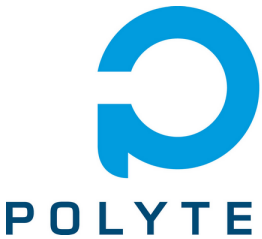

Polytech' Grenoble

RICM 4ème année

NachOS
Etape 3: Multithreading

Étudiants:
Elizabeth Paz
Salem Harrache

Professeur:
Vania Marangozova

février 2012

1 Git

Pour voir les différences entre l'étape 2 et l'étape 3 vous pouvez lancer un diff avec le tag step2 :

```
git diff step2
```

ou alors directement avec le commit 4a23b1...

```
git diff 4a23b18e1b2b597b0454b768347d464f47ecb572
```

2 Test de l'étape 3

Vous pouvez soit lancer le script de test automatique *runtest.sh* soit lancer manuellement les deux programmes *./build-origin/nachos-userprog -rs 1 -x ./build/jointhreads* et *./build-origin/nachos-userprog -rs 1 -x ./build/makethreads*

```
$ ./runtest.sh
Compilation en cours...
Lancement du Test 1 : Create & Exit :)
_____________________________________
./build-origin/nachos-userprog -rs 1 -x ./build/makethreads
Test de thread :
Thread : 0 OK
Thread : 1 OK
Thread : 2 OK
Thread : 3 OK
Thread : 4 OK
Thread : 5 OK
Thread : 6 OK
Thread : 7 OK
Thread : 8 OK
Impossible de cr er de nouveaux Thread
Fin du main :
0
2
1
3
4
6
5
8
7
Machine halting!
Fin Test 1
Lancement du Test 2 : Join :)
_____________________________
./build-origin/nachos-userprog -rs 1 -x ./build/jointhreads
Je lance le Thread 1
1
Fin Thread 1
Je lance le Thread 2
2
Fin Thread 2
Je lance le Thread 3
3
Fin du main :
Machine halting!
```

Sans l'option -rs le thread main ne donne jamais la main aux autres threads. Comme on peut le voir sur le test 1, le nombre de threads simultanés est limité (par la taille de l'espace d'adressage et la taille de la pile d'un thread). Dans ce cas un code d'erreur (-1) est renvoyé lors de l'appel à *UserThreadCreate*.

3 Choix de conception

3.1 La classe UserThread

Pour implémenter les threads utilisateurs nous avons créé une nouvelle classe UserThread qui hérite de la classe Thread. Cette classe surcharge le comportement de la méthode Thread::Fork(..) et implémente quelques mécanismes utiles pour les

threads utilisateurs. Cet objet va servir également à stocker les arguments qu'on passe à la fonction UserThreadCreate(), ce qui nous évite de devoir utiliser une structure dédiée au passage de paramètre à Fork.

code/userprog/userthread.h

```
class UserThread : public Thread {
    public:
        UserThread(const char *name, int f, int a, int callback);
        int func;
        int arg;
        void Fork (); // Make userthread run (*f)(arg)
        void UpdateCallBackRegister (int value); // $31 = value
};
```

code/userprog/userthread.cc

```
void StartUserThread(int thread) {
    UserThread *t = (UserThread *) thread;
    // L'id du thread informe egalement le num ro de page du thread
    currentThread->space->InitThreadRegisters(t->func, t->arg, t->getId());
    currentThread->space->UpdateRunningThreads(1); // appel atomique
    machine->Run();
}

void UserThread::Fork () {
    DEBUG ('t', "Forking userThread \"%s\"\n", getName ());
    Thread::Fork (StartUserThread, (int) this);
}
```

3.2 Appel implicite de UserThreadExit

La fonction UpdateCallBackRegister permet de modifier l'adresse de retour (registre 31 dans MIPS) lors de l'appel système. En effet pour quitter les threads sans appel explicite de la fonction UserThreadExit(), nous devons au moment du lancement du nouveau thread lui affecter comme adresse de retour l'adresse de la fonction UserThreadExit qu'on passe en paramètre dans le fichier start.S (registre 6)

```
    .globl UserThreadCreate
    .ent UserThreadCreate
UserThreadCreate:
    addiu $2,$0,SC_UserThreadCreate
    addiu $6,$0,UserThreadExit // On passe en parametre la fonction de retour
    syscall
    j    $31
    .end UserThreadCreate
```

3.3 Modifications de l'espace d'adressage du processus

Pour gérer plusieurs threads, nous sommes obligés de modifier l'espace d'adressage (AddrSpace) du processus. Nous avons entre autres ajouté le nombre de threads en exécution, un objet bitmap pour gérer l'allocation des zones des threads et des sémaphores pour utiliser ces attributs en section critique. Chaque thread a sa propre pile qui est une partie de la pile du processus (sa taille est de 3 pages). Nous sommes donc naturellement limités en nombre de threads simultanés par processus. Lorsque cette limitation se manifeste, on renvoie simplement un code d'erreur (-1).

Nous devons également différencier un thread d'un autre, pour cela on ajoute un attribut **Thread::id** et un compteur de thread pour éviter de réutiliser les identifiants de threads. On crée ensuite une association entre un thread (son id) et le numéro de la zone auquel il sera affecté lors de la création de ce dernier. Bien évidemment la gestion de ces deux maps (**bitmap** et **id//zone**) se fait en section critique.

L'espace d'adressage est la partie commune de tous les threads (d'un même processus). C'est pour cette raison que nous avons ajouté plusieurs méthodes qui vont "rendre service" aux threads. Par exemple, pour savoir s'il est le seul en cours d'exécution, un thread ne peut pas le savoir directement, il va utiliser la méthode **int AddrSpace::Alone()** qui va garantir que cette appel est atomique. C'est à AddrSpace que revient la responsabilité de garder un état cohérent des variable partagées.

```
int AddrSpace::Alone() {
    int value = 0;
```

```
    this->semRunningThreads->P();
    if (this->runningThreads == 0)
        value = 1;
    this->semRunningThreads->V();
    return value;
}
```

3.4 Le Thread main est comme tous les autres

Notre implémentation des threads utilisateurs considère que même le thread main se comporte comme un thread utilisateur, c'est-à-dire qu'il a sa propre pile (3 pages) et qu'il cède sa place (sa pile) aux autres threads une fois terminé. Ce qui a pour principale consequence que l'appel **Exit()** ne se fait plus automatiquement à la fin du main, mais sera fait par le dernier thread en cours.

Ceci dit, un thread (main ou pas) peut toujours appeler explicitement **Exit()** pour arrêter le processus.

3.5 Mécanisme d'attente des threads (UserThreadJoin)

Nous utilisons un tableau de sémaphore de la taille du nombre maximal de threads. Chaque thread (A) qui veut se bloquer sur un autre (B) va devoir prendre un jeton du sémaphore lié au thread B sur lequel il veut se bloquer. Ces sémaphores sont donc bloquants lorsque le thread B est en cours d'exécution.

Une fois terminé, le thread B va réveiller les threads qui se sont bloqués que lui en déposant un jeton dans le sémaphore concerné. Pour permettre à plusieurs threads de se bloquer sur le même thread, la phase de réveil doit se faire en chaîne, c'est-à-dire chaque thread réveillé doit réveiller le suivant etc..

Ce mécanisme doit également bloquer le prochain thread qui va s'exécuter dans la nouvelle zone liberée par le thread B.

Pour arriver à ce résultat, nous initialisons le tableau de sémaphore à un jeton chacun. A chaque création, il faut prendre un jeton (pour que le sémaphore devienne bloquant pour tous les threads qui appellent **UserThreadJoin()**). S'il y a une phase de réveil qui est en cours, ce nouveau thread sera bloqué, et c'est ce qu'on cherche pour pas qu'un thread bloqué sur un thread spécifique se retrouve bloqué sur d'autres (avec le risque qu'il ne se réveille jamais)

```
int do_UserThreadCreate(int f, int arg, int callback) {
    [...]
    // Avant de commencer on prend le jeton, pour que tout thread qui appelle
    // userThreadJoin sur moi soit bloqu .
    currentThread->space->semJoinThreads[newThread->getZone()]->P();
    [...]
    newThread->Fork();
    return newThread->getId();
}
```

```
void do_UserThreadExit() {
    currentThread->space->UpdateRunningThreads(-1); // appel atomique
    [...]
    // Je libere les threads en attente sur moi
    currentThread->space->semJoinThreads[currentThread->getZone()]->V();
    // Plusieurs threads peuvent attendre que je me termine.
    // Il faut donc que dans la fonction join, les threads en attente se
    // r veillent les uns les autres
    [...]
    currentThread->Finish();
}
```

```
int do_UserThreadJoin(int thread_id) {
    int zone = currentThread->space->GetZoneFromThreadId(thread_id);
    if (zone < 0)
        return zone;
    currentThread->space->semJoinThreads[zone]->P();
    // On r veille le suivant qui peut  tre  soit le prochain thread    qui
    // on a allou  la zone, soit un autre thread qui avait appel  join
    currentThread->space->semJoinThreads[zone]->V();
    return 0;
}
```

3.6 Améliorations possibles

3.6.1 Les identifiants de threads

Pour éviter de réutiliser les identifiants de threads, nous avons mis en place un compteur qu'on incrémente. Il faudrait prévoir un mécanisme qui remet à zéro ce compteur.

3.6.2 Gestion de l'espace des threads

Un handicap important de notre implémentation est la gestion des dépassements mémoire : on ne garantit à **aucun moment** qu'un thread ne va pas dépasser sa pile, et donc il peut très bien écraser la mémoire d'un autre thread d'un autre thread. Il faudrait mettre en place des mécanismes pour limiter l'utilisation de la pile d'un thread à son espace propre. Par exemple, vérifier à chaque lecture memoire, qu'il est autorisé à écrire/lire.

Une des améliorations possibles, serait la gestion dynamique de la taille de la pile d'un thread. Dans notre solution (3 pages par thread) on a soit de la fragmentation interne soit des dépassements, mais même avec un mécanisme de non-dépassement, le thread peut planter, car il n'a pas la possibilité d'étendre sa pile. Une solution possible est d'utiliser pour chaque nouveau thread une taille de pile minimale (une page), mais en cas de dépassement, ajouter une nouvelle page a sa pile...ce qui revient à prévoir un mécanisme de segmentation inter-processus.