L3 – Cours de Système – Devoir dirigé 2
Nachos :Threads

Emmanuel Agullo, Eddy Caron, Nicolas Veyrat Charvillon

Lundi 20 mars 2006

L'objectif de ce devoir est de constituer un premier pas vers l'implantation en Nachos d'un modèle de *multithreading* à la manière de Linux. Il est donc nécessaire que vous ayez bien compris les threads Linux, et que vous ayez achevé les parties consacrées aux verrous et aux variables de condition si vous ne l'avez pas fait.

Le devoir est à faire en binôme. Il est à rendre pour le

mercredi 12 avril 2006.

Tous les développements faits dans le code Nachos pour ce devoir doivent être délimités par `#ifdef CHANGED`, selon la discipline acquise au premier devoir. Si certaines fonctionnalités du premier devoir vous manquent, n'hésitez pas à les demander.

Il vous est demandé de rendre un rapport mis en forme, regroupant les pièces suivantes, soit en Postscript par mail soit imprimé sur papier.
- Les listings des parties modifiées, indentées et commentées.
- Une description de la stratégie utilisée, et une discussion des choix que vous avez faits.
- Une série d'exemples de tests représentatifs, présentant les qualités de votre implantation et ses limites. Chaque test doit être accompagné d'un court commentaire expliquant son intérêt.

L'énoncé est volontairement laissé flou sur plusieurs points. Certains choix de conception *non triviaux* sont donc laissés à votre appréciation.

Partie I. Les threads Nachos

Note *Cette partie d'introduction a déjà été abordée lors de la première séance. Inutile donc d'y passer trop de temps si vous vous en souvenez.*

Consultez le fichier `threads/threadtest.cc`, et faites tourner l'exemple proposé en lançant simplement `nachos`. Cet exemple crée deux threads (noyaux) qui entrelacent leurs écritures. Notez les options de débogage. Essayez par exemple `nachos -d +` et suivez bien ce qui se passe.

Action I.1. *Notez l'invocation explicite des changements de contexte entre threads. Regardez en profondeur le fonctionnement de la fonction `Thread::Yield`. Quelle est la fonction "clé" utilisée pour effectuer un changement de contexte ? Peut-on envisager son implantation de manière un peu plus portable ?*

Action I.2. *Arrangez-vous pour obtenir plusieurs entrelacements différents des exécutions, par exemple en appelant la fonction `Thread::Yield` selon le résultat d'un tirage aléatoire.*

Partie II. Sémaphores et verrous

Regardez `threads/synch.cc` qui définit des sémaphores. Prenez le temps de bien comprendre ce qui se passe.

Il s'agit maintenant d'implanter des verrous (*mutex*) de manière analogue aux sémaphores. Faites-le à la suite des sémaphores, dans le même fichier. Les déclarations sont déjà prêtes !

Action II.1. *Déclarez une classe `Lock` basée sur la classe `Semaphore`, avec son constructeur et son destructeur.*

Action II.2. *Implémentez les fonctions `Lock::Acquire()` et `Lock::Release()`. Notez que threads qui fait `Release`* doit être le même *que celui qui a fait `Acquire`. Si ce n'est pas le cas, il faut* impérativement *provoquer une erreur ! Démontrez vos choix d'implémentation par des petits tests.*

Action II.3. *Écrivez un programme qui met en évidence l'ordre dans lequel les threads bloqués sur un verrou sont réveillés lorsque le verrou se libère.*

Partie III. La partie délicate : 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. Il s'agit donc un modèle *1-on-1* comme dans la librairie `pthread` actuelle de Linux, par opposition aux versions futures (NGPT ou NPTL) qui utiliseront un modèle *M-on-N* (M threads utilisateurs sur N threads noyaux), ou aux LWP (*Lightweight Processes*) de Solaris

Action III.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 propulse ?*

À quoi servent les fonctions `SaveState` et `RestoreState` de `userprog/addrspace.cc` ? À quoi servent les fonctions `SaveUserState` et `RestoreUserState` de `threads/thread.cc` ? Où sont-elles respectivement appelées ?

Action III.2. *Vérifiez que votre programme `putchar` marche toujours. Lancez-le avec les options de trace :*

```
../userprog/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 ThreadCreate(void f(void *arg), void *arg)
```

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

Action III.3. *Créer les fichiers `userprog/userthread.cc` et `userprog/userthread.h`. Y placer les prototypes des fonctions noyau `do_UserThreadCreate(int f, int arg)` et `do_UserThreadExit()`.*

Faire appel à ces fonctions pour mettre en suite en place le traitement des appels système suivants

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

Action III.4. *Faire en sorte que* `do_UserThreadCreate` *crée une nouveau thread* noyau *utilisant le même espace d'adressage MIPS* `space` *que le thread courant. Pour quelle(s) raison(s) la création d'un thread peut-elle échouer ? On retournera* `-1` *dans ce cas.*

Définir le prototype `static void StartUserThread(int arg)` *et utiliser la méthode* `Fork` *du thread créé pour l'exécuter.*

Soyez très vigilants car vous n'avez aucun contrôle sur le moment où cette fonction est appelée ! Tout dépend de l'ordonnanceur...

Trouver une méthode pour passer la fonction `f` *et son argument* `arg` *à* `StartUserThread`*.*

Action III.5. *Dans* `StartUserThread`*, initialiser les registres du nouveau thread à zéro par la fonction* `AddrSpace::InitRegisters` *puis appeler* `AddrSpace::RestoreState` *pour préparer le retour en espace utilisateur. Lancer une nouvelle copie de l'interprète MIPS dans ce thread par* `Machine::Run`*.*

Un test rapide devrait montrer que la création d'un thread fait boucler le programme (il se relance lui-même indéfiniment, pourquoi ?).

Action III.6. *Avant de lancer l'interprète MIPS, lui indiquer, dans les registres, qu'il doit lancer la fonction* `f` *avec l'argument* `arg`*.*

C'est presque prêt ! il reste à faire en sorte que le thread créé ne partage pas la pile utilisateur du thread initial.

Action III.7. *Initialiser le pointeur de pile quelques pages au dessous du haut de la pile du thread initial. Ceci est un* best guess*, bien sûr ! Il faudra faire mieux dans un deuxième temps...*

Ca y est, vous pouvez lancer un thread utilisateur ! Voyons comment terminer son exécution. On s'intéresse donc maintenant à `do_UserThreadExit`.

Action III.8. *Dans* `do_UserThreadExit`*, détruire le thread Nachos propulseur par l'appel de* `Thread::Finish`*. Que doit-on faire pour son espace d'adressage* `space` *?*

Action III.9. *Démontrer sur un petit programme* `test/makethreads.c` *le fonctionnement de votre implémentation. Testez différents ordonnancements.*

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 *noyau* n'est *pas préemptif*. Pourquoi ?

Notez que le programme principal ne doit pas appeler la fonction `Halt` tant que les threads utilisateurs n'ont pas appelé `ThreadExit` ! Il faut donc le faire attendre artificiellement...

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

N'abordez ces parties que si vous terminé les parties précédentes. Ne faites pas toutes les questions ! Choisissez ce qui vous semble le plus intéressant pour vous, et le plus abordable.

Partie IV. Au delà des limites du modèle

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

Action IV.1. *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.*

Rien ne garantit que les tailles prévues pour le tas commun du programme et les piles privées des threads sont suffisantes.

Action IV.2. *Que se passe-t-il si vous exécutez des programmes fortement récursifs en présence de threads (une factorielle par exemple) ? Essayez. Expliquez le comportement observé. Comment corriger cela ?*

Action IV.3. *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 V. Accès sécurisé à la console

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`. En effet, les requêtes d'écriture et d'attente d'acquittement des deux threads se mélangent ! Il faut donc protéger les fonctions noyaux correspondantes par un verrou...

Action V.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. Démontrer le fonctionnement par un programme de test.*

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 V.2. *Définir un appel système `void AtomicPutString(const char s[])` qui écrit une chaîne de manière atomique via un objet de la classe `SynchConsole`.*

Partie VI. Partage non protégé de variables

Maintenant qu'il est possible à plusieurs threads d'utiliser la console simultanément, nous allons le mettre en pratique pour étudier l'effet de l'ordonnancement préemptif sur l'accès *non protégé* aux variables partagées.

Note *Si vous n'avez pas traité la partie précédente, vous pourrez utiliser des boucles vides pour éviter les collisions entre les différents threads lors de l'accès à la console*

Action VI.1. *Écrire un programme utilisateur dans le répertoire `test` qui initialise une variable entière `counter` à zéro. Ensuite, il crée un thread qui la décrémente `N` fois tandis que le programme principal l'incrémente `N` fois, avec `N` assez grand (à vous de voir !). Chaque thread affiche la première valeur et la dernière valeur lue. Conclusion ? Que se passe-t-il si vous lancez nachos sans l'option `-rs` ?*

Action VI.2. *Faites de même avec un tableau de caractères `char tab[M]` avec `M` assez grand. Un thread force toutes les cellules avec un caractère `'a'`, alors que le programme principal le fait avec un autre caractère `'b'`. Trouvez le moyen d'observer à l'écran un état de `tab` où des cellules sont à `'a'` et d'autres à `'b'`. Conclusion ?*

Partie VII. Gestion de la fin d'exécution

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 `ThreadExit`.

Action VII.1. *Que se passe-t-il si le thread initial sort du programme (i.e. en appelant `Halt`) avant que les threads avec lesquels il cohabite n'aient appelé `ThreadExit` ? 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 une variable de condition au niveau Nachos partagée entre tous les threads partageant un même espace d'adressage. Démontrer le fonctionnement par un programme de test.*

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

Action VII.2. *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 `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 VIII. Variables de condition

Le but est maintenant d'implanter des variables de condition par la classe `Condition`. Les déclarations sont après les `Lock` dans `synch.cc`, elles n'attendent que vous !

Action VIII.1. *Implémentez les constructeurs, destructeurs.*

Lisez soigneusement les commentaires placés dans le fichier `threads/synch.h` à propos de la sémantique de `Wait` et `Signal`. En particulier, notez que la suite d'opérations de relâchement du verrou (`Release`) et d'endormissement (`Sleep`) doivent être atomiques dans le `Wait`. `Signal` ne fait que prendre un thread dans la file d'attente de la variable condition pour le remettre dans la file d'attente de l'ordonnanceur. Notez aussi que toutes les actions sur une variable de condition donnée doivent être mutuellement exclusives. En particulier, il doit être garanti que deux threads ne manipulent pas la file d'attente en même temps.

Action VIII.2. *Implémentez `Wait` et `Signal`. Attention, il faut provoquer une erreur s'ils sont employés en dehors de leurs spécifications. En particulier, il est interdit de les utiliser en dehors de l'acquisition du verrou correspondant : ceci doit* impérativement *provoquer une erreur !*

Action VIII.3. *Faire tourner l'exemple du producteur-consommateur vu en TD, avec un tableau de taille importante et des "vitesses" variables pour les threads. À vous de voir comment "ralentir" les threads...*

Action VIII.4. *Implémentez `Broadcast` qui réveille tous les threads en attente. Validez votre implémentation par un exemple* significatif *!*

Action VIII.5. *De même avec un grand nombre de producteurs et un grand nombre de consommateurs. Faites des essais et discutez !*

Partie IX. Quelques autres idées

Action IX.1. *Implémentez des appels système* `Yield` *et* `Sleep` *pour permettre aux threads utilisateur de rendre la main ou d'attendre un certain temps.*

Action IX.2. *Implémentez un appel système* `Join` *permettant à un thread d'attendre un autre thread du même processus.*

On pourra envisager une version bloquante avec un delai maximal d'attente, qui pourra être infini ou nul. Dans le cas d'un délai nul, l'appel se réduira alors à un test de terminaison.

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

Action IX.3. *Remontez l'accès aux sémaphores (type* `sem_t`*, appels systèmes* `P` *et* `V`*) au niveau des programmes utilisateurs. Démontrer leur fonctionnement par un exemple de producteur-consommateur au niveau* utilisateur *cette fois. Vaut-il mieux restreindre les sémaphores utilisateurs à un seul espace d'adressage ? Quels problèmes cela engendre/résoud-il ?*

Action IX.4. *En C, il est possible d'écrire une affectation entre structures. Par exemple :*

```
#define SIZE 4 // Essayez avec 1024!
struct S {char tab[SIZE];} s, s0;
...
s = s0;
```

Cette instruction est traduite par le compilateur avec une itération, éventuellement cachée par un appel à la fonction de bibliothèque `memcpy` *si la taille le justifie. Cette affectation n'est donc pas atomique en général, et l'on doit donc pouvoir observer une inconsistance comme ci-dessus...*

Chiche ? (NB : Vous aurez peut-être besoin de définir cette fonction explicitement dans votre programme, regardez attentivement l'assembleur produit. Faire `man memcpy` *pour sa spécification).*