TP9 : Applications multi-tâches coopératives

XU Weiqin 3410681

HUANG Xingyun 3602284

C) Application producteur / consommateur

Question C1 : En analysant le code C des deux fonctions producer() et consumer() dans le fichier main_bipro.c, décrivez ce que vous pensez voir sur les deux terminaux TTY[0] et TTY[1].

Processeur 0 correspond à TTY[0], Processeur 1 possède correspond à TTY[1]. Chaque TTY affiche la valeur respectivement pour chaque tâche.
Le producteur a la tâche à écrire les valeur de 0 à 49 dans BUFFER et les afficher sur TTY[0] ; le consommateur va lire cettes valeurs dans le BUFFER est les afficher sur TTY[1].

Question C2 : Complétez le code assembleur contenu dans le fichier reset.s pour que les tâches *producer* et *consumer* soient lancées sur les processeurs 0 et 1 respectivement.

Compilez l'application logicielle dans le répertoire soft, en utilisant le Makefile fourni, et exécutez cette application pour différentes valeurs des variables de temporisation définies au début du fichier main_bipro.c:

1. **PRODUCER_DELAY = 10 / CONSUMER_DELAY = 1000**
2. **PRODUCER_DELAY = 1000 / CONSUMER_DELAY = 10**

Exécutez l'application sur le prototype virtuel à deux processeurs, en vous assurant que le mécanisme de SNOOP est activé:

```
./simul.x -NPROCS 2 -SNOOP 1
```

Question C3 : Quel comportement observez vous ? Comment interprétez-vous ces résultat ?

1. PRODUCER_DELAY = 10 / CONSUMER_DELAY = 1000

   Le consommateur ne marche assez vite pour suivre toutes les valeurs de le producteur ;

2. PRODUCER_DELAY = 1000 / CONSUMER_DELAY = 10

   Le producteur produit la valeur trop lentement, donc le consommateur reçoit la même valeur plusieurs fois.

D) Synchronisation par bascule SET/RESET

Question D1 : La valeur de la variable SYNC peut être modifiée par les deux tâches *producer* et *consumer* qui s'exécutent en parallèle. Expliquez pourquoi il n'y a pas de risque d'incohérence liée aux accès concurent à cette variable.

Parce que le produceur ne peut que modifier la variable SYNC de 0 à 1 ; pourtant le consommateur ne peut que modifier SYNC de 1 à 0. Donc il n'y aura pas de risque d'incohérence.

(Dans la condition de 1 seul producteur et 1 seul consommateur, et 1 tampon contenant 1 seul objet.)

Question D2 : Recompilez l'application logicielle. et exécutez cette application logicielle synchronisée en utilisant différentes valeurs des paramètres CONSUMER_DELAY et PRODUCER_DELAY, pour vérifier l'efficacité du mécanisme de synchronisation. Comparez les durées d'exécution avec et sans synchronisation. Pour calculer la durée d'exécution, on prendra la date de terminaison de la tâche qui se termine en dernier.

Les durées d'exécution avec et sans synchronisation (cycles) :

	Producteur	Consommateur
Avec synchronisation	754647	754551
Sans synchronisation	753850	194184

Comme la plupart des segments stockés en mémoire, le segment *seg_data* contenant les variables globales est cachable. Les variables SYNC et BUFFER, vont donc pouvoir être recopiées dans les deux caches associés aux processeur proc[0] et proc[1].

Question D3 : Expliquez pourquoi cette cachabilité introduit-elle un risque de dysfonctionnement?

Parce que c'est possible que le contenu dans la cache a été changé, mais ce n'est pas mis à jour dans la mémoire.

Relancez la simulation en désactivant le mécanisme de snoop, grâce à l'argument (-SNOOP 0) sur la ligne de commande.

Question D4 : Interprêtez le comportement observé.

En raison de la synchronisation, Le consommateur va attendre jusqu'à la valeur est produit par le producteur. Tandis que sans SNOOP, le consommateur ne sait pas que la valeur a été déjà mis à jour.

Question D5 : Quel est, selon vous, le principal coût matériel causé par le mécanisme de SNOOP?

Le consommateur va toujours attendre quand le producteur ne produit pas de nouvelle valeur et il coûte beaucoup de cycles d'attente.

E) Problèmes de synchronisation liés au compilateur

Question E1 : Même si votre programme parallèle semble s'exécuter correctement, modifiez le code pour introduire les macros garantissant strictement l'ordre d'accès aux variables partagées.

Question E2 : Recompilez et re-exécutez l'application logicielle. Comment le code binaire a-t-il été modifié par l'introduction de la macro ? Il faut aller regarder dans le fichier app.bin.txt. Comment le temps d'exécution est-il modifié?

__sync_synchronize() permet d'interdire le compilateur d'inverser l'ordre du code :
BUF = n ; SYNC=1 ; et
val = BUF ; SYNC=0 ;

qui permet de garantir que le consommateur ne peut consommer la valeur que après la valeur a été mis à jour par le producteur.
Le temps d'exécution est 754886(producteur), et 754801(consommateur), ça coûte un peu plus de temps après avoir contrôlé les optimisations réalisées.

F) Synchronisation par FIFO logicielle

Question F1 : Le canal de communication est défini comme une structure C du programme applicatif. Pourquoi cette structure est-elle déclarée comme une variable globale? quelles sont les différents champs de la structure?

Parce que elle va être utilisé par tous les producteurs et tous les consommateurs.
typedef struct fifo{
int buf[DEPTH];
int ptr
int ptw
int sts
int depth
lock_t lock;}fifo_t;

Question F2 : Pourquoi est-il préférable d'utiliser un verrou à ticket plutôt qu'un verrou simple comme dans le TP8 ?

le verrou utilisé dans le TP8 était un verrou "dans l'espace système", utilisé par le système d'exploitation pour séquencialiser les accès au contrôleur de disque. Le verrou utilisé ici pour protéger l'accès exclusif au canal de communication est un verrou *dans l'espace utilisateur*, qui est directement géré par le code applicatif grâce aux deux fonctions *lock_acquire()* pour la prise de verrou, et *lock_release()* pour la libération du verrou.

Pour garantir une allocation équitable entre les différentes tâches, ce verrou implémente une technique de ticket.

Question F3 : Quelles sont les arguments des deux fonctions *lock_acquire()* et *lock_release()*?

Lock_t* plock: une structure contenant *unsigned int current* et *unsigned int free*.

Question F4 : Décrivez précisément ce que fait la fonction *lock_acquire()* pour prendre le verrou.

Dans la fonction **lock_acquire()** pour prendre le verrou, d'abord on acquiert la variable *free* qui représente le numéro du prochain ticket libre, et puis on obtient la variable *current* qui contient le numéro du ticket du proprétaire courant, ensuite on fait le polling sur le *current* jusqu'à le numéro du prochain ticket libre égale le numéro du ticket courant.

Pourquoi la fonction *atomic_increment()*, appelée par la fonction *lock_acquire()*, est-elle écrite en assembleur?

C'est une fonction atomique qui est exécuté directement par CPU, on l'écrit en assembleur pour que le compilateur ne changera pas l'ordre du code.

Expliquez très précisément ce que fait cette fonction *atomic_increment()*.

On enregistre le valeur du numéro du ticket courant et met la valeur suivant comme le début de la file d'attente.

Et puis on compare le numéro du ticket courant et celui du prochain ticket libre, si égale on peut exécuter le code.

Question F5 : Que fait la fonction *lock_release()* pour libérer le verrou? Expliquez pourquoi cette fonction n'a pas besoin d'être écrite en assembleur?

Cette fonction passe le pointeur de l'adresse à l'adresse suivante de l'adresse suivantes, ça veut dire qu'on ne réserve plus l'adresse et relâche le verrou.

cette fonction n'a pas besoin d'être écrite en assembleur parce qu'il y a seulement une instruction dans cette fonction.

Question F6 : pourquoi une tâche qui ne peut effectuer son transfert à cause de l'état de la FIFO (FIFO pleine pour un producteur / FIFO vide pour un consommateur) doit-elle impérativement relâcher le verrou avant de re-essayer? Que se passe-t-il si elle ne le relâche pas?

Les autres tâches vont être empêché si le verrou n'a pas été relâché.

Question F7 : Complétez le code des fonctions *fifo_read()* et *fifo_write()* qui doivent réaliser le protocole de communication décrit ci-dessus.

Compilez l'application logicielle main_fifo.c en modifiant le Makefile dans le répertoire soft et lancez l'exécution pour différente valeurs du paramètre DEPTH définissant le nombre de cases de la FIFO, sans oubliez d'activer le SNOOP:

```
$ ./simul.x -NPROCS 2 -SNOOP 1
```

Question F8 : Relevez les date de terminaison des deux tâches, et reportez ces valeurs dans le tableau ci-dessous. Quelles conclusions peut-on en tirer ?

DEPTH	1	2	4	8
Producer	554340	497265	497951	539458
Consumer	552504	509006	525594	539304

Dans les applications parallèles, il est intéressant d'évaluer séparément le temps de calcul et le temps de communication.

- **Le temps de calcul est une caractéristique intrinsèque de chaque tâche, et correspond au nombre de cycles utilisés par la tâche pour effectuer les calculs dont elle est responsable.**
- **le temps de communication est le temps utilisé pour réaliser les opérations de lectures ou d'écritures dans les canaux de communication (y compris les temps passés dans les boucles d'attente liées à la synchronisation).**

Question F9 :Evaluez le coût moyen (en nombre de cycles) d'une opération de communication entre deux tâches. Il faut pour cela réduire au strict minimum les temps de calcul autres que les appels aux fonctions fifo_read() et fifo_write() dans les tâches *producer()* et *consumer()*:

On a supprimé les temporisations,l'affichage à l'intérieur de la boucle et a augmenté le nombre d'itérations (1000 itérations). Et puis on a relancé les tâches :

Producteur(cycles)	consommateur(cycles)	moyen(cycles)
231465	231589	231527

G) Application logicielle multi-tâches

Question G1 : Que faut-il modifier dans le fichier reset.s pour lancer six tâches sur six processeurs différents ?

Il faut ajouter le code assembleur contenu dans le fichier **reset.s** pour que les tâches *router* soient lancées sur les processeurs 2 à 5 respectivement :

```
la $26, seg_data_base
lw $26, 8($26)
mtc0 $26, $14
```

Question G2 : Pourquoi a-ton choisi de stocker le verrou protégeant l'accès exclusif à un canal de communication dans la structure de donnée représentant l'état du canal?