SE205: Travaux Pratiques sur la Programmation Concurrente en POSIX

Laurent Pautet (pautet@telecom-paristech.fr)

Index

1 TP de Concurrence en POSIX

Objectifs

L'objectif de ce TP consiste à implanter en C un service de file d'attente bloquante de taille fixe comme le service de Java ArrayBlockingQueue. Par cet intermédiaire, vous pourrez découvrir, de manière détaillée, les outils de concurrence de POSIX.

Ce TP ne présente pas **tous** les outils proposés par POSIX pour gérer la concurrence. Il illustre le fonctionnement de quelques uns d'entre eux. Le support de cours sur les tâches se trouve ici. Vous pouvez consulter la documentation complète des fonctions POSIX concernant les threads en suivant ce lien.

Sources

Vous trouverez dans cette archive compressée l'intégralité des sources.

Le TP sur les patrons pour la concurrence s'appuie sur les 4 premières questions, c'est à dire les implantations d'une file d'attente bloquante de taille fixe avec variables conditionnelles pour les sémantiques bloquante, non-bloquante et temporisée.

Debuggage

Pour trouver vos erreurs, nous vous conseillons fortement d'utiliser gdb. Il est **critique** de mettre au point vos programmes C en utilisant gdb et non pas en truffant votre programme de *printf* au petit bonheur. Si vous avez un problème de mémoire (SIGSEGV, ...), faites :

```
gdb ./main_protected_buffer
(gdb) run test-00.txt
```

En cas de problème, le programme stoppera sur l'accès mémoire suspect. Pour identifier la ligne, utiliser les commandes de gdb:

- info threads - pour avoir la liste des threads
- thread <id> - pour accèder au thread de numéro <id>
- bt - pour afficher les différents appels de fonction dans un thread
- f <number> - chaque appel de fonction ayant un numéro, se rendre sur un appel
- list - pour faire afficher le code C correspondant à l'appel
- p <variable> - pour faire afficher la valeur d'une variable
- p *<pointer> - pour faire afficher le contenu d'un pointeur

MacOS

Pour les utilisateurs de MacOS, il faudra rendre votre gdb opérationnel et MacOS ne facilite pas la tâche. Vous trouverez la marche à suivre en suivant ce lien. Si ce lien n'est pas suffisant, il existe de nombreux guides pour règler ce problème.

1.1 Création de processus et terminaison par ordre de création

On considère le fichier *main_protected_buffer.c*. Ce programme crée des producteurs et des consommateurs qui échangent des données en utilisant un tampon partagé. On se concentre en premier lieu sur la création des producteurs et des consommateurs.

Les producteurs exécutent la procédure principale *main_producer* et produisent des données (entiers) dans un tampon circulaire *protected_buffer* que l'on va protéger contre les accès concurrents. Des consommateurs exécutent la procédure principale *main_consumer* et consomment les données produites de la même manière.

Les accès des producteurs et des consommateurs pourront se faire suivant plusieurs sémantiques. Ils pourront être bloquants, non-bloquants, ou temporisés, c'est à dire qu'en cas de blocage, celui-ci ne pourra pas excéder un temps donné.

Producteurs et consommateurs adoptent un comportement périodique. Toutes les périodes à partir d'un instant commun de démarrage *start_time*, ils produisent ou consomment des données puis attendent la période suivante pour recommencer. Par exemple, les producteurs se réactivent toutes les *start_time + n * producer_period* où *n* représente le nombre d'activations.

Données d'entrée

Les scenarii sont fournis dans les fichiers *test-<XY>.txt* et sont décrits comme suit.

Le nombre de producteurs (*n_producers*, celui de consommateurs (*n_consumers*), la taille du tampon circulaire (*buffer_size*), la sémantique des accès à celui-ci (*semantics*) (0 pour bloquante, 1 pour non-bloquante, 2 pour temporisée) ou les périodes des producteurs (*producer_period*) et des consommateurs (*consumer_period*) sont configurés grâce à un fichier dont le nom est passé sur la ligne de commande. La valeur 0 du paramètre *sem_impl* indique que l'on utilise l'implémentation avec variables conditionnelles et la valeur 1 que l'on utilise l'implémentation avec sémaphores.

```
#sem_impl
0
#semantics
0
#buffer_size
1
#n_values
10
#n_consumers
2
#n_producers
5
#consumer_period
5000
#producer_period
5000
```

Données de sortie

Les lignes de sortie se composent de plusieurs champs. Le premier indique la valeur de l'horloge en seconde. Le second le nom de la tâche et le suivant le numéro de la tâche. Les deux derniers champs indiquent l'opération et la donnée d'entrée ou de sortie. Dans l'exemple ci dessous, à la date *000*, le *producteur* attaché au *thread 2* effectue l'opération *put* de la donnée *200* en mode bloquant *(B)*. La valeur *200* correspond au *producteur 2* (valeur des centaines) et à la (première) *production 0* de ce serveur (valeur des unités).

```
000 producer 02 put (B) - data=200
```

Il faut également noter que lorsque l'on a *P* produteurs et *C* consommateurs, les producteurs vont produire *C* fois, et les consommateurs vont consommer *P* fois. Ainsi, en temps normal, toutes les données produites sont consommées.

Complétez le fichier *main_protected_buffer.c* pour créer autant de producteurs et de consommateurs que demandés et pour ensuite attendre la terminaison des producteurs et des consommateurs. Sauvegardez les identificateurs des consommateurs et des producteurs dans le tableau *tasks*.

Vous pouvez tester votre code en faisant :

```
> ./main_protected_buffer test-00.txt
```

Si vous utilisez le fichier *test-00.txt* listé ci-dessus, vous devrez vérifier que vous créez le nombre correct de consommateurs et de producteurs. En consultant le 3ème champs des sorties ci-dessous, on peut effectivement noter qu'il y a 2 consommateurs et 5 producteurs comme spécifié dans *test-00.txt*. **Les autres informations ne sont pas pertinentes à ce stade.**

```
...
start consumer 0
start consumer 1
start producer 2
start producer 3
start producer 4
start producer 5
start producer 6
...
```

1.2 Tampon protégé contre accès concurrents (var cond/bloquants)

Nous allons implanter un tampon circulaire protégé contre les accès concurrents dans le fichier *cond_protected_buffer.c*. Pour ce faire, on pourra utiliser l'implantation d'un tampon circulaire non-protégé qui est fourni dans les fichiers *circular_buffer.c* et *circular_buffer.h*. Consulter les fonctionnalités disponibles dans *circular_buffer.h*.

Complétez la structure *protected_buffer_t*. Elle ne contient qu'un tampon circulaire non-protégé actuellement. Il vous faudra rajouter des outils de synchronisation de POSIX.

Complétez les procédures *cond_protected_buffer_init*, *cond_protected_buffer_get* et *cond_protected_buffer_put* afin qu'il offre les services suivants :

- **cond_protected_buffer_init** : initialise le tampon circulaire protégé contre les accès concurrents et dispose d'un nombre de cases fourni en paramètre de la procédure.
- **cond_protected_buffer_get** : retire du tampon circulaire protégé l'élément le plus anciennement ajouté (donc en politique FIFO). L'opération doit se faire en interdisant les accès concurrents. Si le tampon est vide, l'appel sera bloquant jusqu'à ce qu'une case se remplisse. Par contre, l'opération débloquera des producteurs bloqués si toutefois le tampon était plein précédemment. En effet, on veillera à limiter les déblocages inutiles.
- **cond_protected_buffer_put** : ajoute au tampon circulaire protégé un élément selon une politique FIFO. L'opération doit se faire en interdisant les accès concurrents. Si le tampon est plein, l'appel sera bloquant jusqu'à ce qu'une case se libère. Par contre, l'opération débloquera des consommateurs bloqués si toutefois le tampon était vide précédemment. On veillera à limiter les déblocages inutiles.

Testez le bon fonctionnement du tampon circulaire protégé grâce au programme principal du fichier *main_protected_buffer.c*. Vous pourrez utiliser le scénario de test, *test-00.txt*. On notera les moments où les consommateurs et les producteurs sont supposés bloquer. Normalement, avec cette sémantique, toutes les données produites doivent être consommées. Typiquement, dans l'exemple *test-00.txt*, on s'attend à avoir une sortie comme :

```
...
000 producer 02 put (B) - data=200
000 consumer 00 get (B) - data=200
000 producer 03 put (B) - data=300
000 consumer 01 get (B) - data=300
000 producer 04 put (B) - data=400
005 consumer 00 get (B) - data=400
...
```

dans lequel chaque donnée est consommée, et consommée une seule fois.

1.3 Tampon protégé contre accès concurrents (var cond/non-bloquants)

Nous allons implanter dans le fichier *cond_protected_buffer.c* des accès concurrents au tampon circulaire protégé similaires aux précédents. Cependant, la sémantique de ces accès sera cette fois non-bloquante. Le producteur signalera par un code de retour binaire s'il a réussi à insérer immédiatement un élément dans le tampon. Le consommateur signalera par un pointeur nul ou non sur une donnée s'il a réussi à extraire immédiatement un élément dans le tampon. Le tampon devra toujours être protégé contre les accès concurrents.

Complétez les procédures *cond_protected_buffer_remove* et *cond_protected_buffer_add* afin qu'il offre les services indiqués.

Testez le bon fonctionnement du tampon circulaire protégé grâce au programme principal du fichier *main_protected_buffer.c*.

Vous pourrez utiliser les scénarii de test, *test-01.txt* et *test-02.txt*. On vérifiera que les consommateurs et les producteurs ne restent pas bloqués et s'activent comme prévu. Le scénario *test-01.txt* a plus de producteurs que de consommateurs, le tampon de *taille 1* se trouvera rempli et certains producteurs échoueront à déposer leurs données.

Ci-dessous, à la date *000*, le *consommateur 00* essaye de consommer alors que le tampon est vide (*data=NULL*). Puis, les 5 producteurs produisent des données mais la taille du tampon étant de 1, seule la donnée produite par le premier *producteur 01* sera stockée dans le tampon. Ceci est confirmé à la *date 005*, lorsque le consommateur consomme et retire la *donnée 100* déposée en premier et non la *donnée 500* déposée en dernier. D'ailleurs, à la *date 010*, de nouveau est consommée la *donnée 101* et non les *données 200 ou 501*. Le fait que le producteur échoue est indiqué par une donnée NULL produite. Notez l'indication *(U)* pour signifier la sémantique non-bloquante.

```
...
000 consumer 00 remove (U) - data=NULL
001 producer 01 add (U) - data=100
001 producer 02 add (U) - data=NULL
001 producer 03 add (U) - data=NULL
001 producer 04 add (U) - data=NULL
001 producer 05 add (U) - data=NULL
005 consumer 00 remove (U) - data=100
006 producer 01 add (U) - data=101
006 producer 02 add (U) - data=NULL
006 producer 03 add (U) - data=NULL
006 producer 04 add (U) - data=NULL
006 producer 05 add (U) - data=NULL
010 consumer 00 remove (U) - data=101
...
```

Dans le scenario *test-02.txt*, au contraire, ce sont les consommateurs qui échoueront à extraire des données.

```
...
000 consumer 00 remove (U) - data=NULL
000 consumer 01 remove (U) - data=NULL
000 consumer 02 remove (U) - data=NULL
000 consumer 03 remove (U) - data=NULL
000 consumer 04 remove (U) - data=NULL
000 producer 05 add (U) - data=500
005 consumer 00 remove (U) - data=500
005 consumer 01 remove (U) - data=NULL
005 consumer 02 remove (U) - data=NULL
...
```

1.4 Tampon protégé contre accès concurrents (var cond/temporisés)

Nous allons implanter dans le fichier *cond_protected_buffer.c* des accès concurrents au tampon circulaire protégé similaires aux précédents. Cependant, la sémantique de ces accès sera cette fois temporisée. Si l'opération ne peut pas s'effectuer immédiatement, elle bloque jusqu'à ce qu'elle puisse avoir lieu, mais n'attendra pas au delà d'une échéance donnée (qui correspond à une date ie un temps absolu). En l'occurrence, les accès doivent être débloqués au plus tard à l'échéance suivante comme l'explicite la variable *deadline* dans le code ci-dessous. Si l'opération échoue, elle retourne une valeur spéciale comme précédemment (NULL).

```
struct timespec deadline = get_start_time();
...
add_millis_to_timespec (&deadline, producer_period);
...
done=protected_buffer_offer(protected_buffer, data, &deadline);
```

Complétez les procédures *cond_protected_buffer_poll* et *cond_protected_buffer_offer* afin qu'il offre les services indiqués. On utilisera les outils POSIX dans leur version temporisée.

Testez le bon fonctionnement du tampon circulaire protégé grâce au programme principal du fichier *main_protected_buffer.c*. Vous pourrez utiliser les scénario de test, *test-03.txt* et *test-04.txt*. On vérifiera que les consommateurs et les producteurs restent bloqués au plus le temps imparti et s'activent comme prévu.

Par exemple, dans le scenario *test-04.txt*, le producteur *05* dépose une donnée consommée immédiatement par le premier consommateur. Puis les 4 consommateurs suivants échouent à consommer une donnée et sont débloqués lors de la période suivante ie à la date *05*.

```
...
000 producer 05 offer (T) - data=500
000 consumer 00 poll (T) - data=500
005 consumer 04 poll (T) - data=NULL
005 consumer 03 poll (T) - data=NULL
005 consumer 02 poll (T) - data=NULL
005 consumer 01 poll (T) - data=NULL
005 producer 05 offer (T) - data=501
005 consumer 00 poll (T) - data=501
010 consumer 04 poll (T) - data=NULL
010 consumer 02 poll (T) - data=NULL
010 consumer 03 poll (T) - data=NULL
010 consumer 01 poll (T) - data=NULL
...
```

De même dans le scenario *test-03.txt*,

```
...
```

```
000 producer 01 offer (T) - data=100
000 consumer 00 poll (T) - data=100
000 producer 02 offer (T) - data=200
005 producer 05 offer (T) - data=NULL
005 producer 03 offer (T) - data=NULL
005 producer 04 offer (T) - data=NULL
005 consumer 00 poll (T) - data=200
005 producer 01 offer (T) - data=101
010 producer 05 offer (T) - data=NULL
010 producer 03 offer (T) - data=NULL
010 producer 04 offer (T) - data=NULL
010 producer 02 offer (T) - data=NULL
010 consumer 00 poll (T) - data=101
...
```

1.5 Délai d'attente jusqu'à une date donnée

Dans le code du producteur *main_producer* et du consommateur *main_consumer*, vous avez utilisé la fonction *delay_until* pour suspendre le thread jusqu'à l'échéance suivante (*deadline*).

Analysez la procédure *delay_until* dont le code se trouve dans le fichier *utils.c*. Expliquez comment cette procédure peut faire attendre la tâche bien au delà de l'échéance si l'exécution de *gettimeofday* n'est pas immédiatement de l'exécution de *nanosleep*.

En utilisant les fonctions POSIX précédentes, notamment celles de la question précédente, vous fournirez une implantation alternative de *delay_until* qui ne présentera pas l'inconvénient relevé dans le paragraphe précédent.

Vous vérifierez que les scénarii précédents fonctionnent normalement.

1.6 Tampon protégé contre accès concurrents (sémaphore/bloquants)

On reprend la sémantique des accès concurrents précédents, mais cette fois ci nous allons réaliser une implémentation fondée sur les sémaphores. Au lieu d'utiliser des variables conditionnelles pour bloquer l'exécution, vous allez utiliser des sémaphores. Vous noterez que, contrairement à la version avec les variables conditionnelles, vous utiliserez nécessairement deux files d'attente. Vous modifierez les fichiers *sem_protected_buffer.h* et *sem_protected_buffer.c*.

Le travail demandé et le résultat attendu sont les mêmes qu'à la section Tampon protégé contre accès concurrents (var cond/bloquants). Cependant, il vous faudra utiliser le fichier de scenario *test-10.txt* pour tester votre travail.

1.7 Tampon protégé contre accès concurrents (sémaphore/non-bloquants)

On reprend la sémantique des accès concurrents précédents, ici celle des accès non-bloquants, mais en utilisant une implémentation fondée sur les sémaphores. Modifiez en conséquence les fichiers *sem_protected_buffer.h* et *sem_protected_buffer.c*.

Le travail demandé et le résultat attendu sont les mêmes qu'à la section Tampon protégé contre accès concurrents (var cond/non-bloquants). Cependant, il vous faudra utiliser les fichiers de scenario *test-11.txt* et *test-12.txt* pour tester votre travail.

1.8 Tampon protégé contre accès concurrents (sémaphore/temporisés)

On reprend la sémantique des accès concurrents précédents, ici celle des accès temporisés, mais en utilisant une implémentation fondée sur les sémaphores. Modifiez en conséquence les fichiers *sem_protected_buffer.h* et *sem_protected_buffer.c*. Vous pourrez utiliser les mêmes scénarii que ceux de l'implémentation précédente.

Le travail demandé et le résultat attendu sont les mêmes qu'à la section Tampon protégé contre accès concurrents (var cond/temporisés). Cependant, il vous faudra utiliser les fichiers de scenario *test-14.txt* et *test-13.txt* pour tester votre travail.