TD 2 nachos

Master 1 informatique université de Bordeaux 2019-2020

par Julien Dauliac, Thomas Veloso

I. Bilan

Durant ce second devoir nous avons mis en places les threads utilisateurs dans nachos. Ceci nous a également permis de jouer avec la mémoire virtuelle de nachos ainsi qu'avec les sémaphores/mutex pour sécuriser tout ça.
L'utilisation des `DEBUG` s'est vue particulièrement utile afin de voir quel thread appelle quelle fonction kernel. Nous avons utilisé la lettre t pour les appeler.

```
nachos -d t -rs 1297 -x
```

II Limitations

Dans notre implémentation, il nous utilisons beaucoup de mutex, il aurait peut être été plus simple de développer une classe de moniteurs afin de rendre cette utilisation plus simple.
Il nous manque l'implémentation des bonus ce qui à pour effet de rendre moin lisible le main car nous devons le terminer systématiquement avec `ThreadExit()`, sous peine de ne pas voir les autres threads démarrer.

Après l'implémentation du bitmap, nous avons introduit un bug qui empêche l'exécution des threads.

III Difficultés rencontrés

Lors de tp la difficulté fut triple.

Mon binôme étant malade la moitiés du temps, je fus seul à réaliser la partie 1.5 à 2.x

Étant en formation continue, ce qui normalement n'aurait pas posé de problème, je fus désavantagé car n'ayant pas pu profiter des vacances pour travailler autant.

Mise à part ça, je pense que le défis est relevé, même si les bonus n'ont pas pu être fait.

Plus particulièrement sur le TD, j'ai eu des problèmes sur la partie de la gestion mémoire qui fut un peu plus complexe mais pas insurmontable.

I Partie 1

C'est la fonction de start de thread qui l'utilise en appelant `AllocateUserStack` qui réserve une zone mémoire de `256 + 16`.

Un thread pourrait échouer si il n'a plus de ressources mémoire pour s'exécuter.

Pour tester la suite:

```
./nachos -d t -x ../test/makethread
```

Nous avons ajouté un décalage de 16 bits pour séparer chacune des sections mémoires.

Le schmurtz n'a également pas facilité la chose, le nom n'est pas explicite pour cette variable et à été source de confusion. Nous avons donc utilisé tout simplement une structure pour passer un seul argument au `newThread->Start();`

II Partie II

Oui il faut ajouter un sémaphore car sinon les `synchPutString` se retrouvent coupé avant la fin de l'écriture par un autre thread.

Pour tester la suite:

```
./nachos -d t -x ../test/multithreads
```

Si ce thread écrit lui aussi du texte, alors il se passe n'importe quoi.

La difficulté dans cette partie à été de faire la fonction `AllocateUserStack` pour qu'elle découpe la ram pour chacun des processus: le calcul étant:

```
numPage*PageSize  - (256 + 16) * foundedAddres -16
```

Nous avons mis en place un conteur, protégé par des sémaphores utilisés comme des mutex pour protéger son incrémentation, décrémentation. Le conteur utilise un int et un sémaphore utilisé comme un mutex. Nous avions déclaré dans un premier temps ces 2 variables dans `addspace.cc` mais nous les avons finalement

déplacés dans le `userthread.cc` car nous trouvions que ca avait plus de sens vu qu'il s'agissait du fichier de threads.

Il a bien évidemment fallu utiliser le class Semaphore pour importer la classe semaphore.

Ces variables s'appelant respectivement: `semThread` et `semThread`

Le défaut de cette méthode et qu'on ne vérifie pas que le nombre de thread est respecté, on ne réaloue pas les zones mémoires, et il faut terminer manuellement le thread main sinon, il ne laissera pas sa place aux autres threads

Dans le exit, il y a un check pour savoir si le thread est le dernier à se terminer, si c'est le cas, on ne stoppera pas le thread mais la machine.

Par la suite, les appels kernels se coupaient les uns les autres à cause des changement de threads, il a fallu les entourer de sémaphores.

Nous avons utilisés 2 sémaphores: ceux pour les écrivains, et ceux pour les lecteurs.

Ces sémaphores sont déclarés dans le `system.cc` et importé dans le exception.cc

Pour se faire nous avons utilisé le conteur comme indice de décalage dans la ram. Ce qui marche pour lancer des threads mais pas pour les terminées, réutiliser leur zone mémoire.

Nous avons donc instancié un Bitmap initié à `0` son nombre d'entrée étant la taille de la ram virtuelle divisée par `256+16`, la taille d'un bloc mémoire et de son espace de séparation avec les autres. `-1` car il faut enlever un pour ne pas écraser, ni allouer la zone mémoire du main qui est spéciale.

Le bitmap se mettra dans la fonction `AllocateUserStack` de `addspace.cc` entouré par un semaphore

Il a également fallu créer une fonction `unAllocateUserStack` pour libérer le bitmap.

Ce dernier prend tout simplement en paramètre l'adresse du thread récupérer grâce à `readRegister`. C'est en effet le thread qui exécute son arrêt.