Rapport Devoir 2

Système d'exploitation

NACHOS : Threads utilisateur

Guillaume NEDELEC et Alfred Aboubacar SYLLA

Rendu le : mardi 20 Novembre 2018

I. Bilan

Ce TP consiste à la mise en place de threads utilisateurs au moyen d'appels système qui utilisent des threads noyau pour propulser les threads utilisateurs.

Part 1. Multithreading dans les programmes utilisateurs

Dans un premier temps nous avons développé un seul thread supplémentaire. Il a donc fallu créer les deux appels système `ThreadCreate` et `ThreadExit`. L'appel système `ThreadCreate` permet de créer un nouveau thread utilisateur.
Pour créer ce thread, on enregistre la fonction que le thread doit exécuter ainsi que les arguments de cette fonction.
On exécute ensuite le thread (grâce à la fonction `StartUserThread(void *schmurtz)`) en ré-écrivant la valeur de `PCReg` avec l'adresse de notre fonction permettant d'indiquer que la prochaine instruction à lancer est la fonction que l'on souhaite. On alloue ensuite de la mémoire à notre thread en attribuant à la valeur `StackReg`, l'adresse du haut de la nouvelle pile retournée par `AllocateUserStack()` de `addrspace.h`
Le thread est ensuite lancé grâce à `machine->Run()`.

La fonction `ThreadExit` quant à elle permet de terminer le thread avec la méthode `Finish()` de `thread.h`.

Part2. Plusieurs threads par processus

L'objectif de cette partie est de pouvoir lancer plusieurs threads en parallèles. Pour ce faire il a fallu rajouter des verrou sur nos fonctions de `synchconsole.cc` afin que les threads n'interfèrent pas sur le fonctionnement des autres threads en utilisant ces méthodes (comme de la lecture simultanée de caractères qui entraîne l'oubli de lecture d'un caractère puisque les 2 threads auront lu le même).

Il a ensuite fallu gérer le cas où, dès que le `main` appelle `Exit`, nachos s'arrête et ne laisse pas les autres threads se terminer.
Pour remédier à ce problème, on a considéré que le `main` était un thread comme un autre et qu'il se termine avec la méthode `ThreadExit` lui aussi.

Nous avons utiliser la classe `BitMap` nous permettant de savoir s'il reste de la place dans l'espace d'adressage, pour allouer de la mémoire aux nouveaux threads.
Nous avons donc rajouter un champ dans `thread.h` représentant le slot dans l'espace d'adressage qui est attribué à chaque thread.

Dans un premier temps, dans le constructeur de `AddrSpace` (appélée par le `main`), nous avons créer le `BitMap` (sa taille correspond à la taille de la pile d'adressage divisé par la taille d'un thread) et alloué le slot 0 au thread principal (le `main`).
Ensuite, à chaque fois que l'on désire créer un thread, on utilise le `BitMap` pour savoir s'il reste un slot dans l'espace d'adressage pour lui allouer de la mémoire (avec la méthode `BitMap::Find()`). Si ce n'est pas le cas, le thread n'est pas créé et on prévient l'utilisateur que l'espace d'adressage est complet.
Si il reste de la place, alors le thread est créé en lui attribuant ce slot (au niveau du thread : `thread->setSlot(slot)` et au niveau de l'espace d'adressage `BitMap::Mark()`).
On incrémente ensuite un compteur afin de savoir combien de threads sont en cours d'exécution (l'incrémentation et la décrémentation de ce compteur sont verrouillées par un sémaphore).

Lors de la suppression on vide l'espace d'adressage du thread avec la méthode `BitMap::Clear()` puis on décrémente le compteur de threads. Si ce compteur est supérieur à 0 alors on utilise la méthode `Thread::Finish()` pour clôturer le thread sinon on utilise `Halt()` pour arrêter nachos (un compteur à 0 indique qu'aucun processus ne s'exécute).

Part3. Terminaison automatique (bonus)
Le dernier problème auquel nous avons été confronté, est le fait de devoir explicitement appelé `ThreadExit()` dans le code pour chaque thread.
On a donc modifier le fichier `start.S` afin que `ThreadExit` soit appelé automatiquement à la fin d'un thread.
Pour le programme `main`, nous avons donc changé la ligne:
"`jal Exit`" en "`jal ThreadExit`".
Mais pour que les autres threads utilise aussi `ThreadExit` en sortie, il a fallu enregistrer l'adresse de la fonction `ThreadExit` dans un registre à l'appel de `ThreadCreate`. On a donc rajouter la ligne "`la $6, ThreadExit`" dans le fichier `start.S`, au niveau de la définition de `ThreadCreate`.
Ensuite dans le programme, il a seulement fallu récupérer l'adresse dans le registre et la passer en paramètre au nouveau thread pour qu'à sa création, on remplace la valeur initiale de `RetAddrReg` par celle de `ThreadExit` récupérée précédemment.
`RetAddrReg` représente l'adresse de retour du processus. Le nouveau thread appellera donc `ThreadExit` pour se terminer à l'issue de cette affectation.

II. Points Délicats

Nous avons eu du mal à prendre en main la classe `BitMap`. Après avoir compris son fonctionnement, nous n'avons pas eu tout de suite l'idée d'enregistrer le slot réservé dans le thread, ce qui a causé beaucoup de problème à la libération de l'espace mémoire. L'ajout d'un attribut dans la classe `Thread` a rapidement résolu le problème.

Pour faire la terminaison automatique des threads, la sauvegarde de l'adresse `ThreadExit` dans un registre au niveau du fichier `start.s` se fait avec l'instruction "`la`". Cette instruction n'a pas été facile à trouver, nous avons essayé de faire des ajout avec l'instruction "`addiu`".

III. Limitations

La taille de la pile fait que l'on ne peut avoir un nombre important de threads qui tournent en même temps. Cette limite de l'espace d'adressage entraîne donc beaucoup de dépassement de pile et donc il n'y a pas de création de nouveau thread.
Lorsqu'il n'y a plus de place dans l'espace d'adressage, nous faisons apparaître dans le terminal "Stack overflow" pour prévenir l'utilisateur.

Nous n'avons pas eu le temps d'implémenter la partie 4 sur l'accès aux sémaphores au niveau des programmes utilisateurs.

IV. Tests

Les tests ont été effectués sur le fichier `test/makethreads.c`.
Dans une premier temps nous avons testé la création d'un thread c'est à dire de l'appel système `ThreadCreate` en lui donnant en paramètres la fonction qu'il doit exécuter ainsi que ces paramètres. On a utilisé les fonctions `printChar`, `printInt`, `printString` qui font appel aux appels systèmes suivant : `PutChar`, `PutInt et PutString` qui permettent d'afficher des caractères dans la console.
En créant un thread appelant une de ces méthodes en plus de l'appel de PutChar (ou des autres méthodes) dans le main, on a pu constater le fonctionnement en parallèles des deux processus par l'affichage mélangé des deux processus.

Dans nos fonctions on a fait appel à `ThreadExit` pour tester la terminaison des threads.
Avant d'implémenter le compteur de thread, nachos ne s'arrêtait jamais puisque à la fin du main, `ThreadExit (Thread::Finish())` était appelé, n'arrêtent jamais nachos avec `Halt()`.
Ensuite pour tester la cohabitation de plusieurs threads, nous avons donc créer une boucle créant 10 threads affichant chacun une des 10 premières lettres de l'alphabet.
Bien sur, l'espace d'adressage n'est pas suffisant pour 10 threads, nous avons donc eu plusieurs affichage de notre alerte "`Stack Overflow`".
Avant d'avoir implémenter les sémaphore dans le fichier `synchconsole.cc` nous nous sommes aperçu que certains affichage n'apparaissait pas (sans compter les threads non créé pour cause de pile pleine).
Après avoir implémenté les sémaphores, tous les threads créés affichaient bien leurs lettres.

Enfin il nous a juste fallu tester la terminaison automatique des threads avec `ThreadExit`. Nous avons donc commenté chaque terminaison de notre fichier de tests. Nous avons eu des erreurs en essayant de changer directement la fonction de retour de `ThreadCreate` en remplacement le "`j $31`" par "`j ThreadExit`". Cela ne marchait pas puisque cela agissait sur le thread parent et non pas sur le nouveau thread créé.
Après avoir faire les bons changement (voir part3 dans le Bilan), tout fonctionne correctement.