Systèmes d'exploitation et programmation concurrente : examen de TP et de programmation concurrente « Producteurs-consommateurs synchrone »

Ensimag 2A

1 Le problème : Producteurs-consommateurs synchrone par moniteur en POSIX threads

L'objectif de ce sujet est la programmation d'une version multi-threadée du problème des producteurs-consommateurs similaire, mais très simplifiée, aux techniques utilisées pour faire du Processing-In-Memory (imbrication du calcul et de la mémoire au niveau matériel).

Dans ce sujet, chaque message transféré est généré par plusieurs threads de type producteurs et chaque message consommé est lu par plusieurs threads de type consommateurs.

Une opération de dépôt, c'est-à-dire de production, ou de retrait, c'est-à-dire de consommation, d'un message sera combinée avec une barrière synchrone qui va attendre que tous les threads impliqués aient réalisé leur action.

Vous aurez donc à combiner le problème des producteurs-consommateurs avec celui d'une barrière.

Vous aurez aussi à sauvegarder le résultat dans un fichier.

Les deux questions (synchronisation sec. 2 et fichiers sec. 3) sont indépendantes.

Chaque opération de dépôt, `void deposer_synchrone(...)`, et de retrait, `void retirer_synchrone(...)`, sera découpée en deux phases successives :

1. écrire ou lire la partie du tampon associée au thread
2. attendre que tous les autres threads aient fini d'écrire ou de de lire leur partie avant de les libérer et indiquer que la case est disponible.

Pour simplifier : le même nombre de threads producteurs et de threads consommateurs lisent et écrivent le tampon ; tous les producteurs et tous les consommateurs participent à chaque écriture et lecture d'une case ;

Un message est composé de la structure suivante :

```c
typedef struct _msg {
    uint64_t valeurs[NBTHREADS_PAR_TYPE];
} Msg;
```

Chaque thread lit ou écrit sa case du champ `valeurs` d'un message, en fonction de son propre numéro.

Chaque thread doit exécuter le code correspondant à un *producteur* ou à un *consommateur* (fichier `prodconso.c`). Les threads utilisent les 2 fonctions d'un moniteur pour organiser la synchronisation de la communication par un tampon (`int tampon[TAILLE]`) déjà défini.

Les fonctions du moniteur sont à implanter dans le fichier `synchro.c`.

— `void deposer_synchrone(int mon_tid, uint64_t message)` qui dépose le morceau de message dans le tampon et attend que tous les threads aient déposé leur morceau.

— `uint64_t retirer_synchrone(int th_id)` qui retire un morceau de message dans le tampon et attend que tous les threads aient retiré leurs morceaux.

Les 2 fonctions précédentes sont à implanter dans le fichier `synchro.c`.

Les threads exécutent le code des producteurs et consommateurs correspondant respectivement aux deux fonctions :

— `void *producteur(void *arg)`

— `void *consommateur(void *arg)`

Vous devrez lancer les threads exécutant ces fonctions dans le fichier `main.c`. Ils prennent en argument leur numéros, de 0 à `NBTHREADS_PAR_TYPE -1` .

2 Le travail à réaliser : synchronisations

2.1 Login et compilation

Vous devez modifier le script `CMakeLists.txt` pour y insérer votre login. La compilation et le lancement des tests sont identiques aux différents TPs réalisés pendant le semestre.

```
cd build
cmake ..
make
make test
make check
```

2.2 Lancement et terminaison des threads

Vous devez ajouter, dans le fichier `src/main.c`, le lancement et l'attente de terminaison des threads exécutant les fonctions `void *producteur(void *arg)` et `void *consommateur(void *arg)` .

Il faudra lancer `NBTHREADS_PAR_TYPE` threads producteurs et `NBTHREADS_PAR_TYPE` threads consommateurs.

Ces fonctions devront être lancées avec comme argument le numéro du thread de 0 à `NBTHREADS_PAR_TYPE -1`.

Il faudra ensuite attendre la terminaison de tous les threads.

Vous aurez à définir vous-même les variables `pthread_t` nécessaires.

Pour tout cela vous utiliserez les fonctions `pthread_create(...)` et `pthread_join(...)`.

La zone où vous devez ajouter le code dans `src/main.c`, est indiquée par le marqueur suivant :

```
// Ajouter votre code ici
```

2.3 Moniteur en Posix Threads

Vous devez ajouter dans le fichier `src/synchro.c` tous les appels aux fonctions de synchronisation d'un moniteur en Posix Threads. Les appels de fonctions manquants sont donc du type : `pthread_mutex_lock(...)`, `pthread_mutex_unlock(...)`, `pthread_cond_wait(...)`, `pthread_cond_broadcast(...)`.

Les `man` des fonctions peuvent vous aider.

Vous devrez aussi mettre en place les variables nécessaires à la synchronisation.

```
// Exemple de variables
static pthread_mutex_t toto = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t titi = PTHREAD_COND_INITIALIZER;
```

Les zones où vous devez ajouter du code sont indiquées par le marqueur suivant

```
// Ajouter votre code ici
```

Pour retirer ou déposer un `message` dans le `tampon` depuis le thread consommateur, ou producteur, numéro `th_id`, vous utiliserez un code de la forme suivante

```
// Pour écrire
...
tampon[...].valeurs[th_id] = message;
...
// Pour lire
...
uint64_t message = tampon[...].valeurs[th_id];
...
```

2.4 Conseil pour l'implantation de la synchronisation

Comme les deux fonctions du moniteur implante une barrière, vous ne devriez **pas** utiliser `pthread_cond_signal(...)` mais plutôt `pthread_cond_broadcast(...)` pour le réveil de tous threads en même temps.

3 Le travail à réaliser : sauvegarde du résultat

Le code du squelette affiche le résultat sur la sortie standard, au format YAML.

Vous devrez créer et ouvrir un fichier `resultats.yaml`, si et seulement s'il n'existe pas déjà. Ce fichier aura seulement les droits de lecture et d'écriture pour l'utilisateur, et uniquement de lecture pour le groupe et tous les autres.

Vous redirigerez ensuite la sortie standard vers ce fichier si vous l'avez créé.

Si le fichier existait déjà, votre code ne fera rien, l'affichage continuera à se faire sur la sortie standard. Il ne faudra pas modifier un fichier que vous n'auriez pas créé.

Vous ne modifierez pas l'affichage du résultat lui-même, à la fin du `main(...)`. Cet affichage est lu par une partie des tests.

La zone où vous devez ajouter le code dans `src/main.c`, est indiquée par le marqueur suivant, avant et après la boucle d'affichage du résultat :

```
1 // Ajouter votre code ici
```

3.1 Conseil pour l'implantation de la synchronisation

Si vous fermez brutalement la sortie standard, vous pourriez vous retrouvez avec un fichier vide. Pour forcer le vidage de tous les tampons, avant la fermeture, vous pouvez utiliser la fonction `fflush(NULL)` ;

4 Utilisation du programme `ensisyncpc`

En plus de make test et/ou make check vous pouvez lancer directement *ensisyncpc* pour tester votre code

Il n'est pas nécessaire d'avoir une compréhension exhaustive du fichier `prodconso.c` pour répondre à l'exercice.

Le programme prend 1 argument :

— `SCENARIO` qui permet de choisir un numéro de scénario.

Lorsqu'il exécute un scénario, une trace indique le résultat du déroulement des lectures et écritures. C'est cette trace finale qui est testée par les tests `scenario*.pl`.

5 Barème indicatif

Le but est de fournir une implantation respectant le sujet et pas uniquement de passer les tests. La qualité du code pourra modifier de façon très significative la notation.

0-5 le rendu ne compile pas dans le répertoire `build` et les synchronisations sont inexistantes.

8 le rendu compile en faisant `make` dans le répertoire `build`. Une ébauche de synchronisation est fournie.

10 le rendu compile et passe les premiers tests. Une synchronisation minimaliste buguée, ou incomplète, est fournie.

12 le rendu compile et passe quelques tests significatifs.

14 le rendu compile et passe presque tous les tests avec un schéma de synchronisation presque correct.

16 le rendu compile et passe tous les tests, mais le code est très perfectible.

20 le rendu compile, passe tous les tests, avec une synchronisation parfaite et le code est très satisfaisant.

6 Pour passer le temps des curieux finissant tôt : Perl

Les scripts de test du sujet sont réalisés en Perl. Vous pouvez lire une courte, mais bonne introduction au langage avec `man perlintro`.