Rapport ENSTA 2020 – ROB305 – Gabriel Riqueti

[TD-1] Mesure de temps et échantillonnage en temps

a) Timers avec callback

b) Fonction simple consommant du CPU

c) Mesure du temps d'exécution d'une fonction

En tournant le programme de ce travail, j`ai obtenu pour une boucle de taille 1.000.000.000 un compteur (counter) de valeur égale à 1.000.000.000 et une durée égale à 2,96288 s, toutefois cette valeur change beaucoup.

d) Amélioration des mesures

Concernant la variable *pStop, on a utilisé leur valeur comme un critère de dédans la boucle for de la fonction incr.

Le étalonnage pareil correcte. Le résultat est ci-dessous :

```
Compteur : 3.129e+08
Temps d'exécution : 999.999 ms
Compteur : 1.35289e+09
Temps d'exécution : 4000 ms
Compteur : 2.0497e+09
Temps d'exécution : 6000.01 ms
Etalonnage
a = 348408 compteurs/ms
b = -8.148e+10
```

Tous ces valeurs change un peu entre exécutions du programme.

e) Gestion simplifiée du temps Posix

Pour améliorer des mesures, on a utilisé la Méthode de Moindres Carrés pour 1000 échantillons. L'intervalle est variable, en effet, il suive la fonction de troisième degré dont le première valeur est la minimum 0.01 ms et la dernière est 500 ms. Cette décision prend en compte le trade-off entre temps d'exécution et nombre d'échantillons. Le resultat est le suivant :

```
Etalonnage
a = 330150 compteurs/ms
b = -956663 compteurs
Erreur = 1.44e+06 compteurs
Temps total d'exécution approximé : 124646 ms
```

Le erreur était calculé comme le racine de la moyenne de la différences aux carrés et montre un erreur considérable pour la variable b toutefois la variable a est plus précise car elle prend en compte le temps d'exécution qui est grand aussi.

Pour s'assurer que la tâche ne soit pas perturbée par l'exécution d'autres, on a utilisé un thread Posix avec SCHED_RR comme sched policy et la priorité maxima possible.

[TD-2] Familiarisation avec l'API multitâches *pthread*

a) Exécution sur plusieurs tâches sans mutex

On a tourné le programme trois fois comme suivant :

```
Boucle de taille 10000
Nombre de tâches : 10
Compteur : 47307
Temps d'exécution : 0.595533 ms
Boucle de taille 10000
Nombre de tâches : 10
Compteur : 89194
Temps d'exécution : 1.95145 ms
Boucle de taille 10000
Nombre de tâches : 10
Compteur : 88387
Temps d'exécution : 1.23204 ms
```

Dans les trois exécutions montrés, on voit que les compteurs bien comme les temps d'exécution sont variable et que la valeur du compteur n'est pas le nombre de tâches multiplié par le nombre de fois que chaque tâches rentre dans la boucle (100000). La cause est que comme toutes les tâches essaient changer la valeur du compteur au même temps et ce commande est composé par d'autres plus petites, ces conflits rendent des effectuations incorrectes.

b) Mesure de temps d'exécution

Les mesures prises peuvent être regardées dans la Figure 1 :

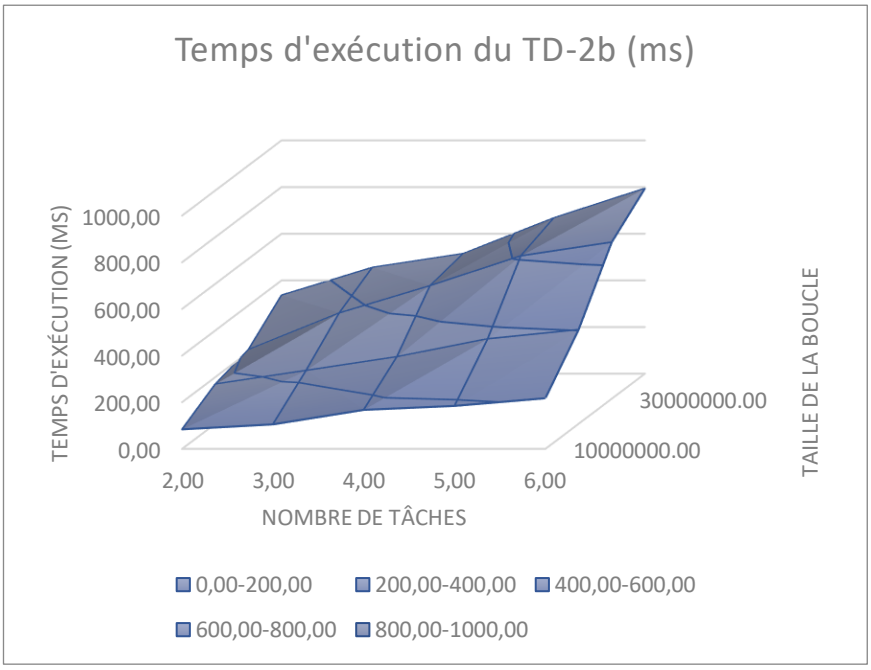

Figure 1. Temps d'exécution du programme 2b en variant les valeurs de nombre de tâches et de taille de la boucle.

D'après le graphique, on peut comprendre que l'architecture du processus change de la même façon par rapport les deux paramètres linéairement. Alors il n'y a pas un coût d'exécution pour paralléliser les tâches.

c) Exécution sur plusieurs tâches avec mutex

Toutes les exécutions étaient faites avec une boucle de taille 10e+7 et dix threads avec ordonnance SCHED_RR.

```
Avec Mutex :
Mutex : PTHREAD_MUTEX_NORMAL
Compteur : 1e+08
Temps d'exécution : 306.766 ms
Mutex : PTHREAD_MUTEX_RECURSIVE
Compteur : 1e+08
Temps d'exécution : 300.687 ms
Mutex : PTHREAD_MUTEX_ERRORCHECK
Compteur : 1e+08
Temps d'exécution : 302.506 ms
Sans Mutex :
Compteur : 1e+08
Temps d'exécution : 302.742 ms
Compteur : 1.26868e+07
Temps d'exécution : 349.051 ms
Compteur : 2.99132e+07
Temps d'exécution : 349.051 ms
```

On voit que le mutex assure la bonne exécution des threads car toujours on a le compteur égal au taille de la boucle multiplié par le nombre de tâches. Le temps d'exécution ne change pas beaucoup parmi les mutex, toutefois les conflits parmi les tâches sans mutex augmentent le temps d'exécution.

[TD-3] Classes pour la gestion du temps

a) Classe Chrono

b) Classe Timer

Tout d'abord, le timer_t tid est une variable privée parce que, dans le paradigme des langages orientés à objets, on laisse les données des classes cachés pour tous qui est dehors la classe et on laisse les méthodes des classes publique pour utiliser les données de ces classes de façon plus sécure. C'est le cas du constructeur, du destructeur et des fonctions start et stop.

Ensuite, la fonction virtuelle pure callback est protégée puisqu'il faut implémenter ce fonction dans les classes dérivées non abstraites afin d'utiliser la classe dérivée et la classe abstraite de manière indirecte.

A la fin, call_callback est privée parce que on ne l'utilise que pour appeler callback et on ne veut pas que cette fonction soit accessible dehors la classe Timer. Leur utilité est appeler callback, il faut l'utiliser parce qu'on veut que les classes dérivées implémentent la fonction callback, donc callback doit être virtuelle pure cependant les fonctions appelées par les timers doivent être statique. Comme les timers ne prennent pas les objets de la classe Timer comme argument, il faut lui envoyer une fonction statique, c'est-à-dire de la classe.

c) Calibration en temps d'une boucle

[TD-4] Classes de base pour la programmation multitâches

a) Classe Thread

D'abord, pour les classes d'exceptions de cette question et de la suivante, on a utilisé la même approche : une variable int retVal pour recevoir la réponse des fonctions de thread, un constructeur pour passer le paramètre de la valeur d'erreur de ces fonctions et une fonction message qui imprime dans l'écran l'erreur attrapé.

Ensuite, pour stocker les valeurs des temps absolus de début et de fin, on a créé le donnée membre chrono de la classe Chrono. Cette classe peut aussi effectuer ces valeurs.

Enfin, selon le paradigme de langages orientés à objets on a créé une classe ThreadCounter héritant la classe Thread. Cette nouvelle classe implémente la fonction run, alors elle n'est pas abstraite, elle contient quelques méthodes et données statiques pour paramétriser la fonction run, accéder la valeur des données de paramétrisation.

b) Classes Mutex et Mutex::Lock

D'abord, on a utilisé les politiques de planification déjà définies dans la questions précédent de la même façon que dans la question 2b en passant le paramètre de SCHED_RR, SCHED_FIFO ou SCHED_OTHER.

Au lieu d'utiliser la classe ThreadCounter, on a créé la classe ThreadCounterSafe qui hérite de la classe Thread aussi. Ce n'est pas possible de développer cette classe comme une classe dérivée de ThreadCounter parce qu'il faut changer la fonction run et si on crée une méthode run dans une classe dérivée à ThreadCounter, on va créer une ambiguïté dans l'appel de la fonction run dans la méthode call_run.

Pour tester les fonctions de mutex qui depent du temps, l'useur doit passer l'argument de temps d'attente en ms. En vue de que les threads sont sécures, on a utilisé le même mutex et condition pour tous les tâches. Enfin, la décision de l'ordre que les tâches peuvent accéder les variables partagées dans la séction critique est réaliser par la fonction nextThreadActive qui permet la thread suivant à accéder ces varaibles.

c) Classe Semaphore

d) Classe Fifo multitâches

[TD-5] Inversion de priorité