NACHOS : Multithreading

Devoir 2, Master 1 Informatique 2019–2020

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

Mardi 12 novembre 2019, 8h00.

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`

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

On rappelle que dans le devoir 0 on a déjà joué avec les threads *noyau* de Nachos, notamment dans la fonction `ThreadTest`. Il s'agit maintenant de permettre aux programmes utilisateurs de créer et manipuler des threads *utilisateur* Nachos au moyen d'appels système qui utiliseront des threads *noyau* pour propulser les threads *utilisateur*. Note : dans cette première partie, on se contentera de ne lancer qu'un seul thread supplémentaire.

Action I.1. *Examinez en détail le fonctionnement des threads Nachos (noyau et utilisateur). Comment ces threads sont-ils alloués et initialisés ? Où se trouve la pile d'un thread Nachos, en tant que thread* noyau *?*

Action I.2. *Lancez votre programme* `putchar` *avec quelques options de trace vues au TP0 :*

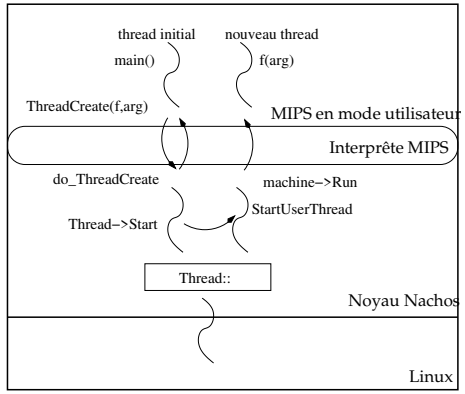
```
./nachos -s -d a -x ../test/putchar
```

Observez en particulier au début du listing comment un programme est installé dans la mémoire (notamment à l'aide d'un objet de type `Addrspace`*), puis lancé, puis arrêté. Repérez en particulier où cela est fait dans* `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 MIPS

```
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'interprête exécutée par un nouveau thread *noyau*). Voici un dessin et un résumé de ce que vous allez implémenter dans les actions I.3 à I.7 (**ne commencez pas à coder ! !**, lisez d'abord ce résumé, et ne commencez à coder qu'une fois atteint le texte de l'action I.3, qui explique par quoi commencer !) :



— 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 dans la file d'attente des threads (noyau) par l'appel

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

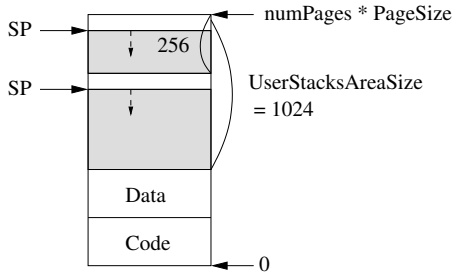
Vous remarquerez que le constructeur positionne au passage la variable `space` de ce nouveau thread `newThtread` à la même valeur que celle du thread courant, de telle manière que la nouvelle copie de l'interprète MIPS partage le même espace d'adressage MIPS que le thread parent.
Notez que la fonction `Thread::Start` ne permettant de passer qu'un seul paramètre `schmurtz` à `StartUserThread`, vous ne pouvez passer à la fois `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` (que vous allez créer). Cette fonction doit initialiser tous les registres MIPS d'une façon similaire à l'interprète primitif (inspirez-vous de la fonction `AddrSpace::InitRegisters`) 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 `AllocateUserStack` retournant l'adresse du haut de cette nouvelle pile. Il vous est suggéré de la placer dans un premier temps 256 octets en-dessous de la fin de la mémoire virtuelle (i.e. en-dessous de la pile du thread principal) (la taille de la mémoire virtuelle est `numPages*PageSize`). Ceci est une méthode simpliste, bien sûr ! Il faudra probablement faire mieux dans un deuxième temps... Pour pouvoir appeler cette méthode, utilisez `currentThread->space`.

— 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 doit appeller `Thread::Finish` pour terminer simplement le thread Nachos. L'objet `Thread` est déjà libéré automatiquement par l'ordonnanceur.

Ce schéma montre l'aspect de notre espace d'adressage avec la méthode simpliste pour allouer la deuxième pile :



Action I.3. *Mettre en place l'interface des appels système*

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

Pour quelle(s) raison(s) la création d'un thread pourrait-elle échouer ? On pensera plus tard à retourner `-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.common` pour tenir compte de `userprog/userthread.cc`.

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

```
static void StartUserThread(void *schmurtz)
```

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 à cette fonction à la fois les arguments `f` et `arg`. À vous de trouver comment faire ! Ajoutez beaucoup d'informations de débuggage : utilisez la macro* `DEBUG('x', "mon debug %d\n", mavar);` *pour imprimer les valeurs que vous mettez dans les registres notamment, pour être bien sûr de vos calculs !*

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` ?*

Action I.7. *Démontrer sur un petit programme `test/makethreads.c` le fonctionnement de votre implémentation, en utilisant dans un premier temps un simple `PutChar` dans le thread créé (mais pas dans le thread principal), et en faisant attention aux remarques importantes ci-dessous. Si vous avez des bugs, vérifiez bien quelle valeur de `PCReg`, `NextPCReg` et `StackReg` vous donnez à votre thread. Testez différents ordonnancements.*

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` après avoir créé le thread avec une boucle vide infinie... Bien sûr, du coup votre programme ne termine pas. Corriger cela sera l'objet de l'action II.2.

Important : Dans un premier temps, dans vos programmes de test, terminez tous vos threads utilisateur en leur faisant invoquer systématiquement l'appel système `ThreadExit()` pour qu'ils se sabordent eux-même depuis le mode utilisateur et ainsi ne "sortent" jamais de leur fonction initiale `f`). Cf l'action III.1 plus loin pour les détails.

Important : Tant que vous n'avez pas encore mis de verrouillage sur votre implémentation de la console, ne faites pas faire d'affichage par différents threads.

Important : Nachos doit alors être lancé avec l'option `-rs` pour forcer l'ordonnancement préemptif (et donc réaliste) des threads utilisateurs :

```
./nachos -rs 1234 -x ../test/makethreads
```

En modifiant le nombre donné en paramètre à l'option, vous pouvez modifier la suite aléatoire utilisée pour l'ordonnancement.

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

Note : Vous pouvez observer les deux threads en ajoutant dans `StartUserThread` un appel `machine->DumpMem("threads.svg");` avant l'appel à `machine->run();`, et constater qu'il y a désormais deux séries de flèches `SP` et `PC` !

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 thread principal et depuis le thread créé, vous aurez probablement un message d'erreur `Assertion failed`. (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 (utilisez des sémaphores, ou mieux, complétez l'implémentation des locks dans `synch.cc` en vous inspirant de celle des sémaphores !)...

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`[1] 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.

1. Sauf exception, on n'utilisera plus l'appel système `Halt`.

Action II.2. *Est-ce que Nachos se termine effectivement si à la fois le thread créé et le thread initial utilisent* `ThreadExit` *? Corrigez la terminaison en assurant une comptage dans* `ThreadExit` *du nombre de threads qui partagent le même espace d'adressage (*`AddrSpace`*) (dans le TP3 on aura plusieurs processus !), pour que le dernier thread appelle* `interrupt->Halt()` *pour terminer Nachos. Démontrez le fonctionnement par un programme de test.*

Attention, vous voudrez éventuellement inclure `synch.h` *depuis* `addrspace.h`*. Cela ne peut pas fonctionner puisque* `synch.h` *inclut* `thread.h`*, qui lui-même inclut* `addrspace.h`*, au final on ne peut rien inclure avant l'autre... Plutôt qu'inclure* `synch.h`*, écrivez simplement une déclaration partielle de classe, par exemple :*

```
class Semaphore;
```

Ainsi la compilation de `addrspace.h` *passe, puisqu'il ne contient que des pointeurs vers des sémaphores, il a juste besoin de savoir que la classe* `Semaphore` *existe.*

Pour le moment, un programme ne peut appeller 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. Avec de la chance, cela fonctionne peut-être, mais par exemple faites faire à vos threads une boucle* `for` *sur une variable locale (donc sur la pile)* `volatile int i;` *qui affiche un* `a` *à chaque tour, et comptez le nombre de* `a`*. Proposer une correction permettant de lancer quelques threads. Avant de désespérer de n'observer que des bugs plus que mystérieux,* ***vérifiez*** *bien que vous donnez aux threads vivants des piles différentes de taille au moins 256 octets par exemple, et qui ne débordent pas dans les données ou le code, et que vous tenez compte de la pile du thread principal. Démontrez le fonctionnement par un programme de test.*

Action II.4. *Que se passe-t-il si un programme lance un grand nombre de threads ? Révisez éventuellement votre mécanisme d'allocation de piles, en utilisant par exemple la classe* `Bitmap` *de* `bitmap.cc` *pour mémoriser quels slots de pile sont déjà utilisés. Faites attention à la pile du thread principal. Discutez avec précision les différents comportements en fonction de l'ordonnancement.*

Partie III. Terminaison automatique (bonus)

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 et de* `ThreadExit`*. Il faudra donc passer cette adresse en paramètre lors de l'appel système... À vous de jouer !*

Partie IV. Sémaphores (bonus)

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.*