ECOLE SUPERIEURE D'INFORMATIQUE

UNITE PROGRES JUSTICE

Ceci est le document descriptif du projet du sujet n°5 (groupe 38). Au sein de ce document vous trouverez :

-la description générale du sujet
-les classes et les méthodes
-les principaux algorithmes
-les complexités en temps
-etc ...

NB : *les algorithmes ont été implémentés en* ***python*** *avec l'utilisation de certaines bibliothèques pour mieux structurer le code*

Remerciement à **Dr Anasthasie OUATTARA / COMPAORE**

SALGO Abdoul Aziz & SANOU Tolo Ambroise

ECOLE SUPERIEURE D'INFORMATIQUE

Spécification Complète du Programme de Gestion de Mémoire Dynamique

Sujet 5 : Écrire un programme qui gère l'attribution d'espace mémoire à un ensemble de tâches. Chaque tâche a une taille en octets. Elle entre dans le système à un moment donné et une place contiguë en mémoire doit lui être allouée. Lorsque la tâche se termine, elle libère l'espace occupée qui peut être réallouée. On souhaite programmer trois façons de choisir l'emplacement à allouer :

• *Le premier emplacement suffisant*

• *Le plus petit emplacement suffisant*

• *Le plus grand emplacement suffisant*

I. Description Générale

Le programme simule un système de gestion de mémoire dynamique avec une interface graphique. Il permet :

- D'allouer de la mémoire à des tâches selon différentes stratégies.
- De libérer la mémoire occupée par des tâches.
- De défragmenter la mémoire pour regrouper les blocs libres.
- De visualiser l'état de la mémoire en temps réel.
- De gérer des tâches avec une durée de vie limitée.

Le programme est composé de trois classes principales :

- **Tache** : Représente une tâche avec un nom, une taille, une durée de vie et une priorité.

- **GestionnaireMemoire** : Gère l'allocation, la libération et la défragmentation de la mémoire.
- **ApplicationMemoire** : Interface graphique pour interagir avec le système.

II. Classes et Méthodes

1. Classe Tache

- **Attributs** :

- nom : Nom de la tâche.

- taille : Taille de la tâche en unités de mémoire.

- duree : Durée de vie de la tâche en secondes.

- priorite : Priorité de la tâche (0 = basse, 1 = haute).

- **Méthodes** :

__init__(self, nom, taille, duree, priorite=0)

- **Description** : Initialise une tâche.

- **Préconditions** :

  - taille > 0 et duree > 0.

- **Postconditions** :

  - Une instance de Tache est créée avec les attributs nom, taille, duree, et priorite initialisés.

- **Axiomes** :

- Une tâche ne peut pas occuper plus de mémoire que la taille totale de la mémoire.

- **Invariants** :

- La taille et la durée de vie de la tâche restent constantes pendant son existence.

---

2. Classe GestionnaireMemoire

- **Attributs** :

- taille_memoire : Taille totale de la mémoire.

- memoire : Liste représentant la mémoire (chaque élément est soit None pour libre, soit une référence à une tâche pour occupé).

- **Méthodes** :

allouer(self, tache, strategie)

- **Description** : Alloue de la mémoire à une tâche selon une stratégie donnée.

- **Stratégies** :

- premier_emplacement : Alloue la première zone libre suffisamment grande.

- plus_petit_emplacement : Alloue la plus petite zone libre suffisamment grande.

- plus_grand_emplacement : Alloue la plus grande zone libre suffisamment grande.

- **Préconditions** :

- tache.taille <= taille_memoire.

- La stratégie est l'une des suivantes : premier_emplacement, plus_petit_emplacement, plus_grand_emplacement.

- **Postconditions** :

- Si un bloc libre suffisamment grand est trouvé, la tâche est allouée dans la mémoire.

- Sinon, la méthode retourne False.

- **Axiomes** :

- La mémoire est allouée de manière contiguë.

- **Invariants** :

- La mémoire reste cohérente après allocation (pas de chevauchement de tâches).

- **Complexité** : O(n), où n est la taille de la mémoire.

obtenir_blocs_libres(self)

- **Description** : Retourne une liste de blocs libres dans la mémoire.

- **Préconditions** :

- La mémoire est représentée par une liste de taille fixe.

- **Postconditions** :

- Une liste de tuples (debut, taille) est retournée, représentant les blocs libres.

- **Axiomes** :

- Les blocs libres sont des séquences contiguës de None.

- **Invariants** :

- La somme des tailles des blocs libres est égale à la mémoire totale moins la mémoire occupée.

- **Complexité** : O(n), où n est la taille de la mémoire.

liberer_memoire(self, tache)

- **Description** : Libère la mémoire occupée par une tâche.

- **Préconditions** :

  - La tâche doit être présente dans la mémoire.

- **Postconditions** :

  - Tous les emplacements mémoire occupés par la tâche sont marqués comme libres (None).

- **Axiomes** :

  - La mémoire libérée est marquée comme None.

- **Invariants** :

  - La mémoire reste cohérente après libération.

- **Complexité** : O(n), où n est la taille de la mémoire.

defragmenter(self)

- **Description** : Défragmente la mémoire en regroupant les blocs libres.

- **Préconditions** :

  - La mémoire contient des blocs libres et occupés.

- **Postconditions** :

  - Les blocs libres sont regroupés en un seul bloc à la fin de la mémoire.

- **Axiomes** :

  - Les tâches sont déplacées vers le début de la mémoire.

- **Invariants** :

  - La mémoire reste cohérente après défragmentation.

- **Complexité**: O(n), où n est la taille de la mémoire.

3. Classe ApplicationMemoire

- **Attributs** :

  - fenetre : Fenêtre principale de l'application.

  - gestionnaire : Instance de GestionnaireMemoire.

  - taches : Liste des tâches actives.

  - couleurs : Dictionnaire associant des couleurs aux tâches.

  - zone_affichage : Zone graphique pour afficher la mémoire.

  - zone_logs : Zone de texte pour afficher les logs.

- **Méthodes** :

ajouter_tache(self)

- **Description** : Ajoute une tâche à la mémoire.

- **Préconditions** :

  - Les champs nom, taille, et duree doivent être valides.

  - taille > 0 et duree > 0.

- **Postconditions** :

  - Si l'allocation réussit, la tâche est ajoutée à la liste des tâches et à la mémoire.

- Sinon, un message d'erreur est affiché.

- **Axiomes** :
  - La tâche est allouée selon la stratégie choisie.
- **Invariants**:
  - La mémoire est mise à jour après l'ajout.
- **Complexité** : O(n), où n est la taille de la mémoire.

supprimer_tache(self)

- **Description** : Supprime la première tâche de la liste.
- **Préconditions** :
  - La liste des tâches n'est pas vide.
- **Postconditions** :
  - La tâche est supprimée de la liste et sa mémoire est libérée.
- **Axiomes** :
  - La mémoire est libérée après suppression.
- **Invariants :**
  - La mémoire est mise à jour après suppression.
- **Complexité** : O(n), où n est la taille de la mémoire.

defragmenter(self)

- **Description** : Défragmente la mémoire.
- **Préconditions** :
  - La mémoire contient des blocs libres.
- **Postconditions** :

- Les blocs libres sont regroupés en un seul bloc à la fin de la mémoire.

- **Axiomes** :

  - Les blocs libres sont regroupés.

- **Invariants** :

  - La mémoire est mise à jour après défragmentation.

- **Complexité** : O(n), où n est la taille de la mémoire.

surveiller_taches(self)

- **Description** : Surveille la durée de vie des tâches et les supprime automatiquement.

- **Préconditions** :

  - La durée de vie des tâches est décrémentée chaque seconde.

- **Postconditions** :

  - Les tâches expirées sont supprimées de la mémoire.

- **Axiomes** :

  - Les tâches expirées sont supprimées.

- **Invariants** :

  - La mémoire est mise à jour après suppression.

- **Complexité** : O(m), où m est le nombre de tâches.

mettre_a_jour_affichage(self)

- **Description** : Met à jour l'affichage graphique de la mémoire.

- **Préconditions** :

  - La mémoire est représentée sous forme de diagramme.

- **Postconditions** :
  - L'affichage reflète l'état actuel de la mémoire.
- **Axiomes** :
  - Les tâches sont affichées avec des couleurs distinctes.
- **Invariants** :
  - L'affichage reflète toujours l'état actuel de la mémoire.
- **Complexité** : O(n), où n est la taille de la mémoire.

III. Conditions Générales

- La taille de la mémoire doit être suffisante pour contenir les tâches.
- Les tâches doivent avoir une durée de vie et une taille strictement positives.
- Les stratégies d'allocation doivent être valides.

Axiomes

- La mémoire est allouée de manière contiguë.
- Les tâches ne peuvent pas chevaucher d'autres tâches.

Invariants

- La mémoire reste cohérente après chaque opération (allocation, libération, défragmentation).
- L'affichage reflète toujours l'état actuel de la mémoire.

6. Complexités

Méthode	Meilleur cas	Pire cas
Tache.__init__	O(1)	O(1)
GestionnaireMemoire.allouer	O(1)	O(n)
GestionnaireMemoire.obtenir_blocs_libres	O(1)	O(n)
GestionnaireMemoire.liberer_memoire	O(1)	O(n)
GestionnaireMemoire.defragmenter	O(1)	O(n)
ApplicationMemoire.ajouter_tache	O(1)	O(n)
ApplicationMemoire.supprimer_tache	O(1)	O(n)
ApplicationMemoire.defragmenter	O(1)	O(n)
ApplicationMemoire.surveiller_taches	O(1)	O(n)
ApplicationMemoire.mettre_a_jour_affichage	O(1)	O(n)

Remarque :

Les complexités des méthodes varient en fonction du nombre de blocs mémoire (n) et du nombre de tâches actives (m). Les meilleurs cas sont souvent constants, tandis que les pires cas sont linéaires par rapport à n ou m. Cela rend l'implémentation efficace pour des simulations de taille modérée, mais nécessite une attention particulière pour les cas extrêmes où n ou m sont très grands.

7. Interactions entre Composants

- L'interface graphique (ApplicationMemoire) interagit avec le gestionnaire de mémoire (GestionnaireMemoire) pour allouer, libérer et défragmenter la mémoire.

- Les tâches (Tache) sont créées par l'utilisateur via l'interface graphique et gérées par le gestionnaire de mémoire.

- L'affichage est mis à jour en temps réel pour refléter l'état de la mémoire.

Récapitulatif de des différentes fonctions ou méthodes qu'implémente le sujet

1. Fonction allouer de taches

```
Fonction allouer(tache, strategie)
    blocs_libres = obtenir_blocs_libres()
    bloc_trouve = Null

    Si strategie == "premier_emplacement"
        Pour chaque bloc dans blocs_libres:
            Si taille(bloc) >= taille(tache):
                bloc_trouve = bloc
                Sortir de la boucle

    Sinon si strategie == "plus_petit_emplacement":
        plus_petit_bloc = Null
        Pour chaque bloc dans blocs_libres:
            Si taille(bloc) >= taille(tache):
                Si plus_petit_bloc == None ou taille(bloc) <
taille(plus_petit_bloc):
                    plus_petit_bloc = bloc
        bloc_trouve = plus_petit_bloc

   Sinon si strategie == "plus_grand_emplacement":
     plus_grand_bloc = None
      Pour chaque bloc dans blocs_libres:
         Si taille(bloc) >= taille(tache):
           Si plus_grand_bloc == None ou taille(bloc)>
taille(plus_grand_bloc):
                    plus_grand_bloc = bloc
        bloc_trouve = plus_grand_bloc

     Si bloc_trouve != Null
        allouer_memoire(bloc_trouve, tache)
        Retourner Vrai
     Sinon:
        Retourner Faux
```

1. Fonction pour obtenir des blocs libres

```
Fonction obtenir_blocs_libres():
    blocs_libres = []
    debut = Null
    compteur = 0

    Pour chaque case dans mémoire
        Si case est libre
            Si debut == Null
                debut = indice(case)
            compteur = compteur + 1
        Sinon:
          Si debut != Null:
           blocs_libres.ajouter(debut, compteur)
                debut = Null
                compteur = 0

    Si debut != Null:
        blocs_libres.ajouter((debut, compteur))

    Retourner blocs_libres
```

1. Fonction pour libérer la memoire Fonction liberer_memoire(tache): Pour chaque case dans mémoire Si case contient tache case = Null	**1. Fonction pour defragmenter la memoire** Fonction defragmenter(): nouvelle_memoire = creer_memoire_vide() Pour chaque case dans mémoire Si case contient tache nouvelle_memoire.ajouter(tache) mémoire = nouvelle_memoire
1. Fonction pour ajouter une tache Fonction ajouter_tache() valeurs = recuperer_valeurs_champs() Si valeurs_valides(valeurs): tache = creer_tache(valeurs) Si allouer(tache, strategie): liste_taches.ajouter(tache) associer_couleur(tache) journaliser("Tâche ajoutée") Sinon: journaliser("Échec de l'allocation") mettre_a_jour_affichage()	**1. Fonction pour supprimer une tache** Fonction supprimer_tache(): Si liste_taches non vide: tache = liste_taches.supprimer_premiere() liberer_memoire(tache) journaliser("Tâche supprimée") mettre_a_jour_affichage() Fonction defragmenter(): defragmenter() journaliser("Défragmentation effectuée")
1. Fonction pour surveiller la durée de vie des taches dans la memoire Fonction surveiller_taches(): Tant que en_cours: Pour chaque tache dans liste_taches: tache.duree_vie = tache.duree_vie - 1 Si tache.duree_vie == 0: supprimer_tache(tache) journaliser("Tâche supprimée (durée de vie écoulée)") mettre_a_jour_affichage() attendre(1 seconde)	**1. Fonction pour mettre à jour l'affichage** Fonction mettre_a_jour_affichage(): effacer_affichage() Pour chaque tache dans liste_taches: afficher_tache(tache) afficher_blocs_libres() mettre_a_jour_statistiques()