Rapport du Devoir 2 : Multithreading dans NACHOS

De Florian Claisse et Maxime Zielinger

Bilan

Table des matières

Questions du devoir

I. 1.

Comment ces threads sont-ils alloués et initialisés ?

Les threads sont alloués dans Thread.cc en initialisant stack_size et main_stack à 0. Leurs sont attribués un nom pour les distinguer et un statut pour indiquer leurs différents états.

Où se trouve la pile d'un thread Nachos,en tant que thread noyau ?

En tant que thread noyau, la pile du thread noyau se trouve dans la zone mémoire réservée au processus. Cette zone fait la taille allouée moins l'espace prit par le code et les data. Dans notre cas, la pile du thread noyau fait une taille de 256 octets et se trouve en haut de la stack moins seize (défini en dur dans le code)

I. 2.

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, ainsi que dans le fichier userprog/addrspace.cc

Dans userprog/progtest.cc on récupère l'exécutable passé en paramètre dans le StartProcess, que l'on donne pour la création de la nouvelle zone mémoire dans new

AddrSpace(executable). On associe alors cette nouvelle zone mémoire au thread courant (dans ce cas le thread main), puis on initialise tous les registres avec InitRegisters(). Enfin, grâce à l'appel de machine->Run() on lance le processus/programme.

Dans addrspace.cc l'installation dans la mémoire commence à se faire dans la fonction AddrSpace::AddrSpace(...) à la ligne 167 avec l'initialisation d'abord, de la zone mémoire réservée au code du programme, puis avec l'initialisation de la zone mémoire réservée aux données que contient le programme. Une fois cela fait, il reste une zone mémoire non initialisée qui est réservée pour la stack (SP).

I. 4.

Pour quelle(s) raison(s) la création d'un thread pourrait-elle échouer ?

La création d'un thread peut échouer pour plusieurs raisons :

Tout d'abord, s'il n'y a plus de place dans l'espace mémoire.

Il peut aussi échouer s'il n'y a plus de place dans l'espace d'adressage.

I. 6.

Doit-on faire quelque chose pour son espace d'adressage space ?

Après la destruction du thread, l'espace d'adressage devient inutilisé. Au moment de recréer un thread à cet emplacement, les données déjà présentes dans l'espace seront simplement écrasées par les nouvelles données. Bien sûr, si aucune nouvelle donnée n'arrive, il serait tout à fait possible de retrouver ces valeurs.

II. 1.

Faut-il également protéger PutString et GetString? Pour quelle raison ?

Nous pouvons utiliser deux verrous différents sous la forme de sémaphores. Il s'agit d'ailleurs de la solution que nous avons privilégiée pour répondre au problème. Un pour GetChar, et un autre pour PutChar.

PutString et GetString ont, eux aussi, besoin d'une sécurité. Par exemple, si nous découpons une chaîne de caractères en deux et que nous passons chaque partie à un thread différent, il faut s'assurer que la chaîne soit restituée en entier et correctement en sortie.

II. 2.

Est-ce que Nachos se termine effectivement si à la fois le thread créé et le thread initial utilisent ThreadExit?

Si aucune vérification du nombre de threads lancés n'est faite, terminer tous les threads revient à empêcher le système de s'arrêter car aucun des threads ne fait appelle à machine-> Shutdown() à l'aide d'un signal Halt ou Exit.

II. 3.

Que se passerait-il si le programme lançait plusieurs threads et non pas un seul ?

En lançant plusieurs threads en même temps, la position de chaque thread dans la pile serait la même. Toutes les variables seraient donc communes et la modification d'une variable dans un thread entraînerait la modification de cette dernière dans les autres threads aussi.

II. 4.

Que se passe-t-il si un programme lance un grand nombre de threads ?

Si le programme lance un grand nombre de threads, les premiers se lanceront correctement, puis les suivants, n'ayant plus de place dans la pile de stack, continueront de descendre dans l'espace mémoire et donc de toucher à des données ou du code. Cela crée une certaine perte de contrôle des threads et plus généralement de l'exécution du programme, ce qui est, logiquement, particulièrement dangereux.

III. 1.

Expliquez ce qui adviendrait dans le cas où un thread n'appellerait pas ThreadExit.

Si un thread n'appelle pas ThreadExit, son programme se termine mais ne libère pas sa mémoire. Le thread main n'ayant pas de retour du thread en question, il ne décrémente jamais son compteur. Appliqué à tous les threads, cela empêche le thread main de créer un nouveau thread. Appliqué aussi au thread main, cela crée une boucle infinie.

Points délicats

De façon assez attendue, nous avons eu des difficultés quant à la gestion de l'espace disponible pour les threads. En effet, ce dernier est assez restreint et ne permet de créer que quatre threads en même temps. Cette limitation nous a poussé à réfléchir sur les façons de réutiliser l'espace, sans quoi notre système se retrouverait limité.

De plus, le TD n'étant que très peu guidé, nous ne savions pas toujours où ajouter les fonctions demandées. Nous avons, au maximum, essayé de respecter une implémentation que nous jugions logique. Certaines fonctions ou déclarations pourraient ne pas être considérées comme à leurs places, mais nous n'avions pas d'informations nous permettant de les placer correctement.

Limitations et choix d’implémentation

Limitations

Comme vu précédemment, l'espace disponible pour gérer les threads est de 1024 octets, coupé en quatre threads de 256 octets. C'est assez peu. Nous avons pensé à augmenter cette limite de 1024 octets, qui est simplement une variable à modifier, mais notre objectif premier est resté de n'utiliser cette solution qu'en dernier recours et essayer au maximum de jouer le jeu. Nous verrons plus tard qu’il existe des solutions, et que nous en avons trouvé une qui nous paraît satisfaisante.

Sans l’option -rs , le thread main ne donne la main aux autres threads que lorsque le code qu’il doit exécuter est terminé. Il s’agit d’une particularité de nachos sur laquelle nous ne pouvons pas impacter.

Le nombres de threads simultanés est limité par la taille de l’espace d’adressage et la taille de la pile d’un thread. Dans le cas où un thread n’aurait pas pu être créé, un code d’erreur -1 est renvoyé lors de l’appel à do_ThreadCreate(). Pour ce faire, lors de l’appel à do_ThreadCreate() dans exception.c, on stock cette valeur de retour puis on se charge de l’écrire dans le registre 2 pour que l’appelant puisse la recevoir.

Choix d’implémentation

Dans addrspace.h nous avons défini le nombre de pages d’un thread à 2, ce qui correspond à 256 octets au total par thread, comme demandé dans le sujet.

Pendant le développement, nous avons réalisé qu’AllocateUserStack combinait deux fonctionnalités qui nous étaient plus utiles séparées. Nous avons donc décidé de la partager en deux. D’un côté, AddrSpace::GetNewZone() donne la nouvelle zone dans la bitmap. De l’autre, AddrSpace::InitThreadRegister() set le StackReg ainsi que tous les autres registres de notre nouveau thread à la bonne valeur.

Nous avions comme limite de ne pouvoir passer qu’un argument à la fonction StartUserThread à cause de Thread::Start(). Pour pouvoir envoyer tout le nécessaire malgré tout, nous avons retenu comme solution un tableau. Celui-ci est alloué dynamiquement et stocke les valeurs des arguments nécessaires à la fonction StartUserThread. Dans ces arguments, nous retrouvons l’adresse mémoire de la fonction que doit appeler le nouveau thread, l’adresse mémoire de l’argument donné par l’utilisateur pour la fonction qui va être appelée et enfin l’adresse mémoire de la fonction ThreadExit() comme fonction appelée par défaut lors de la fin d'exécution d’un thread.

Comme autre solution possible, nous aurions pu utiliser une structure, mais nous avons préféré garder un tableau, plus simple et mieux adapté à nos besoins.

Cette adresse de retour vers ThreadExit() fait partie de l'un des bonus proposés dans l'énoncé. Cela se fait avec start.s dans lequel nous passons cette valeur grace à "addiu $6, $0, ThreadExit"

Exemple du code de ThreadExit dans ThreadCreate :

```
.globl ThreadCreate
.ent   ThreadCreate
ThreadCreate:
addiu $2,$0,SC_ThreadCreate
addiu $6, $0, ThreadExit
syscall
j       $31
.end   ThreadCreate
```

Avec exception.c où il est récupéré puis on l'envoi en troisieme parametre de do_ThreadCreate. Pour finir, au moment d'initialiser les registres, on affecte la valeur du registre 31 avec ce dernier.

Pour gérer plusieurs thread, nous sommes contraints de modifier l'espace d'adressage (AddrSpace) du processus. Nous avons, entre autres, ajouté une variable qui compte le nombre de threads en cours d'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 et chaque pile est une partie de la pile du processus (dont sa taille est de 256 octets soit 2 pages). Nous sommes donc limités en nombre de threads simultanés par processus. Lorsque cette limitation est atteinte, on renvoie un code d'erreur -1 que l'utilisateur peut récupérer pour prendre compte qu'un ou plusieurs threads n'ont pas pu être lancés.

L'espace d'adressage est la partie commune de tous les threads qui tournent sur un même processus. C'est pour cela que nous avons ajouté plusieurs méthodes pour faciliter le développement. Par exemple, pour savoir si un thread est le dernier à s'exécuter, il peut faire appel à AddrSpace::ThreadAlone() qui va garantir que cette vérification est protégée par des sémaphores. ou, comme dit précédemment, AddrSpace::GetNewZone() qui va nous donner la position de la nouvelle zone libre pour ce thread (ou -1 en de cas problème d'espace). Enfin AddrSpace::InitThreadRegisters() pour initialiser tous les registres du nouveau thread.

Comme nous avons ajouté beaucoup d'éléments pour gérer les threads utilisateur, il fallait aussi que le thread main, qui n'est pas instancié à la demande de l'utilisateur, ait

un fonctionnement similaire aux autres threads. C’est pour cela que dans addrspace.cc, nous avons ajouté une fonction InitMainThread qui a pour but que ce thread main occupe une place dans la bitmap qu’il va libérer au moment de sa destruction. AddrSpace::InitMainThread est appelée dans le fichier userprog/progtest.cc dans la fonction StartProcess.

A son lancement, le thread main incrémente le compteur de threads. S’il ne fait pas d’appel explicite à Exit() ou qu’il ne fait pas de return, la machine ne s’arrêtera que lorsque le dernier thread sera terminé. Cependant, un appel à Exit() ou un return arrêtera la machine sans attendre les autres threads.

Pour finir, nous gardons la zone mémoire donnée à un thread grâce à la propriété “int zone” qui contient ladite zone donnée dans la bitmap pour le thread en question.

les fonctions de threads/thread.h Thread::getZone() et Thread::setZone(...) permettent de manipuler cette variable.

Pour le dernier bonus sur l'élévation des sémaphores au niveau des programmes utilisateurs, nous avons d’abord commencé par ajouter dans le fichier userprog/syscall.h les fonctions nécessaires à l’utilisateur pour la création, blocage, déblocage et suppression d’un sémaphore.

A la création d’un sémaphore, il entre dans un tableau duquel sort un entier (sa position dans le tableau des sémaphores) ainsi qu’un numéro dans la bitmap. L’utilisateur peut ensuite manipuler l’entier envoyé, nous permettant de savoir à quel sémaphore il fait référence en appelant semWait(...), semPost(...) et semDelete(...).

Pour réaliser cela, il a fallu modifier le fichier test/start.s pour ajouter les appels système rattachés aux appels SemCreate(), SemWait(), SemPost() et SemDelete().

Le fichier de test demandé se trouve dans test/semaphore.c.