Projet de Gestion de Mémoire Dynamique

Simulation d'allocation mémoire avec stratégies adaptatives

OUEDRAOGO Samiratou DAH Poutieromala

Du 17 février au 6 mars 2025

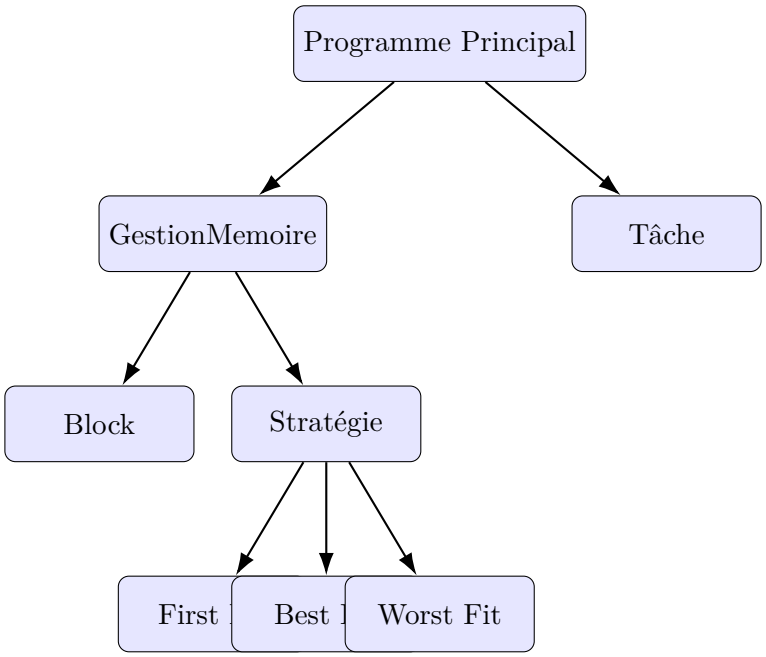
Sujet 5

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 occupé qui peut être réalloué.

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.

1 Architecture Technique



— **Programme Principal** : Gère l'exécution globale.
— **GestionMémoire** : Responsable de l'allocation et de la libération de mémoire.
— **Tâche** : Représente les processus à exécuter.
— **Block** : Structure contenant les espaces mémoire disponibles.
— **Stratégie** : Détermine la méthode d'allocation (*First Fit*, *Best Fit*, *Worst Fit*).

2 Spécification de l'algorithme

Entrées du Système

Taille de la tâche (en octets)
— Durée d'exécution de la tâche (en secondes)
— Stratégie d'allocation choisie (premier, petit, ou grand)

Sorties du Système

Confirmation d'allocation de la mémoire
— Message d'erreur si aucun bloc disponible ne satisfait la demande
— Affichage dynamique de l'état de la mémoire

3 Technologie et Environnement

Catégorie	Technologies
Langage	Python 3.10+
Bibliothèques	rich, threading, random, os, time
Environnement	Windows / Linux

4 Principales Structures de Données

Classe Block

```
classe Block:
        ATTRIBUTS:
        Initialise un bloc de memoire.

        adresse: Position memoire du bloc.
        taille: Taille du bloc en octets.
        libre: Statut du bloc (libre ou occupe).
        tache: Identifiant de la tache associee.
        """
        METHODES:
        INITIALISER(adresse: ENTIER,taille : ENTIER, libre : BOOLEEN,
           tache : CHAINE)
        adresse = adresse
        taille = taille
        libre = libre
        tache = tache

       AFFICHER()-> str:
        """
        Retourne une representation textuelle du bloc.

        retourne: Une chaine de caracteres decrivant le bloc.
        """
       RETOURNER Adresse:  + CONVERTIR_TEXTE(adresse) +
                      Taille:  + CONVERTIR_TEXTE(taille) +
                      Libre: + CONVERTIR_TEXTE(libre) +
                      Tache:  + CONVERTIR_TEXTE(tache)
```

Classe Strategie

```
classe Strategie:
    ATTRIBUTS:
    memoire: Liste des blocs m moire disponibles.
    METHODES:

    Fonction premier_emplacement_suffisant(taille : ENTIER) -> Block:
        taille: Taille requise
        Pour chaque block dans la memoire:
            si block.libre et block.taille >= taille:
                retourne block
        retourne None

    Foncton plus_petit_emplacement_suffisant( taille:) -> Block:
        taille: Taille requise
        retourne: Le plus petit bloc suffisant trouv , ou None si aucun
            ne convient
        meilleur_block = None
        Pour chaque block dans la memoire:
            si block.libre et block.taille >= taille:
                si meilleur_block is None or block.taille <
                   meilleur_block.taille:
                    meilleur_block = block
        retourne meilleur_block

    Fonction plus_grand_emplacement_suffisant(taille) -> Block:
        taille: Taille requise
        retourne: Le plus grand bloc suffisant trouve, ou None si aucun
           ne convient.
        meilleur_block = None
        Pour chaque block dans la memoire:
            si block.libre et block.taille >= taille:
                si meilleur_block is None ou block.taille >
                   meilleur_block.taille:
                    meilleur_block = block
        retourne meilleur_block
```

Classe Tache

```
classe Tache:
         ATTRIBUTS:
        nom : CHAINE            // Nom de la tache
        taille : ENTIER         // Taille de la tache en octets
        temps : ENTIER          // Duree
        strategie : CHAINE      // Strategie

        Initialise une tache.
        nom = nom
        taille = taille
        strategie = strategie
        temps = temps

    Procedure remplir():
        lire les informations de la tache et remplir
        AFFICHER (Strategie utilisee [1:premier, 2:petit, 3:grand] )
        lire strat
```

```
        si strat == 1:
            strategie = premier
        sinon si strat == 2:
            strategie = petit
        sinon si strat == 3:
            strategie = grand
        sinon:
            Afficher(Strategie inconnue, utilisation de premier par
               defaut.)
            strategie = premier

        Afficher(** Tache cree avec succes **)
        Afficher(les informations de la tache cree)
        Appuyez sur Entree pour continuer

    Fonction convertir_en_chaine() -> str:
        """
        Retourne une representation textuelle de la tache.
        retourne (Nom, Taille , Duree, Strat gie)
```

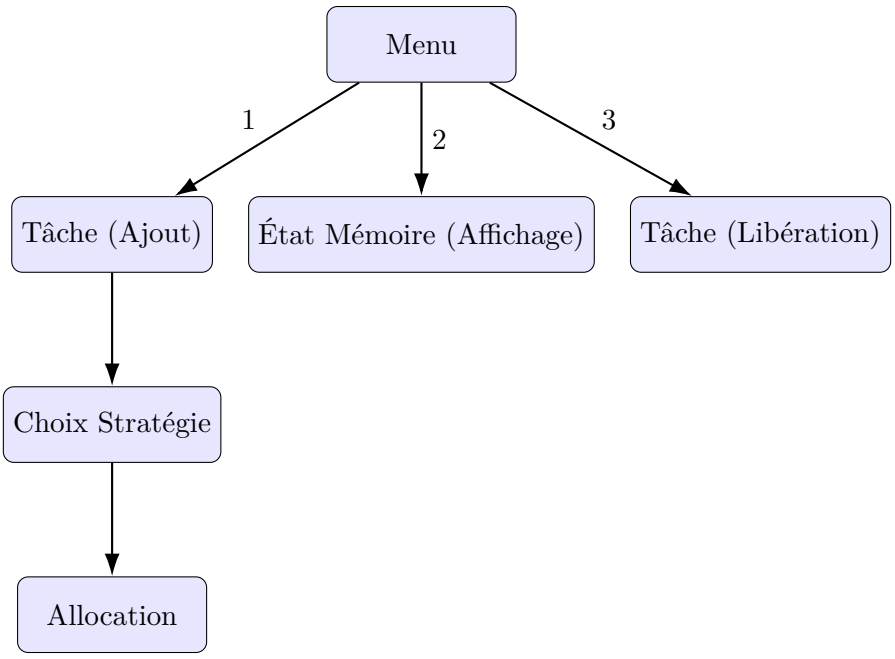
Classe GestionMemoire

```
classe  GestionMemoire:
        ATTRIBUTS:
         tailleMemoire : ENTIER         // Taille de la memoire
        memoire : LISTE DE Block       // Liste des blocs
        algo : Strategie
        verrou : OBJET Verrou          // Verrou pour la synchronisation
        stop : BOOLEEN                 // arret du thread
        compteur : THREAD

        METHODES:
        Procedure INITIALISER(tailleMemoire : ENTIER)
         tailleMemoire <-tailleMemoire)
         memoire <- LISTE_VIDE)
         creerBlock()
         algo <-CREER Strategie(memoire)
         verrou <- CREER Verrou()
         stop <- FAUX
         compteur <-CREER Thread(decrementeTemps())
         DEMARRER_THREAD(SELF.compteur)

        Procedure creerBlock()
        adresse<-0
        tailleRestante<-tailleMemoire
            TANT QUE tailleRestante > 0 FAIRE
                tailleBlock <-NOMBRE_ALEATOIRE(100, 500)
                SI tailleBlock < tailleRestante ALORS
                    AJOUTER_BLOCK(adresse, tailleBlock, VRAI, NULL)
                    adresse<-adresse + tailleBlock
                    tailleRestante <- tailleRestante - tailleBlock
                SINON
                    AJOUTER_BLOCK(adresse, tailleRestante, VRAI, NULL)
                    tailleRestante <- 0
                FIN SI
            FIN TANT QUE

```

```
36      Procedure AFFICHERMemoire()-> str:
37           Etat actuel de la memoire
38              EFFACER_ECRAN()
39              TABLEAU memoireTable<-CREER Table()
40
41              AJOUTER_COLONNES(memoireTable, Adresse, Taille, Statut,
                    Tache)
42              POUR CHAQUE bloc dans la memoire FAIRE
43                  SI bloc.tache !=NULL
44                       tach <-bloc.tache
45                  SINON None
46                  SI bloc.libre = VRAI
47                       statut <- Libre
48                  SINON Occupe
49                  AJOUTER_LIGNE(memoireTable, bloc.adresse, bloc.taille,
                        statut, tach)
50              FIN POUR
51              AFFICHER_TABLE(m moireTable)
52              AFFICHER(Actualiser [Entree] ou Retour Menu [0])
53              LIRE actualiser
54              SI actualiser = "" ALORS
55                  afficherMemoire()
56              FIN SI
57
58      PROCEDURE liberer(nom_tache : CHAINE) : BOOLEEN
59          liber<- FAUX
60              POUR CHAQUE bloc DANS la memoire FAIRE
61                  SI bloc.tache !=NULL ET bloc.tache.nom = nom_tache ALORS
62                      bloc.libre <- VRAI
63                      bloc.tache <-NULL
64                      AFFICHER(La tache , nom_tache,  a ete liberee.)
65                      liber<- VRAI
66                  FIN SI
67              FIN POUR
68
69              SI liber = FAUX ALORS
70                  AFFICHER(Erreur: Tache , nom_tache,  non trouvee.)
71              FIN SI
72              RETOURNER liber
73      PROCEDURE allouer(tache : Tache) : CHAINE
74           ACQUERIR_VERROU(verrou)
75              SI tache.strategie = premier ALORS
76                  bloc <- algo.premier_emplacement_suffisant(tache.taille)
77              SINON SI tache.strategie = petit ALORS
78                  bloc <- algo.plus_petit_emplacement_suffisant(tache.
                        taille)
79              SINON SI tache.strategie = grand ALORS
80                  bloc<- algo.plus_grand_emplacement_suffisant(tache.
                        taille)
81              SINON
82                  LIBERER_VERROU(verrou)
83                  RETOURNER Strategie non reconnue
84              FIN SI
85
86              SI bloc = NULL ALORS
87                  LIBERER_VERROU(verrou)
88                  RETOURNER Pas de place disponible
89              SINON
```

```
                bloc.libre<- FAUX
                bloc.tache<- tache
                LIBERER_VERROU(verrou)
                RETOURNER Tache en cours
            FIN SI
        RETOURNER Adresse:  + CONVERTIR_TEXTE(adresse) +
                        Taille:  + CONVERTIR_TEXTE(taille) +
                        Libre:  + CONVERTIR_TEXTE(libre) +
                        Tache:  + CONVERTIR_TEXTE(tache)
    PROCEDURE decrementeTemps()
        // Reduit progressivement le temps des t ches
            TANT QUE stop = FAUX FAIRE
                ATTENDRE(1) // Pause
                ACQUERIR_VERROU(verrou)
                POUR CHAQUE bloc DANS la memoire FAIRE
                    SI bloc.libre = FAUX ET bloc.tache !=NULL ALORS
                        bloc.tache.temps<-bloc.tache.temps - 1
                        SI bloc.tache.temps < 0 ALORS
                            liberer(bloc.tache.nom)
                        FIN SI
                    FIN SI
                FIN POUR
                LIBERER_VERROU(verrou)
            FIN TANT QUE
```

5 Spécifications Fonctionnelles



- **Menu** : Point d'entrée principal pour l'utilisateur.
- **Ajout** : Permet d'ajouter une tâche dans le système.
- **État Mémoire (Affichage)** : Affiche l'état actuel de la mémoire.
- **Libération** : Permet de libérer une tâche terminée et réallouer l'espace mémoire.
- **Choix Stratégie** : Sélectionne la stratégie d'allocation de mémoire.
- **Allocation** : Procède à l'allocation de mémoire selon la stratégie choisie.

6 Algorithmes d'Allocation

Stratégie	Mécanisme	Limitation	Complexité
First Fit	Premier bloc suffisant	Fragmentation potentielle	O(n)
Best Fit	Plus petit bloc admissible	Coût de recherche	O(n)
Worst Fit	Plus grand bloc disponible	Sous-utilisation mémoire	O(n)

7 Analyse Critique

Points Forts

— Interface utilisateur intuitive grâce à `rich`.
— Gestion asynchrone efficace des tâches.
— Extensibilité des stratégies d'allocation.

Contraintes

— Fragmentation : Absence de mécanisme de compaction.
— Portabilité : Utilisation d'`os.system` pour le nettoyage d'écran.
— Évolutivité : Limité aux simulations de petite ou moyenne taille.