NACHOS : Multithreading

Manon Philippot - Anthony Delasalle

20 Novembre 2017

1 Bilan

1.1 Objectifs

L'objectif de ce devoir est de permettre d'exécuter des applications multithreads sous Nachos.

1.2 Multithreading dans les programmes utilisateurs

Dans cette partie, l'objectif était 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*.

NachOS permet d'instancier des threads par un appel à leur constructeur. Ce thread dispose d'un nom, d'une pile, d'un espace d'adressage ainsi que d'un état qui au moment de sa création est "JUST_CREATED". Le thread nouvellement instancié hérite son espace d'adressage de son père. Ce thread est ensuite placé dans la file d'attente des threads avec la fonction *Start* qui prend en paramètre un pointeur vers une fonction que le thread devra exécuter et un pointeur vers les arguments de cette fonction. Dans le cas du tout premier thread NachOS, celui-ci est intancié dans la mémoire virtuelle de nachOS en tant que thread noyau. Nous avons donc tout d'abord mis en place l'interface des appels systèmes *ThreadCreate* et *ThreadExit* sur le même modèle que dans le devoir précédent : ajout de l'appel système et de la déclaration de la fonction dans le header */userprog/syscall.h*; ajout de la définition en assembleur de l'appel système dans le fichier *test/start.S*; mise en place du traitant d'interruption dans *userprog/exception.cc*. L'appel système doit créer un thread *utilisateur* avec comme argument la fonction *f* et les arguments *arg*. La création d'un thread pouvant échouer lorsque toute la mémoire virtuelle est occupée, cet appel système renvoit *-1* en cas d'erreur. L'appel système *ThreadExit* a pour but de détruire le thread courant.

Nous avons alors commencé par l'écriture de la fonction *do_ThreadCreate(int f, int arg)* qui créé un thread à partir de la fonction située à l'adresse f et des arguments situés à l'adresse arg. Cette fonction instancie un nouveau thread et

retourne -1 si jamais l'instanciation ne s'est pas correctement effectuée. On a ensuite besoin de faire un appel à *Start* pour que notre thread puisse être lancé. Le premier argument que doit prendre en paramètre *Start* est une référence vers la fonction *StartUserThread* dont nous parlerons juste après. Ce qui ne nous laisse qu'un seul argument disponible selon la déclaration de *Start* pour transmettre le nom de la fonction que devra exécuter notre thread et ses arguments. Nous avons remédié a ce problème en créant une structure *Package* qui contient deux entiers et qui servira d'argument . Le premier représente un pointeur vers la fonction à éxécuter et le second un pointeur vers les arguments de cette dernière. Ainsi on instancie un élément *Package* qui contient *f* et *arg* les paramètres de *do_ThreadCreate* et on le passe en argument de *StartUserThread*.

La deuxième fonction qu'on a implémentée est donc la fonction *StartUserThread(void *recup)*. On commence par initialiser tous nos registres à 0. On écrit ensuite dans le registre PCREG l'adresse de la fonction *f*, dans le registre 4 ses arguments *arg* et on alloue au thread un espace mémoire pour sa pile grâce à *AllocateUsertStack*. Cette dernier nous permet d'éviter qu'un thread fils écrive par dessus la pile de son thread parent en effectuant un calcul permettant de connaître la position dans la pile père à partir de laquelle le thread fils peut écrire.

Il ne nous reste plus qu'à implémenter la fonction *do_ThreadExit()* qui correspond à l'appel système *ThreadExit*. Celle-ci nous permet de mettre fin au thread créé grâce à *currentThread->Finish()*.

1.3 Plusieurs threads par processus

L'implémentation ci-dessus était encore primitive et pouvait être améliorée. Lorsqu'on essayait de faire des écritures à la fois depuis le thread principale et le thread créé, un message *assertion failed* intervenait. Cela était du aux requêtes d'écriture et d'attente d'acquittement des deux threads qui se mélangeaient. Pour corriger ce problème nous avons décidé d'utiliser deux sémaphores dans *exception.cc*, déclarés et initialisés dans *system.h*, pour placer les appels système de lecture et d'écriture en section critique. Cette solution marche.

Si un thread appelle *Exit* 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 remédier à ce problème, nous avons déclaré en static un entier *nbThreads* initialisé à 1 (thread principal), qui est incrémenté à chaque création de thread avec *do_ThreadCreate* et décrémenté à chaque destruction de thread avec *do_ThreadExit*. À la destruction d'un thread, si celui-ci est le dernier, alors Nachos se termine avec *interrupt->Halt()*. Nous avons utilisé un sémaphore pour mettre *nbThreads* en section critique.

Nous avons implémenté notre mécanisme d'allocation de pile en utilisant la classe Bitmap dans la classe *addrspace.cc*. Dans le constructeur *AddrSpace*, un

bitmap a été déclaré et initialisé avec comme paramètre le nombre de threads possibles. Ce nombre est déterminé en divisant *UserStacksAreaSize* défini dans *addrspace.h* par la taille d'un programme (256+16). La fonction AllocateUserStack retourne une adresse disponible pour la pile d'un nouveau thread. Elle cherche dans le bitmap un espace disponible, puis calcule l'adresse en mémoire correspondant et la retourne. La fonction DeAllocateUserStack prend une adresse en paramètre et désalloue l'espace du bitmap correspondant à cette adresse. Un mutex a été utilisé pour mettre tous les traitements sur notre bitmap en section critique.

1.4 Terminaison automatique (bonus)

Lorsqu'un thread n'appelle pas ThreadExit, Nachos nous informe qu'il y a une erreur (segmentation fault). Pour éviter de devoir appeler explicitement ThreadExit pour que chaque thread se termine, nous avons modifié les fichiers *start.S*, *userthread.cc* et *exception.cc*.

Dans *start.S*, au niveau de *__start*, nous avons remplacé *jal exit* par *jal ThreadExit* afin de changer le registre de retour et que la fonction *main* finisse sur un appel système *ThreadExit*. Dans le même fichier au niveau de ThreadCreate nous avons ajouté la ligne *addiu $6,$0,ThreadExit* afin de mettre l'adresse de ThreadExit dans le registre 6 lors de la création d'un thread.

Dans le fichier *exception.cc*, nous récupérons la valeur stockée dans le registre 6 et la passons en 3ème argument de la fonction *do_ThreadCreate*.

La fonction *do_ThreadCreate* prend maintenant 3 arguments. Le but de cette manoeuvre est de passer au niveau de la fonction *StartUserThread* la valeur stockée dans le registre 6. Pour cela nous avons modifié la structure Package afin de contenir un troisième entier *ret* et l'avons initialisé avec le troisième argument passé en paramètre. Dans *StartUserThread* nous avons changé la valeur du registre de retour *RetAddrReg* par la valeur stockée dans le registre 6 récupérée dans *ret*.

La terminaison automatique marche.

1.5 Sémaphores (bonus)

Nous avons remonté l'accès aux Sémaphores au niveau des programmes utilisateurs en implémentant l'interface des appels systèmes *SemaphoreCreate*, *SemaphoreDelete*, *P* et *V*, les traitants d'interruption dans *userthreads.cc* et les fonctions utilisées par ces traitants dans *userthreads.cc*.

Dans *userthreads.cc*, la gestion des sémaphores utilisateurs est effectuée grâce à un tableau de Sémaphore associé à un BitMap, et grâce aux fonctions *do_SemaphoreCreate*, *do_SemaphoreDelete*, *do_SemaphoreP* et *do_SemaphoreV*.

La fonction *do_SemaphoreCreate* créé un Sémaphore (de valeur passée en paramètre) dans le tableau *semTab* à l'indice trouvé par la méthode *Find* du bitmap et retourne l'indice.
La fonction *do_SemaphoreDelete* supprime le sémaphore situé à l'indice donné en paramètre du tableau de sémaphores.
La fonction *do_SemaphoreP* effectue un verrou *P* sur le sémaphore situé à l'indice donné en paramètre du tableau *semTab*.
La fonction *do_SemaphoreV* effectue un verrou *V* sur le sémaphore situé à l'indice donné en paramètre du tableau *semTab*.
Le sémaphore *mutexBitmap* est utilisé pour placer tous les traitements du BitMap en section critique.

Dans le fichier *exception.cc* sont implémentés les traitants d'interruption pour les appels systèmes des sémaphores.
Un appel système *SemaphoreCreate* consiste à appeler la fonction *do_SemaphoreCreate(nbCredit)* avec comme paramètre la valeur récupéré dans le registre des arguments, et à placer sa valeur de retour dans le registre de retour de l'appel système.
Un appel système *SemaphoreDelete* consiste à appeler à récupérer l'identifiant (entier) du sémaphore passé en paramètre de l'appel système et à appeler la fonction *do_SemaphoreDelete(id);* avec en paramètre l'identifiant.
Un appel système *P* consiste à appeler à récupérer l'identifiant (entier) du sémaphore passé en paramètre de l'appel système et à appeler la fonction *do_SemaphoreP(id);* avec en paramètre l'identifiant.
Un appel système *V* à récupérer l'identifiant (entier) du sémaphore passé en paramètre de l'appel système et à appeler la fonction *do_SemaphoreV(id);* avec en paramètre l'identifiant.

2 Points délicats

L'aspect qui nous a demandé le plus de travail est sûrement l'utilisation de la classe BitMap pour le mécanisme d'allocation de pile ainsi que la terminaison automatique.

Ce qui nous a posé le plus problème avec l'utilisation de la classe BitMap pour le mécanisme d'allocation a été les calculs de conversion entre indice du bitmap et adresse de la pile du thread créé. De plus, lorsque nous créions plus de 2 threads *utilisateur*, nous obtenions un message d'erreur lors de l'exécution. Cependant on s'est rendu compte que cela était normal, la variable *UserStacksAreaSize* étant de 1024 et ne permettant donc la création que de 2 threads *utilisateur* de taille (256+16), en plus du thread principal.

La terminaison automatique nous a posé problème dans le sens où nous avons mis du temps à comprendre comment appréhender le problème.

3 Limitation

Notre implémentation de *SemaphoreDelete* ne semble pas fonctionner. En effet elle entraîne un message d'erreur lorsqu'utilisée. En exécutant valgrind sur un simple programme, on remarque une fuite mémoire. Cela s'explique en partie par le fait que nos sémaphores ne sont jamais supprimés, et également car certaines structures n'ont pas été supprimées (bitmap dans *userthread.cc* par exemple).

4 Tests

Nous avons implémentés 2 tests. Le premier *makethreads.c* teste la création de threads ainsi que le fonctionnement des sémaphores utilisateurs. Il suffit de saisir par exemple la commande *./userprog/nachos -rs 12345 -x ./test/makethreads* lorsque situé dans le dossier *code*. Il fonctionne. Le second *producteurconsommateur.c* est censé illustrer un exemple de producteurs-consommateurs au niveau utilisateurs. Il ne fonctionne pas encore mais compile.