TP6 : Programmation concurrente

Dans ce TP, on s'intéresse à un programme concurrent dans lequel plusieurs threads coopèrent pour réaliser une tâche commune. Une première version du programme vous est fournie, mais la synchronisation n'est pas faite correctement et le programme calcule un résultat faux. Votre travail va consister à modifier le code de façon à corriger les différents problèmes de synchronisation.

1 Prologue : prise en main du code fourni

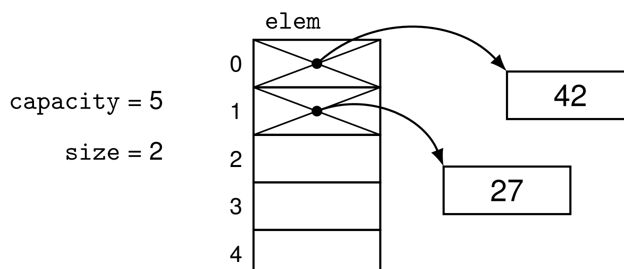
Le scénario étudié dans ce TP est similaire au scénario producteur-consommateur vu en cours, mais avec N producteurs et N consommateurs. Tous ces threads coopèrent en se partageant une structure de données commune. Il s'agit d'un *multiensemble*[1], aussi appelé «sac», c'est à dire un conteneur dans lequel nos threads vont déposer ou retirer des éléments. Contrairement à l'exemple du cours, l'ordre des éléments dans le sac n'est pas significatif : on dira simplement qu'on «ajoute un élément dans le sac» ou qu'on «retire un élément du sac».
Plus précisément, on s'intéresse dans ce TP à un sac de taille bornée (i.e. un *«bounded bag»*) avec les propriétés suivantes :

1. La capacité du sac (qu'on notera C) est fixée une fois pour toutes au moment de son initialisation.
2. Si le sac est «plein» (i.e. s'il contient C éléments) alors l'opération d'ajout est bloquante : elle ne rendra la main au thread appelant que lorsqu'une place se sera libérée et que l'élément aura bien été ajouté.
3. De même, l'opération de retrait est bloquante sur un sac vide.

Attention, dans l'implémentation qui vous est fournie, ces garanties de synchronisation ne sont pas implémentées correctement.

Exercice 1 Téléchargez puis décompressez l'archive correspondant à ce TP. Ouvrez et lisez `bag.h` en vous aidant des explications ci-dessous. Posez des questions sur ce que vous ne comprenez pas.

- Le `typedef struct` définit les attributs d'un sac (taille max, etc). La déclaration `void **elem` ne doit pas vous effrayer : `elem` est simplement un tableau de pointeurs universels (i.e. un tableau de `void*`). Notre cas d'utilisation d'aujourd'hui est illustré ci-dessous : on va stocker dans le sac des valeurs entières, donc notre tableau `elem` contiendra des pointeurs d'entiers.
- Le drapeau `is_closed`, et la méthode `bb_close()` sont hors-sujet pour l'instant, vous pouvez les ignorer. On s'y intéressera de nouveau à la partie 3.
- La fonction `bb_create()` est le constructeur : elle alloue et initialise un sac avec une capacité donnée.
- Les méthodes `bb_add()` et `bb_take()` sont définies dans `bag.c`. Allez lire leur implémentation. Vous remarquerez que les synchronisations par attente active (boucles `while`) sont simplistes et n'ont aucune chance de fonctionner correctement.



1. Pour les curieux, voir `https://fr.wikipedia.org/wiki/Multiensemble`

Exercice 2 Ouvrez et lisez `main.c` en vous aidant des explications ci-dessous. Posez des questions sur ce que vous ne comprenez pas.

- Les «éléments» de notre sac seront des `int*` pointant sur les nombres qui nous intéressent.
- Le programme principal crée N producteurs et N consommateurs, et leur indique à chacun leur numéro de thread.
- Chaque thread consommateur retire (avec `bb_take`) des éléments du sac et ajoute chaque valeur obtenue à une variable partagée `sum`.
- Chaque thread producteur, en fonction de son numéro i, alloue i entiers de valeur 1 et les ajoute au sac.
- Donc en comptant tous les producteurs, on aura au total $\frac{N\times(N+1)}{2}$ fois le nombre 1 (mais pas forcément en même temps dans le sac, car C peut être beaucoup plus petit que cette valeur).
- Le programme principal affiche le résultat «théorique» en utilisant la formule, ainsi que la valeur de la variable `sum`. Si les synchronisations sont correctes, alors les deux valeurs sont identiques.

Exercice 3 Tapez `make` puis `./main 500 10` pour exécuter votre programme avec $N = 500$ et $C = 10$. Respectivement, essayez avec `./main 10 500`. Répétez l'expérience plusieurs fois en variant les paramètres, et constatez que vous obtenez des résultats fantaisistes, et volontiers différents d'une fois sur l'autre : somme incorrecte, *segmentation fault*, messages *assertion failed*, ou encore des erreurs dans l'allocateur mémoire (par exemple, erreur *double free*). Le comportement exact du programme dépend de votre système : matériel, version de l'OS, etc. Si vous obtenez un comportement toujours identique, essayez avec des plus grandes valeurs pour N et/ou C (vous pouvez aller jusqu'à des nombres à quatre chiffres). Vous pouvez aussi décommenter les divers `assert()` marqués *sanity check*, pour observer plus précisément les erreurs à l'exécution. Dans tous les cas, l'objectif de cet exercice est de se convaincre, par l'observation, que le code distribué est faux. Au besoin, demandez de l'aide à un enseignant.

Exercice 4 L'objectif du TP est de corriger un à un les bugs de concurrence du programme : accès aux variables partagées, synchronisation des threads, etc. Un premier problème concerne l'affichage du résultat, qui pour l'instant n'a aucune chance d'être correct. En effet, notre `main()` lit la variable `sum` immédiatement après avoir créé tous les threads, sans leur laisser le temps de s'exécuter. Pour vous en convaincre, décommentez les divers affichages dans les threads pour mieux visualiser le détail de l'exécution.
Dans `main()`, ajoutez un `sleep(2)` juste avant les deux `printf()`. Exécutez de nouveau votre programme plusieurs fois, avec les différents paramètres essayés à l'exercice précédent.

Remarques

- Attention, il s'agit seulement d'un contournement (VO : *workaround*) et pas d'une vraie solution. On s'intéressera de nouveau à ce problème de terminaison dans la dernière partie du TP.

2 Synchronisation des opérations concurrentes

Dans cette première partie, on s'intéresse aux accès concurrents.

Exercice 5 Dans `main.c`, ajoutez des synchronisations pour garantir que tous les accès à la variable partagée `sum` sont faits en exclusion mutuelle. Faites valider par un enseignant (mais n'attendez pas pour avancer sur les exercices suivants)

Remarques

- Pour l'exclusion mutuelle, on pourrait se servir des fonctions `pthread_mutex_bidule()` de `pthread.h` mais dans la suite du TP il faudra aussi faire de la signalisation donc je vous recommande plutôt les fonctions `sem_bidule()` de `semaphore.h`. N'hésitez pas à relire les diapos correspondantes dans le cours, et/ou lire la page `man sem_overview`.

— Si vous êtes sous Linux ou WSL, vous pouvez utiliser des sémaphores anonymes, que vous initialiserez avec `sem_init()`. Mettez le paramètre `pshared` à zéro (booléen *faux*), et lisez `man sem_init` pour en savoir plus.
— Si vous êtes sous macOS, il vous faudra utiliser des «sémaphores nommés», que vous créerez avec `sem_open()`. Un exemple vous est donné dans l'archive de départ (répertoire `misc/`)
— Attention, à ce stade votre programme n'affichera toujours pas un résultat correct ! cf l'exercice suivant.

Exercice 6 Ouvrez `bag.c` et trouvez à quel endroit sont implémentées les synchronisations. Chacune des deux boucles `while` voudrait faire attendre le thread appelant dans le cas où les conditions ne sont pas réunies. Par exemple dans `bb_take()` on scrute le nombre d'éléments du sac jusqu'à ce qu'il devienne positif.
Cette façon naïve de synchroniser les threads pose plusieurs problèmes. D'une part, les accès concurrents à la variable `size` constituent une *race condition*, mais en plus la sychronisation est basée sur une attente active, ce qui occupe inutilement le processeur.

Modifiez `bb_add()` et `bb_take()` pour corriger ces deux problèmes.

Vous devrez à la fois empêcher les accès concurrents aux variables partagées (i.e. exclusion mutuelle) et à la fois assurer les blocages et réveils (i.e. signalisation) en cas de sac plein ou de sac vide.
Pour tout cela, vous utiliserez des sémaphores que vous ajouterez à la structure `bag_t`. N'oubliez pas d'initialiser chacun de ces sémaphores dans `bb_create()`.

Faites valider par un enseignant (mais n'attendez pas pour travailler sur les exercices suivants)

3 Terminaison «propre» du programme

Dormir pendant deux secondes à chaque exécution est clairement sous-optimal. Dans la suite, vous allez implémenter une solution plus élégante car basée sur une contrainte de précédence explicite : avant d'afficher le résultat, on va attendre (se suspendre) jusqu'à ce que tous les threads aient terminé. Évidemment, la tâche sera plus facile pour les producteurs (basés sur une boucle `for`) que pour les consommateurs, qui pour l'instant ne se terminent pas.

Exercice 7 Rajoutez à votre `main()` des appels à `pthread_join()` pour attendre la fin des threads producteurs.

Exercice 8 Attendre la fin des threads consommateurs sera plus difficile car pour l'instant, une fois que le sac est vide ils vont tous se bloquer dans `bb_take()`, et ne jamais en sortir. Pour régler ce problème, vous allez rajouter à votre sac une nouvelle fonctionnalité, qui sera accessible via la méthode `bb_close()` avec les effets suivants :
— une fois qu'un sac est fermé, il devient interdit d'appeler `bb_add()` ou `bb_close()`,
— fermer un sac réveille tous les threads suspendus dans `bb_take()`,
— sur un sac vide *et* fermé, `bb_take()` renvoie NULL au lieu de bloquer.
Implémentez ces fonctionnalités dans `bag.c` et adaptez votre programme en conséquence : il vous faudra appeler `bb_close()` depuis le `main()`, mais également modifier `consumer()` puisqu'on change le contrat de `bb_take()`. Vous pouvez finalement rajouter des `pthread_join()` pour attendre la fin des threads consommateurs avant l'affichage de `sum`.