Université de Bordeaux
Département Master

NACHOS:Multithreading

ANABA Henri Frank
CONSTANS Olivier
TOUATI Amira

1ère Année de Master Informatique
2014-2015

Table des matières

Chapitre 1

Bilan

1.1 Multithreading dans les programmes utilisateurs

Dans cette partie nous avons essayé de permettre aux programmes utilisateurs de créer un thread fils avec un appel système qui utilisera le thread noyau. Nous avons donc mis en place l'interface des appels système *ThreadCreate* et *ThreadExit* .

1.1.1 ThreadCreate

L'appel système *ThreadCreate* crée un nouveau thread, l'initialise et le place dans la file d'attente des threads grâce à la fonction *do_ThreadCreate* qu'on a implémenté, Cette dernière instancie le thread et lance la fonction *StartUserThread*. Celle-ci permet de mettre à jour les registres. Le registre *PCReg* contiendra la fonction qui sera exécuté par le thread fils, le registre 4 aura les paramètres de la fonction, tandis que le registre *StackReg* aura l'adresse du thread fils, on mettra également à jour le registre - *NextPCReg* pour y mettre l'adresse de la prochaine instruction.
Quant à l'espace d'adressage du thread crée on a du implémenter la méthode *AllocateUserStack* qui retourne l'adresse de la pile (les premiers 256 octets en-dessous de la pile du thread principal). Mais la création du thread peut quand même échouer dans le cas où il existe plusieurs threads et que la pile du thread principal *UserStacksAreaSize* est saturée.

1.1.2 ThreadExit

L'appel système *ThreadExit* appelle la fonction *do_ThreadExit* qui détruit le thread propulseur par l'appel *currentThread-¿Finish* . on utilise cela pour que le thread se termine lui même mais pas le processus.

1.2 Plusieurs threads par processus

Dans cette partie on a cherché à améliorer notre implémentation pour gérer la création et l'exécution de plusieurs threads.

1.2.1 Gestion de la synchronisation des threads

Pour permettre au différents threads de s'exécuter en même temps nous avons protégé les fonctions noyau *PutChar* et *GetChar* par un verrou.
On peut utiliser deux verrous différents, car un thread peut lire pendant qu'un autre thread écrit sans que cela ne pose problème.
Par contre on a pas eu besoin de protéger les fonctions *PutString* et *GetString* car ils utilisent les fonctions *PutChar* et *GetChar* qui sont déjà protégées. Cependant nous aurions pu rajouter des sémaphores pour faire en sorte que les fonctions récupèrent des chaînes de caractères consécutives. Or ce comportement n'est pas celui donné dans Unix, donc nous avons décidé de ne pas l'ajouter.

1.2.2 Gestion de la mémoire des threads

La création de plusieurs threads n'était pas possible au début a cause de l'allocation de pile qui était trop simpliste. C'est à dire, la fonction *AllocateUserStack* retournais toujours la même valeur, ainsi tous les threads avaient la même pile. Pour régler ce problème, nous avons utiliser la classe *Bitmap* qui nous a permis de gérer la disponibilité et la répartition des pile threads.

1.3 Terminaison automatique

Au début tous les threads doivent se terminer par l'appel système *ThreadExit* sinon les threads continue à exécuter du code qui n'est pas le leur et provoque des erreurs . Pour régler ce problème, nous avons mis l'adresse de

l'appel système *ThreadExit* dans le registre *RetAddrReg* . Ainsi le thread exécutera *ThreadExit* quand il finira sa fonction.

Chapitre 2

Points délicats et limitation

2.1 Points délicats

Dans un premier temps nous avons eu un problème avec la première partie du TP , en exécutant le fichier *makethreads.c* que nous avons mis en place pour tester le bon fonctionnement de notre implémentation, en utilisant un simple *PutChar* dans le thread crée, nous avons remarqué que le thread se crée mais que la machine s'arrête un peu tôt (avant que le thread fils ne puisse exécuter sa fonction). Cela était du au thread principal qui sortait de la fonction main et arrêtait la machine sans laisser le temps à l'autre thread de continuer à s'exécuter , on l'a donc fait attendre avec une boucle infini après la création du thread.
Mais nous avons amélioré notre implémentation en modifiant *ThreadExit* pour que ça compte le nombre de thread, et permette au dernier d'avertir le thread principal pour qu'il arrête la machine.

L'appel système *ThreadCreate* arrête la création du thread et retourne -1 quand il n'y a plus de place pour sa pile. Pour éviter cela, nous aurions pu utiliser un semaphore initialisé par le nombre de pile de thread maximal. Si un thread est créé alors qu'il n'y a plus de place pour sa pile, il doit attendre la fin d'un autre thread pour allouer sa pile sur emplacement de la pile du thread qui vient de se finir.

2.2 Limitation

Notre système ne peut exécuter plus de 4 threads en même temps. Nous aurons pu implémenter des méthodes pour ré-allouer la mémoire des pile thread.

Chapitre 3

Tests

Nous avons d'abord testé le fonctionnement de la création d'un seul thread exécutant un simple *PutChar* dans le fichier *makethread.c* . Les fichiers *multiThreadAvecPutChar.c* et *multiThreadAvecBoucleFor.c* démontrent le bon fonctionnement de plusieurs threads.
le fichier *multiThreadSansThreadExit.c* contient le programme test qui démontre le bon fonctionnement de la terminaison automatique des threads. Quant au fichier *TestBitMap.c* , il teste le bon fonctionnement de *ThreadCreate* dans le cas ou nous voulons créer plus de threads que possible. Une petite documentation permettant de comprendre le test est disponible en début de chaque fichier.

```
Dans le répertoire userprog :
            ./nachos -rs -x ../test/makethreads
            ./nachos -rs -x ../test/multiThreadAvecPutChar
            ./nachos -rs -x ../test/multiThreadAvecBoucleFor
            ./nachos -rs -x ../test/multiThreadSansThreadExit
            ./nachos -rs -x ../test/TestBitMap
```