Compte rendu TP Pagination

Hoël JALMIN, Mathieu DUMAX-VORZET, Iheb MASTOURA

Les objectifs:

Nanvix suit l'approche de la pagination de la mémoire virtuelle : l'espace d'adressage d'un processus est séparé en pages virtuelles, placées dans des page frames de mémoire physique, dans la RAM ou sur le disque. Le but est de garder dans la RAM les pages les plus utilisées et de les évacuer sur le disque si elles ne sont plus utilisées. Nous cherchons donc à améliorer l'algorithme de remplacement de page, afin d'optimiser les performances: l'objectif est de limiter au maximum le nombre de défauts de pages (quand on doit charger une page depuis le disque dur plutôt que depuis la RAM).

L'algorithme de départ suit un ordre FIFO. Il est peu adapté si on relie plusieurs fois un nombre important de valeurs. Notre fichier de test portant sur de la multiplication de matrices de taille considérable, on met facilement à défaut cet algorithme.

Les algorithmes que nous avons implémenté sont les suivants: Clock, NRU et LRU.

Clock:

L'algorithme de Clock est basé sur le principe de la seconde chance : lorsque l'on cherche à swap-out une page de la RAM au disque, on évacue en priorité les pages les moins accédées. Pour cela, les pages possèdent un bit de deuxième chance (le bit R) qui indique si elles ont été accédées plus d'une fois; et dans ce cas on préfèrera swap-out une page moins utilisée.

Notre algorithme de remplacement de page cherche donc, une fois que le cache des pages est rempli et qu'il faut évacuer une page, la première page appartenant à l'utilisateur courant et n'étant pas partagée. On vérifie ensuite si cette page a une seconde chance, et si c'est le cas on annule la seconde chance (on met de bit de seconde chance à zéro) et on recherche la prochaine page..

Ainsi, notre algorithme regarde les pages les unes après les autres de façon circulaire et évacue la première qui n'a pas de seconde chance, soit dans le pire cas (si toutes les pages avaient une seconde chance), la première page qu'on avait vérifiée et dont on avait annulé la seconde chance.

On a donc la certitude que cet algorithme est plus efficace que l'algorithme FIFO, puisqu'il permet de swap-out en priorité les pages moins utilisées; on aura donc moins besoin de swap-in plus tard quand ces pages seront de nouveau nécessaires.

Cette algorithme est très efficace car il comble le principal défaut de FIFO (qui ne prenait pas en compte la fréquence d'accès) mais reste assez simple pour ne pas alourdir l'étape de sélection. Une optimisation présente dans notre algorithme augmente encore ses performances : tant que la mémoire cache n'est pas pleine, nous ne sommes pas obligé d'étudier les bits d'accès de nos pages pour ajouter une page en mémoire, on cherche simplement un espace non alloué dans notre cache. Cette amélioration permet de remplir notre cache plus rapidement et à moindre coût.

NRU:

Le principe de l'algorithme de NRU est de swap-out les pages en fonction de leur priorité. Pour cela, on utilise les bits M (dirty) et R (accessed) des pages afin de déterminer leur priorité. R est positionné à 1 lorsque la page est référencée et M est positionné à 1 lorsque la page est modifié. Ces bits sont contenus dans chaque entrée du tableau de page, et doivent être mis à jour sur chaque référence en mémoire ; ils sont définis par le matériel.

Régulièrement, une interruption de minuterie déclenche l'effacement des bits référencés de toutes les pages, de sorte que seules les pages référencées dans l'intervalle de minuterie actuel sont marquées avec un bit référencé. Cela permet de savoir si une page a été accédée récemment et pas seulement déjà accéder (pas comme avec le bit de second chance de l'algorithme Clock ci-dessus).

Lorsqu'une page doit être remplacée, le système d'exploitation divise les pages en quatre classes. Elles sont décrites ci-dessous, de la plus à la moins prioritaire à remplacer :

class 0 : non référencé, non modifié
class 1 : non référencé, modifié
class 2 : référencé, non modifié
class 3 : référencé, modifié

La classe 1 peut sembler étrange, un fichier n'a pas été accédé mais quand même été modifié, mais cela s'explique par le rafraîchissement périodique du bit d'accès.

Pour l'implémentation de l'algorithme de NRU, on a défini le macro CLOCK_TICK (en nombre de ticks système), qui indique à quelle fréquence il faut appeler la fonction update_frames_timer, afin de réinitialiser le bit référencé de tout les pages à 0..

Nous avons aussi modifié la fonction allocf, où l'on détermine maintenant le degré de priorité de chaque page frame en fonction de sa classe. Selon leur priorité, on décide donc quelle page va être remplacée. Si on a deux pages de même priorité, on choisit l'une ou l'autre au hasard.

Le fait que cet algorithme classe les différentes pages lui permet de prendre des décisions non plus simplement basées sur la fréquence d'utilisation des pages mais aussi selon la façon dont elles sont utilisées. Cela lui donne une granularité de sélection plus fine que nos autres algorithmes.

Cependant, l'efficacité de cet algorithme reste très dépendante du timer de rafraîchissement. Un timer trop faible entraîne un surcoût de calcul mais un timer trop important empêche une sélection de page à swap out efficace.

Une possible amélioration est donc de calculer en temps réel le timer de rafraîchissement en se basant sur le nombre de défaut de pages, afin d'optimiser nos résultats selon la situation d'exécution.

LRU:

L'algorithme LRU ou *"Last Recently Used"* privilégie les pages récemment accédées (en lecture ou en écriture). Une page est retirée du cache c'est la plus anciennement accédée. On implémente généralement cet algorithme en utilisant une liste chaînée d'élément,classée par ordre d'accès. Cette structure étant complexe à maintenir cohérente et difficile à implémenter sur Nanvix, nous avons utilisé une autre approche.

Nos différentes pages de mémoire en cache sont stockées dans un tableau, nous avons géré un système de vieillissement des pages non accédées : dès qu'une page est accédée on réinitialise son âge (que nous sauvegardons dans un champ).

Les performances de cette version de LRU sont supérieures à celles de la politique de cache FIFO initiale quand il s'agit de réutiliser plusieurs fois les mêmes valeurs (sur notre fichier test, cela correspond à la partie où l'on calcule la matrice résultat) mais se dégrade considérablement quand on a beaucoup de nouvelles valeurs entrantes (phase d'initialisation). Il y a une explication simple à cela: un cache

n'apporte pas un important gain de performance dans ce type de scénario, c'est donc la légèreté de l'algorithme qui prime. Notre LRU étant plus complexe, il est désavantagé dans ce type de situation.

L'inconvénient de notre LRU est que l'on doit en permanence mettre à jour l'âge de nos pages, ce qui oblige à parcourir l'ensemble du tableau à chaque ajout de l'une d'entre elle. Certains de ces parcours pourraient être écourtés: si on a déjà mis à jour toutes les pages stockées et qu'on trouve un emplacement libre pour la nouvelle page, on pourrait s'arrêter. Cela semble particulièrement utile pour la phase de remplissage du cache ou si plusieurs pages sont retirées et mises sur le disque.

C'est cette idée qui a été implémentée dans notre seconde version de LRU. Étonnamment, ce changement n'apporte rien et nuit même légèrement aux performances. On peut supposer que ce n'est pas sur ces parcours supplémentaires que notre algorithme perd beaucoup de temps, cette amélioration n'est donc pas pertinente et rajoute seulement de la complexité.

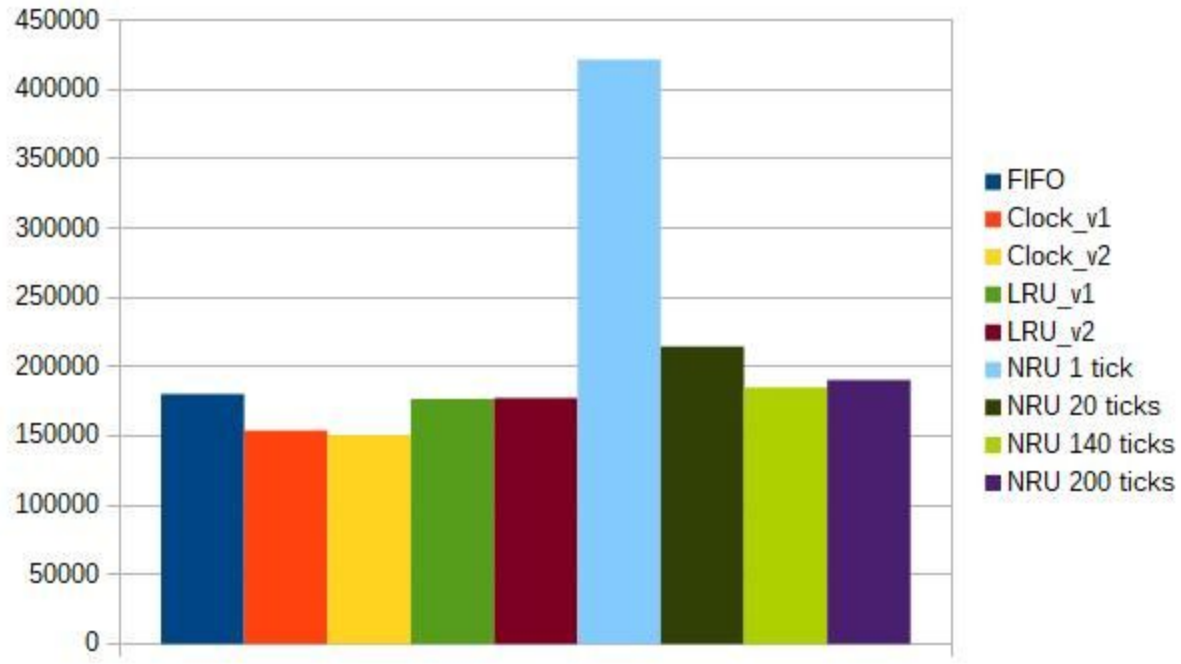
Une autre idée testée fût que si une page est accédée, on ne remet pas son âge à 0, on ne l'a fait juste pas vieillir. Elle fût rapidement abandonnée, générant beaucoup plus de défauts de pages que les versions précédentes, les pages récentes étant trop avantagées par rapport aux autres pages régulièrement accédées. Pareillement, "rajeunir" une page accédée récemment ne semble toujours pas la solution, même si les performances sont légèrement meilleures.

En conclusion notre implémentation, utilisant un compteur par page, n'est pas assez précise pour obtenir de bonnes performances: notre système se retrouve avec trop de pages d'âge similaire pour être efficace. Il existe 2 implémentations pouvant résoudre ce problème, quoique complexes à mettre en place :

- Détecter dès qu'un accès à une page se produit et mettre à jour l'âge de toutes les pages (nécessite des instructions très proches du code machine)
- Implémenter la structure en liste chainée décrite au début (complexe sans l'utilisation de malloc et lourde à maintenir)

Voici ci-dessous le résumé des performances de nos algorithmes, selon le test fourni de multiplication de matrices. Nous avons fait un diagramme représentatifs des temps totaux pour finir le test.

	FIFO	Clock_v1	Clock_v2	LRU_v1	LRU_v2	NRU 1 tick	NRU 20 ticks	NRU 140 ticks	NRU 200 ticks
Init time	22900	18400	16800	34000	34100	38100	37500	36300	38200
Calculation time	155600	133200	131900	139700	140100	383000	175300	146700	148300
Checking time	2000	2400	2200	3200	3300	1000	1800	1900	3900
Total	180500	154000	150900	176900	177500	422100	214600	184900	190400



Nous voyons donc sur ce graphique que l'algorithme le plus efficace que nous ayons implémenté est la seconde version de clock, et que même la meilleure version de NRU n'a pas de meilleures performances que FIFO.