

Travail pratique #3 - IFT-2245

Christophe Apollon-Roy (920403)
et
Amélie Lacombe Robillard (20016735)

April 29, 2018

1 Introduction

Pour ce travail pratique, nous avons dû simuler la gestion de la mémoire avec une table de pages, un *Translation lookaside buffer* -que nous nommerons dorénavant TLB pour le reste de ce rapport-, une mémoire physique et une unité de gestion de mémoire virtuelle. Pour réaliser cela, nous avons, toujours à partir du code fourni, implémenter les fonctions de base du TLB (la recherche et l'ajout d'entrées), de la page de tables (la recherche et l'ajout d'entrées ainsi que le marquage des pages valides ou *dirty*) et de la mémoire physique (le chargement et la sauvegarde de pages sur le disque dur -émulé ici par un fichier contenant des caractères imprimables ASCII- ainsi que la lecture et écriture dans les *frames* de la mémoire physique), l'algorithme de remplacement utilisé par l'unité de gestion de mémoire virtuelle ainsi que divers tests visant à évaluer la performance de ces fonctionnalités. La lecture et l'analyse des requêtes envoyées à l'unité de gestion de mémoire virtuelle étant déjà traitées par les utilitaires fournis avec le code de base, le coeur du travail fut l'implémentation de l'algorithme de remplacement de pages ainsi que l'élaboration des tests.

Le rapport qui suit verra plus particulièrement les deux tâches mentionnées plus haut en examinant différents algorithmes de remplacement de pages et en expliquant la démarche derrière l'élaboration des tests utilisés.

2 Algorithme de remplacement des pages

Il existe plusieurs stratégies pour gérer le remplacement de pages, certaines pouvant permettre une gestion presque optimale des *page faults* et ainsi réduire leur fréquence à un minimum. Étant donné le cadre du travail (nous sommes obligés d'utiliser la pagination sur demande), nos habilités et la complexité de certains algorithmes, nous avons décidé de choisir parmi les algorithmes suivants pour

gérer les *TLB-miss* et les *page faults*:

OPT Optimal, permet de réduire au minimum les *page faults* en remplaçant la page dont la prochaine référence se produira le plus loin dans la queue de requêtes

FIFO *First In First Out*, remplace le contenu de la page la plus vieille

CLOCK Similaire à FIFO, mais, ne remplace une page que si le bit la marquant comme étant "référéncée" est à 0. Si ce bit de référence est à 1, il sera mis à 0 et l'algorithme passera à la prochaine victime potentielle

LRU *Least Recently Used*, remplace la page dont la dernière référence est la plus ancienne

Random Comme le nom le dit, choisir aléatoirement une page à remplacer. Cette victime peut être le choix optimal comme elle pourrait être le pire choix possible (par exemple, une page qui sera référencée très peu de temps après son remplacement), impossible de savoir.

Dans le meilleur des mondes, OPT serait le premier choix, mais, malheureusement, son implémentation demanderait qu'on connaisse toutes les requêtes de mémoire futures, ce qui est impossible (avec les technologies présentement disponibles). En revanche, FIFO et Random peuvent être implémentés très facilement, mais leur performance laisse souvent à désirer. LRU peut s'approcher de OPT du point de vue de la performance, mais le coût en ressources est plutôt élevé; en effet, il est souvent nécessaire d'utiliser des structures de données dispendieuses telles les listes chaînées afin de garder la trace des références à chaque page. Pour sa part, CLOCK se situe entre FIFO et LRU au niveau de la performance (peut s'approcher de LRU dépendant de la situation) et requiert un coût minimal en ressources; les variantes les plus minimalistes ne demande qu'un bit de plus pour chaque entrée de la table des page et un pointeur pour référencer la page sur laquelle se situe l'aiguille de "l'horloge".

En évaluant le rapport performance/coût de chaque algorithme, nous avons donc décidé d'implémenter CLOCK, plus précisément la variante *enhanced second chance* qui prend aussi en compte le statut de modification de la page, car il était beaucoup plus abordable que LRU; en effet, pour notre implémentation, nous n'avions besoin que de surveiller une paire de bits indiquant si la page a été référencée ou modifiée et de garder en mémoire sur la pile deux entiers faisant référence à l'entrée où est rendue la boucle cherchant une victime à remplacer parmi les entrées du TLB et les frames de la mémoire physique respectivement. En plus, le fait de prendre en compte le statut de modification de la page permet de limiter les opération d'écriture sur le disque dur qui sont toujours particulièrement coûteuses.

3 Élaboration des tests

Notre choix de l'algorithme *CLOCK enhanced second chance*, comme vu dans la section précédente, se base sur l'hypothèse que toutes les pages ne sont pas accédées en mémoire avec la même fréquence, et qu'une page accédée récemment a plus de chance d'être accédée à nouveau. Lors de notre premier essai d'élaboration de tests, nous sommes allés pour une approche purement aléatoire, utilisant un générateur de nombres aléatoires pour générer une liste d'adresses réparties également sur tout l'espace d'adressage logique. Cependant, cette méthode ne permettait pas de bien représenter l'avantage de *CLOCK* sur un algorithme comme *Random*.

Pour notre deuxième essai, nous avons décidé de structurer nos tests comme plusieurs *working-set* se succédant dans le temps, ce qui est nous semble plus proche de l'utilisation qu'un système d'exploitation pourrait faire d'une mémoire virtuelle. Ceci permet également de mieux démontrer les avantages de l'algorithme *CLOCK* versus un l'algorithme purement aléatoire ou *FIFO*, puisque les pages accédées récemment prendront plus de temps à être remplacée lors de la transition d'un *working-set* à un autre (et une page commune aux deux *working-set* ne sera pas remplacée). Par soucis de simplicité, les *working-set* ont été défini comme un ensemble de pages contiguës en mémoire, permettant la génération aléatoire d'adresses dans l'intervalle des adresses logiques de ces pages. Les read-write ont été distribués aléatoirement, pour constituer chacun la moitié des requêtes.

Le premier fichier de test, *command1.in*, vise à tester l'efficacité du TLB. Pour éviter que l'algorithme de remplacement de page n'interfère avec ce test, nous avons gardé le nombre total de pages accédées sous 32 (nombre total des frames en mémoire physique). Ce test est composé de deux *working-set* de 16 pages chacun (pages 0-15 et 12-27), avec un chevauchement de 4 pages communes. Chaque *working-set* est composé de 30 requêtes.

Le deuxième test, *command2.in*, vise à tester l'efficacité du remplacement des frames dans la mémoire physique, en plus du TLB. Ce test utilise donc la quasi-totalité de l'adressage logique. Pour éviter que le test ne soit trop long, nous avons défini des *working-set* avec un nombre de pages plus élevé, bien que ce changement engendre évidemment une baisse de l'efficacité du TLB. Le test est composé de 3 *working-set* de 20 pages chaque (pages 0-19, 20-39 et 40-59), sans chevauchement, et 30 requêtes chaque.

Ultimement, il est difficile de tirer des conclusions quant à l'efficacité réelle de l'implémentation de notre TLB et du remplacement de frames puisque qu'elle dépend de plusieurs facteurs inconnus tels que le nombre de pages utilisés dans un intervalle de temps donné et le taux de *page faults* toléré. Nous pensons cependant que cette implémentation offre un ratio d'efficacité pour coût

d'implémentation intéressant dans un contexte d'utilisation tel que décrit plus haut. Malheureusement, ces tests ne permettent pas d'évaluer les gains des d'écriture en mémoire évité grâce à notre implémentation de *CLOCK enhanced*.

4 Conclusion

En conclusion, ce travail s'est révélé plus facile que les précédents grâce aux connaissances acquises quant au fonctionnement des systèmes d'exploitation et à la programmation en C tout au long de la session. Il nous a cependant mis au défi de réfléchir et d'évaluer les répercussions concrètes du choix des algorithmes de remplacement dans le contexte d'utilisation d'une mémoire virtuelle.