Echallier Nicolas

Mahieddine Yaker

1er TP - Prise en main

1- exemple basique d'utilisation du multicore

Nous avons implémenté cet exercice dans le fichier ASE_TP5/ex1.c.

Pour résumer, nous avons lié `IRQVECTOR[0]` avec notre fonction ex1 ; qui imprime le numéro du cœur et compte un certain temps.

2- utilisation de timer / 3- utilisation du lock

Nous avons implémenté cet exercice dans le fichier ASE_TP5/ex3.c.

Tout marche. Lorsque l'interruption TIMER_IRQ arrive, le cœur 2 exécute la fonction irqCoucou qui affiche un message et le cœur sur lequel la fonction est appelé.

En ce qui concerne l'utilisation du lock, elle marche et elle est appelé dans la fonction ex3. En gros, le cœur essai de prendre un lock. Si est libre, il le prend et imprime un message, attends 0xFFFFFF, réimprime un message et rend le lock.

Si le lock n'est pas libre, le cœur attend un peu et ré-essai de le prendre.

Nous avions eu un peu de problèmes dans cette partie car lorsque nous relâchions le lock, nous ne faisions pas attendre le cœur un peu pour donner le temps aux autres cœurs de le prendre. En faisant attendre le cœur un peu après qu'il ait relâché le lock, on permet aux autres cœur d'avoir le temps de le prendre.

2e TP - Gestion de contextes en multicore

Nous avions mis en place la méthode 3 que l'on a utilisé dans la partie 3. Le code peut être trouvé dans le fichier `multiCore.c`. Nous n'avons pas réussi à compiler le fichier `multiCore.c` et `shell_mc.c` en même temps car cela nous donnait des erreurs de compilations.

En gros, lorsque un contexte a fini de s'exécuter il retourne dans la fonction init (on imprime le message back to init en rouge).

3e TP - Partage de structures du noyau

1- Bibliothèque de gestion d'un unique verrou klock

Nous avons implémenté le verrou dans deux fonctions klock et kunlock dans le fichier sched.c. D'ailleurs, nous avons fait un code bien plus efficace pour le klock que celui que nous avions fait

pour la session1.

3- Équilibre de la charge

C'est ici que nous avons eu un problème. Pour une raison que nous ne comprenons pas, nous pouvons lancer un premier contexte sur chaque processeur mais dès que l'on ré-essait d'en créer un nouveau, nous avons une erreur de segmentation.

Nous avons réussi à trouver d’où le problème venait mais nous ne comprenons pas pourquoi : le problème vient lorsque, dans le switch_to_ctx, nous chargeons le context de la fonction pour utiliser son esp et son ebp (ligne 181 du fichier sched.c).

Nous n'avons donc pas pu tester l'implémentation du load_balancer que nous voulions utiliser. Elle se trouve dans shell_mc.c et s'appelle… LoadBalancer.

En gros, le but ici était, lorsque un cœur avait fini ses tâche, de parcourir la liste des tâches des autres processeurs. Si un de ses processeurs avait plus d'un tâche en route, il devait lui « voler » sa tâche.

Un signal « SigUsr 2 » arrive au moment de la création de deux autres processus.