TP3 – Mémoire virtuelle

Dû le 30 avril à 23h00

Préambule

Ce TP vise à vous introduire la gestion de mémoire virtuelle. Vous aurez a implémenter les algorithmes vus en classe. Ce TP ne fait pas partie de la suite de TPs reliés à l'implémentation d'un shell. En fait, le TP2 était le dernier de cette série.

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 dans un espace d'adresses virtuelle, donc de 65536 bits.
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 fichier .zip fournit devrait être directement compatible avec Clion, mais le build process est différent. Il faut utiliser le gnumakefile qui est fournit. Des instructions sont données à la fin de cet énoncé.
Le projet dépend des programmes *bison* et *flex* disponibles sur les environnement GNU/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 8KB (32 frames par 256 bytes), chacune de ses entrées contiennent un caractère parmi les 95 caractères imprimables ASCII 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 table de page
- Taille des pages et des frames de 256 bytes
- 8 entrées dans le TLB
- 32 frames
- Mémoire physique de 8192 bytes

1 Description du projet

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

2 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 commencent à l'index 0 et qu'elles font 256 bytes) et copier son contenu dans une 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 et fermé pour vous. Il contient 65536 caractères imprimables aléatoires. Suggestion: utilisez les fonctions de stdio.h pour simuler l'accès aléatoire à ce fichier.

3 Commandes

Les commandes sont automatiquement lues par les fichiers générés par *flex* et *bison*. Les fonctions *vmm_ read* et *vmm_ write* sont automatiquement appelées. Vous ne devriez donc vous préoccuper 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	commande de lecture
W logical-address 'char-to-write';	*R* logical-address;
ex: *W20'b';*	ex: *R89;*

Bien sûr, vous pouvez rediriger le contenu de fichiers de commande vers le stdin de votre programme pour faciliter vos tests.

4 Travail à effectuer

Vous devez implémenter les fonctions incomplètes de *vmm.c*, *pm.c*, *pt.c*, et *tlb.c*, y compris l'implémentation de l'algorithme de remplacement du TLB et des frames ainsi que la gestion de l'état "dirty" des pages. De plus, vous devez corriger les sorties déjà définies dans les fonctions *vmm_ read* et *vmm_ write* afin de fournir l'ensemble des valeurs qu'il faut afficher.
Finalement, vous devez fournir 2 fichiers de tests *tests/command1.in* et *tests/command2.in*. Le premier devrait principalement tester l'efficacité du TLB alors que le deuxième devrait aussi tester l'algorithme de remplacement des pages.

5 Makefile

Pour vous faciliter le travail, le fichier `GNUmakefile` de l'archive vous permet d'utiliser les commandes suivantes:

- `make` ou `make all`: Compile le projet vers le dossier build
- `make run`: Lance le projet

- `make clean`: Nettoie le dossier build

Petites précisions

Voici des questions-réponses/précisions recueillies pour ce TP:

- Le flag `readonly` de quelques fonctions et structures est une autre façon de parler du `dirty bit`. C'est un `read` au passé, genre "est-ce que ce truc la a seulement été lu, et non modifié?".
- Si vous trouvez qu'un paramètre d'une fonction ne sert a rien, vous pouvez ne pas l'utiliser, si votre fonction est quand même capable de répondre à la spécification. Cependant, vous ne pouvez pas l'enlever de la signature.

Remise

Ce travail est à faire en groupes de 2 étudiants. Le code est à remettre par remise électronique avant la date indiquée.
Chaque jour de retard est -15%, mais après le deuxième jour la remise ne sera pas acceptée.
Indiquez clairement votre nom et votre matricule dans un commentaire en bloc au début de chaque fichier, dans ce format:

```
/*
 *   Prenom1 NomDeFamille1 Matricule1
 *   Prenom2 NomDeFamille2 Matricule2
*/
```

Afin de valider votre entête, le projet `https://github.com/SamuelYvon/NameValidator` est mit à votre disposition. C'est votre responsabilité de vérifier que votre entête correspond au format attendu. Si ce n'est pas le cas, votre devoir aura la note 0. Si le script ne valide pas correctement votre nom, il est de votre responsabilité de nous le signaler (au travers le système de issues de github ou par courriel) afin que nous puissions valider l'erreur et corriger le script.
Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte.
Le programme doit être exécutable sur les ordinateurs du DIRO.
Pour la remise, vous devez remettre l'ensemble des fichiers .c et .h fournis avec cet énoncé en utilisant **ce lien** dans un fichier zip. Assurez-vous que tout fonctionne correctement sur les ordinateurs du DIRO.

Barème

Ce TP compte pour 10 points sur votre note totale dans le cours.
Pour ce TP:

- 60% points sur le fonctionnement correct du code:
  - 15% point pour vmm (virtual memory manager)
  - 15% points pour pt (page table)
  - 15% points pour pm (physical memory)
  - 15% points pour tlb (translation lookaside buffer)
- 20% points sur l'efficacité de vos algorithmes de remplacement (en terme de minimisation des TLB miss et des page faults).

  - 20% point pour les tests fourni

- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, etc.) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat. Si pour une question votre solution est directement copiée, même si il y a attribution de la source, cette question se verra attribuée la note de zéro. Vous pourrez cependant l'utiliser dans les sections suivantes sans pénalité.

- Vous serez pénalisés pour chaque warning lors de la compilation à raison de 1% par warning, sauf pour les warning reliés à l'assignation à NULL, à la comparaison avec NULL, et aux override des fonctions de librairie.

- Si une fuite mémoire est identifiée, vous perdrez 15%. Vous ne perdrez pas plus de points pour les fuites si vous en avez une ou trente.

- Les accès mémoire illégaux identifiés par Valgrind entraîneront jusqu'à 5% de perte, à raison de 1% par accès. La répétition d'un même accès sera comptée comme 1% de plus quand même.

- Votre devoir sera corrigé automatiquement en très grande partie. Si vous déviez de ce qui est demandé en output, les points que vous perdrez seront perdus pour de bon. Si vous n'êtes pas certains d'un caractère demandé, demandez sur le forum studium et nous répondrons de façon à ce que chaque étudiant puisse voir la réponse.

- La méthode de développement recommandée est d'utiliser CLion pour la complétion de code, mais d'utiliser le GNUmakefile pour rouler et compiler. Ceci est compatible avec Valgrind. Si vous voulez utiliser d'autres techniques, vous pouvez le faire, mais nous ne vous aiderons si vous rencontrez des problèmes avec ces techniques. Nous justifions ici l'utilisation d'un GNU makefile pour bien fonctionner avec les librairies flex/bisons ainsi de vous permettre d'apprécier cet outil encore largement utilisé.