National & Kapodistrian University of Athens (NKUA)

Department of Informatics &
Telecommunications

Implementation of Database Systems

Winter Semester 2022 – 2023

Exercise 1 - Deadline: 23/12/2022

Exercise 2 - Deadline: 09/01/2023

The objective of the task

The purpose of this work is to understand the internal workings of Database Systems in terms of block level management but also in terms of record level management. Also, through the work it will be understood whether the existence of indexes on the records can improve the performance of a Database Management System (DBMS). More specifically, as part of the assignment you will implement a set of functions that manage files created based on (1) heap file organization (heap files) and (2) static hashing (Hash Table).

The functions you are asked to implement concern record management and index management. Their implementation will be done on top of the mandatory block management layer, which is provided ready-made as a library. The prototypes (definitions) of the functions you are asked to implement as well as the block-level library functions are given next, along with an explanation of the functionality each will perform.

- Heap files are managed through functions with the HP_ prefix.
- Static hash files are managed via functions with the prefix HT_.

Heap and static hash files have internal records that you manage through the appropriate functions. The entries are in the form given below. The id is the key.

```
typedef struct {
      int id
      char name[15],
      char surname[25],
      char address[50];
} Record;
```

The first block of each file includes "special" information about the file itself. This information is used to identify whether it is a heap file or a static hash file, information about the hash field for the respective files, etc. The following is an example of the records that will be added to the files created by your implementation. Records with which you can test your program will be provided as text files along with appropriate main functions in the course eclass.

```
{15, "Giorgos", "Dimopoulos", "Ioannina"}
{4, "Antonia", "Papadopoulou", "Athina"}
{300, "Yannis", "Yannakis", "Thessaloniki"}
```

BF (Block File) Functions

The block layer (BF) is a memory manager that acts as a cache between the disk and memory layers. The block layer holds disk blocks in memory. Whenever we request a disk block, the BF layer first checks to see if it has already brought it into memory. If the block exists in the memory then it does not read it from the disk, otherwise it reads it from the disk and places it in the memory. Because the BF level does not have infinite memory, at some point we will need to "throw" a block from the memory and put another one in its place. The policies we can throw a block from memory at the block level given to you are LRU (Least Recently Used) and MRU (Most Recently Used).

Next, the functions related to the block layer, which you will rely on to implement the requested functions, are described. The implementation of these functions will be provided ready in the form of a library.

In the bf.h header file you are given, the following variables are defined:

```
BF_BLOCK_SIZE 512 /*The size of a block in bytes*/
BF_BUFFER_SIZE 100 /*The maximum number of blocks we keep in memory*/
BF_MAX_OPEN_FILES 100 /*The maximum number of open files*/
```

and enumerations:

```
enum BF_ErrorCode { … }
```

BF_ErrorCode is an enumeration that defines some error codes that can arise during the execution of the BF level functions.

```
enum ReplacementAlgorithm { LRU, MRU }
```

The Replacement Algorithm is an enumeration that defines the codes for the replacement algorithms (LRU or MRU).

Below are the prototypes of the functions associated with the BF_Block structure.

```
typedef struct BF_Block BF_Block;
```

struct BF_Block is the basic structure that gives entity to the concept of Block. The BF_Block has the following functions.

```
void BF_Block_Init(BF_Block **block
      /* structure that identifies the Block */)
```

The BF_Block_Init function initializes and allocates the appropriate memory for the structure BF_BLOCK.

```
void BF_Block_Destroy(
      BF_Block **block /* structure identifying the Block */)
```

The BF_Block_Destroy function frees the memory occupied by the BF_BLOCK structure.

```
void BF_Block_SetDirty(
      BF_Block *block /* structure that identifies the Block */)
```

The function BF_Block_SetDirty changes the state of the block to dirty. This practically means that the block's data has been changed and the BF layer, when needed, will write the block back to disk. In case we just read the data without changing it then we don't need to call the function.

```
char* BF_Block_GetData(
      const BF_Block *block /* structure specifying the Block */ )
```

The BF_Block_GetData function returns a pointer to the Block's data. If we change the data we should make the block dirty by calling the function BF_Block_SetDirty. Under no circumstances should you release the memory location pointed to by the pointer.

Below are the prototypes of the functions related to the Block level.

```
BF_ErrorCode BF_Init(
      const      ReplacementAlgorithm      repl_alg      /*
                 replacement policy*/)
```

The BF_Init function initializes the BF layer. We can choose between two Block replacement policies that of LRU and that of MRU.

```
BF_ErrorCode BF_CreateFile(
      const char* filename /* filename */)
```

The function BF_CreateFile creates a file named filename which consists of blocks. If the file already exists then an error code is returned. In case of successful execution of the function, BF_OK is returned, while in case of failure, an error code is returned. If you want to see the type of error you can call the BF_PrintError function.

```
BF_ErrorCode BF_OpenFile(
      const char* filename, /* filename */
      int *file_desc /* block file identifier */);
```

The BF_OpenFile function opens an existing blocks file named filename and returns the file identifier in the file_desc variable. BF_OK is returned on success and an error code is returned on failure. If you want to see the type of error you can call the BF_PrintError function.

```
BF_ErrorCode BF_CloseFile(
      int file_desc /* block file identifier */)
```

The function BF_CloseFile closes the open file with identifier number file_desc. BF_OK is returned on success and an error code is returned on failure. If you want to see the type of error you can call the BF_PrintError function.

```
BF_ErrorCode BF_GetBlockCounter(
      const int file_desc, /* block file identifier */
```

```
    int *blocks_num /* return value */)
```

The function Get_BlockCounter accepts as an argument the identifier number file_desc of an open file from block and finds the number of its available blocks, which it returns in the variable blocks_num. BF_OK is returned on success and an error code is returned on failure. If you want to see the type of error you can call the BF_PrintError function.

```
BF_ErrorCode BF_AllocateBlock(
    const int file_desc, /* block file identifier */
    BF_Block *block /* the block returned */)
```

With the function BF_AllocateBlock a new block is allocated for the file with identifier number blockFile. The new block is always committed at the end of the file, so the number of the block is BF_getBlockCounter(...) - 1. The committed block is pinned to the memory (pin) and returned to the block variable. When we no longer need this block then we need to update the block layer by calling the BF_UnpinBlock function. If BF_AllocateBlock succeeds, BF_OK is returned, while if it fails, an error code is returned. If you want to see the type of error you can call the BF_PrintError function.

```
BF_ErrorCode BF_GetBlock(
    const int file_desc, /* block file identifier */ const
    int block_num, /* block identifier number */ BF_Block
    *block /* the block returned */)
```

The BF_GetBlock function finds the block with number block_num of the open file_desc and returns it to the block variable. The block that is committed is pinned to the memory (pin). When we no longer need this block then we need to update the block layer by calling the BF_UnpinBlock function. If BF_GetBlock succeeds, BF_OK is returned, while if it fails, an error code is returned. If you want to see the type of error you can call the BF_PrintError function.

```
BF_ErrorCode BF_UnpinBlock(
    BF_Block *block /* block structure being unpinned */)
```

The BF_UnpinBlock function unpins the block from the BF layer which will eventually write it to disk. BF_OK is returned on success, and an error code is returned on failure. If you want to see the type of error you can call the BF_PrintError function.

```
void BF_PrintError(BF_ErrorCode err /* error code */ )
```

The BF_PrintError function helps print errors that may occur by calling block file level functions. A description of the error is printed to stderr. void BF_Close(). The BF_Close function closes the Block layer by writing to disk any blocks it had in memory.

Exercise 1

HP (heap file) functions

Next, the functions that you are asked to implement in the context of this work and that concern the heap file (heap file) are described.

```
int HP_CreateFile(
    char *fileName,   /*filename*/);
```

The HP_CreateFile function is used to create and appropriately initialize an empty heap file named fileName. If executed successfully, 0 is returned, otherwise -1.

```
HP_info* HP_OpenFile( char *fileName /* file name */ )
```

The function HP_OpenFile opens the file named filename and reads from the first block the information about the heap file. A structure is then updated that holds as much information as is deemed necessary for that file so that you can then process its records. After the file's information structure has been appropriately updated, you return it. In case any error occurs, NULL value is returned. If the file given to open is not a hash file, then this is also considered an error.

An indicative structure of the information to keep is given below:

```
typedef struct {
      int fileDesc; /* block level file opening identifier */
} HP_info;
```

Where fileDesc is the file open identifier as returned by the block management layer. This structure can hold additional information such as: the block in which it is stored, the id of the last block of the heap file, the number of records that fit in each block of the heap file.

```
typedef struct {
    …
} HP_block_info;
```

In addition, at the end of each block there should be a structure in which you will store information related to the block such as: the number of records in the specific block, a pointer to the next data block.

int HP_CloseFile(HP_info* header_info)

The HP_CloseFile function closes the file specified within the header_info structure. If executed successfully, 0 is returned, otherwise -1. The function is also responsible for freeing the memory occupied by the structure passed as a parameter, in case the closure was successful.

```
int HP_InsertEntry(
      HP_info* header_info, /* header of the file*/ Record
      record /* structure identifying the record */ )
```

The HP_InsertEntry function is used to insert an entry into the heap file. Information about the file is in the header_info structure, while the record to import is specified by the record structure. If executed successfully, you return the number of the block in which the entry was made (blockId), otherwise -1.

```
int HP_GetAllEntries(
```

```
    HP_info* header_info, /* file header*/
    int id /* the id value of the record being searched*/)
```

This function is used to print all records present in the hash file that have a value in the key field equal to value. The first structure gives information about the hash file, as returned by HP_OpenFile. For each record that exists in the file and has a value in the id field equal to value, its contents (including the key field) are printed. Also return the number of blocks read until all records are found. In case of success it returns the number of blocks read, while in case of error it returns -1.

HT (Hash Table) Functions

The following describes the functions that you are asked to implement in the context of this work and that concern the static hash file.

```
int HT_CreateFile(
    char *fileName, /* filename */
    int buckets /* number of hash buckets*/)
```

The HT_CreateFile function is used to create and appropriately initialize an empty hash file named fileName. It has as input parameters the name of the file in which the heap will be built and the number of bins of the hash function. If executed successfully, 0 is returned, otherwise -1.

```
HT_info* HT_OpenFile( char *fileName /* file name */ )
```

The function HT_OpenFile opens the file named filename and reads from the first block the information about the hash file. A structure is then updated that holds as much information as is deemed necessary for that file so that you can then process its records. After the file's information structure has been appropriately updated, you return it. In case any error occurs, NULL value is returned. If the file given to open is not a hash file, then this is also considered an error.

An indicative structure of the information to keep is given below:

```
typedef struct {
```

```
    int fileDesc; /* block level file opening identifier */
    long int numBuckets; /* the number of "bins" of the hash file */
    } HT_info;
```

Where fileDesc is the file open identifier as returned by the block management layer, and numBuckets is the number of buckets present in the file. You will also need to keep additional information such as: the block it is stored in, the hash table, the capacity of a block in writes, and whatever else you decide.

```
typedef struct {
    ...
} HT_block_info;
```

In addition, at the end (or at the beginning) of each block there should be a structure in which you will store information related to the block such as: the number of records in the specific block, a pointer to the next block of data (overflow block).

```
int HT_CloseFile(HT_info* header_info )
```

The HT_CloseFile function closes the file specified within the header_info structure. If executed successfully, 0 is returned, otherwise -1. The function is also responsible for freeing the memory occupied by the structure passed as a parameter, in case the closure was successful.

```
int HT_InsertEntry(
    HT_info* header_info, /* header of the file*/ Record
    record /* structure identifying the record */ )
```

The HT_InsertEntry function is used to insert an entry into the hash file. Information about the file is in the header_info structure, while the record to import is specified by the record structure. If executed successfully, you return the number of the block in which the input was made (blockId) , otherwise -1.

```
int HT_GetAllEntries(
```

```
HT_info* header_info, /*file header*/ int id
/*value of key field to search*/)
```

This function is used to print all records present in the hash file that have a value in the key field equal to value. The first structure gives information about the hash file, as returned by HT_OpenFile. For each record in the file that has a value in the key field (as defined in HT_info) equal to value, its contents (including the key field) are printed. Also return the number of blocks read until all records are found. In case of success it returns the number of blocks read, while in case of error it returns -1.

1st paper notes

- The first block of the heap and hash files holds the metadata associated with each structure (HP_info, HT_info). In case we have some operation like import, then the values of HP_info and HT_info should be updated and these changes should be written to the disk at some point. Ideally we want this block to remain in the buffer and be unpinned when the file is closed (after it has become dirty if it contains changes).
    - In order for the entire HT_info structure to fit in the first block, consider a small number of bins, e.g. 10-20, otherwise the HT_info structure will need to be stored in more than one block of the disk.
- Avoid using functions to allocate memory. Instead, have pointers to specific areas of the buffer. In the bf_main example we show how this can be done.
- To complete the exercise you will need to use the memcpy function
  to copy information from a Record to the buffer.
- For both the heap file and the hash file, for your convenience, do not check if an id or record is inserted more than once.

Exercise 2: Functions SHT (Secondary Hash Table)

The following describes the functions that you are asked to implement in the context of this work and which concern the creation of a secondary static hashing index, in the name attribute.

We assume that there is no primary static hash index on id and inserts are made simultaneously on both indexes. Also for simplicity, we assume that the index will always be in the name field.

```
int SHT_CreateSecondaryIndex(
      char *sfileName, /* subindex filename*/ int buckets, /* number
      of hash buckets*/
      char* fileName /* primary index filename*/)
```

The SHT_CreateSecondaryIndex function is used to create and appropriately initialize a secondary hash file named sfileName for the primary hash file fileName. If executed successfully, 0 is returned, otherwise -1.

```
SHT_info* SHT_OpenSecondaryIndex(
      char *sfileName /* subindex file name */)
```

The function SHT_OpenSecondaryIndex opens the file named sfileName and reads from the first block the information about the secondary hash index.

An indicative structure of the information to keep is given below:

```
typedef struct {
      int fileDesc; /* block level file opening identifier */
      int   numBuckets; /* the number of "bins" of the hash file */}
      SHT_info;
```

The fileDesc field is the file open ID number as returned by the block management layer, numBuckets is the number of buckets present in the file. After the file's information structure has been appropriately updated, you return it.

In case any error occurs, NULL value is returned. If the file given to open is not a hash file, then this is also considered an error.

```
typedef struct {
    ...
} HT_block_info;
```

In addition, at the end (or at the beginning) of each block there should be a structure in which you will store information related to the block such as: the number of records in the specific block, a pointer to the next block of data (overflow block).

```
int SHT_CloseSecondaryIndex( SHT_info* header_info )
```

The SHT_CloseSecondaryIndex function closes the file specified within the structure header_info. If executed successfully, 0 is returned, otherwise -1. The function is also responsible for freeing the memory occupied by the structure passed as a parameter, in case the closure was successful.

```
int SHT_SecondaryInsertEntry(
    SHT_info* header_info, /* heading of secondary index*/
    Record record, /* the record for which we have an entry in the subindex*/
    int block_id /* the block of the hash file inserted */);
```

The SHT_SecondaryInsertEntry function is used to insert an entry into thehash file. The information about the file is in the header_info structure, while the record to be inserted is determined by the record structure and the block of the primary index where the record to be inserted exists. If executed successfully, 0 is returned, otherwise -1.

```
int SHT_SecondaryGetAllEntries(
```

```
HT_info* ht_info, /* header of primary index file*/
SHT_info* header_info, /* header of subindex file*/
char* name /* the name being searched for */);
```

This function is used to print all records in the hash file that have a value in the subindex key field equal to name. The first structure contains information about the hash file as returned when it was opened. The second structure contains information about the subindex as returned by SHT_OpenIndex. For each record in the file that has a name equal to value, its contents (including the key field) are printed. Also return the number of blocks read until all records are found. On error it returns -1.

2nd paper notes

- For the 2nd task, assume that the index is created each time on an empty hash file. So you won't need to consider the case where the hash file already contains data that you need to build the index for.
- For the 2nd task, you should create a function that can search for values in a specific block of the hash file, instead of searching all blocks. This will allow you to search for a name, in a specific block.
- For both the first and the second task, you will need to create structures and functions that will make it easier for you.

Exercise 1 and 2: Hash Statistics

Afterimplement the hash functions, evaluate them against:

1. How many blocks a file has,
2. The minimum, average and maximum number of records each bucket of a file has,
3. The average number of blocks each bucket has, and
4. The number of buckets that have overflow blocks, and how many blocks there are for each bucket.

To do this, implement the function:

```
int HashStatistics(
      char* filename /* name of the file of interest */ )
```

The function reads the file named filename and prints the reported statistics previously. In case of success it returns 0, while in case of error it returns -1.

Exercise 2: Functionality Check main function

To check the correctness of your implementation you need to customize the 3 main functions that show the functionality of the different storage modes and indexes, while you should also create a main to check the statistical information.

Notes:

- You should strictly follow the function templates given to you.

Delivery of work

The work is a group work of 2 people.

Implementation language: C / C++

Deliverable

The source code files (sources) and the corresponding header files (headers), readme file with description / comments on your implementation. As well as the corresponding main functions with which you tested the functionality of your code.