esiea

Projet 4 : Simulateur de Système de Gestion de Mémoire

(Niveau avancé)

3A FISE et FISA 2024-2025

Version	Date	Créé/modifié par	Commentaires
1.0	22/09/2024	Radhia GADDOURI	

Table des matières

esiea

1. Objectif du Projet

L'objectif de ce projet est de créer un simulateur de gestion de mémoire dynamique en langage C, illustrant le fonctionnement des opérations d'allocation (malloc) et de libération (free) de mémoire. Ce simulateur permettra aux utilisateurs de visualiser et de manipuler des blocs de mémoire, d'appliquer différents algorithmes d'allocation, et de comprendre les concepts de fragmentation interne et externe.

2. Objectifs Pédagogiques

- **Compréhension de la Gestion de la Mémoire :**
    - Comprendre les concepts fondamentaux de la gestion de la mémoire en C, y compris l'allocation dynamique et la libération de mémoire.
- **Algorithmes d'Allocation de Mémoire :**
    - Apprendre et implémenter différents algorithmes d'allocation de mémoire et comprendre leurs impacts sur les performances du système.
- **Fragmentation de la Mémoire :**
    - Comprendre les concepts de fragmentation interne et externe et comment ils affectent l'efficacité de l'utilisation de la mémoire.
- **Structures de Données Avancées :**
    - Travailler avec des structures de données avancées (listes chaînées) pour gérer la mémoire.
- **Compétences en Débogage et Optimisation :**
    - Développer des compétences en débogage de code complexe et en optimisation des performances.

3. Fonctionnalités Requises

1. **Simulation de l'Allocation Dynamique de Mémoire :**
    - **Description :** Simuler le comportement des fonctions malloc() et free() pour gérer l'allocation et la libération de blocs de mémoire.
    - **Consignes :**

- Implémenter une fonction void* simulerMalloc(size_t taille) qui alloue un bloc de mémoire de la taille demandée.
- Implémenter une fonction void simulerFree(void* ptr) qui libère le bloc de mémoire pointé par ptr.
- Assurez-vous que l'allocation et la libération respectent les contraintes d'alignement de mémoire. **L'alignement de la mémoire** signifie que les blocs de mémoire alloués doivent être alignés à certaines adresses spécifiques, généralement en fonction de la taille des données stockées.

2. **Gestion des Blocs de Mémoire :**
   - **Description :** Gérer les blocs de mémoire alloués et libres à l'aide d'une structure de données appropriée, telle qu'une liste chaînée ou un tableau.
   - **Consignes :**
     - Utiliser une liste chaînée pour représenter les blocs de mémoire, où chaque nœud représente un bloc avec des informations telles que la taille, l'état (alloué ou libre), et un pointeur vers le bloc suivant.
     - Implémenter une structure struct BlocMemoire avec des champs pour la taille, l'état, et un pointeur vers le bloc suivant.
     - Implémenter des fonctions pour ajouter, supprimer, et fusionner des blocs dans la liste.
3. **Implémentation des Algorithmes d'Allocation de Mémoire :**
   - **Description :** Implémenter différents algorithmes d'allocation de mémoire pour gérer l'allocation des blocs (par exemple, First-Fit, Best-Fit, etc.).
   - **Consignes :**
     - Implémentez l'algorithme **First-Fit** qui alloue le premier bloc libre assez grand pour satisfaire la demande.
     - Implémentez l'algorithme **Best-Fit** qui alloue le bloc libre le plus petit capable de satisfaire la demande.

- Implémentez l'algorithme **Worst-Fit** qui alloue le bloc libre le plus grand.
- Permettez à l'utilisateur de choisir l'algorithme d'allocation lors de l'initialisation du simulateur.

4. **Visualisation de l'État de la Mémoire :**
   - **Description :** Visualiser l'état actuel de la mémoire (blocs alloués et libres) dans une interface utilisateur textuelle.
   - **Consignes :**
     - Implémenter une fonction void afficherMemoire() qui affiche la liste des blocs de mémoire et leur état (alloué ou libre).
     - Utiliser des symboles ou des couleurs pour différencier les blocs alloués des blocs libres.
     - Afficher les détails de chaque bloc (adresse de début, taille, état).
5. **Simulation de Scénarios de Fragmentation :**
   - **Description :** Simuler et démontrer des scénarios de fragmentation interne et externe.
   - **Consignes :**
     - Fragmentation **interne** : Illustrer la perte de mémoire due à des blocs alloués qui sont plus grands que nécessaire.
     - Fragmentation **externe** : Illustrer la perte de mémoire due à de multiples blocs libres qui ne sont pas contigus.
     - Implémenter une fonction void simulerFragmentation() qui génère des scénarios de fragmentation pour montrer leurs effets sur l'allocation de mémoire.

4. Fonctionnalités Bonus suggérées

1. **Sauvegarde et Chargement de l'État de la Mémoire :**
   - **Description** : Permettre de sauvegarder l'état actuel de la mémoire et de charger cet état ultérieurement, pour poursuivre la simulation ou pour analyser des scénarios spécifiques.

- **Consignes** :
  - Implémentez une fonction void sauvegarderEtat(const char* fichier) qui écrit l'état des blocs de mémoire dans un fichier.
  - Implémentez une fonction void chargerEtat(const char* fichier) pour lire l'état de la mémoire à partir d'un fichier et restaurer la simulation.

2. Journalisation des Opérations (Logs) :

- **Description** : Ajouter une fonctionnalité de journalisation qui enregistre toutes les opérations d'allocation, de libération, et de fusion des blocs de mémoire dans un fichier de log.
- **Consignes** :
  - Créez une fonction void logOperation(const char* operation) qui écrit les détails des opérations (allocation, libération, fragmentation, etc.) dans un fichier texte.
  - Le fichier de log peut contenir des informations comme l'heure de l'opération, la taille des blocs, et l'état de la mémoire après chaque opération.

3. Visualisation Graphique avec ncurses ou SDL :

- **Description** : Intégrer une interface graphique ou en mode texte avancée pour visualiser l'état de la mémoire de manière plus interactive.
- **Consignes** :
  - Utilisez ncurses pour créer une interface textuelle où les blocs de mémoire sont affichés en utilisant des caractères colorés.
  - Ou bien, utilisez la bibliothèque SDL pour créer une interface graphique où chaque bloc est représenté visuellement comme un rectangle de taille proportionnelle à la mémoire allouée.

4. Gestion des Pages de Mémoire (Pagination) :

- **Description** : Simuler la gestion des pages de mémoire, comme dans un système d'exploitation, où la mémoire est divisée en pages de taille fixe.
- **Consignes** :

- Implémentez un système de pagination, où chaque page a une taille fixe (par exemple, 4 Ko).
- Implémentez des algorithmes de remplacement de pages (comme FIFO, LRU) pour gérer les situations où toutes les pages sont occupées.

5. Analyse Statistique de l'Utilisation de la Mémoire :

- **Description** : Ajouter une fonctionnalité qui analyse et affiche des statistiques sur l'utilisation de la mémoire.
- **Consignes** :
  - Implémentez une fonction void afficherStatistiques() qui calcule et affiche des statistiques telles que :
    - Le pourcentage de mémoire utilisée vs libre.
    - Le nombre de blocs alloués et de blocs libres.
    - La taille moyenne des blocs alloués et libres.
    - Le taux de fragmentation (interne et externe).

6. Support des Commandes Utilisateur en Mode Interactif :

- **Description** : Permettre à l'utilisateur d'interagir directement avec la simulation en entrant des commandes comme "malloc", "free", ou "status" en temps réel.
- **Consignes** :
  - Implémentez une interface interactive où l'utilisateur peut taper des commandes comme :
    - "malloc 1024" : pour allouer un bloc de 1024 octets.
    - "free 0x7fffXXXX" : pour libérer un bloc spécifique.
    - "status" : pour afficher l'état actuel de la mémoire.
  - Ajoutez une fonction pour analyser et exécuter les commandes en direct.

7. Système de Notifications :

- **Description** : Implémenter un système de notifications qui informe l'utilisateur des actions importantes (par exemple, lorsque la mémoire est pleine, ou lorsqu'un bloc a été fusionné).
- **Consignes** :
    - Ajoutez des messages dans la console ou l'interface utilisateur textuelle pour informer l'utilisateur des événements tels que :
        - "Bloc alloué de taille X à l'adresse Y."
        - "Bloc libéré à l'adresse Y."
        - "Mémoire pleine, impossible d'allouer plus de blocs."

5. Contraintes Techniques

- **Structures de Données :**
    - Utiliser des structures comme des listes chaînées pour représenter les blocs de mémoire.
    - Chaque bloc de mémoire doit être représenté par une structure avec des informations pertinentes (taille, état, etc.).
- **Gestion de la Mémoire :**
    - Simuler les fonctions de bas niveau comme malloc() et free() en C.
    - Assurer une gestion efficace de la mémoire allouée pour éviter les fuites de mémoire.
- **Interface Utilisateur :**
    - Créer une interface utilisateur textuelle qui permet aux utilisateurs d'allouer et de libérer de la mémoire.
    - Afficher les états de la mémoire et les messages d'erreur.
- **Compatibilité et Portabilité :**
    - Le programme doit être compatible avec les compilateurs standards C (gcc, clang).
    - Le code doit être portable sur les systèmes d'exploitation principaux (Linux, Windows, macOS).

esiea

6. Plan de Réalisation

1. **Initialisation du Projet :**
    - Créer les fichiers sources (main.c, memoire.h, memoire.c).
    - Définir les structures de données de base (liste chaînée pour les blocs de mémoire, etc.).
2. **Implémentation des Algorithmes d'Allocation :**
    - Implémenter les fonctions de simulation simulerMalloc() et simulerFree().
    - Implémenter les algorithmes d'allocation (First-Fit, Best-Fit, Worst-Fit).
3. **Gestion et Visualisation des Blocs de Mémoire :**
    - Développer les fonctions pour ajouter, supprimer, et fusionner des blocs de mémoire.
    - Implémenter la fonction afficherMemoire() pour visualiser l'état de la mémoire.
4. **Simulation des Scénarios de Fragmentation :**
    - Implémenter les fonctions pour simuler la fragmentation interne et externe.
    - Développer des scénarios de test pour illustrer les différents types de fragmentation.
5. **Tests et Débogage :**
    - Tester chaque fonctionnalité pour s'assurer de son bon fonctionnement.
    - Corriger les erreurs et améliorer la gestion des exceptions.
6. **Optimisation et Documentation :**
    - Optimiser le code pour des performances maximales.
    - Ajouter des commentaires détaillés et rédiger une documentation utilisateur.

7. Critères d'Évaluation

- **Fonctionnalité complète** : Le simulateur doit être entièrement fonctionnel, avec toutes les fonctionnalités requises.
- **Gestion de la mémoire efficace** : Le programme doit gérer correctement la mémoire allouée et libérée, sans fuites de mémoire.
- **Interface utilisateur** : L'interface utilisateur doit être claire et facile à utiliser.
- **Documentation et qualité du code** : Le code doit être bien structuré, avec des commentaires explicatifs et une documentation utilisateur.
- **Originalité et créativité** : Des points bonus seront accordés pour l'implémentation de fonctionnalités bonus ou des améliorations créatives.

8. Livrables

- Code source complet (main.c, memoire.h, memoire.c).
- Fichier README avec instructions de compilation et d'exécution.
- Rapport de projet détaillant l'implémentation, les défis rencontrés, les solutions apportées et la gestion de projet.
- (Optionnel) Démonstration vidéo montrant le fonctionnement du simulateur.

9. Remarques importantes

- Composition des groupes projet et travail collaboratif : Vous pouvez travailler en **binôme ou en groupe de 3 personnes maximum**, **à condition que tous les membres de l'équipe appartiennent à la même classe TD (TD34, TD35, TD36, etc.)**. Assurez-vous bien de bien répartir les tâches et à collaborer de manière efficace.
- **Dépôt Livrable** : Les livrables doivent être déposés exclusivement sur la plateforme **GitLab**. Aucun dépôt ne sera accepté par mail, Teams ou tout autre moyen de communication. Voici le lien, accessible à tous via votre compte 365 : Gitlab.esiea.fr
- **Date limite pour le dépôt des livrables :** Jeudi 19 décembre 2024 à 23h59.
- **Date des soutenances du projet C :** lundi 6 et mardi 7 janvier 2025.