# Travail pratique #3

IFT-2245

April 1, 2019

¡¡ Dû le 30 avril à 23h55 !!

## 1 Survol

Ce TP vise à vous familiariser avec les éléments de la mémoire virtuelles. Les étapes de ce travail sont les suivantes:

1. Lire et comprendre cette donnée. Cela prendra probablement une partie importante du temps total.
2. Lire, trouver, et comprendre les parties importantes du code fourni.
3. Compléter le code fourni.
4. Écrire un rapport. Il doit décrire **votre** expérience pendant les points précédents: problèmes rencontrés, surprises, choix que vous avez dû faire, options que vous avez sciemment rejetées, etc... Le rapport ne doit pas excéder 5 pages.

Ce travail est à faire en groupes de 2 étudiants. Le rapport, au format LaTeX exclusivement et le code sont à remettre par remise électronique avant la date indiquée. Aucun retard ne sera accepté. Indiquer clairement le(s) nom(s) au début de chaque fichier.

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Si un étudiant ne trouve pas de partenaire, il doit me contacter au plus vite. Des groupes de 3 ou plus sont **exclus**.

## 2 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 (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).

## 3 Mise en place

Le code fournis utilise un fichier `GNUmakefile` qui vous aidera lors du développement. 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 page table
- Taille des pages et des frames de 256 bytes
- 8 entrées dans le TLB
- 32 frames
- Mémoire physique de 32768 bytes

## 4 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.

# 5 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.

# 6 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;` |

## 6.1 Makefile

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

- `make` ou `make all`: Compile l'application.
- `make release`: Archive le code dans un `tar` pour la remise.
- `make run`: Lance le client et le serveur ensembles.
- `make clean`: Nettoie le dossier build.

# 7 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 (vous pouvez choisir les algorithmes mais veuillez spécifier et justifier vos choix dans le rapport).

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.

# 8 Remise

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

# 9 Système de classement

La note (sur un maximum de 15 points) sera divisée comme suit:

- 3 points pour le rapport,
- 8 points sur le fonctionnement correct du code,
  - 2 point pour vmm (virtual memory manager)
  - 2 points pour pt (page table)
  - 2 points pour pm (physical memory)
  - 2 points pour tlb (translation lookaside buffer)
- 2 points sur l'efficacité de vos algorithmes de remplacement (en terme de minimisation des TLB miss et des page faults).
- 2 point pour les tests fourni,
- +/- 0.75 points sur la lisibilité du code (avec possibilité de bonus 1 point du code extrêmement lisible),
- -1 segfaults

- -1 fuites de mémoire
- -1 Bugs quelconques

# 10 Détails

- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, ...) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat.
- Le code ne doit en aucun cas dépasser 120 colonnes (-0.5 points).
- Vérifiez la page web du cours, pour d'éventuels errata, et d'autres indications supplémentaires.
- La note sera basée d'une part sur des tests automatiques, d'autre part sur la lecture du code, ainsi que sur le rapport. Le critère le plus important, et que votre code doit se comporter de manière correcte. Ensuite, vient la qualité du code: plus c'est simple, mieux c'est. S'il y a beaucoup de commentaires, c'est généralement un symptôme que le code n'est pas clair; mais bien sûr, sans commentaires le code (même simple) est souvent incompréhensible. L'efficacité de votre code est sans importance, sauf s'il utilise un algorithme vraiment particulièrement ridiculement inefficace.