Projet Système : Nachos et la pagination

Etape 4, Master Informatique 1 2003–2004

15 Juin 2004

L'objectif de cette étape est de constituer un pas de plus vers l'implantation en Nachos d'un modèle de *processus* à la manière d'Unix, chaque processus pouvant contenir un nombre arbitraire de threads. Plus précisément, il s'agit de mettre en place une gestion mémoire paginée simple au sein du système.

Partie I. Adressage virtuel par une table de pages

L'ensemble de la machine MIPS travaille en adressage virtuel, selon deux mécanismes au choix de l'utilisateur : la *table des pages* ou le TLB. On ne s'intéressera qu'au mécanisme de la table des pages. Regarder comment elle est utilisée dans `userprog/addrspace.h`. Une adresse virtuelle est composée d'un numéro de page et d'un décalage dans la page. À chaque numéro de page virtuel, la table associe un numéro de cadre (*frame*, appelé aussi *page* par abus de langage) physique. L'adresse (physique) du cadre et le décalage dans la page déterminent l'adresse physiquement accédée. Le mécanisme est implémenté par la fonction

```
Translate(int virtAddr, int* physAddr, int size, bool writing)
```

dans le fichier `machine/translate.cc`. Tous les accès à la mémoire dans l'interprète se font au travers de cette fonction. Regarder le fonctionnement de `WriteMem` et `ReadMem`.

Action I.1. *Écrire un petit programme de test `test/userpages0` qui lance un ou deux threads utilisateurs et écrit quelques caractères entrelacés à l'écran.*

Nous allons maintenant charger le programme en mémoire en décalant tout d'une page.

Action I.2. *Examiner soigneusement le fonctionnement de `executable->ReadAt` dans la fonction `AddrSpace::AddrSpace` de `userprog/addrspace.cc`. Curieusement (?), elle écrit* directement *en mémoire physique. À quoi voit-on cela ?*

Action I.3. *Définir une nouvelle fonction* locale

*static void ReadAtVirtual(OpenFile *executable, int virtualaddr, int numBytes, int position, TranslationEntry *pageTable, unsigned numPages)*

qui fait la même chose que `ReadAt`, mais en écrivant dans l'espace d'adressage virtuel défini par `pageTable` et `numPages`. Vous pouvez utiliser un tampon temporaire que vous remplirez avec `ReadAt`, puis que vous recopierez en mémoire avec `WriteMem` par exemple...

Utilisez `ReadAtVirtual` en lieu et place de `ReadAt`, et vérifiez que l'exécution de programmes fonctionne toujours.

Action I.4. *Modifiez la table des pages pour que la page virtuelle i soit projetée (*mapped*) sur le cadre physique $i+1$.*

Relancez votre programme. Tout doit marcher, et les threads utilisateurs doivent s'exécuter normalement !

Observez les traductions d'adresse avec l'option de trace `-d a`.

Plus généralement, il est utile d'encapsuler l'allocation des pages physiques aux pages virtuelles dans une classe spéciale `FrameProvider`. Notez que c'est alors à cette classe de remettre à zéro les pages fournies, et non plus au constructeur de `AddrSpace` !

Action I.5. *Créer une classe* `FrameProvider` *dans le fichier* `userprog/frameprovider.cc` *qui s'appuie sur la classe* `Bitmap` *pour gérer les cadres. Elle permet : 1) de récupérer un cadre libre* et initialisé à `0` par la fonction `bzero` *(*`GetEmptyFrame`*); 2) de libérer un cadre obtenu par* `GetEmptyFrame` *(*`ReleaseFrame`*); 3) de demander combien de cadres restent disponibles (*`NumAvailFrame`*). Notez que la politique d'allocation des cadres est complètement locale à cette classe.*

Action I.6. *Restructurer la fonction de création* `AddrSpace` *pour utiliser ces primitives, et faites tourner votre programme avec diverses stratégies d'allocation. Par exemple, allouer les cadres par un tirage aléatoire !*

Vérifiez bien que tous vos programmes fonctionnent toujours.

Partie II. Exécuter plusieurs programmes en même temps...

Puisque seule une partie de la mémoire physique est utilisée pour projeter les pages virtuelles, pourquoi ne pas conserver dans la mémoire *plusieurs* programmes en même temps ?

Action II.1. *Définir un appel système* `int ForkExec(char *s)` *qui prend un nom de fichier exécutable, crée un thread au niveau du système Nachos et lance l'exécution du programme propulsée par ce thread, en parallèle avec le programme courant. Notez que le programme courant et le programme lancé peuvent eux-mêmes contenir des threads (lancé par une fonction du niveau MIPS) ! Le programme suivant doit donc fonctionner :*

```
#include "syscall.h"

main()
{
  ForkExec("../test/userpages0");
  ForkExec("../test/userpages1");
}
```

avec pour `userpages0` *et* `userpages1` *des programmes du genre*

```
#include "syscall.h"
#define THIS "aaa"
#define THAT "bbb"

const int N = 10; // Choose it large enough!

void puts(char *s)
{
  char *p; for (p = s; *p != '\0'; p++) PutChar(*p);
}

void f(void *s)
{
  int i; for (i = 0; i < N; i++) puts((char *)s);
}

main()
{
  int i;
  UserThreadCreate(f, (void *) THIS);
  f((void*) THAT);
```

```
}
```

Action II.2. *Raffiner votre implémentation pour que le dernier processus arrête la machine par un appel à* `Halt()`*. Tout processus qui termine et n'est pas le dernier doit libérer* immédiatement *toutes les ressources qu'il utilise (sa structure* `space`*, etc.).*

Action II.3. *Montrer que vous pouvez lancer un grand nombre de processus (disons une douzaine), chacun avec un grand nombre de threads (une douzaine aussi).*

Action II.4. *Montrer en surveillant la consommation des ressources de votre processus Unix propulseur que les processus MIPS / Nachos libèrent effectivement bien leurs ressources une fois terminés. Pour cela, il vous sera certainement utile de redéfinir les opérateurs* `new` *et* `delete` *de C++.*

Partie III. Bonus : shell

Action III.1. *Implémentez un tout petit shell en vous inspirant du programme* `test/shell.c`*.*

Partie IV. Bonus++ : Malloc et free !

Pour le moment, la taille d'un processus est calculée statiquement au lancement du programme à partir des données fournies par le compilateur, en réservant un espace arbitraire pour la pile. Toutes les pages de l'espace d'adressage du processus sont *valides*. On peut écrire par exemple n'importe où dans la zone de pile (faites un essai !), et même dans le code (essayez, si, si !)

Pour implémenter `malloc`, il faut aussi réserver une partie de l'espace d'adressage additionnelle, entre le code et la pile. Par contre, écrire dans cet espace *doit* être interdit, sauf allocation explicite : les pages doivent exister, mais être *invalides* (et même probablement non allouées). En Unix, c'est l'appel système `sbrk` qui modifie la limite entre la zone allouée et la zone non allouée : `man sbrk`.

Pour simplifier, on travaille par page. On cherche à définir un appel système `AllocEmptyPage` qui renvoie un pointeur vers une nouvelle page fraîche, et `FreePage` qui libère cette page.

Action IV.1. *Ajouter dans la classe* `AddrSpace` *une variable privée* `brk` *et un* appel système `Sbrk(unsigned n)` *qui alloue n nouvelles* `frame`*s* vides *pour l'espace d'adressage (processus), valide les n pages suivantes à partir de celle désignée par* `brk` *et les associe à ces cadres, incrémente* `brk` *de n et retourne la nouvelle valeur de* `brk`*.*

Précisez soigneusement le traitement des cas d'erreurs.

Action IV.2. *Implémenter une* fonction utilisateur `void *AllocEmptyPage()` *retourne un pointeur (dans le monde MIPS !) sur une zone de mémoire initialisée à zéro d'une page. Faites quelques tests.*

Action IV.3. *Implémenter une* fonction utilisateur `void FreePage(void *p)` *qui libère la page suivant l'adresse p supposée avoir été retournée par un appel à* `void *p = AllocEmptyPage()`*.*

Note *Attention :* `free` *doit libérer les cadres utilisés et invalider les pages. Vérifiez ce point par quelques tests : accéder à une zone libérée doit déclencher une exception.*

Action IV.4. *Idée bonus : généralisez la fonction* `void *AllocEmptyPage()` *pour allouer n pages* consécutives *dans l'espace d'adressage. Implémentez les fonctions de bibliothèque* `malloc` *et* `free` *dans le monde MIPS en vous appuyant sur ces fonctions. Exécutez des petits programmes utilisant* `malloc` *et* `free`*...*