Systèmes d'exploitation : Devoir 2

GROUPE 3
Thierno Sambegou DIALLO
Nathan LESNE

Novembre 2019

1 Bilan

Cette partie visait a implémenter la manipulation des threads utilisateurs.
Coté utilisateur nous avons implémenté les appels systèmes suivants:

- int ThreadCreate(f,arg) : Permet la création et le lancement d'un thread utilisateur exécutant f.
- void ThreadExit() : Termine le thread utilisateur courant.

Les stacks des threads utilisateurs sont alloués dans des slots de taille fixe à l'aide d'une BitMap.

Les threads se terminent automatiquement a la sortie de leur fonction, quand il ne reste plus qu'un seul thread, le dernier stoppe Nachos. Cela vaux également pour le thread principal.

Nous avons également implémenté les sémaphores cotés utilisateur.
Pour cela nous avons crée un type sem_t, c'est un entier qui sert d'id et qui correspond a une Semaphore du noyau.
L'utilisateur manipule les sémaphores grâce aux appels système suivants:

- sem_t SemInit(int) : Initialise une sémaphore à la valeur passé en paramètre, renvoie l'id de cette-ci.
- int SemDestroy(sem_t) : Libère la sémaphore correspondant à l'id passé en paramètre si elle existe.
- int SemWait(sem_t) : Appelle P() sur la sémaphore correspondant à l'id passé en paramètre si elle existe.
- int SemPost(sem_t) : Appelle V() sur la sémaphore correspondant à l'id passé en paramètre si elle existe.

Enfin, les fonctions PutChar et GetChar de SynchConsole ont été rendues thread-safe, l'affichage provenant de celle-ci ne sautera donc aucune lettre.
Sachant que nous disposons des sémaphores coté utilisateur, PutString et GetString n'ont pas été rendues thread-safe, nous jugeons que c'est à l'utilisateur de le faire.

2 Points délicats

Un des aspects les plus compliqués et source d'erreurs a été l'allocation des stacks des threads.
Nous sommes passés par plusieurs itérations avant d'avoir cette implémentation, qui n'est pas encore parfaite.

Tout d'abord, la première implémentation allouait juste la place non prise par le thread principal au premier thread utilisateur, les suivants ne sont pas crées, nous avons vite changés.

Ensuite, nous avions un compteur de threads dans userthread, et allouions des blocs de 256 en séquence.
Ceci est évidemment un problème si un thread se termine, on ne peut pas récupérer la place laissée.

Enfin, nous sommes passés à la BitMap qui permet de désallouer les slots de piles quand un thread se termine.

Néanmoins il reste un défaut majeur a cette implémentation, si tout les slots sont pleins et qu'un thread est crée, au lieu d'attendre qu'un slot se libère sa création est avortée.
C'est un défaut que nous n'avons pas corrigé par manque de temps, nous y avons néanmoins réfléchis, et une sémaphore pour que les threads soient bloqués avant l'allocation semble être une bonne solution.

3 Limitations

L'implémentation actuelle comporte plusieurs limitations:

Tout d'abord le nombre de threads qui peuvent être lancés est limité au nombre de slots de pile (BitMap). Si l'utilisateur appelle ThreadCreate alors que tout les slots sont pleins, la fonction renvoie un code d'erreur immédiatement.

Enfin, lancer un certain nombre de thread (nombre variable) cause des erreurs, la plus commune étant que les threads se créent mais ne s'exécutent pas. Une des causes du problème vient de l'implémentation de List, mais cette classe faisant partie de l'implémentation de base, et étant assez cryptique, nous n'avons pas réussi a corriger le problème.

4 Tests

Nous avons crée 2 programmes de tests qui couvrent tout les ajouts et changements dans cette partie:

makethreads illustre la création de threads avec ThreadCreate, et comment passer un paramètre à la fonction exécutée par celui-ci (ici une chaîne de caractères). Il montre également que les ThreadFinish se sont pas nécessaires, et qu'un thread utilisateur continue de s'exécuter après l'arrêt du thread principal.
Cependant, à cause d'un bug de l'implémentation, si l'on essaye de changer NB_THREADS a une valeur plus grande que 1, les threads ne sont pas exécutés.

producer_consumer est une implémentation du problème producteur-consommateur. Il montre tout ce qui est utilisé dans **makethreads** avec en plus une utilisation des sémaphores coté utilisateur.
Le programme principal lance des paires de thread producteurs et consommateurs, qui produisent et lisent des entiers en passant par un buffer commun, il attend que les consommateurs aient fini, puis se termine.
Grâce aux sémaphores, les threads sont coordonnés, et l'affichage est parfait.
En revanche, il existe le même bug que **makethreads** avec 2 paires de producteur-consommateur les threads ne sont jamais exécutés.