

NACHOS - Pagination

Manon Philippot - Anthony Delassalle

22 Décembre 2017

1 Bilan

L'objectif de ce devoir est de mettre en place une gestion mémoire paginée simple au sein d'un système.

1.1 Adressage virtuel par une table des pages

La machine MIPS travaille en adressage virtuel. Lors de ce devoir nous nous sommes intéressés uniquement au mécanisme de la table des pages. Dans ce mécanisme, les programmes sont découpés en bouts que l'on nomme pages, de taille fixe. La mémoire physique est elle aussi découpée en pages de même taille. La table des pages permet la traduction de l'adresse virtuelle en adresse physique, tâche utile lors des accès mémoire. La table des pages est initialisée dans *userprog/addrspace.h* et *userprog/addrspooace.cc*. La conversion entre adresse virtuelle et physique est effectuée dans *machine/translate.cc*.

Lors de ce devoir, nous avons tout d'abord chargé le programme en mémoire en décalant tout d'une page. Pour ce faire nous nous sommes inspiré de la fonction *executable->ReadAt*. Celle-ci lit *numBytes* octets depuis la position *position* dans le fichier *executable* et utilise directement la mémoire MIPS : on le voit au fait qu'elle utilise directement *machine->mainMemory* qui correspond à la mémoire physique dédiée au programme de l'utilisateur (cf *machine.h*). A partir de cette fonction nous avons défini dans *userprog/addrspace.cc* une nouvelle fonction locale *static void ReadAtVirtual(...)* faisant la même chose que *ReadAt* mais en écrivant dans l'espace d'adressage virtuel défini par la table des pages *pageTable* de taille *numPages*. Dans cette fonction, on utilise un tampon temporaire que nous remplissons avec *ReadAt*, puis on sauvegarde la table des pages actuelle avant de copier le tampon en mémoire octet par octet avec *WriteMem*, et enfin on restore l'ancienne page des tables en s'inspirant de *space->restoreState*. On remplace ensuite *ReadAt* par *ReadAtVirtual* dans le constructeur de *Addrspooace* et on modifie la création de la table des pages pour que la page virtuelle *i* soit une projection de la page physique *i+1* (*pageTable[i].physicalPage = i+1;*). Le programme se charge désormais en mémoire en décalant tout d'une page. Tout marche, et les threads utilisateurs s'exécutent normalement.

Plus généralement, il est utile d'encapsuler l'allocation des pages physiques dans une classe spéciale *PageProvider* globale à tout NachOS. Puisque l'on réutilise les pages physiques, cette classe doit mettre à zéro le contenu des pages allouées. On crée donc la classe *PageProvider* dans *pageprovider.cc* et *pageprovider.h* que l'on ajoute lors de la compilation

dans *Makefile.common*. Cette classe, qui s'appuie sur la classe *Bitmap*, recense l'ensemble des pages physiques disponibles et renvoie des pages physiques libres et initialisées à 0. Elle dispose de quatre fonctions : *GetEmptyPage* qui récupère le numéro de la première page libre et l'initialise à 0 par la fonction *memset*; *GetRandomEmptyPage* qui récupère le numéro d'une page libre au hasard et l'initialise à 0 de la même façon; *ReleasePage* qui libère la page définie par *n*; et *NumAvailPage* qui renvoie le nombre de pages disponibles. Lorsqu'aucune page est disponible, la méthode *find* du bitmap *pagesBitmap* peut renvoyer -1 : nous avons traité ce problème grâce à un *ASSERT*. Il faut un seul objet de cette classe puisqu'elle gère l'intégralité des pages physiques. En effet nous n'avons pas besoin d'une deuxième instance qui gère les pages. Pour cela nous créons l'instance de *PageProvider* en même temps que la machine dans *Initialize (sytem.h/.c)* et la détruisons lors du *Cleanup*. Il faut maintenant corriger le constructeur et destructeur d'*AddrSpace* pour utiliser ces primitives : il suffit de faire appel à la fonction *getEmptyPage* ou bien *GetRandomPage* (pour tester et stresser l'implémentation) afin d'allouer une page grâce à notre *PageProvider* dans le constructeur et de désallouer les pages dans le destructeur grâce à *ReleasePage*. L'allocation de pages grâce à notre *PageProvider* marche, que ce soit avec *GetEmptyPage* ou *getRandomEmptyPage*.

1.2 Exécuter plusieurs programmes en même temps

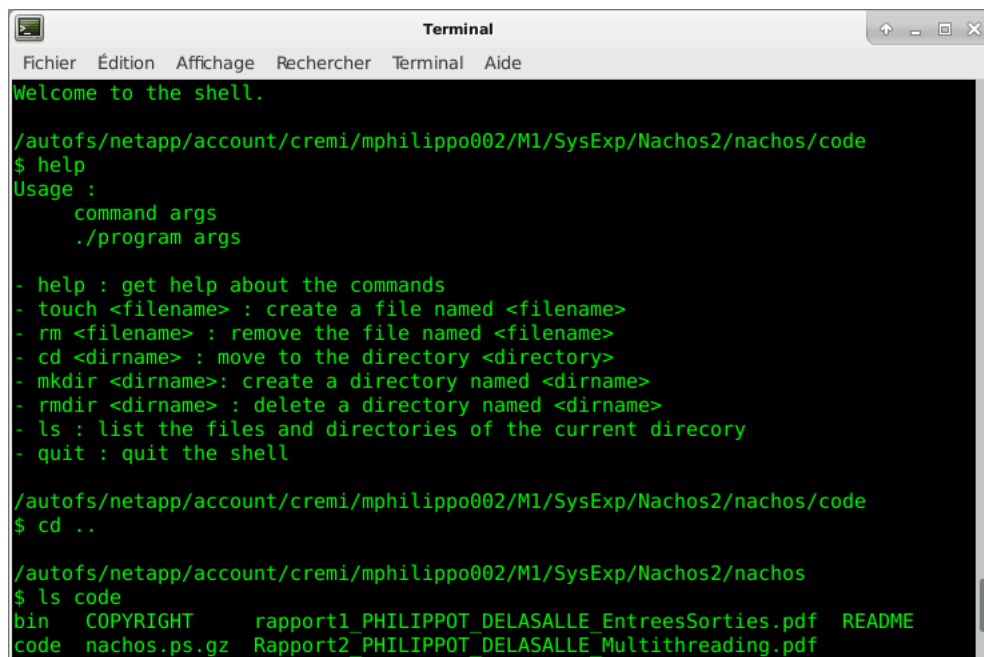
Puisque désormais seule une partie de la mémoire physique est utilisée pour projeter les pages virtuelles, on va désormais conserver dans la mémoire plusieurs programmes en même temps.

Pour ce faire on met en place appel système *int ForkExec(const char *s)* qui prend un nom de fichier exécutable, crée un objet *AddrSpace* à partir de ce fichier exécutable, et crée un thread noyau lançant ensuite l'exécution du nouveau processus, en parallèle avec le processus père. Le traitement d'interruption de cet appel système doit récupérer la chaîne de caractères correspondant au nom du fichier exécutable, puis lancer la fonction *do_ForkExec* implémentée par nos soins. Cette dernière est implémentée dans *userthread.cc* afin de pouvoir implémenter la variable static *nbThread* à chaque création de Thread par le Fork. L'intérêt est que désormais lorsque le dernier processus créé (que ce soit par *do_ThreadCreate* ou *do_ForkExec*) s'arrête, un appel à *Halt* est effectué automatiquement. La création de processus par l'intermédiaire de *ForkExec* marche bien, et le programme courant ainsi que le programme lancé peuvent eux-même contenir des threads (voir tests). La mémoire est donc partagée entre plusieurs programme. Par manque de temps, nous n'avons pas encore implémenté la libération immédiate de toutes les ressources que les threads résultant de *ForkExec* utilisent (sa structure *space*, etc).

Les questions bonus II.5, II.6 et II.7 n'ont pas été traitées par manque de temps. Cependant nous avons réfléchi à la question 5 : nous aurions pu résoudre ce problème en créant pour chaque processus un tableau avec les références vers les threads du processus. Ainsi lorsqu'un thread de cette liste fait appel à *EXIT*, il suffit de terminer tous les autres threads de cette liste.

1.3 Bonus : shell

Nous avons implémenté un tout petit shell dans le fichier *test/littleshell.c*. Pour l'exécuter, il suffit de taper la commande *./userprog/nachos -x ./test/littleshell* lorsque situé dans le dossier *code* de NachOS. Ce petit shell permet grâce à des appels systèmes et le terminal d'effectuer des actions basiques : lancer un programme (*./nomDuProgramme*), créer un document (*touch/SC_Touch*), créer un dossier (*mkdir/SC_Mkdir*), se déplacer dans les dossier (*cd/SC_Cd*) tout en affichant l'arborescence (*SC_PrintCurrentDir*), supprimer un document (*rm/SC_Rm*), supprimer un dossier (*rmdir/SC_Rmdir*), lister les fichiers/dossiers du dossier actuel (*ls/SC_Ls*) et la commande *echo*. La commande *help* permet d'obtenir un récapitulatif des usages et commandes, et la commande *quit* permet de quitter le shell *littleshell*.



```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide

Welcome to the shell.

/autofs/netapp/account/cremi/mphilippo002/M1/SysExp/Nachos2/nachos/code
$ help
Usage :
  command args
  ./program args

- help : get help about the commands
- touch <filename> : create a file named <filename>
- rm <filename> : remove the file named <filename>
- cd <dirname> : move to the directory <directory>
- mkdir <dirname>: create a directory named <dirname>
- rmdir <dirname> : delete a directory named <dirname>
- ls : list the files and directories of the current directory
- quit : quit the shell

/autofs/netapp/account/cremi/mphilippo002/M1/SysExp/Nachos2/nachos/code
$ cd ..

/autofs/netapp/account/cremi/mphilippo002/M1/SysExp/Nachos2/nachos
$ ls code
bin  COPYRIGHT  rapport1 PHILIPPOT DELASALLE EntreesSorties.pdf  README
code nachos.ps.gz Rapport2 PHILIPPOT DELASALLE Multithreading.pdf
```

Figure 1: Exemple du shell implémenté

2 Points délicats

Le point le plus délicat a sûrement été l'implémentation de la fonction *ReadAtVirtual*. En effet il fallait faire attention à bien changer temporairement de table des page dans la machine, afin que *WriteMem* utilise bien la table des pages construite dans le constructeur *AddrSpace*. Pour ce faire nous avons sauvegardé l'ancienne table des pages dans un pointeur *oldPageTable* de type *TranslationEntry* puis nous l'avons restauré en nous inspirant de *space->restoreState*. Un autre point délicat a été les problèmes d'inclusions multiples lorsque nous souhaitions créer l'instance de *PageProvider* dans *system.h/c*.

3 Limitations

La première limitation est l'utilisation dans `ReadAtVirtual` de `WriteMem`, qui recopie le tampon en mémoire octet par octet : c'est peu efficace.

Ensuite, nous n'avons pas encore implémenté la libération immédiate de toutes les ressources que les threads résultant de *ForkExec* utilisent (sa structure *space*, etc), ce qui peut causer des fuites mémoires. Cela peut être particulièrement gênant sur une machine ne possédant que très peu de ressources matérielles.

4 Tests

Nous avons implémenté les programmes suivants :

- **bigNumberProcess.c** : crée 12 processus ayant chacun son propre espace d'adressage par l'intermédiaire de *ForkExec*. Chaque processus exécute le code de *bigNumberThreads.c*.
- **bigNumberThreads.c** crée 12 threads partageant l'espace mémoire du processus père. Chaque thread doit afficher la chaîne de caractère *abcd*.

Commande test 1 : `./userprog/nachos -x ./test/bigNumberProcess` depuis le dossier code de NachOS.

Lorsqu'on exécute *bigNumberProcess* faisant appel à *bigNumberThreads*, on s'attend à voir s'écrire $12 \times 12 \times 4 = 576$ caractères dans le terminal, ce qui est le cas. Les deux tests ci-dessus sont concluant et montrent que notre implémentation marche.

Et les programmes suivants :

- **userpage0.c** : lance un thread qui écrit 10 fois le caractère a et b.
- **userpage1.c** : lance un thread qui écrit 10 fois le caractère c et d.
- **pagination.c** : test de l'exercice II.3. Créé 2 processus avec *ForkExec*, le premier exécutant *userpage0* et le deuxième *userpage1*.

Commande test 2: `./userprog/nachos -x ./test/pagination` depuis le dossier code de NachOS.

Lorsqu'on exécute *pagination.c*, on s'attend à voir s'afficher dans le terminal 10 fois le caractère a, b, c et d, ce qui est le cas. Cela montre donc que le programme courant et le programme lancé peuvent eux-mêmes contenir des threads.