Systèmes d’Exploitation

Devoir 2 : Multithreading dans NACHOS

Laroque Austin & Malmgren Axel

I. Bilan

Nous avons implémenté la gestion des threads utilisateurs dans Nachos en suivant les étapes du TD2 et sommes arrivés à construire un système de multithreading fonctionnel.

La première étape a consisté à créer des threads utilisateurs via l'appel système ThreadCreate. Ces threads partagent le même espace mémoire que le thread principal mais ont chacun leur propre pile. Nous avons ajouté les stubs assembleur dans start.S et implémenté les fonctions noyau do_ThreadCreate et do_ThreadExit. L'appel système ThreadExit permet à un thread de se terminer proprement.

Nous avons ensuite ajouté la synchronisation pour la console. Sans protection, plusieurs threads affichant en même temps provoquaient des crashs. Nous avons utilisé deux sémaphores dans ConsoleDriver : un pour protéger les écritures (writeLock) et un pour les lectures (readLock).

Après cela, nous avons introduit un compteur de threads dans AddrSpace pour suivre le nombre de threads actifs. Quand un thread appelle ThreadExit, le compteur diminue. Si c'est le dernier thread, le système appelle automatiquement Powerdown() pour terminer Nachos proprement. Le thread principal peut donc se terminer avant les autres sans problème.

Nous avons ensuite implémenté la création de plusieurs threads simultanés grâce à une BitMap qui alloue les piles. Chaque thread reçoit un emplacement distinct pour sa pile, évitant les écrasements mémoire.

Tout fonctionne : plusieurs threads peuvent être créés, s'exécuter en parallèle avec l'option -rs, afficher en même temps sur la console, et se terminer proprement. Le système se termine uniquement quand le dernier thread a fini. Le code compile et s'exécute correctement. Les tests montrent un comportement stable même avec plusieurs threads actifs.

II. Points délicats

La partie la plus difficile a été le partage de la mémoire entre plusieurs threads. Chaque thread doit avoir sa propre pile tout en partageant le code et les données. Nous avons dû calculer correctement les adresses de pile et initialiser les registres avant de lancer chaque thread. L'allocation se fait en haut de la mémoire avec la formule numPages * PageSize - slot * StackSlotSize - 16.

L'autre difficulté concernait la destruction de la mémoire. Il fallait éviter qu'un thread supprime la mémoire alors que d'autres tournent encore. Pour cela, nous avons utilisé un sémaphore threadCountLock et un compteur threadCount dans AddrSpace. Quand le compteur atteint zéro, on appelle Powerdown() pour terminer le système. Cette section critique garantit qu'un seul thread vérifie et décrémente à la fois.

Un bug subtil s'est produit avec la console. Nous avions créé des variables globales readAvail et writeDone dans consoledriver.cc, alors que les mêmes noms existaient comme membres de classe. Le constructeur initialisait les globales, mais PutChar et GetChar essayaient d'utiliser les membres de classe qui étaient NULL. Le debugger avec les options -d x, -d t et -d m a aidé à trouver le problème. La solution a été de renommer les globales en g_readAvail et g_writeDone. Ces variables globales sont nécessaires pour que les handlers puissent accéder aux sémaphores, car ils ne peuvent pas utiliser le keyword “this”.

Nous avons aussi dû gérer le passage de deux paramètres (la fonction f et l'argument arg) à travers Thread::Start() qui n'accepte qu'un seul paramètre. Nous avons créé une structure ThreadParams allouée dans do_ThreadCreate et libérée dans StartUserThread après usage. Cette structure sert de conteneur pour transporter les données.

III. Limitations

Notre implémentation a plusieurs limitations. La terminaison est globale : le système s'arrête quand tous les threads sont finis. Cela fonctionne bien mais ne permet pas à un thread principal d'attendre un autre. Comme il n’y a pas de pthread_join, cela nous oblige à contourner.

Les piles ont une taille fixe de 256 octets par défaut. Si un thread déborde sa pile, il peut écraser celle d'un autre thread. Aussi, le nombre maximum de threads est limité par UserStacksAreaSize / StackSlotSize. Même si le système a des ressources disponibles, on ne peut pas créer plus de threads. Cette limite est gênante pour des applications avec beaucoup de threads.

Malgré ces limites, le système fonctionne . Les protections sont bien placées et le code est assez extensible pour ajouter facilement les sémaphores ou les mécanismes de join.

IV. Tests

Le programme makethreads.c crée un seul thread qui affiche des caractères puis se termine. Ce test simple valide les bases : la création et la terminaison d'un thread. Le programme putchar.c vérifie que le thread principal seul fonctionne toujours correctement.

Le test threadtest.c crée un thread pendant que le thread principal affiche aussi des caractères. Les deux threads utilisent PutChar en même temps et appellent ThreadExit. Avec l'option -rs, on voit les caractères s'entrelacer, ce qui prouve que l'ordonnancement fonctionne et que la console est protégée. Le programme doit se terminer proprement avec “Machine halting!”.

Ensuite, le programme multithread.c est notre test principal. Il crée plusieurs threads (selon NB_THREADS) qui affichent tous en boucle. Cela génère beaucoup d'accès simultanés à la console. Nous avons vérifié que tous les threads s'exécutent en parallèle et que le système ne se termine qu'après le dernier thread.

Pour vraiment tester le système, nous avons varié plusieurs choses. Différents seeds (-rs 1234, -rs 5678, etc.) changent l'ordre d'exécution et permettent de trouver des bugs qui dépendent du timing. Enfin, nous avons augmenté le nombre de threads jusqu'à la limite pour vérifier que le système gère bien l'erreur quand il n'y a plus d'emplacements.

Tous les tests donnent les résultats attendus. Le système se termine toujours proprement avec les statistiques. Aucune fuite mémoire ni corruption n'a été observée, confirmant que notre compteur de threads et l'allocation des piles fonctionnent bien.