Projet: Réalisation d'un simulateur de CPU

Un processeur est l'unité centrale de traitement (*Central Processing Unit*, CPU) chargée d'exécuter les instructions d'un programme. Il coordonne l'accès à la mémoire, effectue des calculs et prend en charge le contrôle du flux d'exécution des programmes. Son fonctionnement repose sur un cycle d'instruction, dans lequel il récupère une instruction en mémoire, la décode, puis l'exécute en manipulant les registres et la mémoire. Le processeur est constitué de plusieurs éléments fondamentaux :

- **Les registres** : ce sont des emplacements mémoire internes au processeur permettant de stocker temporairement des valeurs nécessaires aux calculs et à l'exécution des instructions. Parmi eux, on trouve les registres généraux (`AX, BX, CX, DX`), les registres d'indexation et les registres de contrôle (`IP`, `SP`, `FLAGS`).
- **L'unité de contrôle** : elle orchestre l'exécution des instructions en gérant la récupération des instructions, leur décodage et leur exécution.
- **L'unité arithmétique et logique (ALU)** : elle est responsable des opérations mathématiques et logiques (addition, soustraction, comparaisons, etc.).
- **L'accès à la mémoire** : le processeur communique avec la mémoire vive (RAM) pour charger les instructions et stocker temporairement des données. Il utilise différentes stratégies d'adressage pour accéder aux données de manière efficace.

Dans le cadre de ce projet, nous allons implémenter une version simplifiée de ce fonctionnement, en nous concentrant sur un ensemble réduit de registres et de modes d'adressage, tout en conservant les principes fondamentaux du traitement des instructions et de la gestion de la mémoire.

Le projet sera à rendre sur moodle quelques jours avant les soutenances (date à préciser), et les soutenances auront lieu pendant la dernière séance de TD/TP. Sur moodle, vous trouverez un document récapitulant ce qui est attendu dans votre rendu. Pour les étudiants qui ne peuvent être présents le jour de la soutenance, vous devez impérativement contacter votre chargé de TD afin de lui fournir un justificatif et prévoir avec elle/lui une autre date de soutenance (au plus près de la semaine de rendu). En effet, les soutenances sont obligatoires, et toute absence injustifiée sera sanctionnée par la note de zéro. Enfin, nous vous rappelons que la note de ce projet fait partie de la note du module, mais ne fait partie du contrôle continu. Plus précisément, elle ne peut pas être compensée avec la règle du max et sera conservée en seconde session. **Attention :** Ce projet doit obligatoirement être réalisé en binôme. Si vous ne trouver pas de binôme, vous devez envoyer un mail à votre chargé de TD pour qu'il vous en trouve un.

Exercice 1 – Implémentation d'une table de hachage générique

Dans cet exercice, nous allons implémenter une table de hachage générique en C. L'objectif est de créer une structure permettant d'associer des clés (chaînes de caractères) à des valeurs de type générique `void*`. Cette flexibilité nous permettra de concevoir un dictionnaire associant des clés à n'importe quel type de structure. La table de hachage utilisera une taille fixe de `TABLE_SIZE = 128` et appliquera un mécanisme de sondage/probing linéaire en cas de collision. Ainsi on utilisera les structures suivantes :

```
1 typedef struct hashentry{
2     char* key;
3     void* value;
4 } HashEntry;
```

```c
typedef struct hashmap {
    int size;
    HashEntry* table;
} HashMap;
```

Remarque : quand on supprime un élément dans une table de hachage avec adressage ouvert, on peut utiliser une `TOMBSTONE` pour éviter d'interrompre les chaînes de probing. Une `TOMBSTONE` est un élément spécial, distinct de la valeur nulle, qui remplace tout élément supprimé. Contrairement à NULL, elle permet de continuer la recherche et d'assurer l'accessibilité des éléments. Bien sûr, une `TOMBSTONE` doit être supprimée si un nouvel élément souhaite être inséré à sa position. Ici, on utilisera par exemple un `TOMBSTONE` définie par :

```c
#define TOMBSTONE ((void*)-1)
```

Q 1.1 Implémentez une fonction `unsigned long simple_hash(const char *str)` permettant de convertir une chaîne de caractères en un indice dans la table de hachage.

Q 1.2 Écrivez une fonction `HashMap *hashmap_create()` permettant d'allouer dynamiquement une table de hachage et d'initialiser ses cases à `NULL`.

Q 1.3 Implémentez la fonction `int hashmap_insert(HashMap *map, const char *key, void *value)` permettant d'insérer un élément dans la table de hachage.

Q 1.4 Réalisez une fonction `void *hashmap_get(HashMap *map, const char *key)` permettant de récupérer un élément à partir de sa clé.

Q 1.5 Ajoutez une fonction `int hashmap_remove(HashMap *map, const char *key)` permettant de supprimer un élément de la table de hachage tout en assurant la continuité du sondage linéaire.

Q 1.6 Implémentez une fonction `void hashmap_destroy(HashMap *map)` permettant de libérer toute la mémoire allouée à la table de hachage.

Exercice 2 – Gestion dynamique de la mémoire

Dans cet exercice, nous allons implémenter un gestionnaire de mémoire simple en C. Le gestionnaire de mémoire divise l'espace mémoire en **segments** de taille variable. Chaque segment peut être alloué à un programme ou libéré pour être réutilisé. L'allocation et la libération de segments suivent des étapes bien définies. Dans notre cas, lorsqu'un segment est alloué, il est retiré de la liste des segments libres, puis il est ajouté à une **table de hachage** (correspondant aux segments alloués) en lui associant un nom unique. Lorsqu'un segment est libéré, il est supprimé de la table de hachage et réinséré dans la liste des segments libres (en le fusionnant avec d'autres segments libres adjacents s'il en existe, afin de limiter la fragmentation de la mémoire).

L'implémentation du gestionnaire reposera ainsi sur deux structures essentielles : la structure `Segment` représentant un bloc de mémoire (qui peut être libre ou alloué) et la structure `MemoryHandler` représentant l'état global du gestionnaire de mémoire.

```c
typedef struct segment {
    int start; // Position de debut (adresse) du segment dans la memoire
    int size; // Taille du segment en unites de memoire
    struct Segment *next; // Pointeur vers le segment suivant dans la liste chainee
} Segment;

```

```
7  typedef struct memoryHandler {
8      void **memory; // Tableau de pointeurs vers la memoire allouee
9      int total_size; // Taille totale de la memoire geree
10     Segment *free_list; // Liste chainee des segments de memoire libres
11     HashMap *allocated; // Table de hachage (nom, segment)
12 } MemoryHandler;
```

Lors de l'initialisation du gestionnaire, la mémoire est constituée d'un **unique segment libre** couvrant la totalité de l'espace mémoire disponible. Par exemple, pour une mémoire initiale de taille `1024`, la liste des segments libres contient un unique segment : `[start=0, size=1024]`.

Q 2.1 Implémentez une fonction `MemoryHandler *memory_init(int size)` permettant d'initialiser le gestionnaire de mémoire.
On souhaite à présent écrire une fonction qui permet de savoir s'il est possible d'allouer un segment commençant à une position donnée. Cette allocation est possible s'il existe un segment libre contenant le segment à allouer, c-à-d s'il existe un segment libre `seg` tel que `seg->start <= start` et `seg->end >= start+size`, où `start` est l'adresse du segment que l'on souhaite allouer et `size` est sa taille.

Q 2.2 Implémentez une fonction `find_free_segment(MemoryHandler* handler, int start, int size, Segment** prev)` qui retourne un tel segment libre s'il en existe, l'argument `prev` permettant alors de récupérer un pointeur sur le segment libre qui le précède dans la liste chaînée `free_list`. Cette fonction retourne `NULL` dans le cas contraire.

On s'intéresse maintenant à l'allocation de segments. L'allocation d'un segment nommé `X` de taille `size` à l'adresse `start` suit les étapes suivantes :

1. Vérifier qu'un espace mémoire libre suffisant est disponible à l'adresse indiquée (à l'aide de la fonction précédente).
2. Si c'est le cas, créer un nouveau segment `new_seg` de taille `size` à l'adresse `start`, puis ajouter le à la table de hachage `allocated`, avec `X` comme clé.
3. Remplacer dans `free_list` le segment libre qui contient `new_seg` par deux segments libres : un avant `start` et un après `start+size`.

Exemple : supposons une mémoire initiale de taille **1024**. Si l'utilisateur commence par allouer un segment nommé `"data"` de **100 unités** à l'adresse **200**, l'état devient :
- La table des segments alloués contient le segment `[200, 100]`, avec `"data"` comme clé.
- La liste des segments libres comporte désormais deux segments : `[0, 200]` et `[300, 724]`.

Q 2.3 Implémentez une fonction `int create_segment(MemoryHandler *handler, const char *name, int start, int size)` permettant d'allouer dynamiquement un segment de mémoire de taille `size` à l'adresse mémoire `start`.

La libération d'un segment consiste essentiellement à le retirer de la table de hachage `allocated` pour l'ajouter dans la liste `free_list`. Cet ajout doit cependant se faire de manière à optimiser la réutilisation des espaces libres, c'est-à-dire en fusionnant ce nouveau segment libre avec les éventuels segments libres qui lui sont adjacents. Sur l'exemple précédent, la libération de `[200, 100]` doit ainsi conduire à l'unique segment libre `[0, 1024]` résultant de la fusion de `[200, 100]` avec les segments libres adjacents `[0, 200]` et `[300, 724]`.

Q 2.4 Implémentez une fonction `int remove_segment(MemoryHandler *handler, const char *name)` permettant de libérer un segment de mémoire alloué et de l'ajouter à la liste des segments libres comme décrit ci-dessus.

La suite du projet sera mise en ligne la semaine du 10 mars.