université de BORDEAUX

NACHOS : Multithreading

Rapport TD2 - Système d'exploitation

Pierre Cayrol
Thomas Furelaud

12 Novembre 2019

1 Bilan

1.1 Partie 1

Le but de ce TD était d'implémenter le multithreading sur NachOS. Nous avons donc commencé en codant les appels systèmes de base, à savoir : $ThreadCreate$ et $ThreadExit$. Regardons de plus près ces deux méthodes.

Pour ce qui est de $ThreadCreate$, le but est de créer un thread puis de lui allouer un espace d'adressage de même taille que celui de son père. Pour faire ça, nous appelons donc la méthode $Start$. Or avec cette méthode nous devons passer en paramètre un $void*$. Nous avons donc décidé de créer une structure avec deux champs : l'adresse du pointeur de la fontion, et l'adresse du pointeur des arguments. Nous pouvons donc maintenant appeler la méthode $Start$ en passant en paramètre $StartUserThread$ et la structure contenant toutes les informations dont on a besoin.

Pour la fonction $StartUserThread$, nous commençons par de-sérialiser la structure pour retrouver nos informations (fonction et arguments). Ensuite on initialise les registres, puis on place la fonction dans le registre actuel, les arguments dans le registre 4 (registre ou les arguments sont habituellement), et la prochaine instruction à 4 octets plus loin que la fonction. Pour finir on initialise le pointeur de la pile grâce à la méthode $AllocateUserStack$.

Et enfin pour $ThreadExit$, on désalloue la pile, puis on finit le thread grâce à la méthode $Finish$.

1.2 Partie 2

Pour implémenter $SynchPutChar$ et $SynchGetChar$ de telle sorte à ce que les appels soient en section critique nous avons tout simplement décidé de les entourer d'un sémaphore.

Ensuite, nous voulions que Nachos se termine que lorsque le dernier thread qui tourne encore appel $ThreadExit()$. Nous avons donc décidé d'utiliser des sémaphores pour pouvoir bloquer l'interruption de la machine si il reste des threads "en fonction". Du coup, nous avons initialisé un compteur de threads, qui incrémente lorsque qu'on crée un thread et décrémente lorsqu'on en finit un. Ensuite, dans $exception.cc$, dans les cas $default$ et $Halt$, tant qu'il reste encore des threads qui n'ont pas appelé $ThreadExit()$ alors on bloque. Lorsque le dernier thread finit, alors on envoie un signal et on interrompt la machine.

Pour ce qui est de la mémoire, pour le point II.3 nous utilisions juste un compteur. Il était initialisé à $numPages * PageSize$ et dés qu'un thread été créé, notre compteur perdait 256. Lors qu'un thread se finissait il regagnait

256.

Mais nous avons décidé à la fin, d'utiliser une BitMap. Celle-ci est initialisée à 1024/(2*PageSize) car la taille de la pile utilisateur fait 1024 octets et qu'on a deux pages par thread car une page fait 128 octets. La BitMap a donc une taille de 4(on pourra donc avoir 4 threads qui s'exécutent en même temps, au maximum). Lorsqu'on crée un thread, on regarde si il y a un bit de libre, au quel cas, le thread aura donc son espace mémoire. Sinon alors, à l'aide d'un sémaphore on se place en attente, et lorsqu'un thread se finit on clear le bit qu'il utilisait et on envoie un signal sur ce même sémaphore. Nous avons du utiliser des identifiants de thread pour cette partie car nous avons besoin de connaître la bit que nous devons libérer quand un thread se termine.

Voici le résultat quand plusieurs threads sont lancé dans notre programme et qu'ils exécutent 2 PutInt chacun :

```
pcayrol@latitude-e6320:~/M1/S1/SE/nachos/code/userprog$ ./nachos -rs 1234 -x ../
test/makethreads
10
10
56
11
11
56
6
6
90
12
-20
-20
12
90
5
5
Machine halting!
```

2 Points Délicats

Le point qui nous a été le plus délicat est le point II.2, donc la terminaison finale des threads. Il nous a été compliqué de bien comprendre ce qui été demandé et plus précisément comment le faire (le lien entre les fichiers, pour les sémaphores). Comme dit dans la partie précédente, nous avons donc opté pour des sémaphores ainsi qu'un compteur de threads. Ainsi lorsque l'on crée un thread, on incrémente le compteur et lorsque qu'on en finit on décrémente le compteur. Lorsque l'appel système $default$ est appelé, on regarde donc ou en est notre compteur. Si il est supérieur à 0 alors des threads n'ont pas

fini leur exécution, nous devons donc attendre. Lorsque le dernier a fini, il envoie un signal et donc la machine peut finir.

3 Limitations

Une des limitations connue est l'entremêlage des threads. Comme on peut le voir sur la photo, nos threads s'exécutent tous à la fois et pas un par un. Une solution serait de créer l'appel ThreadJoin pour qu'un thread attende que celui d'avant ait fini pour pouvoir s'exécuter.

4 Tests

Pour ce qui est des tests de la partie 1, nous testions simplement qu'un thread se créait bien et qu'il exécutait bien la fonction associée. Nous testions ça à l'aide d'un simple *PutChar*. Nous testions aussi si celui-ci initialisait les registres comme il le fallait et si il se finissait convenablement(à l'aide de $DEBUG('x',"...");$).
Pour la partie 2, nous avons testé si un thread qui se finissait après le main, se finissait bien et que le *return* du main ne fasse pas que la machine s'arrête. Nous testions aussi, si avec un while(1) dans la fonction du thread, le programme terminait ou pas.
Pour ce qui est dans la pile, nous regardons si le nombre maximal de thread qui s'exécutent en même temps était au maximum de 4. Nous testons aussi si les threads occupe bien un place de 256 octets et si le cinquième thread se met bien en attente jusqu'à ce qu'un des 4 premiers ait fini pour ensuite s'exécuter. Nous avons aussi testé pour plusieurs dizaines de threads. Nous créons beaucoup de thread dans le fichier test *makethreads* et à l'aide de la commande $DEBUG('x',"...")$; nous regardons ce qu'il se passe et si les informations concordent.

5 Questions

5.1 Partie 1

5.1.1 Action I.3

La création d'un thread peut échouer si le nombre maximal de thread déjà créé est atteint.

5.1.2 Action I.5

Lorsqu'un thread finit, nous devons désallouer son esapce d'aressage pour qu'un autre thread puisse s'exécuter sur celui-ci.

5.2 Partie 2

5.2.1 Action II.1

Nous n'avons pas besoin de protéger les appels SynchPutString et SynchGetString car les deux méthodes codant ces appels appellent eux mêmes, respectivement, SynchPutChar et SynchGetChar. Ainsi ces appels sont déjà protégés.

5.2.2 Action II.3

Si il y avait plusieurs threads, ils auraient tous le même espace mémoire et donc des informations seraient perdues.

5.2.3 Action II.4

Si un programme lance un trop grand nombre de threads, l'espace mémoire maximal serait atteint.