

# NACHOS : Multithreading

Devoir 2, Master 1 Informatique 2009–2010

11 octobre 2009

L'objectif de ce devoir est de permettre d'exécuter des applications multi-threads sous Nachos. **Lisez l'ensemble du sujet avant de commencer pour avoir une vue d'ensemble des problèmes et l'ordre dans lesquels vous allez les aborder.**

Le devoir est à faire en binôme et à rendre avant

*Lundi 12 Novembre 9h00.*

Comme pour le TP précédent, il vous est demandé de rendre vos fichiers sources ainsi qu'un rapport de quelques pages (5 pages maximum) selon la procédure décrite à l'adresse suivante

[http://dept-info.labri.fr/~guermouc/SE/procedure\\_nachos.txt](http://dept-info.labri.fr/~guermouc/SE/procedure_nachos.txt)

Il vous est demandé de rendre les pièces suivantes :

- une description de la stratégie d'implémentation utilisée, et une discussion des choix que vous avez faits (5 pages maximum). Un plan pour ce document se trouve sur le site <http://dept-info.labri.fr/~guermouc/SE/>;
- Vos sources.
- Des programmes de test représentatifs présentant les qualités de votre implémentation et ses limites. Chaque test doit contenir un commentaire expliquant comment le programme doit être lancé (arguments,...) et être accompagné d'un court commentaire (5–10 lignes) expliquant son intérêt.
- Il **n'est pas** demandé de répondre aux questions de ce sujet dans l'ordre où elles sont posées.

Avant de commencer à coder, lisez bien chaque partie **en entier** : les sujets de TP contiennent à la fois des passages descriptifs pour expliquer vers où l'on va (et donc il ne s'agit que de lire et comprendre, pas de coder), et des **Actions** qui indiquent précisément comment procéder pour implémenter pas à pas (et là c'est vraiment à vous de jouer).

## Partie I. Multithreading dans les programmes utilisateurs

Il s'agit maintenant de permettre aux programmes utilisateurs de créer et manipuler des threads Nachos au moyen d'appels système. Dans cette première partie, on ne lancera qu'un seul thread supplémentaire.

**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,

```
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 qui exécuteront des fonctions du programme, c'est-à-dire effectuer un appel système

```
int ThreadCreate(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'interprète exécutée par un nouveau thread *noyau*). Voici un résumé de ce que vous allez implémenter dans les actions I.3 à I.7 :

- Sur l'appel système `ThreadCreate`, le thread noyau courant doit créer (dans une nouvelle fonction `do_ThreadCreate`) un nouveau thread `newThread`, l'initialiser et le placer la file d'attente des threads (noyau) par l'appel

```
newThread->Start(StartThread, schmurtz)
```

Il doit positionner en passant la variable `space` de ce nouveau thread `newThread` 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::Start` ne prenant qu'un seul paramètre `schmurtz`, 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 `StartThread` (que vous allez créer). Cette fonction doit initialiser les registres MIPS d'une façon similaire à l'interprète primitif (fonctions `AddrSpace::InitRegisters` et `Thread::RestoreUserState`) et lance l'interprète (`Machine::Run`).

Notez que vous aurez à initialiser le pointeur de pile, ajoutez pour cela à la classe `AddrSpace` une méthode `AllocateStack` retournant l'adresse du haut de cette nouvelle pile. Il vous est suggéré de la placer 256 octets en-dessous de la fin de la mémoire virtuelle (i.e. en-dessous de la pile du thread principal). Ceci est une évaluation empirique, bien sûr ! Il faudra probablement faire mieux dans un deuxième temps...

- Pour terminer, un thread exécutant du code en espace utilisateur doit simplement se détruire par un appel système `ThreadExit`, qui appelle une fonction `do_ThreadExit` dans un nouveau fichier source `userprog/userthread.cc`. Cette fonction active `Thread::Finish` au niveau Nachos.

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

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

et `void ThreadExit(void)`. 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_ThreadCreate(int f, int arg)
```

activée au niveau Nachos lors de l'appel de `ThreadCreate` 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_ThreadCreate(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.

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

```
static void StartThread(int f)
```

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

**Important** : Pour le moment, dans vos programmes de test, terminez vos threads en invoquant systématiquement l'appel système `ThreadExit()` (ils ne "sortent" donc jamais de leur fonction initiale).

**Action I.6.** Définir le comportement de l'appel système `ThreadExit()` par une fonction `do_ThreadExit`, 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` ?

**Important** : Notez que pour que vos nouveaux threads utilisateurs aient une chance de s'exécuter, le thread principal utilisateur ne doit pas se terminer (e.g. sortir de la fonction `MIPS main`) tant que les threads utilisateurs n'ont pas appelé `ThreadExit`! Dans un premier temps, faites donc attendre la fonction `MIPS main` avec une boucle infinie...

Attention! Nachos doit être lancé avec l'option `-rs` pour forcer l'ordonnancement préemptif (et donc réaliste) 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. Pour quelle raison ?

**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'acquiescement des deux threads se mélangent! Il faut donc protéger les fonctions noyau correspondantes par un verrou (utilisez des sémaphores, ou mieux, complétez l'implémentation des mutexes!)

**Action II.1.** Modifier votre implémentation de la classe `SynchConsole` pour placer les traitements effectués par `SynchPutChar` et `SynchGetChar` 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.

Faut-il également protéger `SynchPutString` et `SynchGetString`? Pour quelle raison?

Si un thread appelle `Exit`<sup>1</sup> ou que le thread principal sort de la fonction `main`, nachos est arrêté sans donner une chance aux autres threads de continuer à s'exécuter. Pour laisser les autres threads tourner dans le processus, `main` peut utiliser `ThreadExit` pour se terminer lui-même mais pas le processus.

<sup>1</sup>Sauf exception, on n'utilisera plus l'appel système `halt`.

**Action II.2.** Que se passe-t-il si à la fois le thread créé et le thread initial utilisent `ThreadExit`? Corrigez ce comportement en assurant une synchronisation au niveau des appels à `ThreadExit`, par exemple en comptant le nombre de threads qui partagent le même espace d'adressage (`AddrSpace`). Démontrez le fonctionnement par un programme de test.

Pour le moment, un programme ne peut appeler qu'une seule fois `ThreadCreate`, à cause de l'allocation de pile qui est trop simpliste. Il faut lever cette limitation.

**Action II.3.** Que se passerait-il si le programme lançait plusieurs threads et non pas un seul? Faites un essai pour voir. Proposez une correction permettant de lancer un grand nombre de threads. Avant de désespérer d'observer des bugs plus que mystérieux, vérifiez bien que vous donnez aux threads vivants des piles différentes de taille au moins 256 octets par exemple. Démontrez le 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. Terminaison automatique

Pour le moment, un thread doit explicitement appeler `ThreadExit` pour se terminer. 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 principal (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 `ThreadCreate`? NB: votre solution doit être indépendante de l'adresse réelle de chargement de la fonction. Il faudra donc passer cette adresse en paramètre lors de l'appel système... À vous de jouer!

## Partie IV. Sémaphores

**Action IV.1.** 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.