FILE MANAGER

"A Simplified File Management Simulator for Memory Allocation and File Operations in C"

Our GitHub

https://github.com/ahmedyassinebenayache/File_Manager

CONTENTS

"Programs must be written for people to read, and only incidentally for machines to execute"
Harold Abelson

INTRODUCTION

This project focuses on the development of a simplified file management simulator, which involves implementing various functions and algorithms related to file system operations. In this report, I will explain the core functions of the simulator, including memory allocation, file management, and metadata handling. Additionally, I will detail the underlying algorithms used to perform tasks such as file creation, insertion, deletion, and memory compaction, along with an overview of how the user interface presents these operations.

"Talk is cheap. Show me the code"
Linus Torvalds

LINKED STORAGE

Initialize Linked Disk and Metadata

The ***Initialize_Disk_Ch*** function initializes a virtual disk specifically for the linked memory organization. It begins by marking all blocks as free in the allocation table, which is then written to the first block of the disk. The function proceeds to set up default metadata values, such as placeholders for file name, size, and allocation mode. These metadata values are stored in separate metadata blocks, which are written sequentially after the allocation table. This setup ensures that the disk starts with an empty state, using the linked memory model where each file block can be scattered across the disk with pointers linking them, ready for file operations and management.

```c
void Initialize_Disk_Ch(FILE *ms){
    int Allocation_Table[NbBloc];

    for(int i = 0; i < NbBloc; i++){
        Allocation_Table[i] = 0;  // mark all blocks as free
    }
    fseek(ms, 0, SEEK_SET);

    fwrite(Allocation_Table, sizeof(int), NbBloc, ms); // write the allocation table to the first
block
    FDmeta meta;             // default values for metadata blocks
    BLOC_meta meta_bloc;
    strcpy(meta.FDnom, " ");
    meta.taille = 0;
    meta.nbEtudiant = 0;
    meta.adresse = -1;
    meta.modeglobal = 0;
    meta.modeinterne = 0;
    BLOC_ch buffer;
    buffer.ne = 0;
    buffer.next = -1;
    fseek(ms, NbBloc * sizeof(int), SEEK_SET);  // move the cursor to the next block
    // default values for metadata blocks
    meta_bloc.ne = 0;
    for(int j = 0; j < NbBlocmeta; j++){
        for(int i = 0; i < FB; i++){
            meta_bloc.t[i] = meta;
        }
        fwrite(&meta_bloc, sizeof(BLOC_meta), 1, ms); // write the metadata blocks to MS
    }
}
```

LINKED STORAGE

Memory Allocation and Management Functions for Linked Storage

The ***allouer*** function allocates a free block from the memory. It reads the current allocation table from the disk and searches for the first free block (where the value is 0). Once a free block is found, it marks it as allocated (1), and returns the block's index (k). If no free blocks are found, it returns -1.

```c
int allouer (FILE *ms ){
    int k=-1 ,table[NbBloc]  ;
    fseek(ms, 0, SEEK_SET);
    fread(table,sizeof(int),NbBloc,ms) ;
    for (int i = 0; i < NbBloc; i++)
    {
        if (table[i]==0) {
            table[i]=1 , k=i , i=NbBloc+1 ;
        }
    }
    return k ;
}
```

```c
void update_Allocation_Table(FILE *ms, int bloc_adress, int b) {
    int Allocation_Table[NbBloc];
    fseek(ms, 0, SEEK_SET);
    fread(Allocation_Table, sizeof(int), NbBloc, ms);

    Allocation_Table[bloc_adress] = b; // Update the allocation status
    fseek(ms, 0, SEEK_SET);
    fwrite(Allocation_Table, sizeof(int), NbBloc, ms);
}
```

The ***update_Allocation_Table*** function updates the allocation status of a specific block in the allocation table. It reads the current allocation table from the disk, updates the status of the specified block (bloc_adress) to either allocated (1) or free (0), and then writes the updated table back to the disk.

The ***empty_MS_Ch*** function empties the memory space by reinitializing it for linked storage. It calls the ***Initialize_Disk_Ch*** function, which sets up the disk with all blocks marked as free and initializes metadata for the linked memory model.

```c
void empty_MS_Ch(FILE *ms) {
    Initialize_Disk_Ch(ms); // Initialize the disk for linked storage
}
```

LINKED STORAGE

Storage Space Management for Linked Storage

The ***Manage_Storage_Space_Ch*** function manages storage space by checking if there are enough free blocks to accommodate a specified number of students (num_Etudiant). It calculates the number of blocks required based on the number of students, reads the allocation table from the disk, and counts the free blocks. If enough free blocks are found, the function returns 0 (success); if not, it prints "MS IS FULL" and returns 1 (failure).

```
nt Manage_Storage_Space_Ch(FILE *ms, int num_Etudiant) {
    int num_Blocs = ceil((double)num_Etudiant / FB); // Calculate required blocks
    int Allocation_Table[NbBloc];
    fseek(ms, 0, SEEK_SET);

    fread(&Allocation_Table, sizeof(int), NbBloc, ms); // Read the allocation table from the
diskint counter = 0;
    for (int i = 0; i < NbBloc; i++) { // Count the free blocks
        if (Allocation_Table[i] == 0) {
            counter++;
        }
        if (counter == num_Blocs) {
            break;
        }
    }
    if (counter < num_Blocs) { // If not enough blocks
        printf("MS IS FULL\n");
        return 1;
    } else {
        return 0; // Enough blocks available
    }
}
```

```
Position Searchmetadata(FILE *ms, FDmeta M) {
    BLOC_meta meta;
    fseek(ms, NbBloc * sizeof(int), SEEK_SET);
    for (int j = 0; j < NbBlocmeta; ++j) {
        fread(&meta, sizeof(BLOC_meta), 1, ms);
        for (int i = 0; i < FB; ++i) {
            if (strcmp(meta.t[i].FDnom, M.FDnom) == 0) {
                Position X;
                X.nbrbloc = j;
                X.mov = i;
                return X;
            }
        }
    }
    Position y;
    y.mov = -1;
    y.nbrbloc = -1;
    rewind(ms);
    return y;
}
```

The ***Searchmetadata*** function searches for a file's metadata in the metadata blocks of the disk. It reads the metadata blocks sequentially and compares the file name (FDnom) with the target file name (M.FDnom). If a match is found, it returns the position (block number and index within the block) of the metadata. If no match is found, it returns a position with -1 values, indicating the file does not exist.

LINKED STORAGE

Student File Creation and Storage Management in Linked Storage

The ***Creer_du_fichierchainee*** function is responsible for creating and managing student files in a linked storage system. It begins by constructing the metadata for the student file, which includes the number of students, file name, and memory allocation. The function updates the allocation table to mark the necessary blocks as occupied. It then checks for available space on the disk and creates a metadata entry for the student file.
Next, the function gathers student information such as ID, name, surname, and section. If required, it sorts the students by their IDs. After gathering the student data, the function divides it into blocks of fixed size and writes each block to the file. The process continues until all students are stored, and finally, the function frees the allocated memory and finishes the process.
This function ensures efficient storage management by writing data in blocks and updating the metadata accordingly.

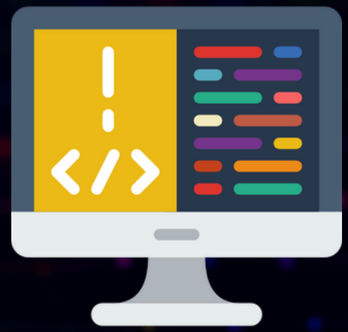
Loading Linked File into Secondary Memory

The ***chargement_fichier_chainee*** function loads a file from the source into secondary memory. It first searches for the file's metadata in the master storage (MS). Once found, it reads the file's data in blocks and inserts them into MS, linking the blocks using the next pointer. If the block isn't the last one, a new block is allocated, and the allocation table is updated. The function repeats this process until all blocks are loaded. If the file can't be found or the disk is full, an error message is displayed.

LINKED STORAGE

Displaying File Data from Secondary Memory

The ***Displayfile*** function displays the contents of a file stored in secondary memory. It first searches for the file's metadata using the given filename. If found, it validates the starting address and begins reading the file data block by block. For each block, it reads the information (e.g., ID, name, section, status) and prints it to the console. The function continues until all blocks are displayed or an error is encountered, such as an invalid block address or the file not being found.



Renaming a File in Linked Storage

The ***Rename_File_Ch*** function renames a file stored in secondary memory. It begins by searching for the file's metadata based on the current filename. If the file is found, it modifies the filename in the metadata structure to the new name provided. After updating the filename, the function writes the updated metadata back to the disk. If the file is not found during the search, an error message is displayed. This function ensures that the file name is changed in the metadata, which is essential for maintaining file consistency.

LINKED STORAGE

Deleting a File from Linked Storage

The ***supprime_fichier_chainee*** function is responsible for deleting a file from secondary memory. It starts by searching for the file's metadata using the provided filename. If the file is found, the function iterates through all the blocks associated with the file, starting from the file's address. Each block is freed by updating the allocation table, marking the blocks as available. After removing all the file blocks, the function updates the metadata to reset the file's details, such as its size and address, effectively removing the file from the system. If the file is not found during the search, an error message is displayed.

Adding a Student to an Unsorted or Sorted Linked File

The ***add_student_to_unsorted_linked_file*** function adds a student to an unsorted linked file. It first locates the file's metadata, searches for the file by its name, and navigates through blocks until it finds the file. If there's space in the last block, it adds the student there; if the block is full, a new block is allocated, and the student is added to it. The metadata is updated to reflect the new student count and block allocation.

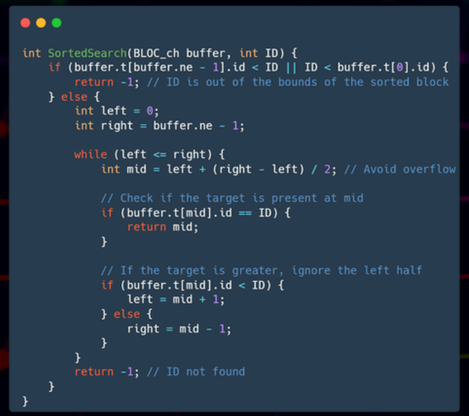
The ***add_student_to_sorted_linked_file*** function works similarly, but it maintains the students in sorted order based on their ID. The function traverses the blocks, reads existing student data, and inserts the new student at the correct position in the sorted order. The blocks are then written back to the file, and the metadata is updated to reflect changes in the number of students and blocks.

LINKED STORAGE

Defragmenting a Linked File

The ***defragmentation_fichier_chainee*** function optimizes a linked file by reorganizing it. It first locates the file's metadata, then collects active students from the file's blocks. The students are rewritten into new blocks, and the metadata is updated with the new student and block counts. The function aims to remove fragmentation and improve space utilization in the file.

Search for Student Record in Linked File by ID



```
int SortedSearch(BLOC_ch buffer, int ID) {
    if (buffer.t[buffer.ne - 1].id < ID || ID < buffer.t[0].id) {
        return -1; // ID is out of the bounds of the sorted block
    } else {
        int left = 0;
        int right = buffer.ne - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2; // Avoid overflow

            // Check if the target is present at mid
            if (buffer.t[mid].id == ID) {
                return mid;
            }

            // If the target is greater, ignore the left half
            if (buffer.t[mid].id < ID) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1; // ID not found
    }
}
```

The ***SortedSearch*** function performs a binary search within a sorted block to find a student by their ID. It first checks if the ID is within the bounds of the block. Then, it repeatedly divides the search space in half, adjusting the left and right pointers based on whether the target ID is smaller or larger than the middle element. If the ID is found, it returns the index; otherwise, it returns -1. This ensures efficient searching with a time complexity of O(log n).

The ***Search_Linked_File*** function searches for a student's record in a linked file by name and ID. It starts by searching for the file's metadata in the main storage. If found, it reads through the file's blocks and checks each block for the student's ID. The search can be sorted or unsorted, depending on the sorted flag. If the student is found, it returns the block and position within the block; otherwise, it reports that the student was not found.

LINKED STORAGE

Student Record Deletion: Physical vs Logical Methods in Linked Files

The ***physical_deletion_from_linked_file*** function performs a physical deletion of a student's record from a linked file by removing the record entirely and updating the blocks and metadata. The remaining records are written back to the file.

The ***logical_deletion_from_linked_file*** function performs a logical deletion by marking a student's record as deleted (setting etat = 0) without removing the data. The metadata and blocks are updated to reflect the deletion, but the data remains for potential future use.

CONTIGUOUS STORAGE

Disk Initialization for Contiguous Storage

The ***Initialize_Disk_Co*** function prepares the disk for contiguous storage by creating and writing an allocation table to track free and occupied blocks. It initializes all blocks as free and saves this table at the start of the file. Additionally, the function sets up default metadata values for each block using the FDmeta and BLOC_meta structures. These metadata blocks are then written to the disk to ensure it is properly configured for storage operations.

```c
void Initialize_Disk_Co(FILE *ms){

    int Allocation_Table[NbBloc];

    for(int i=0; i<NbBloc; i++){
        Allocation_Table[i] = 0;  // mark all blocks as free
    }
    fseek(ms,0,SEEK_SET);

    fwrite(Allocation_Table,sizeof(int),NbBloc,ms); // write the allocation table in the first block

    FDmeta meta;              // default values for metadata blocks
    BLOC_meta meta_bloc;
    strcpy(meta.FDnom,"");
    meta.taille = 0;
    meta.nbEtudiant = 0;
    meta.adresse = -1;
    meta.modeglobal = 0;
    meta.modeinterne = 0;
    fseek(ms,NbBloc * sizeof(int) ,SEEK_SET);  // move the cursor to the next block
    // default values for metadata blocks
    meta_bloc.ne = 0;
    for(int j=0; j<NbBloc; j++){

        for(int i= 0; i<FB; i++){

        meta_bloc.t[i] = meta;
        }
    fwrite(&meta_bloc ,sizeof(meta_bloc), 1,ms); // write the metadata blocks in ms
    }
}
```

Updating the Allocation Table for Contiguous Storage

```c
void update_Allocation_Table_co(FILE *ms,int bloc_adress , int b){
    int Allocation_Table[NbBloc];
    fseek(ms,0,SEEK_SET);                                   // Move the file pointer to the beginning
of the file

    fread(Allocation_Table,NbBloc *sizeof(int),1,ms);

    Allocation_Table[bloc_adress] = b;   // Update the allocation table at the specified block address

    fseek(ms,0,SEEK_SET);                                  // Move the file pointer to the beginning of
the file

    fwrite(Allocation_Table, NbBloc * sizeof(int),1,ms);       // Write the updated allocation table to
the file
 }
```

The ***update_Allocation_Table_co*** function updates the allocation table of the disk for a specific block. It first reads the current allocation table from the disk file into an array. The function then modifies the entry corresponding to the specified block address with the provided value (b), indicating whether the block is free or occupied. Finally, the updated allocation table is written back to the file, ensuring the changes are saved.
4o

CONTIGUOUS STORAGE

Clearing the Disk for Contiguous Storage

The ***empty_MS_Co*** function clears the disk by reinitializing it using the ***Initialize_Disk_Co*** function. This effectively resets the allocation table and metadata, preparing the disk for fresh storage operations.

```c
void empty_MS_Co(FILE *ms) {
    Initialize_Disk_Co(ms); // Initialize the disk for contiguous storage
}
```

Allocating Contiguous Blocks

```c
void allouer_co(int *start, int tableAllocation[NbBloc], int nbEtudiant) {
    int freeBlocks = 0;
    *start = -1;
    for (int i = 0; i < NbBloc; i++) {
        if (tableAllocation[i] == 0) { // Unused block
            if (freeBlocks == 0) {*start = i; } // Start of the free sequence
            freeBlocks++;
        } else {
            freeBlocks = 0; // Reset if a block is used
        }

        if (freeBlocks == nbEtudiant) break; // Sufficient sequence found
    }

    if (freeBlocks < nbEtudiant) {
        *start = -1; // Not enough free blocks
    }
}
```

The ***allouer_co*** function allocates a sequence of contiguous blocks for storing data. It scans the allocation table to find the first sequence of free blocks that matches the required size (nbEtudiant). If such a sequence is found, the starting block index is stored in *start. If not, *start is set to -1, indicating that allocation is not possible due to insufficient contiguous free blocks.

Sorting Students by ID

The ***trierTetudiants*** function sorts an array of students (Tetudiant) in ascending order based on their IDs. It uses the bubble sort algorithm, repeatedly comparing and swapping adjacent elements if they are out of order. The process continues until the entire array is sorted.

```c
void trierTetudiants(Tetudiant *tTetudiant, int taille) {
    for (int i = 0; i < taille - 1; i++) {
        for (int j = 0; j < taille - i - 1; j++) {
            if (tTetudiant[j].id > tTetudiant[j + 1].id) {
                Tetudiant temp = tTetudiant[j];
                tTetudiant[j] = tTetudiant[j + 1];
                tTetudiant[j + 1] = temp;
            }
        }
    }
}
```

CONTIGUOUS STORAGE

Disk Compaction for Contiguous Storage

The ***compactdisk_co*** function reorganizes the disk by relocating used blocks to earlier free blocks, ensuring that all occupied blocks are contiguous. It starts by reading the allocation table and scans the disk to find free blocks. When a free block is identified, the function searches for the next used block to relocate. If a relocated block is the first block of a file, the metadata is updated to reflect the new address. The used block is then moved to the free block, and its original location is cleared. Finally, the updated allocation table is written back to the disk, completing the compaction process.

Creating a File in Contiguous Storage

The ***creer_un_fichier_co*** function creates a file in a disk using contiguous storage. It starts by reading the allocation table and calculating the required number of blocks based on the number of students. It then searches for an available metadata block to store file information. Once found, it assigns values to the file metadata, including the file name, size, starting block, and storage mode. The function calls ***allouer_co*** to allocate contiguous blocks for the file and ensures sufficient space is available. Depending on the storage mode, it loads student data into the allocated blocks, either sorted or unsorted. Finally, the allocation table is updated to reflect the occupied blocks, and the file creation is confirmed.

CONTIGUOUS STORAGE

Loading and Storing Student Data in Contiguous Storage

The ***Load_unsorted_students_into_data_file*** function collects student information from the user and stores it in blocks in the data file, without any sorting.

The ***Load_sorted_students_into_data_file*** function first collects all student data, sorts them by ID, and then stores the sorted records into the data file.

The ***chargerFichier_co*** function loads a specified file from the disk into secondary memory by reading its metadata, finding the corresponding blocks, and transferring them to the data file.

Searching for File Metadata

```
FDmeta Searchmetadata_Co(FILE *ms, char FDnom[20]) {
    FDmeta m;
    BLOC_meta meta;
    fseek(ms, NbBloc * sizeof(int), SEEK_SET);
    for (int j = 0; j < NbBlocmeta; ++j) {
        fread(&meta, sizeof(BLOC_meta), 1, ms);
        for (int i = 0; i < FB; ++i) {
            if (strcmp(meta.t[i].FDnom, FDnom) == 0) {
                m = meta.t[i];
                return m;
            }
        }
    }
    // If file is not found, set the address to -1 to indicate failure
    m.adresse = -1;
    return m;
}
```

The ***Searchmetadata_Co*** function searches for a file's metadata in the disk storage by checking each metadata block. It scans through the metadata blocks, looking for a matching file name (FDnom). When a match is found, the corresponding file metadata (FDmeta) is returned. If no matching file is found, the function sets the address field of the metadata to -1, indicating a failure to locate the file.

CONTIGUOUS STORAGE

Updating File Metadata

The ***MAJMETADATA*** function updates the metadata of a specific file stored in disk. It searches through the metadata blocks for a file with the name matching the provided FDnom. When the file is found, its metadata is replaced with the new values from the FDmeta structure (M). The function writes the updated metadata back to the appropriate block. If the file is not found in any of the metadata blocks, an error message is printed indicating the file was not found.

```c
void MAJMETADATA(FILE *ms, FDmeta M){
    BLOC_meta meta;
    int cont_block_meta = 0;
    while (cont_block_meta < NbBlocmeta) {
        rewind(ms);
        fseek(ms, (NbBloc + cont_block_meta) * sizeof(int), SEEK_SET);
        fread(&meta, sizeof(BLOC_meta), 1, ms);
        int cont_meta = 0;
        while (cont_meta < meta.ne) {
            if (strcmp(meta.t[cont_meta].FDnom, M.FDnom) == 0) {
                meta.t[cont_meta] = M;
                fseek(ms, (NbBloc + cont_block_meta) * sizeof(int),
SEEK_SET);          fwrite(&meta, sizeof(BLOC_meta), 1, ms);
                return;
            }
            cont_meta++;
        }
        cont_block_meta++;
    }
    perror("file not found");
}
```

Displaying File Content from Contiguous Storage

The ***Display_fichier_co*** function retrieves and displays the content of a specified file stored in contiguous storage. It first searches for the file's metadata using ***Searchmetadata_Co***. If the file exists (indicated by a valid address), the function reads and prints the student records from each block of the file. The content is displayed student by student, showing their ID, name, first name, and state. If the file cannot be found, an error message is printed.

CONTIGUOUS STORAGE

Renaming a File in Contiguous Storage

The ***Renommer_co*** function allows renaming a file in contiguous storage. It first searches for the file by its old name in the metadata blocks. Once the file is found, its name is updated with the new name. The metadata block is then rewritten to reflect this change. If the file is not found, an error message is displayed. The function ensures that the renaming process is done correctly and updates the storage accordingly.

Deleting a File in Contiguous Storage

The ***supprime_fichier_contigue*** function handles the deletion of a file in contiguous storage. It first searches for the file based on its name in the metadata blocks. Once the file is found, it retrieves the starting address and the number of blocks allocated to the file. These blocks are then marked as free in the allocation table. The metadata for the file is updated to reflect that it is deleted, and the file's information is cleared. Finally, the updated metadata is saved back to the storage, and a confirmation message is displayed, including the number of blocks released.

CONTIGUOUS STORAGE

Inserting a New Student into a Contiguous File

The ***insertion_co*** function adds a new student to an existing file in contiguous storage by first prompting the user for the student's details, including name, first name, ID, and section. It then searches for the file's metadata using the provided file name. If the file exists, the function checks whether it is in unsorted or sorted mode. In unsorted mode, it tries to find space in the current blocks to insert the new student; if space is available, the student is added, and the metadata is updated. In sorted mode, a temporary array is created to hold all the students, and the new student is inserted in the correct position based on the ID, ensuring the list remains sorted. The updated student list is then written back to the file, and the metadata is updated with the new student count. If there is no space left or if the file cannot be found, appropriate error messages are displayed.

Defragmentation and Reorganization of Contiguous Storage File

The ***defragmentation_co*** function reorganizes the data in contiguous storage by defragmenting a file. It first searches for the file's metadata by scanning through the blocks dedicated to metadata. Once the file is found, the function creates an array to hold all the students in the file and reads each block, extracting student information into this array. It then updates the allocation table to mark the blocks as free and writes the students back into new blocks, ensuring that no empty space is left between them. Afterward, the metadata is updated with the new number of students and blocks, and the file is reloaded into secondary memory. The function finally displays the number of students and blocks after the defragmentation process and frees the allocated memory used to store the student data.

CONTIGUOUS STORAGE

Student Search in Contiguous Storage File

The ***Recherche_co function*** is used to search for a specific student's record by their ID in a file, which can either be in unsorted or sorted mode. It reads through the file blocks and updates the num_block and deplacement variables with the block number and the position of the student, respectively. In unsorted mode, it uses a linear search to find the student, while in sorted mode, it performs a binary search to locate the student more efficiently.

Student Record Deletion: Physical vs Logical Methods in Contiguous Files

The ***Suppression_Enregistrement_logique_co*** function is responsible for logically deleting a student's record from a file. It marks the student's record as deleted by setting their state to 0 without physically removing their data from the file. It updates the file's metadata accordingly. The

The ***Suppression_Enregistrement_physic_co*** function, on the other hand, performs a physical deletion, shifting the remaining students' records within the block and reducing the block's size. After deleting the record, it reorganizes the blocks and updates the metadata to reflect the changes in the file, ensuring the file remains properly structured. Both functions utilize the ***Recherche_co*** function to locate the student by their ID before performing the deletion.

DISPLAY

When we run the program, main menu appears:

```
Choose an option:
1. Linked Storage
2. Contiguous Storage
3. < Exit >
```

We will start with the linked storage so we choose 1, then the linked menu appears:

```
1
Choose an action for linked storage:
1. Initialize linked storage
2. Empty linked storage
3. Create linked file (Sorted & Unsorted)
4. Load linked file
5. Display linked file
6. Rename linked file
7. Delete linked file
8. Add to linked list (Sorted & Unsorted)
9. Defragment a linked file
10. Search in linked file (Sorted & Unsorted)
11. Delete record from linked file (Physical & Logical)
12. < Return >
```

First we initialize the linked storage:

```
1
Initializing disk for linked storage...
Disk initialized for linked storage.
Choose an action for linked storage:
```

We can now create two files, one sorted and one unsorted, lets do the sorted one in this mode:

```
3
Enter the number of students: 5
Enter the file name: SStudent
Choose file type (0 for unsorted, 1 for sorted): 1
Starting Creer_du_fichiertrieechainee
Metadata created: nbEtudiant=5, FDnom=SStudent, taille=1, adresse=0
Initial meta block read: ne=1
Meta block directly updated: ne=2
ID : 1
Name : Ahmed
Surname : Benayache
Section : C
Student 0: ID=1, Name=Ahmed, Surname=Benayache, Section=C
ID : 2
Name : Anis
Surname : Ben Azza
Section : B
Student 1: ID=2, Name=Anis, Surname=Ben Azza, Section=B
ID : 0
Name : Amine
Surname : Hamidi
Section : A
Student 2: ID=0, Name=Amine, Surname=Hamidi, Section=A
ID : 5
Name : Youcef
Surname : Alliche
Section : C
Student 3: ID=5, Name=Youcef, Surname=Alliche, Section=C
ID : 3
Name : Taib
Surname : Salma
Section : A
Student 4: ID=3, Name=Taib, Surname=Salma, Section=A
Students sorted by ID
Sorted Student 0: ID=0, Name=Amine, Surname=Hamidi, Section=A
Sorted Student 1: ID=1, Name=Ahmed, Surname=Benayache, Section=C
Sorted Student 2: ID=2, Name=Anis, Surname=Ben Azza, Section=B
Sorted Student 3: ID=3, Name=Taib, Surname=Salma, Section=A
Sorted Student 4: ID=5, Name=Youcef, Surname=Alliche, Section=C
Final buffer written with remaining students
Function completed
```

DISPLAY

After that we load the linked file:

```
4
Loading a linked file...
Enter the file name to load: SStudent
[DEBUG] Searching for metadata for the file: SStudent
[DEBUG] Reading metadata block 0
[DEBUG] Checking file:
[DEBUG] Checking file: SStudent
[DEBUG] File found: SStudent, Address: 0, Size: 1
[DEBUG] Starting to read the file and insert it into secondary memory.
[DEBUG] Reading block 0 from the source file.
[DEBUG] Record found: Hamidi
[DEBUG] Record found: Benayache
[DEBUG] Record found: Ben Azza
[DEBUG] Record found: Salma
[DEBUG] Record found: Alliche
[DEBUG] Last block: next = -1
[DEBUG] Block 0 inserted at address 0
Choose an action for linked storage:
```

Now we display the content to test if the file have been created succesfuly:

```
5
Displaying a linked file...
Enter the file name to display: SStudent
[INFO] The file 'SStudent' exists. Loading data...
[DEBUG] Metadata: Address = 0, Size = 1
[INFO] Block 1:
  ID: 0, Name: Amine, Surname: Hamidi, Section: A, Status: 1
  ID: 1, Name: Ahmed, Surname: Benayache, Section: C, Status: 1
  ID: 2, Name: Anis, Surname: Ben Azza, Section: B, Status: 1
  ID: 3, Name: Taib, Surname: Salma, Section: A, Status: 1
  ID: 5, Name: Youcef, Surname: Alliche, Section: C, Status: 1
  Next Block Address: -1
[INFO] File loading completed.
```

Now lets rename the file from ***SStudent*** to ***Student***:

```
6
Renaming a linked file...
Enter the current file name: SStudent
Enter the new file name: Student
```

Now we continue with the name ***Student,*** lets add an element to the file:

```
8
Inserting into a linked file...
Enter the file name: Student
Do you want to insert in sorted or unsorted order? (0 for unsorted, 1 for sorted): 1
Enter the new student's information:
ID: 7
Name: Yani
Surname: Doudou
Section: C
DEBUG: Starting search for metadata for file: Student
DEBUG: Reading metadata block #0
DEBUG: Checking metadata entry:
DEBUG: Checking metadata entry: Student
DEBUG: Metadata found - Address: 0, nbEtudiant: 5, taille: 1
DEBUG: Successfully allocated memory for 7 students
DEBUG: Reading blocks to populate student array...
DEBUG: Moved to linked list address: 0
DEBUG: Read block #0 with next = -1, ne = 5
DEBUG: Student ID: 0 added to array at index 0
DEBUG: Student ID: 1 added to array at index 1
DEBUG: Student ID: 2 added to array at index 2
DEBUG: Student ID: 3 added to array at index 3
DEBUG: Student ID: 5 added to array at index 4
DEBUG: Inserting new student with ID: 7
DEBUG: Writing updated blocks back to the file...
DEBUG: Writing student ID: 0 to block 0 at index 0
DEBUG: Writing student ID: 1 to block 0 at index 1
DEBUG: Writing student ID: 2 to block 0 at index 2
DEBUG: Writing student ID: 3 to block 0 at index 3
DEBUG: Writing student ID: 5 to block 0 at index 4
DEBUG: Allocated new block: 1 at iteration 1
DEBUG: Current buffer.next = -1
DEBUG: Updated meta.adresse = 0
Debug: Reading block meta #1
Debug: Comparing   with Student
Debug: Comparing Student with Student
Debug: Found matching metadata! Address: 0, Taille: 2
updating the Metadata DEBUG: Insertion completed successfully
```

DISPLAY

Now, lets search for the added element wich is the ID*7:

```
10
Searching in a linked file...
Enter the file name: Student
Enter the student ID to search: 7
Is the file sorted? (1 for yes, 0 for no): 1
Debug: Calculated z = 200
Debug: fseek to metadata completed.
Debug: Reading block meta 1
Debug: Comparing   with Student
Debug: Comparing Student with Student
Debug: Found matching metadata! Address: 0, Size: 2
Debug: fseek to first block at address 0 completed.
Debug: Read first block. Buffer details - Next: 1, NE: 5
Debug: SortedSearch returned -1
Debug: Moving to next block. Address: 1
Debug: Read next block 2. Buffer details - Next: -1, NE: 1
Debug: SortedSearch in block 2 returned 0
Found student at block 2, position 0
```

Good, it did create a new block and inserted the new element to it, we can recheck later with the display but now, lets delete logicaly the statue of the student with ID*0:

```
11
Deleting a record from a linked file...
Enter the file name: Student
Enter the student ID to delete: 0
Choose deletion type (0 for logical, 1 for physical): 0
DEBUG: Starting logical_deletion_from_linked_file...
DEBUG: Searching for metadata for file 'Student'...
DEBUG: Moved to metadata section (offset = 200).
DEBUG: Reading metadata block 0...
DEBUG: Checking file ' ' in metadata index 0...
DEBUG: Checking file 'Student' in metadata index 1...
DEBUG: File 'Student' found in metadata.
DEBUG: File metadata located. Starting address = 0, number of students = 6
DEBUG: Reset 'found' to 0 for student search.
DEBUG: Reading block at address 0...
DEBUG: Moved to block address 0 (offset = 2240).
DEBUG: Block read. Number of entries = 5, Next block = 1.
DEBUG: Checking student ID 0 in block 0, position 0...
DEBUG: Student with ID 0 found. Performing logical deletion...
DEBUG: Writing updated block back to MS.
DEBUG: Student logically deleted. Exiting loop.
DEBUG: Updated metadata. New student count = 5.
DEBUG: Writing updated metadata to block index 0.
DEBUG: Logical deletion process completed successfully.
Logical deletion completed.
Updated number of students: 5
```

Now, lets check by displaying the content of the file:

```
5
Displaying a linked file...
Enter the file name to display: Student
[INFO] The file 'Student' exists. Loading data...
[DEBUG] Metadata: Address = 0, Size = 2
[INFO] Block 1:
  ID: 0, Name: Amine, Surname: Hamidi, Section: A, Status: 0
  ID: 1, Name: Ahmed, Surname: Benayache, Section: C, Status: 1
  ID: 2, Name: Anis, Surname: Ben Azza, Section: B, Status: 1
  ID: 3, Name: Taib, Surname: Salma, Section: A, Status: 1
  ID: 5, Name: Youcef, Surname: Alliche, Section: C, Status: 1
  Next Block Address: 1
[INFO] Block 2:
  ID: 7, Name: Yani, Surname: Doudou, Section: C, Status: 1
  Next Block Address: -1
[INFO] File loading completed.
```

DISPLAY

Great, it seems like we did add an element and delete logically the student with ID*0, now lets try the defragmentation:

```
9
Defragmenting a linked file...
Enter the file name: Student
Checking file:
Checking file: Student
File found: Student (Index: 1, BlocMetaIndex: 0)
Size: 2
Reading block at address: 0
Block read: 5 students, Next: 1
Student ID: 0
Student ID: 1
Student ID: 2
Student ID: 3
Student ID: 5
1Reading block at address: 1
Block read: 1 students, Next: -1
Student ID: 7
Updating allocation table for block: 1
2Total number of students after defragmentation: 5
Block written: 5 students
Metadata updated: Number of students = 5, Number of blocks = 1, Address = 0
[DEBUG] Searching for metadata for the file: Student
[DEBUG] Reading metadata block 0
[DEBUG] Checking file:
[DEBUG] Checking file: Student
[DEBUG] File found: Student, Address: 0, Size: 1
[DEBUG] Starting to read the file and insert it into secondary memory.
[DEBUG] Reading block 0 from the source file.
[DEBUG] Record found: Benayache
[DEBUG] Record found: Ben Azza
[DEBUG] Record found: Salma
[DEBUG] Record found: Alliche
[DEBUG] Record found: Doudou
[DEBUG] Last block: next = -1
[DEBUG] Block 0 inserted at address 0
Defragmentation completed.
Number of students updated: 5
Number of blocks updated: 1
```

Now, lets see if it worked by the display operation:

```
5
Displaying a linked file...
Enter the file name to display: Student
[INFO] The file 'Student' exists. Loading data...
[DEBUG] Metadata: Address = 0, Size = 1
[INFO] Block 1:
  ID: 1, Name: Ahmed, Surname: Benayache, Section: C, Status: 1
  ID: 2, Name: Anis, Surname: Ben Azza, Section: B, Status: 1
  ID: 3, Name: Taib, Surname: Salma, Section: A, Status: 1
  ID: 5, Name: Youcef, Surname: Alliche, Section: C, Status: 1
  ID: 7, Name: Yani, Surname: Doudou, Section: C, Status: 1
  Next Block Address: -1
[INFO] File loading completed.
```

Yeah, now lets delete physically the student with ID*3, and display than the file content:

DISPLAY


```
11
Deleting a record from a linked file...
Enter the file name: Student
Enter the student ID to delete: 3
Choose deletion type (0 for logical, 1 for physical): 1
Starting physical deletion of the file Student for the student with ID: 3
Checking file:
Checking file: Student
File found: Student (Index: 1, BlocMetaIndex: 0)
Number of students before deletion: 5
Size 1
Reading block at address: 0
Block read: 5 students, Next: -1
Student ID: 1
Student ID: 2
Student ID: 3
Student with ID 3 found and will be deleted.
Student ID: 5
Student ID: 7
1Total number of students after deletion: 4
Block written: 4 students
Metadata updated: Number of students = 4, Number of blocks = 1, address 0
[DEBUG] Searching for metadata for the file: Student
[DEBUG] Reading metadata block 0
[DEBUG] Checking file:
[DEBUG] Checking file: Student
[DEBUG] File found: Student, Address: 0, Size: 1
[DEBUG] Starting to read the file and insert it into secondary memory.
[DEBUG] Reading block 0 from the source file.
[DEBUG] Record found: Benayache
[DEBUG] Record found: Ben Azza
[DEBUG] Record found: Alliche
[DEBUG] Record found: Doudou
[DEBUG] Last block: next = -1
[DEBUG] Block 0 inserted at address 0
Physical deletion completed.
Updated number of students: 4
Updated number of blocks: 1
```

Displaying:

```
5
Displaying a linked file...
Enter the file name to display: Student
[INFO] The file 'Student' exists. Loading data...
[DEBUG] Metadata: Address = 0, Size = 1
[INFO] Block 1:
  ID: 1, Name: Ahmed, Surname: Benayache, Section: C, Status: 1
  ID: 2, Name: Anis, Surname: Ben Azza, Section: B, Status: 1
  ID: 5, Name: Youcef, Surname: Alliche, Section: C, Status: 1
  ID: 7, Name: Yani, Surname: Doudou, Section: C, Status: 1
  Next Block Address: -1
[INFO] File loading completed.
```

Excellent, now lets delete the file and try to desplay it:

```
7
Deleting a linked file...
Enter the file name to delete: Student
Starting to search for file metadata...
Searching in metadata block 0...
Checking file:
Checking file: Student
File found: Student
Starting to delete file blocks...
Deleting block 0...
Allocation table updated for block 0 (freed).
Updating metadata...
File deletion completed successfully.
```

DISPLAY

Now, Desplaying:

```
5
Displaying a linked file...
Enter the file name to display: Student
[INFO] The file 'Student' exists. Loading data...
[DEBUG] Metadata: Address = 43, Size = 415
[ERROR] Unable to read block at address 43. Aborting...
```

Okey, it seems like the file is no longer there, lets empty the memory:

```
2
Emptying linked storage...
Linked storage emptied.
```

Now we return to the main menu to switch the mode:

```
12
Choose an option:
1. Linked Storage
2. Contiguous Storage
3. < Exit >
```

Now, lets jump to the Contiguous Storage mode:

```
2
Choose an action for contiguous storage:
1. Initialize contiguous storage
2. Empty contiguous storage
3. Create contiguous file (Sorted & Unsorted)
4. Compact a contiguous file
5. Load contiguous file
6. Display contiguous file
7. Rename contiguous file
8. Delete contiguous file
9. Add to contiguous list (Sorted & Unsorted)
10. Defragment a contiguous file
11. Search in contiguous file (Sorted & Unsorted)
12. Delete record from contiguous file (Physical & Logical)
13. < Return >
```

First we initialize the memory:

```
1
Initializing disk for contiguous storage...
Disk initialized for contiguous storage.
```

Now we create unsorted file:

DISPLAY


```
3
Creating a contiguous file...
Enter the number of students: 5
Enter the file name: UStudent
Choose file type (0 for unsorted, 1 for sorted): 0
Enter student information (ID, Name, First name, Section):
Student 1:
ID: 5
Name: Ahmed
First name: Benayache
Section: C
Student 2:
ID: 1
Name: Amine
First name: Yalaoui
Section: A
Student 3:
ID: 2
Name: Amine
First name: Hamidi
Section: A
Student 4:
ID: 0
Name: Anis
First name: Benazza
Section: B
Student 5:
ID: 4
Name: Youcef
First name: Alliche
Section: D
File 'UStudent' created successfully.
```

Now we comapct the created file:

```
4
Compacting a contiguous file...
```

Then, we load the file:

```
5
Loading a contiguous file...
Enter the file name to load: UStudent
File 'UStudent' successfully loaded into secondary memory.
```

Then, we display the content of the file:

```
6
Displaying a contiguous file...
Enter the file name to display: UStudent
Reading block 0
Content of the read block:
Student ID: 5, Name: Ahmed, First Name: Benayache, State: 1
Student ID: 1, Name: Amine, First Name: Yalaoui, State: 1
Student ID: 2, Name: Amine, First Name: Hamidi, State: 1
Student ID: 0, Name: Anis, First Name: Benazza, State: 1
Student ID: 4, Name: Youcef, First Name: Alliche, State: 1
```

Now, lets rename the file from ***UStudent*** to ***Student***:

```
7
Renaming a contiguous file...
Enter the current file name: UStudent
Enter the new file name: Student
```

DISPLAY

Now lets try to delete one of the elements physically, for example ID*0:

```
12
Deleting a record from a contiguous file...
Enter the file name: Student
Enter the student ID to delete: 0
Choose deletion type (0 for logical, 1 for physical): 1
File name Student
File address 1
File size 1
Number of students in the file 5
Reading block 0
Content of the read block:
Student ID: 5, Name: Ahmed
Student ID: 1, Name: Amine
Student ID: 2, Name: Amine
Student ID: 0, Name: Anis
Student found in block 1, position 3
The student with ID 0 has been successfully deleted.
```

Okey, now we delete one logically, for exmaple ID*4:

```
12
Deleting a record from a contiguous file...
Enter the file name: Student
Enter the student ID to delete: 4
Choose deletion type (0 for logical, 1 for physical): 0
File name Student
File address 1
File size 1
Number of students in the file 4
Reading block 0
Content of the read block:
Student ID: 5, Name: Ahmed
Student ID: 1, Name: Amine
Student ID: 2, Name: Amine
Student ID: 4, Name: Youcef
Student found in block 1, position 3
The student has been successfully deleted.
```

Now, lets defragment to see if the element deleted logically is gone:

DISPLAY


```
10
Defragmenting a contiguous file...
Enter the file name: Student
Checking file: Student
File found: Student (Index: 0, BlocMetaIndex: 0)
Size: 1
Reading block at address: 1
Block read: 4 students
Student ID: 5
Student ID: 1
Student ID: 2
Student ID: 4
Total number of students after defragmentation: 3
Block written: 3 students
Metadata updated: Number of students = 3, Number of blocks = 1, Address = 1
File 'Student' successfully loaded into secondary memory.
Defragmentation completed.
Number of students updated: 3
Number of blocks updated: 1
```

Lets now search for the student with ID*5:

```
11
Searching in a contiguous file...
Enter the file name: Student
Enter the student ID to search: 5
File name Student
File address 1
File size 1
Number of students in the file 3
Reading block 0
Content of the read block:
Student ID: 5, Name: Ahmed
Student found in block 1, position 0
```

Now lets display to check if everything we done is done correctly:

```
6
Displaying a contiguous file...
Enter the file name to display: Student
Reading block 0
Content of the read block:
Student ID: 5, Name: Ahmed, First Name: Benayache, State: 1
Student ID: 1, Name: Amine, First Name: Yalaoui, State: 1
Student ID: 2, Name: Amine, First Name: Hamidi, State: 1
```

Great, since everything is working correctly, lets add an element to the file:

```
9
Inserting into a contiguous file...
Enter the file name: Student
Enter the information for the new student
Name: Yasmine
First name: Benkari
ID: 9
Section: A
DEBUG: New student details - Name: Yasmine, First Name: Benkari, ID: 9, Section: A
Insertion successful
```

Now lets display the content of the file to see if the student we added is there:

DISPLAY


```
Displaying a contiguous file...
Enter the file name to display: Student
Reading block 0
Content of the read block:
Student ID: 5, Name: Ahmed, First Name: Benayache, State: 1
Student ID: 1, Name: Amine, First Name: Yalaoui, State: 1
Student ID: 2, Name: Amine, First Name: Hamidi, State: 1
Student ID: 9, Name: Yasmine, First Name: Benkari, State: 1
```

Great, now we ddelete the file:

```
8
Deleting a contiguous file...
Enter the file name to delete: Student
File deletion completed.
Number of blocks released: 4
```

We try to display the deleted file:

```
6
Displaying a contiguous file...
Enter the file name to display: Student
Error: File 'Student' does not exist.
```

Since everything is working correctly, lets empty the memory

```
2
Emptying contiguous storage...
Contiguous storage emptied.
```

WE ARE DONE NOW!

CONCLUTION

In conclusion, this project demonstrates the fundamental principles of file system operations through the development of a simplified file management simulator. By implementing and analyzing key functions such as memory allocation, file management, and metadata handling, the simulator provides a clear understanding of how these processes work in real-world systems. The exploration of algorithms for file creation, insertion, deletion, and memory compaction highlights the challenges and solutions in optimizing file system performance. Additionally, the user interface effectively bridges the technical operations and user interaction, showcasing the importance of usability in system design. This project serves as a valuable learning tool, offering insights into the complexities of file system management and the algorithms that support it.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
Martin Fowler

CREDITS

Name	Matricula	Job
Benayache Ahmed Yassine	232331691710	Report & Main Developer
Hamidi Amine	232331640902	Linked Developer
Ben Azza Mohamed Anis	232331674106	Linked Developer
Yalaoui Mohamed Elamine	232331717704	Contiguous Developer
Alliche Ali Youcef	232331443618	Contiguous Developer
Salma Taib	232331640905	SM Developer
Doudou Yani	232331633907	SM Developer

Big Thanks to You!