Rapport : Multithreading dans NACHOS

MAHDI Yanis, LUBET Damien

November 12, 2024

Contents

1 Bilan

Dans le cadre de ce projet, nous avons implémenté un système permettant l'exécution de programmes multi-threads sous Nachos. Nous avons suivi une série d'actions méthodiques pour ajouter et tester les appels système `ThreadCreate` et `ThreadExit`, et assurer la création, l'exécution, et la terminaison de threads utilisateur ainsi qu'une gestion dynamique de la pile des threads avec l'aide de `Bitmap` permet de suivre les slots de pile utilisés, évitant ainsi les débordements de pile et facilitant une gestion plus efficace de la mémoire, particulièrement lorsque le nombre de threads est élevé. Nous avons de plus mis en place un système de verrou : Lock, pour les appels concurrents `PutString, PutChar` et `GetString, GetChar`. Nous avons également fait attention aux erreurs liées à la mémoire grâce à valgrind, cependant nous avons eu des erreurs de mémoire que nous n'avons pas réussi à résoudre quand `machine->DumpMem` est utilisé lorsque l'on créer plusieurs threads. Nous ne savions pas si c'est normal ou pas et avons donc décidé de le noter en commentaires dans le code car elles ne bloquent pas le bon fonctionnement du programme. Ainsi que l'ajout de 3 tests pour valider le bon fonctionnement de la création des threads. Plus de détails seront donnés dans la section Tests. Le bonus Terminaison automatique a aussi été particulièrement difficile à implémenter, le 2ème bonus quant à lui, n'est pas

terminé faute de temps. Plus de détails seront donnés dans la section Points délicats.

2 Points délicats

Le premier point délicat a été la gestion des piles des threads. En effet, on avait au départ alloué à chaque thread la même zone mémoire pour sa pile, ce qui pouvait causer des conflits d'accès notamment lors de l'incrémentation de `volatile int i` et donc à des résultats incorrects. Pour cela, il fallait s'assurer que chaque thread ait une pile distincte. La gestion de la pile des threads était complexe, car il fallait éviter que plusieurs threads se chevauchent en mémoire, tout en assurant une allocation efficace pour un nombre variable de threads. De plus, lors de l'implémentation, nous ne savions pas comment gérer un thread lorsqu'il n'y avait plus suffisament de place pour faire sa pile (avant l'implément avec `Bitmap`). Nous avons donc décidé de faire une boucle while pour attendre qu'un slot de pile soit disponible et de passer la main aux autres threads grâce à `currentThread->Yield()`.

Le point qui nous a demandé le plus de temps a été la gestion de la terminaison automatique des threads. Nous avons donné en 3ème argument à l'appel système `ThreatCreate`, l'adresse de `ThreadExit` directement depuis start.S. La solution que nous proposions au départ était de passer par une fonction ThreadFunction qui exécute systématiquement ThreadExit afin de s'assurer que chaque thread utilisateur se termine correctement. Cette fonction sert de wrapper pour garantir une gestion propre de la fin de chaque thread. Cependant nous n'avons pas réussi à l'implémenter car si ThreadExit n'est pas appelé explicitement, lorsque disont dans notre test `makethreads`, le premier thread a affiché ses 10 caractères "a", au lieu de s'arrêter normalement comme il devrait le faire quand il y a l'appel explicite de ThreadExit, le programme se termine avec une erreur Assertion False qui n'était pas présente chez Yanis. De plus, on n'est pas certain de comment utiliser l'adresse de ThreadExit pour l'appeler dans ThreadFunction afin de terminer le thread automatiquement. On a donc regardé si il n'y avait pas un registre à modifier pour que l'adresse de ThreadExit soit appelée automatiquement à la fin de chaque thread, et nous avons donc essayé avec `RetAddrReg` en lui donnant l'adresse de ThreadExit, cela nous à permis de nous assurer que les threads crées puissent se terminer automatiquement.

Un dernier point compliqué a été la partie sur les sémaphores (la partie bonus). En effet, nous avons rencontré plusieurs erreurs pour remonter l'accès des sémaphores au niveau utilisateur. Bien que nous ayons tenté de résoudre ces problèmes, nous n'avons pas réussi à compiler correctement le fichier de test associé. Malgré tout, nous avons décidé de push notre tentative que l'on a mis en commentaire pour la montrer

3 Limitations

L'allocation de la pile des threads est géré par un while, ce qui permet d'attendre qu'un slot de pile soit disponible et assure une gestion correcte des ressources en passant aux autres Thread grâce à currentThread->Yield() qui permet à d'autre thread d'être exécuté en attendant la libération d'un slot de pile. Cependant, si aucun thread ne libère de slot de pile, le programme restera bloqué dans la boucle while, étant donné que rien n'est fait si aucun slot n'est libéré.

Un des avantages est l'utilisation d'un lock dans `addspace.cc` pour éviter les erreurs liées aux appels concurrents pour l'incrémentation et la décrémentation du nombre de threads. 2 autres lock ont été ajouté dans `userprog.cc` pour les fonctions `PutString, PutChar` et `GetString, GetChar` pour éviter les appels concurrents et les erreurs liées à l'accès concurrent à la console avec un lock pour les fonctions `PutString, PutChar` et un lock pour les fonctions `GetString, GetChar` afin de pouvoir les appeler en même temps sans conflit. L'ajout de lock pour les fonctions `PutString` et `GetString` est nécessaires pour éviter des conflits d'accès afin que les informations ne se mélangent pas ou causent d'erreur, comme écrire 2 chaine de caractère en même temps pourrait la rendre illisible.

Bien que la terminaison automatique des threads ait été implémentées, et qu'elle fonctionne correctement pour les threads créés, il reste qu'il faille appeler explicitement ThreadExit pour que le programme se termine correctement en tant que Thread sans avoir à terminer le programme entièrement lorque qu'il fait un return ou Halt.

4 Tests

Nous avons conçu plusieurs tests pour valider le bon fonctionnement de nos appels systèmes. Le code source des tests contient tous les commentaires nécessaires pour les exécuter avec les bons paramètres et pour connaître la sortie normale attendue à l'écran.

4.1 Stratégie de test pour Makethread

Pour valider le bon fonctionnement de l'appel système `ThreadCreate` et du système de gestion des threads, nous avons conçu un test avec un grand nombre de threads `NumberOfThreads` affichant chacun un caractère spécifique. L'objectif est de s'assurer que `ThreadCreate` fonctionne correctement dans des conditions variées, que plusieurs threads peuvent être créés sans problème, et que la pile des threads est bien gérée, peu importe le nombre de threads lancés. Ce test permet de vérifier la robustesse de `ThreadCreate` en situation de charge élevée et assure que le système gère correctement la

mémoire et la pile pour chaque thread, même lorsque le nombre de threads est important.

4.2 Stratégie de test pour threadsArgument

Pour valider le bon fonctionnement de l'appel système `ThreadCreate` avec des arguments, nous avons conçu un test avec un certain nombre de Threads avec en argument leur numéro, et un dernier prenant un caractère en argument. L'objectif est de s'assurer que `ThreadCreate` prend bien en compte les arguments passés, et que les threads peuvent les utiliser correctement.

4.3 Stratégie de test pour threadGet

Ce test a pour objectif de valider le bon fonctionnement des appels système `GetChar` et `GetString` lorsqu'ils sont exécutés dans un environnement multi-thread. Nous avons conçu deux threads distincts, l'un pour lire un caractère via `GetChar` et l'autre pour lire une chaîne de caractères via `GetString`, chacun affichant ensuite les données reçues. Cette approche permet de vérifier plusieurs aspects du fonctionnement des appels système dans un contexte concurrent.

Ce test permet de s'assurer que les appels système `GetChar` et `GetString` sont sûrs en contexte multi-thread, notamment pour la gestion correcte des locks, des entrées de l'utilisateur, et des débordements de buffer.

4.4 Stratégie de test pour ProducerConsumer

Ce test a pour objectif de valider le bon fonctionnement des sémaphores dans un contexte multi-thread. Nous avons conçu un test avec un producteur et un consommateur, chacun utilisant des sémaphores pour gérer l'accès à une ressource partagée. L'objectif est de s'assurer que les sémaphores fonctionnent correctement en situation de concurrence, en garantissant un accès sécurisé aux ressources partagées.

Cependant, ce test n'a pas compilé correctement, en raison d'une erreur dans l'implémentation des appels systèmes relatifs aux sémaphores. Par conséquent, nous n'avons pas pu le tester comme prévu. Nous avons donc choisi de commenter le test dans notre code source pour illustrer notre tentative