or PF_PAGE_LIST_END */
int     numpages; /* # of pages in the file */
    } PFhdr_str;
```

Each page on the disk contains the following information:

```
 typedef struct PFfpage {
  int nextfree;  /* page number of next free page in the linked
           list of free pages, or PF_PAGE_LIST_END if
           end of list, or PF_PAGE_USED if this page is not free */
   char pagebuf[PF_PAGE_SIZE];      /* actual page data visible to
                            the user */
 } PFfpage;
```

The free pages on the disk are chained so that allocating a new page would involve only getting the page from the head of the free list. The used pages are not chained in any way, which means that a linear scan of the file will also have to pass through the free pages.

The operations on the Paged File as provided include the following:

pf.c: Paged File Interface Routines+ support routines

```
void PF_Init()
```

SPECIFICATIONS:
Initialize the PF interface. Must be the first function called in order to use the PF ADT.

RETURN VALUE: none

```
PF_CreateFile(fname)
        char *fname;      -- name of file to create
```

SPECIFICATIONS:
Create a paged file called "fname". The file should not have already existed before.
RETURN VALUE:
PFE_OK   if OK
PF error code if error.

```
PF_DestroyFile(fname)
    char *fname;          /* file name to destroy */
```

SPECIFICATIONS:
Destroy the paged file whose name is "fname". The file should exist, and should not be already open.

RETURN VALUE:
PFE_OK   if success
PF error codes if error

```
PF_OpenFile(fname)
 char *fname;             /* name of the file to open */
```

SPECIFICATIONS:
Open the paged file whose name is fname. It is possible to open a file more than once. Warning: Openinging a file more than once for

write operations is not prevented. The possible consequence is the corruption of the file structure, which will crash the Paged File functions. On the other hand, opening a file more than once for reading is OK.

RETURN VALUE:
The file descriptor, which is >= 0, if no error.
PF error codes otherwise.

IMPLEMENTATION NOTES:
A file opened more than once will have different file descriptors returned. Separate buffers are used.

```
PF_CloseFile(fd)
int fd;         /* file descriptor to close */
```

SPECIFICATIONS:
Close the file indexed by file descriptor fd. The file should have been opened with PFopen(). It is an error to close a file with pages still fixed in the buffer.

RETURN VALUE:
PFE_OK   if OK
PF error code if error.

```
PF_GetFirstPage(fd,pagenum,pagebuf)
    int fd;     /* file descriptor */
     int *pagenum;         /* page number of first page */
     char **pagebuf;      /* pointer to the pointer to buffer */
```

SPECIFICATIONS:
Read the first page into memory and set *pagebuf to point to it.
Set *pagenum to the page number of the page read.
The page read is fixed in the buffer until it is unixed with
PFunfix().

RETURN VALUE:
PFE_OK   if no error.
PFE_EOF if end of file reached.(meaning there is no first page.)
other PF error code if other error.

```
   PF_GetNextPage(fd,pagenum,pagebuf)
      int fd;  /* file descriptor of the file */
      int *pagenum;       /* old page number on input, new page number on
output */
   char **pagebuf;        /* pointer to pointer to buffer of page data */
```

SPECIFICATIONS:
Read the next valid page after *pagenum, the current page number,
and set *pagebuf to point to the page data. Set *pagenum
to be the new page number. The new page is fixed in memory
until PFunfix() is called.
Note that PF_GetNextPage() with *pagenum == -1 will return the
first valid page. PFgetFirst() is just a short hand for this.

RETURN VALUE:
PFE_OK   if success
PFE_EOF if end of file reached without encountering
        any used page data.
PFE_INVALIDPAGE if page number is invalid.
other PF errors code for other error.

```
 PF_GetThisPage(fd,pagenum,pagebuf)
    int fd;              /* file descriptor */
     int pagenum;  /* page number to read */
     char **pagebuf;       /* pointer to pointer to page data */
```

SPECIFICATIONS:
Read the page specifeid by "pagenum" and set *pagebuf to point
to the page data. The page number should be valid.

RETURN VALUE:
PFE_OK   if no error.
PFE_INVALIDPAGE if invalid page number is specified.
other PF error codes if other error encountered.

```
PF_AllocPage(fd,pagenum,pagebuf)
int fd;         /* file descriptor */
int *pagenum;     /* page number */
char **pagebuf;   /* pointer to pointer to page buffer*/
```

SPECIFICATIONS:
Allocate a new, empty page for file "fd".
set *pagenum to the new page number.
Set *pagebuf to point to the buffer for that page.
The page allocated is fixed in the buffer.

RETURN VALUE:
PFE_OK   if ok
PF error codes if not ok.

```
PF_DisposePage(fd,pagenum)
     int fd;           /* file descriptor */
     int pagenum; /* page number */
```

SPECIFICATIONS:
Dispose the page numbered "pagenum" of the file "fd".
Only a page that is not fixed in the buffer can be disposed.

RETURN VALUE:
PFE_OK   if no error.
PF error code if error.

```
PF_UnfixPage(fd,pagenum,dirty)
   int fd;      /* file descriptor */
   int pagenum;    /* page number */
   int dirty;   /* true if file is dirty */
```

SPECIFICATIONS:
Tell the Paged File Interface that the page numbered "pagenum"
of the file "fd" is no longer needed in the buffer.
Set the variable "dirty" to TRUE if page has been modified.

RETURN VALUE:
PFE_OK   if no error
PF error code if error.

```
void PF_PrintError(s)
 char *s;     /* string to write */
```
SPECIFICATIONS:

Write the string "s" onto stderr, then write the last error message from PF onto stderr.

RETURN VALUE: none

This implementation uses an array, called an open file table, to keep track of the files that have been opened with PFopen(). Each element of the array contains the following information:

```
typedef struct PFftab_ele {
        char *fname;       /* file name, or NULL if entry not used */
        int unixfd;   /* unix file descriptor*/
        PFhdr_str hdr;     /* file header */
        short hdrchanged; /* TRUE if file header has changed */
} PFftab_ele;
```

Whenever a file is opened, an entry in this table is allocated, and the information in the table is initialized.
At this level no actual I/O is performed except reading/writing the file header. The buffer manager decides when to read/write the file pages.

Error handling is done in the Unix style, with a global variable PFerrno keeping track of the last error. PFperror() can be called to print out the last error message. In the case where PFerrno is equal to PFE_UNIX, meaning a unix error, the unix function perror() is called by PFperror() to print the error message.
Only in very few occasions, all of which involving an internal error, does the PF interface exit the program by itself.
It is suggested that the caller also implement error handling

in this hierarchical manner. A list of the error codes follows:

```
/************** Error Codes *********************************/
#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      /* incomplte 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 to the TA */
#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 exist in hash table */
```

II. The buffer manager:

The buffer manager is implemented as a separate abstract data type.
It can be made into a completely independent module if separate error
code from that of the PF error code assignment is used.
Here is a list of the interface routines:

```
PFbufGet(fd,pagenum,fpage,readfcn,writefcn)
int fd;     /* file descriptor */
int pagenum;  /* page number */
PFfpage **fpage;     /* pointer to pointer to file page */
int (*readfcn)();        /* function to read a page */
int (*writefcn)();       /* function to write a page */
/****************************************************************************
SPECIFICATIONS:
            Get a page whose number is "pagenum" from the file pointed
            by "fd". Set *fpage to point to the data for that page.
            This function requires two functions as input:
                    readfcn(fd,pagenum,fpage)
                    int fd;
                    int pagenum;
                    PFfpage *fpage;
            which will read one page whose number is "pagenum" from the file "fd"
            into the buffer area pointed by "fpage".
                    writefcn(fd,pagenum,fpage)
                    int fd;
                    in pagenum;
                    PFpage *fpage;
            which will write one page into the file.
            It is an error to read a page already fixed in the buffer.

RETURN VALUE:
            PFE_OK    if no error.
            PF error code if error.
****************************************************************************/



PFbufUnfix(fd,pagenum,dirty)
int fd;             /* file descriptor */
int pagenum;  /* page number */
int dirty; /* TRUE if page is dirty */
/****************************************************************************
SPECIFICATIONS:
        Unfix the file page whose number is "pagenum" from the buffer.
        If dirty is TRUE, then mark the buffer as having been modified.
        Otherwise, the dirty flag is left unchanged.

RETURN VALUE:
        PFE_OK if no error.
        PF error codes if error occurs.
****************************************************************************/



PFbufAlloc(fd,pagenum,fpage,writefcn)
int fd;          /* file descriptor */
int pagenum;  /* page number */
PFfpage **fpage;    /* pointer to file page */
int (*writefcn)();
/****************************************************************************
SPECIFICATIONS:
        Allocate a buffer and mark it belonging to page "pagenum"
        of file "fd".  Set *fpage to point to the buffer data.
        The function "writefcn" is used to write out pages. (See PFbufGet()).

RETURN VALUE:
        PFE_OK if successful.
        PF error codes if unsuccessful
****************************************************************************/



PFbufReleaseFile(fd,writefcn)
int fd;          /* file descriptor */
int (*writefcn)();        /* function to write a page of file */
/************************************************************************
SPECIFICATIONS:
            Release all pages of file "fd" from the buffer and
            put them into the free list

RETURN VALUE:
            PFE_OK if no error.
            PF error code if error.

IMPLEMENTATION NOTES:
            A linear search of the buffer is performed.
            A better method is not needed because # of buffers are small.
************************************************************************/

PFbufUsed(fd,pagenum)
int fd;           /* file descriptor */
int pagenum;  /* page number */
/************************************************************************
SPECIFICATIONS:
            Mark page numbered "pagenum" of file descriptor "fd" as used.
            The page must be fixed in the buffer. Make this page most
            recently used.


RETURN VALUE: PF error codes.
************************************************************************/



void PFbufPrint()
/************************************************************************
SPECIFICATIONS:
            Print the current page buffers.
```

**/

A doubly linked list of the buffer pages plus a singly linked list of free pages is maintained by the buffer manager. When the caller tries to get a page using PFbufGet(), and the page is already in the buffer, the buffer manager will return that page immediately. If the page is not in the buffer, and there is a page in the free list, the page data is read into the free buffer page, and the page is returned to the caller. If there are no pages in the free list, but the number of buffer pages in use is less than PF_MAX_BUFS, then a new page is allocated using malloc(). When all of the above fails, a page is chosen as a victim and written to the disk. The desired page is then read into now free page, and the page is returned to the user.

The buffer manager currently uses the global LRU algorithm. When searching for a victim to page out to disk, it searches from the back of the list of buffer pages. Whenever a page is used, it is moved to the head of the list.

III. The Hash Table

The hash table, like the Buffer Manager, is an independnet ADT except for the error code assignments. The functions provided include the following:

void PFhashInit()
/**
SPECIFICATIONS:

Init the hash table entries. Must be called before any of the other hash functions are used.

```
*****************************************************************************/


PFbpage *PFhashFind(fd,page)
int fd;             /* file descriptor */
int page;/* page number */
/*****************************************************************************
SPECIFICATIONS:
            Given the file descriptor "fd", and page number "page",
            find the buffer address of this particular page.

RETURN VALUE:
            NULLif not found.
            Buffer address, if found.
*****************************************************************************/


PFhashInsert(fd,page,bpage)
int fd;             /* file descriptor */
int page;/* page number */
PFbpage *bpage;    /* buffer address for this page */
/*****************************************************************************
SPECIFICATIONS:
            Insert the file descriptor "fd", page number "page", and the
            buffer address "bpage" into the hash table.

RETURN VALUE:
            PFE_OK    if OK
            PFE_NOMEM     if nomem
            PFE_HASHPAGEEXIST if the page already exists.
*****************************************************************************/

PFhashDelete(fd,page)
```

```
int fd;             /* file descriptor */
int page;/* page number */
/****************************************************************************
SPECIFICATIONS:
            Delete the entry whose file descriptor is "fd", and whose page number
            is "page" from the hash table.

RETURN VALUE:
            PFE_OK    if OK
            PFE_HASHNOTFOUND if can't find the entry
****************************************************************************/



PFhashPrint()
/****************************************************************************
SPECIFICATIONS:
            Print the hash table.
****************************************************************************/
```

The hash table is used by the buffer manager in order to efficiently find out
the buffer address for a given page of a given file descriptor.
The data strucutre for the hash table is quite simple:
an array of pointer to the linked hashed entries. There is no need for
further explanations here.

Performance Analysis:

1.)
Block Size: 20
Pagesize: 4096
AttrLen: 11

No. of Entries: 4269
6 passes required

Number of seeks required in btree scan for reads: 11613
Number of seeks required in btree scan for writes: 11729
Number of transfers required in btree scan for reads: 11761
Number of transfers required in btree scan for writes: 11898

Number of seeks required in mergesort for reads: 7
Number of seeks required in mergesort for writes: 134
Number of transfers required in mergesort for reads: 132
Number of transfers required in mergesort for writes: 233

2.)
Block Size: 20
Pagesize: 4096
AttrLen: 11

No. of Entries: 33
1 pass required

Number of seeks required in btree scan for reads: 1
Number of seeks required in btree scan for writes: 2
Number of transfers required in btree scan for reads: 1
Number of transfers required in btree scan for writes: 3

Number of seeks required in mergesort for reads: 2
Number of seeks required in mergesort for writes: 2
Number of transfers required in mergesort for reads: 2
Number of transfers required in mergesort for writes: 4

3.)

Block Size: 20
Pagesize: 4096
AttrLen: 11

No of entries: 18,65,648
3 passes required

Number of seeks required in mergesort for reads: 742
Number of seeks required in mergesort for writes: 27624
Number of transfers required in mergesort for reads: 27436
Number of transfers required in mergesort for writes: 41536

Number of seeks required in btree scan for reads: 3009747
Number of seeks required in btree scan for writes: 1871569
Number of transfers required in btree scan for reads: 3010741
Number of transfers required in btree scan for writes: 1872356

4.)

Block size: 20
Pagesize: 4096
AttrLen: 11

No of entries: 35447312
4 passes required

```
Number of seeks required in mergesort for reads: 14099
Number of seeks required in mergesort for writes: 736118
Number of transfers required in mergesort for reads: 651605
Number of transfers required in mergesort for writes: 1049809
\
Number of seeks required in btree scan for reads: 71745461
Number of seeks required in btree scan for writes: 35904979
Number of transfers required in btree scan for reads: 71747679
Number of transfers required in btree scan for writes: 35906301
```