Eric Larsen
Jean-Philippe Raymond
IFT2245
19 avril 2015

Travail pratique 3
Rapport d'activité

1 Architecture et simplifications du modèle de gestion de mémoire

L'architecture de mémoire abstraite définie par le devis se caractérise par l'usage d'un mécanisme de pagination sur demande muni d'une table de pagination à un seul niveau et d'une cache de table de pagination (en anglais 'translation look aside buffer' ou 'TLB') à un seul niveau dont les entrées sont remplacées suivant l'une ou l'autre des règles 'premier entré-premier sorti' (en anglais 'first in-first out' ou 'FIFO') ou 'moins récemment utilisé' (en anglais 'least recently used' ou 'LRU'). Cette abstraction représente la mémoire physique, la table de pagination et la cache de pagination par des tableaux d'objets et le dépôt (en anglais 'backing store') par un fichier. Elle comporte notamment les simplifications suivantes:

- Seules des demandes de lecture de caractères ASCII individuels dans la mémoire physique et leur écriture initiale dans cette dernière à partir du dépôt sont modélisées : la réécriture d'objets de taille fixe ou variable dans la mémoire n'est pas prévue.
- Le mécanisme de pagination sur demande est de type paresseux : la prépagination n'est pas modélisée.
- La modélisation d'une réserve de cadres de mémoire physique disponibles n'est pas requise. (La solution proposée comporte toutefois une telle réserve.)
- Le nombre de pages de la mémoire virtuelle est égal au nombre de cadres de la mémoire physique de sorte que l'entreposage et le remplacement des cadres de la mémoire dans et depuis le dépôt ne sont pas modélisés. Les possibilités de remplacement synchrone ou asynchrone des cadres, d'inclusion de bits de validité des cadres, d'entreposage anticipé des cadres susceptibles de remplacement et de remplacement par grappes de cadres sont donc sans objet.
- La table de pagination de même que la cache de table de pagination comportent un seul niveau et les pages virtuelles sont de taille unique et fixe.
- Le partage de la mémoire entre des processus concurrents n'est pas modélisé de sorte que l'affectation de la mémoire entre des processus, la multiplicité des tables de pagination et la réallocation et la réécriture de la mémoire lors des permutations des processus (en anglais 'swap') sont sans objet.
- La différenciation des vitesses d'accès à la mémoire dépendamment de la localisation des cadres n'est pas modélisée.

2 Fonctionnement de l'application

Outre la classe *VirtualMemoryManager* qui est responsable de la gestion du modèle de mémoire et assure les principales fonctionnalités de l'application, l'application comporte deux classes fondamentales, *PhysicalMemory* et *TLB*.

La classe *PhysicalMemory* décrit la structure de la mémoire physique et les opérations qui la concernent, soit la lecture dans un cadre et au décalage spécifiés (avec *getValueFromFrameAndOffset*), l'identification d'un cadre libre (avec *findFreeFrame*) et la récupération directe et en bloc d'une page depuis le dépôt et sa copie dans un cadre de la mémoire (avec *demandPageFromBackingStore*). Une pile des numéros des

cadres de mémoire libres est constituée et mise à jour afin d'accélérer la recherche des cadres libres avec *findFreeFrame*.

La classe *TLB* décrit la structure de la cache de la table de pagination et les opérations qui la concernent, soit la recherche linéaire de l'entrée de la cache correspondant à un numéro de page spécifié (avec *findPage*) et l'ajout d'une entrée dans la cache, précédé si nécessaire par l'éviction d'une entrée existante suivant l'une ou l'autre des règles LRU et FIFO (avec *addEntry*). La différence de traitement entre LRU et FIFO est prise en charge par la méthode *findPage* suivant la valeur attribuée au paramètre *TLB_MODE* apparaissant dans le fichier c*ommon*. La classe *TLB* contient également les méthodes *findPageSearchMap* et *addEntrySearchMap* qui, au contraire de *findPage* et *addEntry*, utilisent une table de hachage permettant un accès quasi-direct à l'entrée désirée de la cache de table de pagination. Ces méthodes ont été introduites pour permettre la comparaison des performances des deux types d'accès et ne sont pas utilisées lors du fonctionnement normal de l'application.

La classe *VirtualMemoryManager* est responsable de la logique du modèle de gestion de mémoire de même que de la comptabilité, de l'affichage et de l'entreposage des résultats de son fonctionnement. Celle-ci instancie la mémoire physique, la table de pagination (sous forme d'un tableau de pages virtuelles de la classe imbriquée *Page*) et la cache de la table de pagination. Outre *printResults*, elle définit deux méthodes importantes: D'une part, *addCommandFromLogicAddress* entrepose dans la liste *commandList* l'adresse décimale en mémoire virtuelle d'un objet dont la lecture en mémoire physique est désirée. D'autre part, *applyCommands* parcourt la liste d'adresses décimales *commandList* et, après s'être assuré de leur validité, effectue les opérations d'accès à la mémoire appropriées: Le constructeur de la classe imbriquée *Command* est d'abord invoqué afin de séparer l'adresse de mémoire décimale entre le numéro de page virtuelle et le décalage correspondants. Lorsque le numéro de la page virtuelle requise est identifié à une entrée de la cache de la table de pagination par *findPage*, l'objet spécifique requis est lu en mémoire physique par *getValueFromFrameAndOffset* dans le cadre indiqué par la cache. Dans l'alternative, la cache doit être mise à jour par *addEntry* suivant la règle de remplacement choisie, LRU ou FIFO. Au préalable, toutefois, lorsque l'adresse de la page virtuelle requise ne correspond pas à une page valide de la table de pagination, sa récupération depuis le dépôt et sa copie dans un cadre libre de la mémoire physique identifié par *findFreeFrame* est effectuée par *demandPageFromBackingStore*. Finalement, l'objet désiré est lu par *getValueFromFrameAndOffset* dans le cadre de la mémoire physique indiqué par la table.

Veuillez noter ce qui suit : L'**instruction de compilation** de l'application se trouve dans le fichier *makefile* de sorte que la commande *make* peut être utilisée pour la compilation et que le nom du **fichier de sortie** est *tp3*. Le fichier *main* contient le programme principal où est spécifié le nom du fichier d'adresses de mémoire dont la lecture est désirée.

3 Stratégies de remplacement implémentées : propriétés et comparaisons

Qu'il soit utilisé pour le remplacement dans la cache de la table de pagination ou dans la mémoire, le choix d'un algorithme doit poursuivre deux objectifs concurrents, à savoir la minimisation du taux de défaut (en anglais 'miss rate' ou 'fault rate') et la minimisation des ressources de calcul requises pour identifier l'objet du remplacement et pour assurer le fonctionnement de l'algorithme. Bien qu'il ne soit pas réalisable puisqu'il requiert la connaissance des accès futurs à la mémoire, l'algorithme de remplacement dit 'optimal' (ou 'minimal') fournit une borne inférieure pour le taux de défaut par rapport à laquelle les algorithmes réalisables peuvent être comparés. Parmi ceux-ci l'algorithme du LRU présente habituellement un taux de défaut parmi les plus bas, au prix cependant de la mise en œuvre de ressources de calcul importantes pour identifier l'objet du remplacement et assurer le fonctionnement de l'algorithme. Au contraire, l'algorithme du FIFO requiert peu de ressources pour son fonctionnement mais présente un

piètre taux de défaut. En outre, a contrario de l'algorithme du LRU, l'algorithme du FIFO est affecté par l'anomalie de Belady : l'augmentation de la taille de la cache peut conduire à une détérioration du taux de défaut.

Les algorithmes dits 'de la deuxième chance' ou 'de l'horloge' ou qui plus généralement comportent des bits de référence à mettre à jour périodiquement constituent des approximations de l'algorithme du LRU qui visent à atteindre des taux de défaut comparables tout en obviant à ses exigences de ressources de calcul élevées. Finalement, l'algorithme de l'utilisation la moins fréquente (en anglais 'least frequently used' ou 'LFU') présente généralement un taux de défaut moins avantageux que le LRU ou ses approximations tout en requérant des ressources de calcul qui demeurent importantes. Son usage est donc peu avantageux. Cela est peu étonnant puisque le nombre cumulatif d'accès passés fournit une piètre approximation des accès à venir, notamment parce que des accès nombreux à certaines pages survenant lors de la mise en place d'un processus ne sont pas répétés par la suite. Pour cette raison, lorsque le LFU est implémenté, les statistiques d'accès sur lesquelles celui-ci repose sont réinitialisées régulièrement.

Nous avons choisi de présenter une comparaison contrastée entre les deux algorithmes réalisables dont les propriétés sont symétriquement opposées, soit LRU qui présente habituellement les taux de défaut les plus avantageux mais requiert les ressources de calcul les plus importantes pour son fonctionnement et FIFO qui au contraire présente haibituellement les taux de défaut parmi les moins avantageux et requiert les plus faibles ressources de calcul.

Deux implémentations des règles d'éviction du FIFO et du LRU sont envisageables. La première repose sur l'usage d'une file doublement chaînée de pages virtuelles (ou de leur numéro). Ainsi, suivant la règle FIFO, les numéros des pages virtuelles sont placés à l'arrière de la file lors de leur inclusion dans la cache et progressent vers l'avant au fur et à mesure des inclusions successives tandis que suivant la règle LRU les numéros des pages sont aussi placés à la fin de la file lors de leur consultation dans la cache. La page dont le numéro est à l'avant de la file est évincée de la cache. La seconde repose sur un compteur dont la valeur est incrémentée et associée à une page lors de son inclusion (FIFO et LRU) ou lors de sa consultation dans la cache (LRU seulement). L'entrée de la cache associée à la plus faible valeur de compteur est évincée. La solution reposant sur une file requiert davantage d'opérations pour effectuer les mises à jour nécessaires à son fonctionnement mais permet toutefois d'obvier à la renumérotation qui doit survenir lorsque le compteur atteint sa valeur maximale. Elle permet également d'accéder directement à la page choisie pour l'éviction, a contrario de la solution reposant sur un compteur qui requiert une recherche parmi les entrées de la table.

Quatre implémentations de la recherche et du remplacement dans la cache de table de pagination ont été réalisées: recherche linéaire (avec *findPage* et *addEntry*) ou encore quasi-directe (avec *findPageSearchMap* et *addEntrySearchMa*p) dans la cache et usage d'une file ou encore d'un compteur pour implémenter les règles de remplacement. Des essais chronométrés ont révélé que la recherche quasi-directe dans la cache était moins avantageuse dans le contexte du modèle à l'étude (Rappelons que la cache de pagination comporte un très petit nombre d'entrées (16) de sorte que le temps de mise à jour supplémentaire associé à un dictionnaire de hachage présente peu de chances d'être justifié par des gains en temps de recherche.). Des essais ont également révélé que l'usage d'un compteur ou d'une file pour implémenter les règles de remplacement présentaient des temps comparables. Nous avons choisi l'usage d'une file pour sa plus grande élégance et n'incluons que cette solution pour alléger le code. Bien entendu, de tels essais comparatifs fondés sur une interprétation programmatique particulière d'un modèle de mémoire abstrait ne constituent pas des simulations réalistes et généralisables du fonctionnement d'une mémoire réelle et sont simplement utiles à titre d'illustration des conséquences importantes des choix d'implémentation particuliers.

Les résultats des simulations effectuées à partir des lots d'adresses indépendantes et dépendantes repectivement contenus dans les fichiers *addressRandom.txt* et *address.txt* sont rapportés dans le tableau ci-dessous. Celui-ci indique le taux de succès de la cache (en anglais 'translation look aside buffer hit rate'), le taux d'échec de la table (en anglais 'page fault rate') et le temps de calcul moyen en fonction de l'algorithme de remplacement et du type de lot d'adresses. En raison de l'usage d'un mécanisme de pagination sur demande paresseux et de l'égalité des nombres de pages et de cadres, le nombre d'échecs de la table de pagination sera simplement égal au nombre de pages référencées dans le fichier d'adresses. La très faible différence des taux de succès de la cache entre les règles LRU et FIFO **lorsque les adresses sont indépendantes** est en accord avec l'attente selon laquelle cette différence est dans ce cas spécifique statistiquement non significative et tend vers zéro lorsque le nombre d'accès à la mémoire augmente. Tel qu'attendu, le temps moyen de calcul est plus élevé lorsque la règle LRU est appliquée.

résultats des simulations

algorithme	adresses indépendantes		adresses dépendantes		
	taux de succès cache ('TLB hit rate')	taux d'échec table ('page fault rate')	taux de succès cache ('TLB hit rate')	taux d'échec table ('page fault rate')	temps de calcul moyen (sec) (écart-type)
LRU	4337/70000	256/70000	974/1000	18/1000	.00101 (.00030)
FIFO	4353/70000	256/70000	943/1000	18/1000	.00055 (.00015)

4 Extension : nombre de pages virtuelles supérieur au nombre de cadres

Une extension du modèle de gestion de mémoire à l'étude pour permettre un nombre de pages virtuelles supérieur au nombre de cadres de mémoire physique disponibles requerrait bien entendu de prévoir le remplacement de cadres à partir du dépôt. Sous l'hypothèse que toutes les autres caractéristiques et simplifications du modèle de mémoire décrites à la section 1 demeureraient inchangées et notamment que les réécritures en mémoire continueraient d'être exclues de la modélisation, les remplacements de pages dans les cadres de la mémoire physique seraient analogues aux remplacements présentement effectués dans la cache de la table de pagination virtuelle et pourraient appliquer l'une ou l'autre des règles de remplacement LRU ou FIFO en introduisant une liste doublement chaînée de numéros de cadres de mémoire, disons *lruFifoFrameQueue*, dans la classe *PhysicalMemory*. Pour implémenter cette extension, il suffirait en outre pour l'essentiel de modifier les méthodes *applyCommands* et *findFreeFrame* des classes *VirtualMemoryManager* et *PhysicalMemory* et d'ajouter une méthode responsable de la sélection du cadre à remplacer, disons *selectFrame*, dans la classe *PhysicalMemory*.

Ainsi, la logique de la méthode *applyCommands* serait modifiée comme suit: Lorsqu'un numéro de page virtuelle serait disponible en mémoire physique d'après la recherche dans la table de pagination et que la règle de remplacement LRU serait en vigueur, *applyCommands* mettrait à jour la file *lruFifoFrameQueue* servant à l'implémentation de cette règle. Au contraire, lorsque le numéro de page serait indisponible dans la mémoire physique d'après la recherche dans la table de pagination, la méthode *findFreeFrame* serait appelée pour identifier un cadre disponible et soit retourner son numéro en cas de disponibilité ou soit encore retourner *-1* en cas d'indisponibilité. Dans cette dernière éventualité, *applyCommands* appellerait la méthode *selectFrame* qui retournerait le numéro du cadre situé à l'avant de la file *lruFifoFrameQueue* après avoir mis celle-ci à jour. Finalement, *applyCommands* appellerait alors *demandPageFromBackingStore* pour effectuer l'insertion de la page désirée dans le cadre de mémoire indiqué. Le reste de la logique de *applyCommands* serait essentiellement inchangé.