Simulation d'un méchanisme d'allocation de mémoire

Claire Bilat

Juin 2020

1 Introduction

Le but de ce travail, effectué dans le cadre du cours *Systèmes d'exploitation*, est d'écrire un programme "thread safe" qui simule le système d'allocation de la mémoire utilisant un mécanisme de pure *demand paging*. Plusieurs **paramètres** ont été fixés dans le fichier `memory_simulation.h` et peuvent être modifiés selon les besoins de l'utilisateur:

1. `PAGE_SIZE` qui correspond à la taille (en bytes) d'une page (la mémoire virtuelle étant divisées en pages). Normalement c'est une puissance de 2 mais dans un but de lisibilité elle a été fixée à 100.
2. `FRAME_SIZE` qui correspond à la taille (en bytes) d'un frame (la mémoire physique étant divisée en frames). Celle-ci a été fixée à la taille d'une page pour éviter la fragmentation externe de la mémoire.
3. `VIRTUAL_ADDRESS_SIZE` correspondant à la taille de la mémoire virtuelle allouée à un programme, généralement supérieure à la taille de la mémoire physique installée dans le système et fixée ici à 5000.
4. `PHYSICAL_ADDRESS_SIZE` correspondant à la taille de la mémoire physique installée dans le système. Celle-ci a été fixée à 2000.
5. `MAX_TLB_LENGTH` correspondant au nombre d'entrées maximum qui peuvent être enregistrées dans le TLB. Celui-ci a été fixé à 4.

2 Description du programme

Les **structures** définies dans ce programme sont les suivantes (cf. `memory_simulation.h`):

1. `tlb_item` est une structure implémentant une entrée dans le TLB et constituée d'un entier correspondant au numéro de page, d'un entier correspondant au numéro de frame, et d'un entier correspondant au numéro du processus ayant entré le couple page-frame.

2. `free_frame` est une structure implémentant un frame vide et constituée d'un entier correspondant au numéro de frame vide et un pointeur sur le `free_frame` suivant.

3. `TLB` est une structure implémentant le TLB et constituée d'un pointeur sur le premier élément `tlb_item` et d'un `mutex` pour pouvoir protéger l'accès au TLB.

4. `free_frame_list` est une structure implémentant la liste des frames vides et constituée d'un pointeur sur le premier élément `free_frame` et d'un `mutex` pour pouvoir protéger l'accès à la liste des frames vides.

5. `memory_context` est une structure constituée d'un pointeur sur le TLB et d'un pointeur sur la liste de frames vides afin de pouvoir passer ces deux objets en paramètre de la routine d'un thread.

Les **variables** utilisées par le programme (cf. `memory_test.c`) sont les suivantes :

1. `pt` qui est un pointeur sur un tableau d'entiers stockant les numéros de frames associés à chaque page. Ainsi, l'indice est le numéro de la page et l'entier correspondant est le numéro du frame. Cette variable est locale à la routine d'un thread (cf. `routine()` dans `memory_simulation.c`).

2. `tlb` qui est un objet `TLB` contenant donc un tableau de `tlb_item` et constituant ainsi les entrées du TLB.

3. `ffl` qui est un objet `free_frame_list` contenant donc un pointeur sur le premier `free_frame`, constituant ainsi une liste chaînée de tous les frames libres.

4. `mc` qui est un objet `memory_context` contenant l'adresse des deux variables susmentionnées.

Les différentes **étapes** du programme (cf. `memory_test.c`) sont les suivantes:

1. Le programme va appeler `init_free_frame_list()` pour initialiser tous les frames comme vides, et `init_TLB()` pour initialiser toutes les entrées du `tlb` à INVALID. Les addresses des deux variables `ffl` et `tlb` ainsi initialisées sont stockées dans la variable `mc`.

2. Trois threads sont créés et servent à simuler l'existence de trois processus distincts voulant récupérer le contenu de la mémoire physique correspondant à une adresse de mémoire virtuelle.

3. Les trois threads exécutent la même routine (ici, le choix a été fait de les faire exécuter la même routine pour simplifier la démonstration mais il est clair que ce n'est pas toujours le cas), qui prend en paramètre l'adresse de la variable `mc` et qui décrit les étapes suivantes :

   (a) `init_page_table()` est appelée pour initialiser les entrées de `pt` à INVALID

   (b) Pour chaque adresse virtuelle déclarée et initialisée dans cette routine, le programme va tenter de retourner le numéro du frame (et l'adresse physique) où est stockée la page correspondant à cette adresse virtuelle en appelant la fonction `request_memory()`.

   (c) Pour ce faire, il va d'abord déterminer le numéro de page et l'offset correspondant (cf. fonction `get_page_number()`). Il va ensuite vérifier si le numéro de page est déjà dans le TLB via `ask_tlb()`. Si ce n'est pas le cas, il va vérifier si cette page est déjà dans la page table via `ask_page_table()`. Si ce n'est pas le cas, il va vérifier si un frame est vide et charger la page dans ce frame puis introduire ce couple page-frame dans la `page_table` et le `tlb` via `load_page()`. Lors de cette étape, les mutex sont "lockés" pour éviter des conflits entre les différents processus. Si il n'y a plus de frame vide, le programme se termine.

   (d) Le numéro de frame correspondant à la page ainsi que l'adresse physique correspondant à la page et le numéro du processus vont être affichés en output (l'adresse physique ayant été récupérée via `get_physical_address()`).

.

3 Choix techniques

Les adresses virtuelles décrites dans la `routine` sont exprimées en décimal par souci de lisibilité et de pédagogie. Il serait possible de modifier le programme de manière à ce qu'il fonctionne avec des adresses entrées en notation binaire; en effet, il suffirait de modifier les fonctions `get_page_number()` et `get_physical_address()` afin que celles-ci en prennent compte. Au lieu de faire les calculs de décomposition en base décimale, il faudrait traduire la variable `PAGE_SIZE` en binaire et effectuer les calculs de décomposition sur cette base. À titre d'exemple, la fonction `get_page_number()` pourrait prendre la forme suivante:

```
int get_page_number(int virtual_address) {
    int offset_size = log2(PAGE_SIZE - 1) + 1;
    int offset = virtual_address & ((1 << offset_size) - 1);
    int page = (int) ((unsigned) virtual_address >> offset_size);
    return page;
```

Il en irait de même pour la fonction `get_physical_address()`.