Université de Montréal

IFT2245 - Systèmes d'exploitation

Tp3

Nicolas Richard

Mathieu Matos

remis au
Prof. Liam Paull

06 Mai 2019

Problèmes Rencontrés et Surprises

1. **Incompatibilité Mac / Linux**
   Un des premiers problèmes rencontrés a été causé par l'utilisation d'un Mac pour ce travail. Ce qui compilait sur Linux ne compilait pas nécessairement sur Mac, donc la mise à jour et la division des tâches a dû être testée sur Linux à chaque itération. Nous devions utiliser les ordinateurs à l'université à chaque fois. Cela a demandé plus de temps pour tester (push/pull sur git à chaque fois pour tester sur Linux ainsi que le déplacement à l'université) et un nombre considérable de temps a été mis dans le débug avant de réaliser l'incompatibilité avec Mac.

2. **Compréhension et questions théoriques**
   Une partie importante de ce travail était la compréhension de la matière et de s'assurer de commencer la partie codage sur une bonne lignée. Cette partie n'était pas un gros contretemps puisque nous comprenions assez bien la matière étudiée. Nous avons bien pris le temps de discuter et schématiser le plan de match avant d'écrire une ligne de code. Les quelques questions qui sont parvenues dans notre chemin ont pu être répondues dans l'implémentation, au fur à mesure, par une simple recherche internet ou même par un auxiliaire d'enseignement. Des exemples de ces questions sont : le choix de l'algorithme de replacement de frames/TLB; la correspondance entre le nombre de frames et le nombre de pages selon la taille du *backing_store*; l'implémentation d'un backup efficace. Ces questions sont abordées dans la section suivante.

3. **Erreur de segmentation en mémoire physique**
   Un autre problème, qui nous a pris quelques heures et un peu à la dernière minute, était une faute de segmentation causée par un changement dans une classe pour faire des tests modulaires qu'on avait oublié de retourner à normal pour les tests sur l'ensemble et nous avons pris plus de temps que voulu pour se souvenir de ce changement et de le remettre correctement.

4. **Initialisation des pages**
   Aussi, lors de l'initialisation de la TLB, nous avons réalisé que les pages n'étaient pas initialisées d'une manière conforme à notre code et nous avons donc opté pour ajouter une ligne dans *tlb_init* afin d'initialiser les

pages à quelque chose qui correspond avec notre code, soit les initialiser à -1 comme les frames.

5. **Backup final**
   Finalement, nous avons presque oublié de faire un backup de la mémoire physique final à la fin du roulement du programme. Pour ce faire, nous avons été dans le main du parse.y et avons ajouté une fonction vmm_backup_final qui est implémentée dans $vmm.c$, servant à passer à travers les frames qui ont été modifiés et de les mettre à jour juste avant l'impression de l'état final des compteurs et la terminaison du programme.

Choix faits

1. **Algorithmes de remplacement**
   Un des choix les plus importants est sans doute le choix de l'algorithme pour la TLB ainsi que le remplacement des frames dans la mémoire physique lorsque ces deux espaces mémoire sont pleins et requièrent un swap.

2. **Remplacement dans la TLB**
   Pour ce qui est de la TLB, la maintenance de la cache et l'algorithme de replacement a lieu à l'intérieur même de la fonction $tlb\_\_add\_entry$, que nous appelons à chaque fois qu'on fait une recherche dans la TLB. Cette fonction traite donc tous les cas possibles d'une lecture/écriture, soit tlb-hit ou tlb-miss et une page fault ou pas. Nous avons opté pour un algorithme LRU (Least Recently Used) avec une certaine particularité : nous avons créé un tableau d'index qui ne sert qu'à suivre les entrées de la TLB sans nécessairement déplacer tout le contenu d'un espace dans la TLB. Donc, par exemple, si une certaine entrée avec index i était utilisé dans la TLB, de déplacer seulement cet index i au début du tableau d'index, qui prend moins de place et demande moins d'opérations de repositionnement que si nous modifions le contenu de la TLB au complet à chaque fois. Aussi, lorsque nous devions swap un item de la TLB, il ne s'agissait seulement de prendre que le dernier index de ce tableau d'index, de swap la TLB à cet index, qui est le $Least\ Recently\ Used$, et de réitérer les valeurs du tableau d'index pour mettre l'ancien $Least\ Recently\ Used$ comme étant le $Most\ Recently\ Used$.

Un autre point important de cette technique est l'initialisation de ce tableau d'index, qui n'est réalisée que lorsqu'on a les 8 premières page faults.

3. **Pourquoi LRU**
   La raison pour laquelle nous avons choisi l'algorithme de LRU est à cause son efficacité. Cet algorithme devient très efficace lorsque des read/write se font sur des pages ayant une fréquence d'utilisation plus élevées, d'où l'utilité d'une cache pour ce cas, pour un accès plus rapide. Par contre, ce système n'est pas sans faille : dans la situation comme la nôtre, si les commandes sont générées sur les 256 pages aléatoirement, sans notion de fréquence, une cache de 8 entrées n'est pas gérée de manière très efficace avec LRU puisque les nouvelles commandes seront très peu présentes dans la TLB. Donc, pour minimiser l'effort de calcul utilisé avec LRU, qui serait très peu efficace, ce serait probablement plus avantageux d'utiliser un algorithme plus simple en termes d'opérations et presqu'autant efficace, soit FIFO. Aussi, il est important de mentionner l'utilisation d'un tableau additionnel pour notre choix d'algorithme, qui demande un espace mémoire plus élevé (de très peu).

4. **Remplacement de frames**
   Très similaire à l'algorithme de la TLB, nous avons implémenté un algorithme de LRU pour ce qui concerne les frames à la mémoire physique. Lorsqu'on doit remplacer un frame en mémoire, nous regardons quel frame à été utilisé en dernier et le remplaçons. Encore, nous avons créé un nouveau tableau d'index, comme avec la TLB, pour swap les frames seulement lorsque nécessaire. La même logique a été appliquée dans cette partie et celle de remplacement d'entrées en TLB.

5. **État dirty de pages et effet sur backup**
   Pour l'état *dirty* des pages, ce que nous avons fait est très simple. Une page est structurée avec un bit *readonly*, qui garde l'état indiquant si cette page a été modifiée ou non. Lorsqu'on *read* une page, nous ne modifions pas ce bit et lorsqu'on *write* une page, ce bit est mis à true si ce n'était pas déjà le cas. Ce bit est actualisé dans la page table et ensuite dans la TLB. Cette implémentation a un effet sur le backup des frames, qui est élaboré dans la section suivante. En bref : si on swap un frame ayant été modifié (avec un dirty bit *true*) on doit s'assurer de faire le backup avant de swap pour mettre l'important à jour.

6. **Génération de commandes et tests d'efficacité d'algorithmes**
   Les commandes dans les fichiers de test ont été générées aléatoirement. Pour le fichier qui teste l'efficacité du TLB, nous avons produit des commandes demandant l'accès à un total de 9 pages, ce qui devrait engendrer peu de $TLB\ MISS$ lorsque celle-ci comporte 8 entrées. Pour tester l'efficacité de l'algorithme de remplacement, nous avons généré des commandes ayant des adresses pouvant aller jusqu'à 65 536, ce qui devrait engendrer davantage de $page faults$. Nous avons pensé que 300 commandes était une quantité raisonnable dans le cadre de ce tp.

7. **Compteur d'initialisation des frames**
   Nous avons implémenté un compteur qui garde une trace des frames de la mémoire physique qui sont encore vides. Ceci nous permet d'accélérer le processus de sélection de frame disponible pour accueillir une nouvelle page en début d'exécution. En effet, ce compteur nous permet de remplir les frames de la mémoire physique de façon séquentielle d'abord et d'éviter l'algorithme LRU. Une fois que toutes les frames ont été remplis au moins une fois, ce compteur n'est plus utile et nous nous référons à LRU pour décider du frame à évincer.

Options Rejetées

1. **Algorithmes rejetés**
   Nous avons fait le choix de l'algorithme de remplacement *Least Recently Used*, au détriment des algorithmes *First In First Out*, *Random* et *Optimal* en raison du fait que LRU est considéré plus performant en général. *Random* est plus simple à implémenter, et dans certains cas est meilleur que LRU, mais demeure un des algorithmes les moins performants. $FIFO$ est également simple à coder, mais n'est pas souvent utilisé en pratique, c'est pourquoi nous avons opté pour LRU. Bien sûr, *Optimal* est celui qui aurait eu les meilleurs statistiques et aurait été réalisable puise que nous avons accès à toutes les commandes dès le départ. Cependant, il n'est pas utilisé en pratique et dans les applications concrète de la vie de tous les jours, il est impensable et peu pertinent d'implémenter l'algorithme *Optimal*. LRU demande un support matériel et nous avons choisi d'implémenter une pile plutôt que d'y aller avec l'algorithme de l'horloge. Nous n'avons pas vu de

différence significative entre les deux méthodes et celle de la pile nous semblait plus réaliste.

2. **Méthode de Backup**
   En ce qui concerne les *backups*, nous avons choisi d'implémenter notre programme pour qu'il fonctionne de façon asynchrone. Nous pensons qu'il aurait été trop coûteux en temps de faire un *backup* à chaque fois qu'il y a une écriture sur une page. Nous avons donc décidé de sauvegarder les modifications dans le *backing store* uniquement lorsque la page en question a dû être évincée de la mémoire physique. Ceci permet d'économiser du temps, surtout lorsque plusieurs modifications sont effectuées sur la même page. Un seul *backup* est nécessaire pour toutes les modifications sur la page. En suivant ce mode asynchrone, il est important de faire un *backup* sur tous les frames modifiés avant que le programme ne se termine, puisque ces pages en mémoire physique ne seront pas évincées par le biais de l'algorithme de remplacement de pages. En tenant en compte les contraintes de la vie réelle, nous comprenons que la méthode de *backup* asynchrone (vs synchrone) est celle qui souffre le plus face aux pannes, puisqu'il y a davantage de modifications qui n'auront pas été enregistrées. Considérant l'étendue de ce tp, nous avons choisi de ne pas prendre de mesure de résistance aux pannes.