NACHOS : Multithreading

Etape 3, Master Informatique 2003–2004

9 Juin 2004

L'objectif de cette etape est de permettre d'exécuter des applications multi-threads sous Nachos.

Partie I. Mise en place des threads *utilisateurs*

Il s'agit maintenant de rendre accessibles les threads Nachos depuis les programmes utilisateurs, comme `putchar`. Dans notre cadre, chaque thread "utilisateur" sera directement supporté par un thread Nachos.

Action I.1. *Examinez en détail le fonctionnement des threads Nachos. Comment ces threads sont-ils alloués et initialisés ? Où se trouve la pile d'un thread Nachos, en tant que thread* noyau *? Et la pile de la copie de l'interprète MIPS (c'est-à-dire du thread* utilisateur*) qu'il exécute ? À quoi servent les fonctions* `SaveState` *et* `RestoreState` *de* `userprog/addrspace.cc` *?*

Action I.2. *Lancez votre programme* `putchar` *avec les options de trace :*

```
nachos -s -x ../test/putchar
```

pour le pas à pas,

```
../userprog/nachos -d + -x ../test/putchar
```

pour une trace détaillée (voir le source `threads/system.cc` *pour les autres options, en particulier* `-d t`*).*

En suivant pas à pas l'exécution dans le listing, examinez comment un programme est installé dans la mémoire (notamment à l'aide d'un objet de type `Addrspace`*), puis lancé, puis arrêté. Regardez en particulier* `userprog/progtest.cc`*, puis* `userprog/addrspace.cc`*.*

On souhaite maintenant qu'un programme utilisateur puisse créer des threads au niveau utilisateur, c'est-à-dire effectuer un appel système

```
int UserThreadCreate(void f(void *arg), void *arg)
```

Cet appel doit lancer l'exécution de `f(arg)` dans une nouvelle copie de l'interprète MIPS (autrement dit, une nouvelle instance de l'interprête exécutée par un nouveau thread *noyau*).

- Sur l'appel système `UserThreadCreate`, le thread noyau courant doit créer un nouveau thread `newThread`, l'initialiser et le placer la file d'attente des threads (noyaux) par l'appel

  ```
  newThread->Fork(StartUserThread, f)
  ```

  Il positionne en passant la variable `space` de ce nouveau thread `newThtread` à la même adresse que lui, de telle manière que la nouvelle copie de l'interprète MIPS partage le même espace d'adressage MIPS.

  Notez que la fonction `Thread::Fork` ne prenant qu'un seul paramètre, vous ne pouvez passer `f` et `arg` directement par ce moyen. À vous de voir comment faire !
- Lorsqu'il est finalement activé par l'ordonnanceur, ce nouveau thread lance la fonction `StartUserThread`. Cette fonction initialise les sauvegardes des registres d'une nouvelle copie de l'interprète MIPS à la manière de l'interprète primitif (fonctions `Machine::InitRegisters` et `Machine::RestoreState`) et lance l'interprète (`Machine::Run`).

  Notez que vous aurez à initialiser le pointeur de pile. Il vous est suggéré de le placer 2 ou 3 pages en dessous du pointeur du programme principal. Ceci est une évaluation empirique, bien sûr ! Il faudra probablement faire mieux dans un deuxième temps...

– Pour terminer, un thread utilisateur doit simplement se détruire par un appel système `UserThreadExit`, qui appelle une fonction `do_UserThreadExit` dans un nouveau fichier source `userprog/userthread.cc`. Cette fonction active `Thread::Finish` au niveau Nachos. Notez que la fonction MIPS `UserThreadExit` ne retourne jamais de valeur, à la manière de l'appel système `exit` de Unix. N'oubliez pas de détruire aussi les structures `AddrSpace`.

Action I.3. *Mettez en place les appels système*

```
int UserThreadCreate(void f(void *arg), void *arg)
```

et `void UserThreadExit()`. *Pour quelle(s) raison(s) la création d'un thread peut-elle échouer ? Retournez* `-1` *dans ce cas.*

Action I.4. *Écrire la fonction*

```
int do_UserThreadCreate(int f, int arg)
```

activée au niveau Nachos lors de l'appel de `UserThreadCreate` *par le thread appelant. Vous aurez à beaucoup travailler sur cette fonction : la placer dans le fichier* `userprog/userthread.cc` *en ne plaçant que la déclaration*

```
extern int do_UserThreadCreate(int f, int arg);
```

dans le fichier `userprog/userthread.h`. *Inclure ensuite ce fichier dans* `userprog/exception.cc`. *De cette manière, cette fonction est invisible par ailleurs. Pensez à ajuster les* `Makefile` *pour tenir compte de ce nouveau fichier si nécessaire. Il vous faudra probablement relancer toute la compilation pour reconstruire les dépendances correctes :* `make clean; make`.

Action I.5. *Définir dans le fichier* `userprog/userthread.cc` *la fonction*

```
static void StartUserThread(int f)
```

appelée par le nouveau thread Nachos créé par la fonction `do_UserThreadCreate`. Soyez très vigilants car vous n'avez aucun contrôle sur le moment où cette fonction est appelée ! Tout dépend de l'ordonnanceur... *Notez aussi qu'il faut passer l'argument* `arg` *d'une autre façon (*sérialisation*). À vous de trouver comment faire !*

Pour le moment, nous allons considérer qu'un thread se termine en invoquant systématiquement l'appel système `UserThreadExit()` (il ne "sort" donc jamais de la fonction initiale).

Action I.6. *Définir le comportement de l'appel système* `UserThreadExit()` *par une fonction* `do_UserThreadExit`, *placée elle aussi dans le fichier* `userprog/userthread.cc`. *Pour le moment, elle se contente de détruire le thread Nachos propulseur par l'appel de* `Thread::Finish`. *Que doit-on faire pour son espace d'adressage* `space` *?*

Notez que le programme principal ne doit pas appeler la fonction `Halt` tant que les threads utilisateurs n'ont pas appelé `UserThreadExit` ! Il faut donc le faire attendre artificiellement... Comment ? À vous de trouver !

Attention ! Nachos doit être lancé avec l'option `-rs` pour forcer l'ordonnancement préemptif des threads utilisateurs :

```
nachos -rs -x ../test/makethreads
```

En ajoutant un paramètre à l'option, vous modifiez la suite aléatoire utilisée pour l'ordonnancement :

```
nachos -rs 1 -x ../test/makethreads
```

Notez que l'ordonnancement des threads noyaux *n'est pas préemptif*. Pourquoi donc ?

Action I.7. *Démontrer sur un petit programme* `test/makethreads.c` *le fonctionnement de votre implémentation. Testez différents ordonnancements. Que se passe-t-il en l'absence de l'option* `-rs` *? Expliquez !*

Partie II. Plusieurs threads par processus

L'implémentation ci-dessus est encore bien primitive, et elle peut être améliorée sur plusieurs points.

Si vous essayez de faire des écritures (par exemple par la fonction `putchar`) depuis le programme principal et depuis le thread, vous aurez probablement un message d'erreur `Assertion Violation`. (Essayez !) En effet, les requêtes d'écriture et d'attente d'acquittement des deux threads se mélangent ! Il faut donc protéger les fonctions noyau correspondantes par un verrou...

Action II.1. *Modifier votre implémentation de la classe `SynchConsole` pour placer les écritures et les lectures en section critique. Pouvez-vous utiliser deux verrous différents ? Notez que ces verrous sont privés à cette classe. Démontrez le fonctionnement par un programme de test.*

Pour le moment, l'utilisateur doit garantir que le programme principal n'appelle pas la fonction `Halt` tant que le thread n'a pas appelé `UserThreadExit`.

Action II.2. *Que se passe-t-il si le thread initial sort du programme (c'est-à-dire en appelant `Halt`) avant que les threads avec lesquels il cohabite n'aient appelé `UserThreadExit` ? Corrigez ce comportement en assurant une synchronisation au niveau des appels `Halt` et `ThreadExit`, par exemple en comptant le nombre de threads qui partagent le même espace d'adressage (`AddrSpace`). Vous aurez sans doute à utiliser un sémaphore au niveau Nachos utilisé par tous les threads partageant un même espace d'adressage. Démontrer le fonctionnement par un programme de test.*

Pour le moment, un programme n'appelle qu'une seule fois `UserThreadCreate`. Il faut lever cette limitation.

Action II.3. *Que se passe-t-il si le programme lance plusieurs threads et non pas un seul ? Faites un essai, et expliquez ce que vous observez. Proposer une correction permettant de lancer un grand nombre de threads. Démontrer son fonctionnement par un programme de test.*

Action II.4. *Que se passe-t-il si un programme lance un très grand nombre de threads ? Discutez avec précision les différents comportements en fonction de l'ordonnancement.*

Partie III. BONUS : Terminaison automatique

Pour le moment, un thread doit explicitement appeler `UserThreadExit`. De même, le programme principal doit explicitement appeler `Halt`. Ceci est évidemment peu élégant, et surtout très propice aux erreurs !

Action III.1. *Expliquez ce qui adviendrait dans le cas où un thread n'appellerait pas `ThreadExit`. Comment ce problème est-il résolu pour le thread initial (avec `nachos -x`) ? Regardez notamment dans le fichier `test/start.S`. Que faut-il mettre en place pour utiliser ce mécanisme dans le cas des threads créés avec `UserThreadCreate` ? NB : votre solution doit être indépendante de l'addresse* réelle *de chargement de la fonction. Il faudra donc passer cette adresse en paramètre lors de l'appel système... À vous de jouer !*

Pour le moment, vous n'avez que la fonction `putchar` à votre disposition pour écrire à la console. À cause de l'entrelacement préemptif, il n'est pas possible de rentre *atomique* l'écriture d'une chaîne.

Action III.2. *Définir un appel système `void PutString(const char s[])` qui écrit une chaîne de manière atomique via un objet de la classe `SynchConsole`. Notez que le paramètre qui est passé lors de l'appel système est un pointeur MIPS ! Il faut donc que la fonction noyau Nachos aille chercher la chaîne dans le monde MIPS caractère par caractère. À vous de jouer !*

Et pourquoi ne pas maintenant implémenter un schéma producteur-consommateur au niveau utilisateur ?

Action III.3. *Remontez l'accès aux sémaphores (type* `sem_t`*, appels système* `P` *et* `V`*) au niveau des programmes utilisateurs. Démontrez leur fonctionnement par un exemple de producteurs-consommateurs au niveau* utilisateur *cette fois.*