IFT2245 - Systèmes d'exploitation

Travail Pratique 3 (15%)

DIRO - Université de Montréal

À faire en équipe de deux.

Disponible : 30/04/2016 - Remise : 24/04/2016 avant 23h55

LISEZ TOUT LE DOCUMENT AVANT DE COMMENCER LE TRAVAIL.

Introduction

Dans ce travail pratique, vous devrez implémenter en langage C un programme qui simule un gestionnaire de mémoire virtuelle par pagination (paging). Votre solution simulera des accès mémoire consécutifs en traduisant les adresses logiques en adresses physiques de 16 bits (`uint16_t`) dans un espace d'adresses virtuelle (virtual address space) de taille $2^{16} = 65536$ bytes.

Votre programme devra lire et exécuter une liste de commandes sur des adresses logiques. Pour y arriver il devra traduire chacune des adresses logiques à son adresse physique correspondante en utilisant un TLB (Translation Look-aside Buffer) et une table de pages (page table).

Mise en place

Le code fournis utilise un fichier `makefile` qui vous aidera lors du développement. Le projet dépend des programmes `bison` et `flex` disponnibles sur les environnement `Linux`, `cygwin` et `OSX`.

Ce TP est basé sur un projet du livre de référence utilisé dans le cours, et il vous aidera à mettre en pratique les sections 8.5 (paging), 9.2 (demand paging) et 9.4 (page replacement).

Vous trouverez dans ces section tous les concepts nécessaires. Pour vous simplifier la tâche, on simulera une mémoire physique qui ne contient que des caractères imprimables (ASCII). C'est-à-dire, pour une mémoire physique de 32754 bytes (128 frames par 256 bytes), chacune de ses entrées contiennent un caractère parmi les 95 caractères imprimables possibles. Plus spécifiquement, le code fourni comprends déjà les structures de données de base pour un gestionnaire de mémoire avec les paramètres suivants (`src/conf.h`) :

- 256 entrées dans la page table
- Taille des pages et des frames de 256 bytes
- 16 entrées dans le TLB

- 128 frames
- Mémoire physique de 32754 bytes

Description du projet

Puisqu'on a un espace de mémoire virtuelle de taille 2^{16} on utilisera des adresses de 16 bits (`uint16_t`) qui encodent le numéro de page et le décalage (offset).

`Structure d'une addresse logique`

```
   Page Number          Offset
|---------------|----------------|
 16              8 7              0
```

Par exemple l'adresse logique 1081 représente la page 4 avec un décalage (offset) de 57.

Votre programme devra lire de l'entrée standard (`stdin`) une liste de commandes de lecture ou d'écriture. Vous devrez décoder le numéro de page et l'offset correspondant et ensuite traduire chaque adresse logique à son adresse physique, en utilisant le TLB si possible (TLB-hit) ou la table de page dans le cas d'un TLB-miss.

Traitement de Page Faults

Dans le cas d'un TLB-miss (page non trouvé dans le TLB), le page demandée doit être recherchée dans la table de pages. Si elle est déjà présente dans la table de pages, on obtient directement le frame correspondant. Dans le cas contraire, un page-fault est produit.

Votre programme devra implémenter la pagination sur demande (section 9.2 du livre). Lorsque un page-fault est produit, vous devez lire une page de taille 256 bytes du fichier `BACKING_STORE.txt` et le stocker dans un frame disponible dans la mémoire physique (au début du programme la mémoire physique commence toujours vide).

Par exemple, si l'adresse logique avec numéro de page 15 produit un page-fault, votre programme doit lire la page 15 depuis le `BACKING_STORE.txt` (rapellez-vous que les pages commence a l'index 0 et qu'elles font 256 bytes) et copier son contenu dans une page frame libre dans la mémoire physique. Une fois ce frame stocké (et que la table de pages et le TLB sont mis à jour), les futurs accès à la page 15, seront adressé soit par le TLB ou soit par la table de page jusqu'à ce que la page soit déchargé de la mémoire (swapped out). Le fichier `BACKING STORE.txt` est déjà ouvert en lecture et écriture par `pm_init` et fermé par `pm_clean`. Il contient 65536 caractères imprimables aléatoires. Suggestion: utilisez les fonctions de `stdio.h` pour simuler l'accès aléatoire à ce fichier.

Commande (`command.{l,y}`)

Les commandes sont automatiquement lues par les fichiers générés par `flex` et `bison`. Les fonction `vmm_read` et `vmm_write` sont automatiquement appelées. Vous ne devriez donc vous préoccupé du fonctionnement du programme qu'à partir de la gestion des commandes lues.

La lecture des commandes se fait par l'entrée standard (`stdin`) et les commandes invalides sont ignorées. Les commandes sont insensible à la casse et les espaces sont ignorés.

Les commandes sont de la forme suivante:

```
commande d'écriture
===================
W <logical address> <char to write>;
ex: W20'b';

commande de lecture
===================
R <logical address>;
ex: R89;
```

Virtual Memory Manager (`vmm.{c,h}`)

La structure `virtual_memory_manager` est la principale du gestionnaire de mémoire. Toutes les demandes d'adresses logiques sont converties par le virtual memory manager. Pour faire la conversion, vous devrez rechercher dans le TLB et la table de page, le numéro de frame correspondant à la page demandée. Vous devrez comptabiliser le nombre de fois où une demande est résolue par le TLB (`tlb_hit_count` et `tlb_miss_count`) et le nombre de fois où une demande de page échoue et doit être chargée en mémoire (`page_fault_count` et `page_found_count`).

struct page (`vmm.{c,h}`)

La `struct page` représente la structure de page utilisée dans la table de page. Le projet utilise 256 pages de taille 256 bytes. Chaque page possède un identifiant du frame auquel elle pointe (frame_number) et un ensemble de bit (`flags`) permettant de garder certaines information tel que le bit de vérification (verification bit) qui sert à savoir si une page est chargée en mémoire ou un dirty bit servant à savoir si une page a été modifiée. Vous être libre d'utiliser les 6 bits non utilisés pour vos algorithmes.

page_table (dans `struct virtual_memory_manager`)

La variable `page_table` est un tableau de 256 structures `page` pour représenter les 256 pages disponibles. Vous devrez utiliser ce tableau afin de trouver le numéro de frame correspondant à une page déjà chargée en mémoire physique.

Si la page n'est pas chargée, on a alors un page-fault, et il faut la charger en mémoire, pour enfin l'ajouter dans la `page_table` pour de futures demandes.

physical memory (`physical_memory.{c,h}`)

La structure `physical_memory` représente la mémoire physique sous la forme d'un tableau de 32754 charactères correspondant aux 128 frames de 256 bytes. Elle contient aussi l'accès au fichier `BACKING_STORE.txt` qui vous servira à charger en mémoire les pages se trouvant sur le backing store.

tlb (`tlb.{c,h}`)

La structure TLB utilise un tableau de 16 entrées qui représente un sous-ensemble des pages chargées en mémoire physique. Vous devrez implémenter la fonction de recherche d'une page dans la TLB et une fonction pour ajouter des entrées dans la table. Dans le cas où le TLB est plein, une stratégie de remplacement devra être mise en place afin de supprimer une page du TLB.

Choisissez deux méthodes de remplacement différents parmi : first-in,first-out (FIFO), least-recently-used (LRU), deuxième chance (clock), least-frequently-used (LFU), etc.

Nous vous demandons de remettre deux versions du programme (dossier `1` et `2`). Vous devriez être capable de copier-coller tout le code de votre première version à l'exception des fichiers `tlb.h` et `tlb.c`.

Travail à effectuer

Pour commencer

Vous pouvez commencer en utilisant la table de page en ignorant le TLB. Ajouter ensuite la fonctionnalité du TLB. Il est possible de séparer ainsi le travail étant donné que le TLB n'est en fait qu'un cache de traduction des adresses.

Sortie d'exécution

Vous devez changer les sorties déjà définies dans les fonctions `vmm_read` et `vmm_write` afin de fournir l'ensemble des valeurs qu'il faut afficher.

Votre programme doit aussi afficher à la fin de l'exécution les 4 compteurs de hit/miss que vous aurez maintenus à jour tout au long de l'exécution du programme

Les états finaux de la mémoire physique, de la table de pages et du tlb peuvent être enregistrés dans des fichiers données en option à l'exécutable. Exécutez `../build/vm -h` pour voir comment utiliser ces options.

Notez que l'état final du TLB dépend fortement de la stratégie de remplacement implémenté.

Code à compléter 11pts

Pour ce TP, vous devrez compléter les fonctionnalitées demandés ainsi que les fichiers fournis.

- 6pts sont attribués à la mise en oeuvre des 2 TLB et de leur algorithme de remplacement(3% chacuns)
- 3pts sont attribués aux autres fonctionnalités.
- 2pts sont attribués à la qualité du code (`valgrind` et compagnie).

Il est important de remettre des programmes fonctionnels. Une pénalité minimale de 3pts est donnée pour tout programme qui ne compile pas ou programme qui plante systématiquement (même si cela ne demande qu'une petite correction).

Rapport 4pts

Décrivez brièvement le fonctionnement de votre solution et énoncez clairement les fonctionnalités mises en oeuvre. Si vous avez des bugs, le rapport est l'occasion d'essayer d'expliquer les causes de ces bugs. Évitez de raconter «l'histoire» de votre programme.

Répondez explicitement à chacune des question suivantes :

1. Quelles stratégies de remplacement avez-vous utilisées pour le TLB? Quels sont leurs avantages/inconvénients par rapport aux autres solutions?
2. Comparez les valeurs de TLB Hit ratio et Page Fault Ratio de ces stratégies. Qu'est-ce que vous en tirez?
3. Faites différet tests où vous avez fait varié certains paramètres du fichier de configuration (`conf.h`) (tout en gardant un espace addressable de 65536 bytes). Quels sont les effets sur les différentes statistiques? Est-ce que vos stratégie s'adaptent aussi bien l'une que l'autre à ces changements? Assurez-vous de la pertinence des tests effectués (2 cas d'études sont suffisants).

Format des documents à remettre

Remettez sur Studium l'archive générée par la commande `make release`.