Travail pratique #3 - IFT-2245

Maude Sabourin et Jérémy Coulombe

April 30, 2018

1 Expérience

Arrivé à ce travail, nous pensions être rendu des pros de gestion de mémoire et de programmation en C. Cependant, nous avons été confronté à des difficultés différentes, principalement reliée au concept de TLB et de page de table. Nous avions une compréhension de ces concepts plutôt en surface, donc les implémenter a été difficile. Après une première écriture des éléments clés du programme, nous avions plusieurs incohérence : backup de la même page qui doit être loadée, écriture sur une page readonly, mauvaise utilisation des fonctions données à notre disposition, incapacité de récupérer la page à partir d'une frame, etc. Nous avons utilisé les statistiques obtenues (principalement tlb miss) comme indicatif nous permettant de savoir si nous étions dans la bonne direction, le score de 100% étant nécessairement mauvais.

2 Difficultés

Unsigned Integer : Une première difficulté a été les UInt qui étaient utilisés dans certaines parties du programme. N'ayant jamais vraiment travaillé avec ces éléments, nous avions un bug où nous faisions une comparaison if UInt <0. Il a fallut rechercher l'origine du bug pour comprendre que cette ligne était toujours fausse. Une recherche sur google nous a ensuite expliqué pourquoi.

Compréhension des fichiers : Même si les fichiers peuvent sembler intuitif de par leur nom, il a été difficile au début de comprendre le flow de l'application (qui appelle qui) et comment le contrôler. Nous avons premièrement codé le TLB et la PT, donc nous avions de la difficulté à faire le lien avec PM et VMM.

Bugs variés : À un moment donné, lorsque nous écrivions dans le backing store, nous ajoutions des caractères NULL ce qui faisait en sorte que le fichier texte

devenait instable. Plus tard, nous ne pouvions déboguer via l'impression de la ligne de commande pour une raison qui nous reste encore inconnue à ce jour ce qui semble être une tendance lorsque nous travaillons en C. Un autre problème a été que GDB ne nous laissait pas mettre de breakpoint, donc nous n'avons jamais pu débogguer ligne par ligne. Nous avons dû effectué des printf sans fin.

Tests : Finalement, nous avons voulu être fancy et nous créer un script qui génère automatiquement des opérations avec des adresses aléatoires. Cependant, nous avons eu quelques bugs dans le code et ça a pris plus de temps que nous pensions.

3 Bons coups

Commandes Stream : Chaque TP nous découvrons de nouvelles commandes, donc nous étions plus à même de savoir ce que nous devions chercher. Nous hésitions cependant entre fwrite et fputs. Après une recherche rapide, les deux utilisations semblaient acceptées. Nous avons également trouvé lors de notre recherche fseek et fread que nous avons utilisé.

Memset & Strncpy : Nous les avions utilisés dans les TPs passés donc nous avons pu rapidement créer un buffer et l'utiliser pour transférer les éléments en mémoire.

Adresse Logique : Faire la conversion entre adresse logique et physique s'est bien passé, car nous avions bien étudié pour l'examen et étions des pros des offset et de la séparation de la plage mémoire.

4 Résultats

Afin de vraiment tester notre code, nous avons créé plusieurs tests variés. Ainsi, nous avons créé un dossier 'Extra' contenant plus de tests. Les tests ont été réalisé avec NUM_FRAMES = 3 et TLB_NUM_ENTRIES = 3

Command.in 1 TLB Miss (la première écriture) et 1 seul PF → Normal, on utilise seulement 1 page pour toute l'exécution

worstCase.in 98.9% TLB Miss et 90 PF → Chaque appel est sur la prochaine page (1,2,3,1,2,3,1,2,3...) ce qui force le TLB et la table des pages à swap pour constamment reloader les pages.

TLBLRU.in → command1.in : BESOIN DE LA CONFIG 3,3 pour voir l'algorithme Permet de voir que le LRU fonctionne bien via un test simple (frame 3 remplace

1 et 4 remplace 2)

Tests réalisés avec NUM_FRAMES = 32 et TLB_NUM_ENTRIES = 8

bestCase.in → !Attention va écrire par dessus le backingstore! Écrit de façon séquentiel, TLB Miss de 0.4%.

normalCase.in Opérations aléatoires avec les valeurs possibles réduits à 3000, TLB Miss de 33%. On suppose le pourcentage, car on accède aux pages de façon aléatoire, mais que les valeurs sont plus restreintes que worstcase.in

5 Algo et choix implantés

Algorithme : Nous avions décidé d'implémenter le LRU, car le FIFO offre généralement une performance moindre tel que vu en classe. Notre première approche a été d'implémenter une liste ordonnée qui se met à jour chaque fois qu'un accès est effectué. Cependant, l'implémentation avait quelques problèmes et il semblait compliqué de toujours déplacer tous les éléments. En effet, le déplacement de plusieurs éléments dans un tableau pour chaque accès au tlb ou à un frame revenait lourd. Nous avions aussi eu des problèmes avec la librairie dyn_array au dernier TP donc nous voulions éviter de réutiliser des arrays dynamiques.

Finalement, nous nous sommes inspirés d'une implémentation expliquée dans le livre expliquant qu'on incrémente le premier bit d'une séquence lors d'un accès et qu'on shift tous les bits subséquents. Cet algorithme est une variante du bit d'accès où on peut avoir un historique des derniers accès. Les bons côtés sont que cet algorithme était facile à implémenter et permet de rapidement voir les derniers accès.

La difficulté de l'algorithme était de déterminer la grandeur du chiffre. Nous voulions que ça fonctionne pour les architecture 32 bits et 64 bits. Nous avons finalement décidé d'implémenter le 32 pour qu'il soit compatible avec les deux (via l'utilisation de 0x80000000). Au départ, nous utilisions seulement un unsigned int, mais par la suite nous avons eu besoin de plus d'informations relatives aux éléments, donc nous avons transformé notre paramètre en une structure contenant plusieurs attributs (framenumb,pagenumb,valid,UInt).

Il est à noter que l'algorithme pour le TLB et pour le swapping des frames (paging) est le même soit un algorithme LRU.

Nous avons décider de faire une implémentation qui est la plus réaliste possible . Ainsi, nous ne copions le contenu des frames (backup) que lorsque nous avons écrit sur celles-ci (dirty). Cela permet de minimmiser le nombre d'opérations

IO (celles-ci sont généralement les opérations les plus coûteuses sur de vrais systèmes).

Copy on write : Nous avons jugé qu'il serait pertinent d'utiliser une stratégie permettant de gérer l'écriture sur une page readonly. C'est à ce moment que nous nous sommes rendus compte que nous devions modifier notre structure initiale du LRU pour inclure plus d'éléments. Plusieurs difficultés sont apparues, car nous n'étions pas certain comment gérer la présence de 2 frames pointant vers la même page dans la table de pages. Après avoir lu en ligne et révisé les notes, il n'était pas clair comment agir avec ce cas-là. Nous avons donc fait au meilleur de nos connaissances pour l'implémentation de cette variante. De façon surprenante, après avoir ajouté cette gestion des bugs readonly, nous avons amélioré nos statistiques sur tous les tests.

6 Améliorations possibles

Nous pourrions implanter une table de page multi-niveaux nous permettant ainsi d'augmenter la portée de notre algorithme. Cependant, on se souvient que cette manière de faire augmente également les accès mémoire, ce qui peut faire baisser les performances globales du système. Sinon, il serait aussi intéressant de gérer plusieurs tables de pages soit une pour chaque processus fictif afin d'avoir une implentation qui se rapproche encore plus de la réalité.

Une autre idée est d'implémenter de manière plus complète le "copy on write" permettant de gérer des accès en parallèle.

7 Fonctionnement du code

Pour l'implémentation du "copy on write" : 1) Nous obtenons un frame (libre ou victime) 2) Nous sauvegardons son contenu en mémoire 3) Nous allons chercher le contenu de la page sur laquelle nous voulions écrire 4) Nous téléchargeons le contenu de celle-ci dans le frame que nous avions obtenu 5) Nous modifions la table de page. 6) Pour la sauvegarde en mémoire nous n'écrivons le contenu des frames en mémoire que s'ils ont été modifiées (i.e. s'ils sont dirty)

Pour les algorithmes d'ordonnancement nous utilisons les bits shifts afin savoir qu'elle frame a la plus récente utilisation.