Compte Rendu TP5

Aurèle Barrière & Jérémy Thibault

23 novembre 2016

Introduction

Ce TP consistait en l'écrituer de plusieurs fonctions essentielles au bon fonctionnement du système Nachos : les outils de synchronisation (sémaphores, verrous et variables de condition) et la gestion des threads.

1 Implémentation des outils de synchronisation

1.1 Sémaphores

Pour commencer, nous avons dû implémenter les fonctions de création, de destruction, ainsi que `P` et `V` des sémaphores, dans le fichier `kernel/synch.h`. Nous avons pris soin de désactiver les interruptions dans les fonctions `P` et `V`, puis de restaurer le statut d'interruption précédent.

Il faut également mettre à jour les champs `ReadyToRun`, `alive` pour indiquer le statut des `threads`.

1.2 Verrous

Les verrous sont très similaires aux sémaphores. Cette fois-ci, en plus des fonctions de création et de destruction, le verrou possède deux méthodes `Acquire` et `Release`. Un thread peut demander à acquérir un verrou si celui est libre. Dans ce cas, le thread peut continuer. Sinon, le thread est mis en attente et éventuellement prendra le contrôle du verrou, s'il est libéré.

Le verrou possède une liste des processus qui attendent sa libération : quand la fonction `Release` est appellée, il ne faut déverrouiller le verrou que si aucun thread ne l'attend. Sinon, il faut lui attribuer ce thread et le réveiller.

1.3 Variables de condition

Les variables de condition sont là encore semblables aux cas précédents. Un thread peut demander d'attendre sur une variable de condition avec la fonction `Wait`. Ensuite, il est possible d'envoyer un signal à un thread particlier avec `Signal` ou à tous avec `Broadcast`. Les threads signalés peuvent alors reprendre leur déroulement.

1.4 Code des appels systèmes

Pour pouvoir utiliser ces fonctions depuis un programme utilisateur, il est nécessaire de fournir une interface. Celle-ci est définie sous forme d'appels systèmes dans le fichier `userlib/syscall.h`.

Nous avons dû écrire le code de ces appels systèmes. Lors d'un appel système, les arguments sont stockés dans les registres 4, 5, 6 et 7. La valeur de retour est, elle, stockée dans le registre 2. Dans le cas général, nous lisons les arguments envoyés en paramètres, puis effectuons les actions nécessaires, et certains tests (par exemple, vérifier que l'objet demandé est bien du type voulu : un sémaphore n'est pas un verrou !). Enfin, nous stockons dans le registre 2 la valeur de retour, selon qu'il y ait eu une erreur ou non.

2 Implémentation des threads

La seconde partie de ce TP était dédiée à l'implémentation des threads. En effet, cela est nécessaire pour executer n'importe quel programme.

Plusieurs fonctions sont nécessaires : une fonction de démarrage d'un thread `Start`, une fonction de fin de thread `Finish` et enfin des fonctions de sauvegarde et de restauration de contexte `SaveProcessorState` et `RestoreProcessorState`

La méthode `Start` ajoute le thread au processus actuel, puis initialise son contexte en allouant de la mémoire sur le stack. Enfin, le thread est ajouté à la liste des threads actifs et prêts. La méthode `Finish` indique que le thread peut être détruit, le retire de la liste des threads actifs puis appelle la function `Sleep`. C'est ensuite le scheduler qui détruit définitivement le thread.

Les deux méthodes `SaveProcessorState` et `RestoreProcessorState` sauvegardent et restaurent le contenu de chaque registre entier ou flottant ainsi que du registre contenant le *condition code*.

3 Programmes de test

Pour vérifier notre implémentation, nous avons créé des programmes de tests utilisant les différents outils de synchronisation présentés dans la partie précédente.

Du fait d'un ordonnanceur très basique et déterministe (un thread ne peut être interrompu que lors d'une attente sur un `P` ou `wait`), avoir une exécution conforme à nos attentes ne garantit pas que le programme se comporterait toujours correctement dans un autre contexte.

Dans un premier temps, nous avons lançé les programmes de tests déjà présents dans Nachos. Ils eurent le comportement attendu (sauf le programme `shell` qui ne fonctionne pas pour tous les programmes). Les sections suivantes présentent les tests que nous avons écrit.

3.1 Rendez-vous entre 3 threads

Le premier test implique des sémaphores et nous permet de vérifier le comportement des méthodes `SemCreate`, `SemDestroy`, `P` et `V`. Nous y créons également des nouveaux threads (avec la fonction `threadCreate`). Dans ce programme, nous exécutons 3 threads qui effectuent chacun un calcul (de durée variable) puis attendent à un point de rendez-vous. Ils ne peuvent sortir de cette attente que lorsque tous les threads ont atteint leur point de rendez-vous.

Les résultats obtenus sont les suivants :

```
Starting thread
Starting thread
FStarting thread
Starting thread
irst thread arrived
Third thread arrived
Second thread arrived
Third thread is gone
Second thread is gone
First thread is gone
Destroying semaphores
Done destroying, bye!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

On constate que le programme se comporte comme attendu à une exception près : l'affichage de `Starting thread` se mélange avec celui de `First thread arrived`. En effet, le premier affichage vient de la fonction `printf` utilisée dans le code de nachos, alors que le deuxième provient de la fonction `n_printf` et aucun mécanisme ne permet de garantir la synchronisation entre les deux.

3.2 Test des verrous

Le second programme de test vérifie le bon fonctionnement des verrous. Deux processus incrémentent une variable partagée 1000 fois de suite. Autour de chaque écriture, on place un verrou d'exclusion mutuelle. On attend obtenir la valeur 2000 dans la variable partagée à la fin.

```
Loop 999: n = 1997
Loop 999: n = 1998
Loop 1000: n = 1999
Loop 1000: n = 2000
Everything is OK!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

On montre ici les deux derniers tours de boucles des deux programmes. On obtient bien le comportement attendu.

3.3 Test des variables de conditions

Ce test s'est avéré plus compliqué. En effet, il n'y a en général aucun moyen de garantir qu'un processus attend sur une variable de condition. Si celui-ci prévient juste avant d'arriver (par un sémaphore ou une variable partagée par exemple), il est toujours possible qu'il se fasse interrompre entre le moment où il prévient et le moment où il commence à attendre. Si, pour éviter ce cas, on encadre les deux instructions d'une section critique (avec un sémaphore par exemple), alors on risque l'interblocage.

Pour résoudre ce problème, nous avons utilisé la particularité de notre ordonnanceur : un processus ne sera interrompu que pendant une attente. On peut donc prévenir d'une attente (avec un `V` d'un sémaphore) puis commencer à attendre sur la variable de condition. Sur un autre système d'exploitation, le programme pourrait avoir un comportement différent.

Le test lance trois processus. Au début, deux d'entre eux attendent sur une variable de condition jusqu'à ce que le troisième fasse un appel à `CondBroadcast`. Le second thread s'arrête alors mais le premier attend que le troisième fasse appel à `CondSignal`.

```
Starting thread
Starting thread
Starting thread
Starting thread
Broadcasting
Broadcast received for second thread
Broadcast received for first thread
Sending signal
Signal received for first thread
Everything is OK!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Le programme se comporte comme prévu.

3.4 Producteur-Consommateur

Un autre test implémentait un producteur consommateur. Le processus producteur écrit dans un tampon et le consommateur lit ce tampon. On utilise deux sémaphores pour synchroniser les deux processus.

```
Starting thread
Starting thread
WStarting thread
rote fibo  0
```

```
Fibo(0) = 1
Wrote fibo  1
Fibo(1) = 1
Wrote fibo  2
Fibo(2) = 2
Wrote fibo  3
Fibo(3) = 3
Wrote fibo  4
Fibo(4) = 5
Everything is OK!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Le programme se comporte comme prévu. La ressource calculée est une valeur de la suite de Fibonacci. Le consommateur affiche ensuite le résultat.