Rapport ENSTA 2022 – ROB305 – Yufei HU

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

L'ensemble du code est placé dans le dossier TD_1. Il faut aussi utiliser Les deux bibliothèques <timer.h> et <signal.h> .

a) Gestion simplifiée du temps Posix

Dans ce TD, on utilise la structure ***timespec*** qui représente les mesures de temps dans l'API Posix pour générer des fonctions et des opérateurs. Dans le même temps, ces fonctions sont également les fonctions de support de TD2~5. Pour accéder à cette structure, on doit inclure la bibliothèque <time.h>. Dans toutes les fonctions suivantes, les variables de type ***timespec*** résultant du calcul doivent être normalisées avec les contraintes *tv_sec* et *tv_nsec*.

Le script *main_td1a.cpp* dans la dossier TD_1 permet de tester l'ensemble des fonctions implémentées dans cet exercice.

b) Timer avec callback

Le code de test de ce TD est écrit en *main_td1b.cpp* qui peut être trouvé dans le dossier TD_1. On implémente un temporisateur Posix périodique avec une fréquence de 2 Hz en imprimant un message avec une valeur de compteur incrémentée périodiquement. Le programme doit s'arrêter après 15 incréments.

Le hander de cet Timer est le suivant :

```
void myHandler(int, siginfo_t* si, void*)
{
    int* p_counter = (int*)si->si_value.sival_ptr;
    *p_counter += 1;
    std::cout <<*p_counter << std::endl;
}
```

c) Fonction simple consommant du CPU

On a implémenté une fonction d'incrémentation qui doit exécuter une boucle qui incrémente la valeur du compteur pointé par *pCounter* de 1.0, elle doit exécuter cette boucle *nLoops* fois. *nloop* est défini manuellement. De plus, on utilise les fonctions de *timespec.h* pour calculer le temps d'exécution de ce processus.

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

Dans ce TD, onmodifie la fonction *incr* pour ajouter un troisième paramètre *pStop*. Cette fonction peut donc s'exprimer sous la forme suivante :

On peut contrôler le fonctionnement de cette fonction en notifiant la valeur de *pStop* via le callback. Soit l(t) le nombre de boucles que la fonction *incr* exécute dans l'intervalle de temps t ; on suppose que cette fonction est affine : $l(t)=a\times t+b$. On calcule la valeur a et b à deux moments différents. Enfin, on utilise la valeur a et b calculés pour simuler l'augmentation de la valeur à 8 secondes et comparer avec la valeur réelle.

```
unsigned int incr(unsigned int nLoops, double* pCounter, bool* pStop){
    unsigned int iLoop;
    for(iLoop = 0; iLoop < nLoops; ++iLoop)
    {
        if (*pStop == true) break;
        *pCounter += 1.0;
    }
    return iLoop;
}
```

e) Gestion simplifiée du temps Posix

Afin d'obtenir des valeurs plus précises de a et b, nous utilisons la méthode des moindres carrés pour calculer plusieurs points dans le temps et le nombre de cycles correspondant.

```
void calculateParametersWithLeastSquares(struct parameters & paras){
    double x_average=0.0,y_average=0.0;
    double xy_sum=0.0, x2_sum=0.0;
    double y_value=0.0;
    int n = 10; // samples

    for (int i =0;i < n;i++){
        std::cout<<"Working on "<<i+1<<"th sample..."<<std::endl;
        y_value=0.0;
        x_average+=i+1;
        Timer(i+1,y_value);
        y_average+= y_value;
        xy_sum += (i+1)*y_value;
        x2_sum += (i+1)*(i+1);
    }
    x_average /= n;
    y_average /= n;
    paras.a = (xy_sum-n*x_average*y_average)/(x2_sum-n*x_average*x_average);
    paras.b = y_average-paras.a*x_average;
}
```

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

L'ensemble du code est placé dans le dossier TD_2. On utilise les fonctions de ***timespec*** de TD1.

a) Exécution sur plusieurs tâches sans mutex

Dans ce TD, on utilise la bibliothèque <thread> pour démarrer plusieurs threads, puis observer la valeur de *pCounter* sans utiliser Mutex. On constate que lorsque plusieurs threads sont activés et que la valeur de *nLoops* est élevée, les threads se disputent les ressources de pCounter, ce qui entraîne un écart important de la valeur réelle du résultat du calcul.

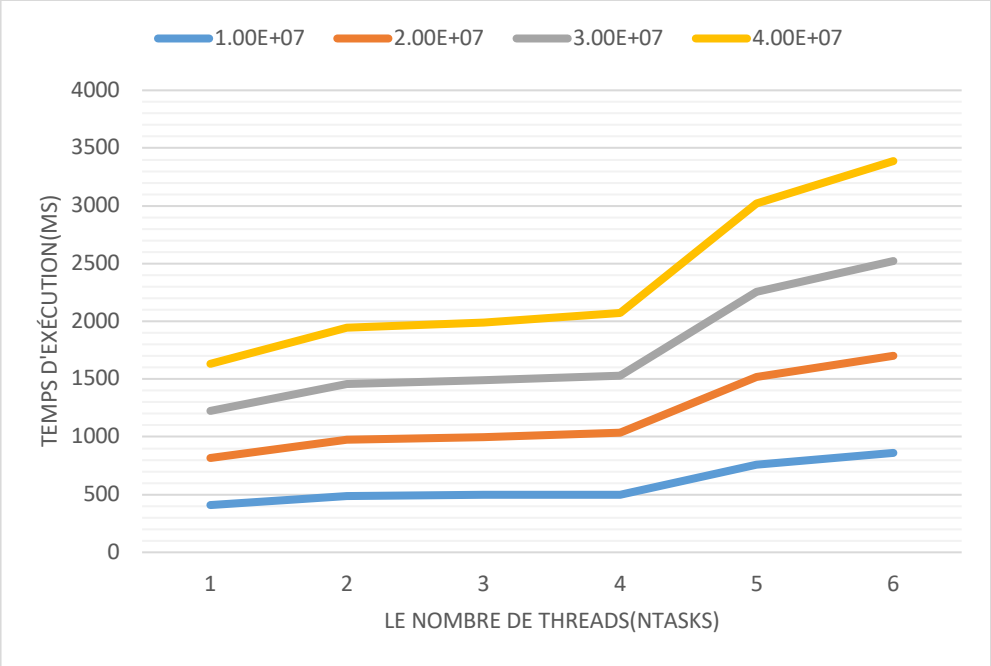
Par exemple, lorsque le nombre de threads est de 5 et que *nLoops* est de 10 000, la valeur obtenue à partir de la Raspberry Pi2 est **26 570** et la valeur théorique doit être de **50 000.**

b) Mesure de temps d'exécution

On planifie les coureurs avec *SCHED_RR* et Utilise la fonction *clock_gettime* pour compter le temps d'exécution dans différentes situations avec l'aide de la Raspberry Pi2. Comme indiqué dans le tableau ci-dessous.

Unité (ms)

nTasks\nLoops	1e7	2e7	3e7	4e7
1	409.198	816.764	1224.46	1632
2	487.606	972.69	1459.38	1944.58
3	499.258	994.693	1491.98	1988.5
4	496.489	1035.01	1526.93	2072.76
5	759.262	1517.39	2253.03	3022.17
6	860.827	1701.47	2522.17	3387.58



c) Exécution sur plusieurs tâches avec mutex

Lorsqu'on utilise *mutex*, on constate que la valeur finale du compteur est égale à *nLoops* * *nTasks*. Cela signifie que le problème de plusieurs threads en concurrence pour la même ressource est résolu. On constate également que dans ce cas le temps d'exécution est plus long.

[TD-3] Classes de base pour la programmation multitâche

L'ensemble du code est placé dans le dossier TD_3. On utilise les fonctions de ***timespec*** de TD1.

a) Classe Chrono

On a créé une nouvelle classe qui est Chrono pour implémenter la fonctionnalité de mesure du temps du chronomètre. On a également fini de tester la classe dans *main_td3a.cpp*.

b) Classe Timer

On a implémenté une classe Timer qui encapsule la fonctionnalité de minuterie Posix. Cette classe est abstraite, où l'opération callback ne peut pas être implémentée, et elle doit être implémentée pour un timer spécifique dérivé de la classe Timer.

Pour tester la classe Timer, on crée une nouvelle classe appelée *CountDown* qui hérite la classe de Timer.

c) Calibration en temps d'une boucle

On a utilisé une méthode carrée similaire à TD1 pour vérifier la valeur de et b. Le code clé est indiqué ci-dessous.

```
Calibrator::Calibrator(double samplingPeriod_ms, unsigned int nSamples) {
    a=0.0, b=0.0;

    double x_average=0.0,y_average=0.0;
    double xy_sum=0.0, x2_sum=0.0;

    this->nSamples = nSamples;
    start(samplingPeriod_ms);
    looper.runLoop();
    stop();

    for (unsigned int i =0;i < nSamples;i++){
        std::cout<<"Working on "<<i+1<<"th sample..."<<std::endl;
        x_average+=(i+1)*samplingPeriod_ms;
        y_average+= samples[i];
        xy_sum += (i+1)*samplingPeriod_ms*samples[i];
        x2_sum += (i+1)*(i+1)*samplingPeriod_ms*samplingPeriod_ms;
    }
    x_average /= nSamples;
    y_average /= nSamples;
    a = (xy_sum-nSamples*x_average*y_average)/(x2_sum-
nSamples*x_average*x_average);
    b = y_average-a*x_average;

    std::cout << "a :" << a << std::endl;
    std::cout << "b :" << b << std::endl;

}
```

On produit également des valeurs réelles et simulées. Par comparaison, on constate que le résultat calculé avec les valeurs des paramètres(a et b) qu'on a vérifiés est très faibles par rapport au valeur réelle.

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

Dans ce TD, il s'agit d'encapsuler la gestion des tâches Posix dans les classes *PosixThread*, *Thread*, *Mutex* et *Lock*. L'ensemble du code est placé dans le dossier TD_4.

a) Classe Thread

On implémente la structure indiquée par TD, où la classe *Thread* hérite de la classe *PosixThread*. De plus, pour tester ces deux classes abstraites, on crée une nouvelle classe *IncrThread* et spécifie *ntasks* et *nLoops* manuellement. En l'absence de *Mutex*, On constate qu'il existe encore plusieurs threads en concurrence pour la même ressource(*counter*).

b) Classes Mutex et Mutex::Lock

On a créé une nouvelle classe appelée *Mutex* pour résoudre le problème des courses de données multithread. Autrement dit, lorsque chaque thread exécute le code de clé, il est

verrouillé avec Lock first, c'est-à-dire que seul le thread actuel peut accéder à la ressource. De même, à l'aide du verrou de Mutex, la valeur de compteur est égale à *nTasks*nLoops*.

c) Classe Semaphore

Dans cet exercice, on a créé une nouvelle classe qui est *Semaphore*, qui hérite de la classe *Mutex*. Ensuite, pour tester la classe, on crée deux nouvelles classes qui sont *SemaphoreProducer* et *SemaphoreConsumer*. Parmi eux, *SemaphoreProducer* est responsable de l'augmentation de la valeur de *Token*. Par contre, *SemaphoreConsumer* est responsable de la diminution de la valeur de *Token*. On constate que lorsque *nPros =5* et *nCons=4*, le jeton(*Token*) augmentaient d'abord à 5e5, puis diminuaient à 1e5 comme prévu. Il n'y a pas de problème de course aux données dans ce processus.

d) Classe Fifo multitâches

On utilise le conteneur C++ std::queue pour programmer la classe *Fifo*. On teste également la classe en y accédant simultanément à partir de plusieurs tâches de production et de consommation. Pour ce faire, utilisez un entier Fifo<int> et demandez à chaque tâche de production de produire une séquence d'entiers de 0 à n. On constate que le nombre d'éléments créés est égal au 5e5 et le nombre d'éléments réduite à 1e5 lorsque *nPros =5, nCons=4* et *maxCount = 1e5*.

[TD-5] Inversion de priorité

Dans cet exercice, avec l'aide de TD3 et Td4, on utilise le contenu créé pour mettre en œuvre l'inversion de priorité. On sait que le processeur étant multicœur, on doit utiliser 100 % de tous les autres cœurs. C'est-à-dire qu'on doit se préparer à l'avance à ne définir qu'un seul processeur pour les tests.