

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

Les fichiers correspondant à ce TD sont situés dans le fichier TD1/src. Le fichier README contient les instructions pour l'exécution de chacun des exercices mis en œuvre dans les fichiers main_td1x.cpp.

a) Gestion simplifiée du temps Posix

Cette section présente l'implémentation de fonctions et d'opérateurs permettant une utilisation simplifiée de la structure *timespec* représentant la mesure des temps dans l'API Posix. Pour cela, il était nécessaire d'utiliser la bibliothèque *time.h*. Compte tenu du fait que la structure *timespec* est constituée d'un champ de secondes et d'un autre de nanosecondes, les conventions suivantes sur ces paramètres ont été prises en compte pour régulariser le calcul des fonctions et opérateurs à concevoir :

- Le champ *tv_sec* (de type *time_t*) peut prendre des valeurs négatives
- Le champ *tv_nsec* (de type *long*) doit être positif ou nul et toujours inférieur à 1000000000 (109) et ce, même pour un temps globalement négatif.

Ces fonctions et opérateurs se trouvent, en dehors du dossier TD1, dans le répertoire *Timer_Lib* décrit dans les fichiers *Operateurs.h* et *Fonctions.h*. Dans ce sens, dans le fichier TD1/src/main_td1a.cpp un test des fonctions développées est présenté. Il est à noter que la bibliothèque *Time_Lib* est utilisée lors de plusieurs exercices des autres TP du cours et pour cette raison elle n'est pas incluse dans le dossier TD1.

b) Timers avec callback

Dans cette partie, on nous a demandé d'implémenter un timer Posix périodique avec une fréquence de 2 Hz qui imprime un message avec la valeur d'un compteur régulièrement incrémenté, jusqu'à 15 incréments.

Ceci a été réalisé en implémentant une fonction de rappel appelée *myHandler* (vue ci-dessous), qui effectue l'incrément d'une variable également associée au *timer* et dont la référence est reçue comme paramètre de *myHandler*. De plus, pour que le compteur effectue un incrément périodique avec une fréquence de 2Hz, une valeur de 5e8 a été utilisée dans le paramètre *its.it_interval.tv_nsec* de *l'itimerspec* "its" et en laissant presque tous les autres paramètres à zéro sauf pour *its.it_value.tv_sec* dont la valeur était de deux, de sorte qu'un délai de 2 secondes était imposé avant le démarrage du compteur.

Dans le fichier TD1/src/main_td1b.cpp un test de la fonction callback et le *timer* développées est présenté.

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

c) Fonction simple consommant du CPU

Voici l'implémentation d'une fonction « incr » d'incrément unitaire d'une variable double pour la consommation du CPU. Dans ce cas et comme on peut le voir dans l'implémentation trouvée dans le fichier TD1/src/main_td1c.cpp, le paramètre « nLoops » pour la gestion du compteur est pris directement au moment de l'exécution du fichier binaire en utilisant la commande *std::stoi(argv[1])* pour non seulement prendre la valeur

mais aussi la passer comme un entier. On peut voir ici comment, en utilisant les fonctions développées dans la bibliothèque Time_Lib, il est possible de mesurer le temps d'exécution de la fonction bloquante d'incrément.

```
void incr(unsigned nLoops, double* pCounter)
{
    for(unsigned i=0; i<nLoops; i++) *(pCounter) += 1.0;
}
```

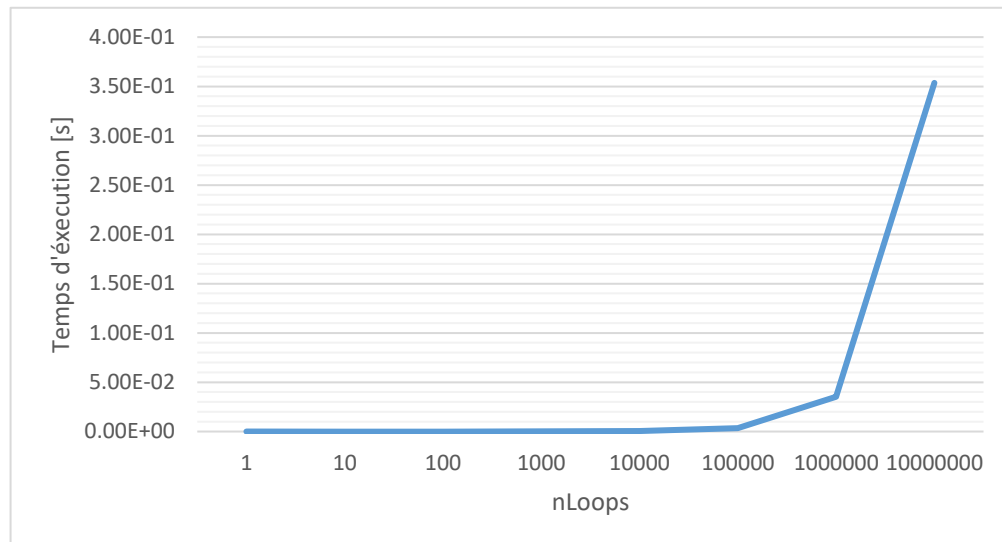


Figure 1. Temps d'exécution en variant nLoops

La figure 1 montre les temps d'exécution obtenus pour différentes valeurs de « nLoops » sur une échelle logarithmique.

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

Maintenant, la fonction de consommation CPU développée dans le paragraphe précédent est utilisée en définissant « nLoops » comme `UINT_MAX` incluant un paramètre « pStop » associé à un *timer* non périodique de temps variable qui permet d'arrêter le comptage, comme indiqué ci-dessous :

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

Pour modifier la valeur de la variable « pStop » associée au *timer*, on a utilisé la fonction *callback* indiquée ci-dessous, qui fixe la valeur de la variable à « true » une fois la minuterie expirée.

```
void myHandler(int, siginfo_t* si, void*)
{
    volatile bool& pStop = *((bool*) si->si_value.sival_ptr);
    pStop = true;
}
```

Enfin, une fonction de calibrage est mise en œuvre pour fixer les valeurs d' a et b en mesurant la valeur de « iLoop » (nombre total d'itérations effectuées avant l'arrêt du minuteur) pendant 4 secondes et pendant 6 secondes. Ainsi, une fonction affine $l(t)$, qui représente le nombre de boucles effectuées par la fonction « incr » pendant l'intervalle de temps t , est définie comme $l(t)=a\times t+b$.

```
linear calib()
{
    linear coeffs;
    double iLoops1 = timerT((time_t)4);
    double iLoops2 = timerT((time_t)6);

    coeffs.a = (iLoops2 - iLoops1)/2;
    coeffs.b = ((iLoops2 + iLoops1) - 10*(coeffs.a))/2;

    return coeffs;
}
```

Dans le fichier TD1/src/main_td1d_1.cpp et TD1/src/main_td1d.cpp un test de la fonction callback, laarrêt du compteur et la fonction « *calib()* » développées est présenté.

e) Amélioration des mesures

Dans la section C du TD3, une optimisation de la régression linéaire discutée dans la section précédente est mise en œuvre en utilisant un plus grand volume de données comme entrées dans la formulation.

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

Les fichiers correspondant à ce TD sont situés dans le fichier TD2/src. Le fichier README contient les instructions pour l'exécution de chacun des exercices mis en œuvre dans les fichiers main_td2x.cpp.

a) Exécution sur plusieurs tâches sans mutex

A ce stade, nous commençons par prendre en entrée les paramètres « nLoops » et « nTasks » qui désignent respectivement le nombre d'itérations à effectuer par la fonction « incr » ci-dessous et le nombre de *threads* à désigner pour ce calcul. Ci-dessous, il est possible également de voir l'implémentation d'une fonction « call_incr » qui associe la tâche d'incrémenter un compteur commun contenu dans « data » aux *threads*.

```
void* call_incr(void* data)
{
    Data* p_data = (Data*) data;
    incr(p_data->nLoops, (double*)p_data->pCounter);
    return data;
}

void incr(unsigned nLoops, double* pCounter)
{
    for(unsigned i=0; i < nLoops; i++)
        *(pCounter) += 1.0;
}
```

Normalement, on s'attendrait à ce que si 4 threads sont affectés à l'exécution d'un incrément de 1e6 au compteur, le résultat obtenu soit 4e6. Cependant, on observe que lors de l'exécution de cette expérience, la valeur obtenue est considérablement plus basse, spécifiquement 1.11955e+06, ceci est dû au manque d'atomicité de l'opération qui génère que simultanément les *threads* peuvent accéder à la même ressource et donc confondre la valeur actuelle du compteur avec sa valeur cible et donc finir par obtenir un sous-comptage.

Dans le fichier TD2/src/main_td2a.cpp un test de la fonction « call_incr » développée est présenté pour le calcul sans mutex d'une opération *multithread*.

b) Mesure de temps d'exécution

Le code implémenté dans le fichier TD2/src/main_td2b.cpp ajoute un paramètre de ligne de commande qui spécifie l'ordonnancement (SCHED_RR(2), SCHED_FIFO(1) ou SCHED_OTHER(0), ce dernier étant la valeur par défaut) ; le choix définit la valeur d'une variable « schedPolicy » qui est appliquée dans la fonction *pthread_setschedparam()*, pour définir les paramètres du thread.

```
struct sched_param schedParam;
schedParam.sched_priority = sched_get_priority_max(schedPolicy);
pthread_setschedparam(pthread_self(), schedPolicy, &schedParam);
```

Ensuite, en choisissant la politique d'ordonnancement SCHED_RR et en utilisant les fonctions de la bibliothèque Time_Lib développée dans le TD1, le temps d'exécution est mesuré pour les configurations suivantes :

$nLoops \in \{107, 2 \times 107, 3 \times 107, 4 \times 107\}$ et $nTasks \in \{1, 2, 3, 4, 5, 6\}$

Les résultats de cette expérience sont présentés ci-dessous :

nTasks	nLoops / temps d'exécution[s]			
	1.00E+07	2.00E+07	3.00E+07	4.00E+07
1	3.52E-01	7.02E-01	1.05E+00	1.40E+00
2	4.91E-01	9.83E-01	1.47E+00	1.96E+00
3	4.20E-01	8.30E-01	1.24E+00	1.66E+00
4	4.28E-01	8.43E-01	1.27E+00	1.69E+00
5	6.56E-01	1.21E+00	1.67E+00	2.16E+00
6	6.55E-01	1.28E+00	2.04E+00	2.75E+00

Tableau 1. Temps d'exécution - calcul sans mutex

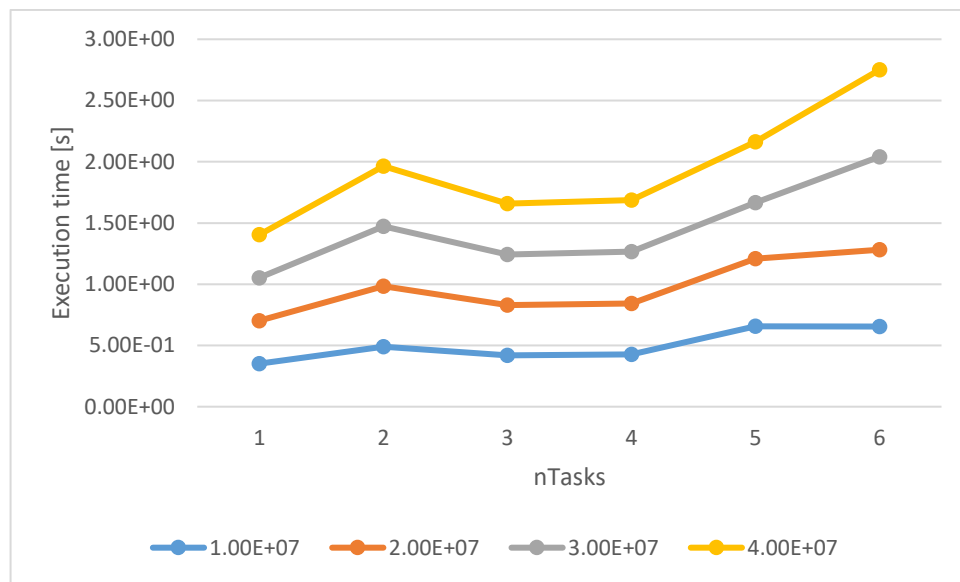


Figure 2. Temps d'exécution - calcul sans mutex

On peut voir que lorsque « nTasks » dépasse la valeur de 4, il y a une augmentation soutenue du temps d'exécution du programme, on peut donc conclure que la carte possède 4 processeurs, 1 *thread* chacun, ce qui est cohérent avec les spécifications de la carte Raspberry Pi 2B qui a été utilisée. En plus de cela, on peut observer que lors de l'implémentation de 2 *threads*, il y a une augmentation soutenue du temps d'exécution de l'algorithme, qui bien qu'elle soit inférieure à celle observée après 4, est significative par rapport aux cas où « nTasks » est 3 et 4.

c) Exécution sur plusieurs tâches avec mutex

Pour le cas trouvé dans le fichier main_td2c.cpp, on observe l'implémentation du processus réalisé dans le littéral a, à l'exception du fait qu'à travers le paramètre « protected » introduit dans la ligne de commande il est possible d'appliquer un mutex sur l'opération d'incrément à l'intérieur de la fonction *call_incr()* de façon à ce que la ressource soit protégée pendant qu'un des threads l'utilise et que le résultat attendu soit obtenu à la sortie

du compteur. C'est-à-dire que si l'on fixe un « nLoops » de 1e6 et un « nTasks » de 4, à la fin on obtiendra 4e6 comme sortie de l'opération.

```
void* call_incr(void* data)
{
    Data* p_data = (Data*) data;
    if(p_data->protect == 1)
    {
        pthread_mutex_lock(&p_data->mutex);
        incr(p_data->nLoops, (double*)p_data->pCounter);
        pthread_mutex_unlock(&p_data->mutex);
    }
    else
    {
        incr(p_data->nLoops, (double*)p_data->pCounter);
    }

    return data;
}
```

[TD-3] Classes pour la gestion du temps

Les fichiers correspondant à ce TD sont situés dans le fichier TD3/src. Le fichier README contient les instructions pour l'exécution de chacun des exercices mis en œuvre dans les fichiers main_td3x.cpp.

a) Classe Chrono

Initialement, dans le fichier main_td3a.cpp, on procède à l'implémentation des fonctions Chrono.h, qui contient une classe Chrono qui émule les fonctionnalités de mesure du temps d'un chronomètre. Dans ce sens, il dispose des fonctions *lap()*, *stop()* et *start()*, formées à partir des fonctions de la bibliothèque Timer_Lib réalisées lors du TD1, celles-ci permettent respectivement de prendre le temps contenu dans le chronomètre, de le compter et de le démarrer.

En outre, on dispose des fonctions *isActive()*, *startTime()* et *stopTime()* qui permettent de récupérer l'état du chronomètre et les temps absolus de démarrage et d'arrêt de celui-ci.

b) Classe Timer

Dans le fichier main_td3b.cpp, vous trouverez l'implémentation respective des classes contenues dans Time.h, PeriodicTimer.h et Countdown.h.

Dans cet ordre, la classe Time contient les fonctions d'implémentation d'un timer Posix, de sorte que le constructeur contient les définitions liées à *sigaction* et *sigevent*, ainsi qu'une fonction *start()* qui prend une durée en millisecondes comme argument pour configurer le paramètre « it_value » d'un « itimerspec », et une fonction *stop()* qui annule les paramètres d'un « itimerspec » avant de relancer le timer et donc de l'arrêter.

Ensuite, la classe PeriodicTimer est créée comme classe enfant de Timer qui permet de surcharger la méthode start() afin que le timer créé soit périodique et avec une période déterminée égale à la valeur initiale saisie.

Donc, la classe Countdown qui hérite de PeriodicTimer est implémentée, qui met en œuvre la fonction de rappel virtuelle définie dans la classe Timer pour effectuer un compte à rebours à partir d'une valeur donnée dans le constructeur de Countdown.

Enfin, il est important de mentionner que la fonction callback() est déclarée comme virtuelle (tout comme le destructeur de la classe) et protégée, et mise à zéro, dans la classe Timer car elle est destinée à être mise en œuvre par ses classes dérivées.

c) Calibration en temps d'une boucle

de mentionner que la fonction callback() est déclarée comme virtuelle (tout comme le destructeur de la classe) et protégée, et mise à zéro, dans la classe Timer car elle est destinée à être mise en œuvre par ses classes dérivées.

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

Les fichiers correspondant à ce TD sont situés dans le fichier TD4/src. Le fichier README contient les instructions pour l'exécution de chacun des exercices mis en œuvre dans les fichiers main_td4x.cpp.

- a) Classe Thread
- b) Classes Mutex et Mutex::Lock
- c) Classe Semaphore
- d) Classe Fifo multitâches

[TD-5] Inversion de priorité