Guide du développeur

Simulation de la gestion de la mémoire

Présenté par :

- RIGHI Racim
- TRABELSI Lydia

2020/2021

1. Table des matières

2. INTRODUCTION

Ce document a pour but d'aider les développeurs qui souhaiteraient améliorer et ajouter des fonctionnalités à ce programme, il décrira tous les algorithmes utilisés, ainsi que les structures de données, fonctions et librairies qui ont servi au développement, et comment utiliser les dépendances.

3. DESCRIPTION

Ce projet implémente une simulation de la gestion de la mémoire d'un système d'exploitation, il propose trois modes d'exécutions (interactif, ligne de commande, batch).

Le but est de fournir des fonctions similaires aux fonctions *malloc*, *calloc* et *free* de la librairie système standard, et d'implémenter les algorithmes de gestion de mémoire connus.

4. Environnement de développement

Le projet a été développé sur une machine sous Pop.OS 20.04 LTS. Avec le logiciel *Visual Studio Code* qui permet l'ajout d'extensions qui permettent d'effectuer la vérification syntaxique au fur et à mesure qu'on code, en plus des extensions de *linting* qui permettent d'unifier le style de code lors de la collaboration.

Le projet est stocké dans un dépôt G*ithub*, ce qui donne la possibilité de voir toutes les étapes du développement et de revenir vers d'anciennes versions, et de collaborer à distance.

Lien : https://github.com/RacimRgh/OS-memory-simulation

- **Dépendances :**
    - Cmake : La compilation nécessite une version de **cmake** >= 3.13
    - Makefile
    - Curses : la librairie ncurses.h a été utilisé pour l'interface graphique du mode interactif de l'application.
    - Doxygen : pour générer la documentation HTML.

5. Utilisation

● Arborescence de fichiers

La racine contient les documents essentiels, à savoir le point d'entrée du programme ***main.c***, le fichier de compilation ***CmakeLists.txt***, et les fichiers qui concernent le dépôt github (documentation ***README.MD***, de la licence ***LICENSE***, et le fichier de configuration de git ***.gitignore)***

Le dossier ***src*** contient les codes sources qui implémentent les en-têtes du dossier ***headers***.

Le dossier ***config_files*** contient les fichiers qui vont servir à l'exécution du programme (un fichier qui contient les partitions de la mémoire ***memory.txt***, et un qui contient les processus ***process.txt***, et le fichier de configuration ***config.txt***).

Le dossier ***build*** va contenir les fichiers temporaires (cache) générés par **c*make***.

● Compilation

Un fichier ***CMakeLists.txt*** est situé dans le dossier racine du projet, il permet de générer un fichier ***Makefile*** qui servira à compiler le programme, l'exécuter et générer la documentation ***doxygen***.

Syntaxe:

$ cmake -B build/ -G "Unix Makefiles"

$ make -C build/

● Exécution

Le programme fournit quatre différents modes d'exécution

- Mode test: **$ ./Mémoire -t**
- Mode interactif : **$ ./Mémoire -i**
- Mode ligne de commande : **$ ./Mémoire -m <taille_memoire> -p <partition_1> <fit_1>… <partition_n> <fit_n>**
  où <fit> == 1: first fit, 2: bestfit, 3: worst fit
- Mode fichier de configuration : **$ ./Mémoire -f <nom_du_fichier>**

● Documentation

Le fichier ***makefile*** contient aussi une entrée pour générer la documentation par le biais de ***doxygen*** dans le dossier ***docs***.

Syntaxe:

$ make -C build/ docs

6. Structures de données

Le programme utilise principalement 4 structures de données présentées dans la documentation

- Process.
- Partition.
- Memory.
- Proc_Queue.

7. Algorithmes de gestion de mémoire

Pour simuler la mémoire d'un système d'exploitation, on a utilisé la technique de partitionnement de la mémoire, chaque partition ayant une adresse de début, taille, état (occupé ou libre) et dans certains cas, un processus qui l'occupe.

L'allocation d'un processus ou d'une zone mémoire s'effectue l'un des modes suivants :

- **First fit :** La première partition libre et assez grande pour accueillir le processus/zone sera choisie.
- **Best fit :** La partition choisie sera celle qui laissera le moins de résidu possible après l'allocation.
- **Worst fit :** La partition choisie sera celle qui laissera le plus de résidu après l'allocation, ce qui nous permettra d'y allouer d'autres processus/zones après.

Après chaque allocation, une nouvelle partition est créée selon la taille du résidu avec la fonction *new_partition()* (documentation ci-dessous).

Pour éviter d'avoir une mémoire trop fragmentée, la mémoire est optimisée après chaque allocation, ou à la demande de l'utilisateur. Le principe de cette réallocation est de déplacer les partitions occupées vers le début de la mémoire, et d'additionner les partitions fragmentées, pour en créer une grande partition à la fin de la mémoire, ce nouveau bloc sera à son tour partitionné en un groupe de partitions de tailles similaires (par défaut 1000 octets).

● Fonctions

Pour initialiser la mémoire, deux fonctions peuvent être utilisées :

- **initMemory() :** En lui fournissant la taille du bloc en octets, elle crée plusieurs partitions de 500 octets. Ces dernières pourront être utilisées plus tard pour exécuter des processus et allouer des espaces.
- **initMemoryFile() :** Cette fonction agit de la même manière que la précédente, mais les partitions allouées sont issues du fichier *memory.txt* du dossier *config_files/*

- **myAlloc() :** Alloue un espace d'une taille donnée dans la mémoire, suivant la fonction d'allocation choisie.
- ***myAllocProc() :*** Similaire à la fonction *myAlloc()*, cependant elle alloue la mémoire pour un processus qui commence son exécution et libère l'espace à la fin de son temps.
- ***myFree():*** Fonction qui désalloue une partition de la mémoire. Notre projet ayant intégré le principe de processus, elle a été remplacée par la fonction *myFreeProc()* qui, en utilisant des *threads* vérifie l'état de la mémoire, et libère les partitions contenant des processus qui ont fini leur exécution.
- **freeMemory() :** Libère toute la mémoire allouée avec *init()*
- ***myRealloc() :*** Réorganise la mémoire de façon à rassembler les partitions vides pour permettre à des processus d'être servis.
- ***show_memory() :*** Affiche la mémoire sous forme graphique grâce à la librairie n*curses*
- ***« file_proc.h » :*** Fichier qui décrit une file de processus à allouer dans la mémoire et exécuter.

8. Librairies

● Utilisation de cmake

Le fichier ***CMakeList.txt*** permet de générer un makefile complet qui sert à compiler le programme, générer la documentation et automatiser la recherche et liaison des librairies utilisées.

9. Axes d'amélioration

- Automatiser l'exécution des processus
- Ajouter un compteur selon le temps d'exécution
- Définir d'autres systèmes de gestion de la mémoire (pagination,)