Université des Sciences
et de la Technologie
Houari Boumediene.

Simulateur Simplifié d'un Système de Gestion de Fichiers (SGF)

DAHDAH Leticia

ZIANE Fatima Zohra

Université des Sciences et de la Technologie Houari-Boumédiène
Faculté d'Informatique
2e Année Ingénieur Informatique
Structures de Fichiers & Structures de Données (SFSD)

USTHB, January 2025

Université des Sciences
et de la Technologie
Houari Boumediene.

Simulateur Simplifié d'un Système de Gestion de Fichiers (SGF)

DAHDAH Leticia
Etudiant No. 232331425902

ZIANE Fatima Zohra
Etudiant No. 232337422719

Professeur: Dr. LAHRECHE Abdelmadjid
Docteur, Université des Sciences et de la Technologie Houari-Boumédiène

Université des Sciences et de la Technologie Houari-Boumédiène
Faculté d'Informatique
2e Année Ingénieur Informatique
Structures de Fichiers & Structures de Données (SFSD)

USTHB, January 2025

Table des matières

Liste des tableaux

1

Introduction

1.1 Contexte

Les systèmes de gestion de fichiers (SGF) jouent un rôle fondamental dans le fonctionnement des systèmes informatiques modernes. Ils permettent de stocker, organiser et manipuler les données sur des supports de mémoire secondaire tout en assurant une gestion efficace des ressources disponibles. Les fonctionnalités telles que l'allocation de mémoire, la recherche d'informations, et la défragmentation de l'espace sont essentielles pour garantir des performances optimales. Dans un contexte pédagogique, concevoir un simulateur simplifié de gestion de fichiers permet de comprendre les principes sous-jacents et d'appliquer des concepts théoriques à des problématiques concrètes.

1.2 Objectifs

Ce projet a pour principal objectif de développer un simulateur simplifié d'un système de gestion de fichiers, capable de reproduire les opérations de base telles que :

- L'allocation et la gestion de blocs mémoire.
- La création, la suppression, et la recherche d'enregistrements.
- La défragmentation et le compactage de la mémoire secondaire.

À travers ce projet, nous visons à renforcer la compréhension des concepts fondamentaux des SGF tout en appliquant les connaissances acquises en programmation et en algorithmique.

2

Conception

2.1 Structure de la Mémoire Secondaire

2.1.1 Blocs et Enregistrements

1. **Premier Bloc**
   - Réservé à la table d'allocation uniquement.
2. **Blocs Suivants**
   - Les premiers blocs après le premier sont utilisés pour les métadonnées.
   - Le reste des blocs est réservé aux fichiers de données.
3. **Contenu d'un Bloc**
   - Chaque bloc contient plusieurs enregistrements logiques.
   - Un enregistrement logique (`Element`) inclut trois champs :
     - `ID` (un entier)
     - `row` (une chaîne de caractères)
     - `flag` (un entier)
   - Chaque bloc contient également :
     - `nbr_element` (un entier)
     - `last` (un entier)
     - `next` (un entier)

2.1.2 Table d'Allocation

1. **Contenu**
   - Elle contient deux types d'informations :
     - L'adresse du bloc
     - Son état (libre ou occupé)
2. **Représentation**
   - Chaque bloc et son état sont codés sous forme de chaînes de 4 caractères :

  - Adresse du bloc sur 3 caractères
  - État sur 1 caractère

3. **Stockage**
   - Un certain nombre de ces chaînes de 4 caractères est stocké dans le champ `row` des enregistrements logiques du premier bloc.
   - Le champ `ID` contient l'adresse du premier bloc dont l'information est stockée dans le champ `row` du même enregistrement.
   - Le champ `flag` est mis à `-1` dans tous les enregistrements de ce bloc, également les champs `last` et `next` du bloc.
   - Le champ `nbr_element` contient le nombre total de blocs réservés pour les fichiers de données.

2.1.3 Métadonnées

1. **Contenu**
   - Nom du fichier.
   - Taille du fichier en blocs.
   - Taille du fichier en enregistrements.
   - Adresse du premier bloc.
   - Adresse du dernier bloc.
   - Mode d'organisation globale.
   - Mode d'organisation interne.
   - Flag.
2. **Représentation**
   - Nombre d'enregistrements occupés (3 caractères)
   - Adresse du premier bloc (3 caractères)
   - Adresse du dernier bloc (3 caractères)
   - Mode d'organisation globale (1 caractère) :
     - `1` si contigu
     - `0` si chaîné
   - Mode d'organisation interne (1 caractère) :
     - `1` si trié
     - `0` sinon
   - Le reste du champ `row` est réservé pour le nom du fichier.
3. **Stockage**
   - Le champ `ID` contient le nombre de blocs occupés par le fichier.
   - Le champ `row`.
   - Le champ `flag` est défini comme suit :
     - `0` si le fichier est supprimé
     - `1` sinon

- Les autres champs du bloc ne sont pas utilisés sauf le champ `last` du premier bloc des blocs de métadonnées (Il garde le nombre d'enregistrements occupés ou supprimés logiquement dans les métadonnées).

2.1.4 Fichiers de Données

1. **Stockage**
   - `ID` : Stocke l'identifiant unique de l'enregistrement.
   - `row` : Contient l'information proprement dite.
   - `flag` :
     - `0` si l'enregistrement est supprimé logiquement.
     - `1` sinon.
   - `last` et `next` du bloc sont définis comme suit :
     - Mis à `-1` si le mode d'organisation globale est contigu.
     - Sinon, elles contiennent respectivement l'adresse du bloc précédent et du bloc suivant dans le fichier.
     - Le champ `last` (ou `next`) est également mis à `-1` si le bloc est le premier (ou le dernier) du fichier.

2.2 Algorithme de Conception

Les algorithmes développés dans le cadre de ce projet sont conçus pour gérer les opérations principales des Systèmes de Gestion de Fichiers (SGF), permettant de gérer les fichiers et leur manipulation au sein d'un système. Ces algorithmes sont organisés en différentes catégories, reflétant les opérations fondamentales.

MS/ Fichier de données	**Fonctionnalités**	**Fonctions**
MS	**(Table d'allocation)**	initAllocationTable(FILE* ms)
		UploadAllocationTableToMC(FILE* ms)
		TablefromMCtoMS(AllocationTableBuffer* MC_Table, FILE* ms)
		MAJTable(FILE* ms, int contigue, int size, int Bloc_list[], int x)
	Initialiser	initMS(FILE* ms)
	Vider	deleteMS(FILE* ms)
	Compacter	Compact(FILE* ms)
	Vérifier l'espace	Check_available(FILE* ms, int n, int contigue)
		CheckMSavailability(FILE* ms, int n, int contigue)

MS/Fichier de données	Fonctionnalités	Fonctions
Fichier de données	**(Meta_Données)**	UpdateMeta(FILE* ms, char curr_name[], char new_name[], int bloc, int record, int first_address, int last_address, int Add_Addresses[], int added_a, int delete_Addresses[], int deleted_a, int delete)
	Créer	Create_file(FILE* ms, FILE* inputFile, char name[], int ne, int contigue, int ordered)
		ChargeFile(FILE* ms, FILE* inputFile, int size, int nbr_e, int* addressList, int contigue)
	Rechercher	binarySearchElements(Element* elements, int nbr_element, int id)
		binarySearchContiguous(FILE* ms, int id, int first-Block, int lastBlock)
		binarySearchChained(FILE* ms, int id, int first-Block)
		linearSearchContiguous(FILE* ms, int id, int first-Block, int lastBlock)
		linearSearchChained(FILE* ms, int id, int firstBlock)
		searchElementByID(FILE* ms, char fileName[], int id)
	Insérer	binarySearchPosition(Element* elements, int nbr_element, int id)
		Insertion_Contigue_NotOrdered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)
		Insertion_Chainnee_NotOrdered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)
		Insertion_Contigue_Ordered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)
		Insertion_Chainnee_Ordered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)
		Adding_Contiguous_2(FILE* ms, char name[], int bloc, int record, int first_address, int last_address, int ID, char data[])
		Adding_Chainnee_2(FILE* ms, char name[], int ID, char data[], int first_address, int bloc, int added_bloc)
		Adding_Chainnee_1(FILE* ms, char name[], int ID, char data[], int first_address, int bloc, int record)
		Adding_Contiguous_1(FILE* ms, char name[], Bloc Buffer[], int* ptr, int size, int ID, char data[])
		insertElement(FILE* ms, char name[], int ID, char data[])
	Supprimer	LogicalSuppression(FILE* ms, char fileName[], int id)

MS/Fichier de données	**Fonctionnalités**	**Fonctions**
		PhysicalSuppression(FILE* ms, char filename[], int id)
	Défragmenter	defragmentation(FILE *ms, char fileName[])
	Autres	ElementToMetaBufferPart(Element meta)
		MetaBufferToElement(MetaBuffer Buffer)
		Get_ChainneeAddresses(FILE* ms, int first_address)

Table 2.1: *Les Fonctions du SGF*

3

Implementation

3.1 Gestion de la mémoire (MS)

`void initAllocationTable(FILE* ms)`

Initialise les adresses des blocs et met leurs états à vide (0).

`void UploadAllocationTableToMC(FILE* ms)`

Charge le premier bloc de la mémoire secondaire dans un buffer, extrait les adresses des blocs de fichiers de données ainsi que leurs états, et les stocke dans une table dans la mémoire centrale.

`void TablefromMCtoMS(AllocationTableBuffer* MC_Table, FILE* ms)`

Charge la table d'allocation de la mémoire centrale dans un buffer et écrit le bloc dans la mémoire secondaire (fait l'inverse de la fonction UploadAllocationTableToMC).

`void initMS(FILE* ms)`

Définit la taille et la structure de la mémoire secondaire, puis initialise la table d'allocation.

`void deleteMS(FILE* ms)`

Réinitialise la mémoire secondaire en fermant le fichier, le ré-ouvrant en mode écriture afin de le vider, et en réinitialisant la table d'allocation des blocs.

`void MAJTable(FILE* ms, int contigue, int size, int Bloc_list[], int x)`

- Si contigu : Change l'état de tous les blocs entre la première et la deuxième adresse dans Bloc_list vers x.
- Si non contigu : Parcourt Bloc_list et met à jour l'état de chaque bloc individuellement.

`void Compact(FILE* ms)`

Réorganise le fichier ms en copiant les blocs occupés vers le début d'un fichier, mettant à jour la table d'allocation et les métadonnées, puis remplace le fichier original par la version optimisée.

`int* Check_available(FILE* ms, int n, int contigue)`

- Si non contigu : Parcourt la table d'allocation à la recherche de blocs libres (état = 0), les ajoute à une liste, et retourne cette liste si le nombre de blocs libres atteint n. Sinon, retourne NULL.
- Si contigu : Cherche une séquence de n blocs libres, en réinitialisant le compteur à chaque bloc occupé rencontré. Retourne la séquence si n atteint, sinon NULL.

`int* CheckMSavailability(FILE* ms, int n, int contigue)`

- Si non contigu : Appelle Check_available pour rechercher des blocs libres. Retourne la liste trouvée ou NULL.
- Si contigu : Vérifie d'abord la disponibilité avec Check_available. Si insuffisant, appelle Compact pour réorganiser la mémoire, puis vérifie à nouveau. Retourne une liste de n blocs contigus ou NULL.

3.2 Gestion des fichiers de métadonnées

`MetaBuffer* ElementToMetaBufferPart(Element meta)`

Extrait les données de meta et crée un MetaBuffer.

`Element* MetaBufferToElement(MetaBuffer Buffer)`

Fait l'inverse de ElementToMetaBufferPart.

`int* Get_ChainneeAddresses(FILE* ms, int first_address)`

Lit les blocs chaînés dans le fichier, à partir de l'adresse donnée, et les stocke dans un tableau d'adresses.

`MetaBuffer* Get_MetaData(FILE* ms, char name[])`

Parcourt les blocs de métadonnées, cherche un nom correspondant et retourne un MetaBuffer si trouvé, sinon NULL.

`int UpdateMeta(FILE* ms, char curr_name[], char new_name[], int bloc, int record, int first_address, int last_address, int Add_Addresses[], int added_a, int delete_Addresses[], int deleted_a, int delete)`

Met à jour les métadonnées selon les paramètres fournis, permettant de : - Renommer le fichier si un nouveau nom est donné et qu'il n'existe pas déjà.
- Supprimer les métadonnées en réinitialisant les adresses et en mettant à jour la table d'allocation si delete est égal à 1, ou si bloc ou record sont égaux à 0.
- Mettre à jour les informations de bloc, d'enregistrement, d'adresses, et appliquer les changements à la mémoire si nécessaire.

`void defragmentation_Meta(FILE* ms)`

Réorganise les métadonnées pour éliminer les éléments supprimés.

`void Create_MetaData(FILE* ms, char name[], int bloc, int ne, int first, int last, int contigue, int ordered)`

Crée et ajoute de nouvelles métadonnées dans la mémoire secondaire, en vérifiant d'abord si une défragmentation est nécessaire avant d'ajouter les métadonnées.

3.3 Gestion des fichiers de données

`void ChargeFile(FILE* ms, FILE* inputFile, int size, int nbr_e, int* addressList, int contigue)`

Charge des données depuis un fichier d'entrée dans la mémoire secondaire.

`int Create_file(FILE* ms, FILE* inputFile, char name[], int ne, int contigue, int ordered)`

Crée un fichier dans la mémoire secondaire en initialisant ses métadonnées et en chargeant les données depuis un fichier d'entrée. Elle vérifie d'abord si la mémoire est disponible pour le nombre de blocs requis. Si l'espace est disponible et que le nom du fichier n'existe pas déjà, elle crée les métadonnées, charge les données et retourne 0. Si le nom du fichier existe déjà, elle retourne 1. Si l'espace mémoire n'est pas suffisant, elle retourne -1.

`void defragmentation(FILE* ms, char fileName[])`

Supprime les fragmentations en réorganisant les enregistrements valides et libère l'espace occupé par les blocs inutilisés.

`searchResult searchElementByID(FILE* ms, char fileName[], int id)`

Cette fonction permet de rechercher un élément dans un fichier donné en utilisant l'identifiant (id). Elle utilise différentes méthodes de recherche selon les caractéristiques des données (ordonnées ou non, contiguës ou chaînées). Elle fait appel aux fonctions suivantes :

- `MetaBuffer* Get_MetaData(FILE* ms, char fileName[])`
  - Récupère les métadonnées du fichier, telles que si les éléments sont ordonnés ou non, et si l'organisation des blocs est contiguë ou chaînée.
- `searchResult binarySearchContiguous(FILE* ms, int id, int firstBlock, int lastBlock)`
  - Utilisée lorsque les éléments sont ordonnés et organisés de manière contiguë. Elle effectue une recherche binaire pour localiser l'élément.
- `searchResult binarySearchChained(FILE* ms, int id, int firstBlock)`
  - Utilisée lorsque les éléments sont ordonnés et organisés de manière chaînée. Elle effectue une recherche binaire sur les blocs chaînés.
- `int binarySearchElements(Element* elements, int nbr_element, int id)`
  - Fonction auxiliaire utilisée dans les recherches binaires (binarySearchContiguous et binarySearchChained) pour effectuer une recherche binaire sur les éléments d'un bloc.
- `searchResult linearSearchContiguous(FILE* ms, int id, int firstBlock, int lastBlock)`
  - Utilisée lorsque les éléments ne sont pas ordonnés et organisés de manière contiguë. Elle effectue une recherche linéaire pour localiser l'élément.
- `searchResult linearSearchChained(FILE* ms, int id, int firstBlock)`
  - Utilisée lorsque les éléments ne sont pas ordonnés et organisés de manière chaînée. Elle effectue une recherche linéaire à travers les blocs chaînés.

`bool LogicalSuppression(FILE* ms, char fileName[], int id)`

Marque l'élément comme supprimé en mettant son flag à 0 et décrémente le nombre d'éléments dans le bloc. Si le bloc devient vide, il est supprimé via `MAJTable`, et les métadonnées sont mises à jour (enregistrements et blocs).

`void PhysicalSuppression(FILE* ms, char filename[], int id)`

Remplace l'élément à supprimer par le dernier élément. Si le dernier bloc devient vide, il est supprimé. Pour les données ordonnées, elle appelle `LogicalSuppression` suivie d'une défragmentation. Les métadonnées sont ajustées pour refléter les changements.

`int insertElement(FILE* ms, char name[], int ID, char data[])`

La fonction `insertElement(FILE* ms, char name[], int ID, char data[])` insère un élément dans la mémoire, en utilisant les métadonnées obtenues via `MetaBuffer* Get_MetaData(FILE* ms, char fileName[])` pour déterminer si l'élément doit être inséré de manière contiguë ou chaînée, et s'il doit être ordonné ou non.

- Si l'insertion est ordonnée :

- `int Insertion_Contigue_Ordered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)` qui utilise :
  - `int Adding_Contiguous_1(FILE* ms, char name[], Bloc Buffer[], int* ptr, int size, int ID, char data[])`
  - `int Adding_Contiguous_2(FILE* ms, char name[], int bloc, int record, int first_address, int last_address, int ID, char data[])`
- `int Insertion_Chainnee_Ordered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)` qui utilise :
  - `int Adding_Chainnee_1(FILE* ms, char name[], int ID, char data[], int first_address, int bloc, int record)`
  - `int Adding_Chainnee_2(FILE* ms, char name[], int ID, char data[], int first_address, int bloc, int added_bloc)`

  en combinaison avec `int binarySearchPosition(Element* elements, int nbr_element, int id)`

- Si l'insertion est non ordonnée :

- `Insertion_Contigue_NotOrdered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)`
- `int Insertion_Chainnee_NotOrdered(FILE* ms, char name[], int ID, char data[], MetaBuffer* metaBuffer)`

4

Discussion

Dans ce projet, nous avons cherché à optimiser les fonctions et à simplifier la gestion de la mémoire. Une des actions entreprises a été la création d'une interface utilisateur graphique (GUI), facilitant ainsi les interactions avec le système. Nous avons opté pour un modèle de bloc unique pour toute la mémoire, permettant ainsi une standardisation de l'organisation des données. Cependant, certains champs comme `last` et `next`, qui sont utilisés dans des structures de gestion non contiguë, ne sont pas utilisés dans le cas de l'allocation contiguë, ce qui peut être vu comme une limitation.

4.1 Choix des partitions et taille des blocs

Pour le dimensionnement de la mémoire, nous avons pris le parti de considérer le pire des cas, où chaque fichier est limité à un seul bloc. Cette hypothèse permet de garantir que même si chaque fichier nécessite un bloc séparé, cela sera correctement géré sans gaspillage de mémoire. En fonction de cette hypothèse, nous avons utilisé les formules suivantes pour calculer la taille de la mémoire et des partitions :

4.1.1 Calcul du nombre total de blocs en mémoire (BM)

La mémoire doit être composée de plusieurs blocs, principalement pour les données et les métadonnées. Le nombre total de blocs en mémoire est donc calculé comme suit :

$$BM = 1 + BFD + BMD$$

Où :

- (BM) : nombre total de blocs en mémoire,
- (BFD) : nombre de blocs nécessaires pour stocker les fichiers de données,
- (BMD) : nombre de blocs nécessaires pour stocker les métadonnées des fichiers.

4.1.2 Calcul des blocs pour les métadonnées ((BMD))

Les métadonnées de chaque fichier sont stockées dans des enregistrements logiques distincts. Le nombre de blocs nécessaires pour les métadonnées dépend du nombre de blocs de données et du facteur de blocage (FB). La formule utilisée est :

$$BMD = \lceil \frac{BFD}{FB} \rceil$$

Où :

- (BMD) : nombre de blocs pour les métadonnées,
- (BFD) : nombre de blocs de données,
- (FB) : facteur de blocage, qui détermine la taille maximale d'un bloc (plus (FB) est grand, moins de blocs sont nécessaires pour stocker les métadonnées).

4.1.3 Facteur de blocage ((FB))

Le facteur de blocage (FB) est un paramètre essentiel qui détermine la taille maximale des données qu'un bloc peut contenir. Il doit être suffisamment grand pour que la table d'allocation tienne dans un seul bloc. Ce facteur est déterminé en fonction de la taille de la chaîne de caractères `row` :

$$FB \geq \frac{BFD}{\lceil \frac{TR}{4} \rceil}$$

Où :

- (FB) : facteur de blocage (taille du bloc),
- (BFD) : nombre de blocs de données,
- (TR) : taille de la chaîne de caractères `row`.

4.2 Optimisation des fonctions

Dans le cadre du projet, nous avons cherché à optimiser les fonctions de gestion de la mémoire, principalement en réduisant les appels redondants et en centralisant les calculs afin d'éviter les répétitions. Cela permet non seulement de rendre le système plus rapide, mais aussi plus efficace, en réduisant la complexité des processus et en évitant des calculs inutiles.

5

Conclusion

En résumé, les choix effectués pour le dimensionnement et la structure de la mémoire ont permis d'assurer une gestion efficace des données. En minimisant l'utilisation de la mémoire et en garantissant une extensibilité pour des fichiers de tailles variables, nous avons conçu un système qui pourra s'adapter à différents besoins tout en restant optimisé.