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

Les deux bibliothèques <timer.h> et <signal.h> sont utilisées afin de réaliser ce TD. Le code associé peut être trouvé dans le répertoire TD1/src.

a) Gestion simplifiée du temps Posix

Dans cette section, on a produit des fonctions et opérateurs qui nous permettent d'utiliser plus simplement la structure de *timespec*, qui représente la mesure des temps dans l'API Posix. Ils sont définis dans *timespec.h* et implémentés dans *timespec.cpp*, qui sont aussi réutilisés dans les TDs suivants.

La structure de timespec est consitituée de deux parties : *tv_sec* qui représente les seconds du temps et *tv_nsec* qui représente la partie de nanoseconds. Il faut noter pendant la programmation que ce dernier est toujours positif.

b) Timers avec callback

Dans cette section, on a implémenté un timer Posix périodique de fréquence 2Hz dont le but est d'incrémenter régulièrement la valeur d'un compteur en l'imprimant. Le programme s'arrête après 15 incrémentations.

La définition d'un timer est constituée de deux parties : *it_value* qui indique le délai de la première période et *it_interval* qui représente l'intervalle entre deux périodes. À la fin de chaque période, la fonction *Myhandler* va être executée et le compteur qui vale intialement 0 augmente d'une unité à chaque appel du handler.

```
namespace td1b
{
void myHandler(int, siginfo_t* si, void*)
{
    int* p_counter = (int*)si->si_value.sival_ptr;
    *p_counter += 1;
    std::cout << "Counter: "<<*p_counter << std::endl;
}
}
```

c) Fonction simple consommant du CPU

Dans cette section on a implémenté une fonction *de signature* simple qui consomme du CPU *: void incr(unsigned int nLoops, double* pCounter).* Cette fonction effectue une boucle nLoops fois. Et dans chaque boucle, elle incrémente de 1.0 la valeur du compteur pointée par pCounter.

La signature standard du point d'entrée d'un programme est : int main(int argc, char* argv[]) où le paramètre argc indique le nombre de chaines de caractères de la ligne de commande et argv est le tableau où on sauvegarde l'ensemble de ces chaines.

La fonction timespec_now() implémentée dans *timespec.cpp* qui concerne la fonction *clock_gettime* est utilisée pour mesurer et afficher la valeur du temps d'exécution à l'ecran.

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

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

Dans cette section on a modifié la fonction *incr* en lui rajoutant le paramètre *bool* pStop* qui sera initialisé à false en permettant l'incrément du compteur jusqu'au moment où il devient true.

pStop est déclaré volatile pour que sa valeur puisse être modifiée par un callback. À un instant donné, *pStop* va devenir true et la boucle de la fonction incr va être arrêtée.

Soit *l(t)* le nombre de boucles effectuées par la fonction *incr* durant l'intervalle de temps *t,* on suppose que ⁋ cette fonction est affine : $l(t)=a\times t+b$. On a donc construit une fonction *calib* pour calculer les valeurs a (pente) et b(constante).

Pour vérifier si la calibration est correcte, on fait une comparaison entre le temps d ' exécution réel et le temps calculé par le nombre d'itérations.

e) Amélioration des mesures

Pour améliorer la précision, on peut effectuer plus de mesures, puis effectuer une régression linéaire. En théorie, la performance de régression linéaire est meilleure que TD1d mais on n'a pas obtenu le même résultat.

```
long double sum_x = 0;
long double sum_xx = 0;
long double sum_y = 0;
long double sum_xy = 0;

for (int i= 0; i< numCalibs; i++)
{
        sum_x += t[i];
        sum_y += r[i];
        sum_xx += t[i]*t[i];
        sum_xy += r[i]*t[i]; // x*y
}

a = (double) (numCalibs * sum_xy - sum_x * sum_y) / (double)(numCalibs * sum_xx - sum_x * sum_x);
b = (double) sum_y / numCalibs - a * sum_x / (double)numCalibs;
```

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

Dans cet exercice, on a utilisé l'API multitâche pthread. Le code associé peut être trouvé dans le répertoire TD2/src.

a) Exécution sur plusieurs tâches sans mutex

Dans cette section, on a essayé de créer des threads afin de réaliser une variable compteur qui incrémente. Plusieurs threads sont utilisés.

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

En théorie, si on démarre 50 threads et que chaque thread effectue 1000 itérationss , une valeur de compteur de 50, 000 peut être obetenu mais ce n'est pas le cas. La valeur obtenue est toujours inférieure à la valeur attendue.

En fait, les threads modifient la valeur du compteur en même temps, ce qui peut entraîner des problèmes liés à l'accès parallèle aux données par plusieurs processus (problème d ' écritures concurrentes). Le *MUTEX va* être utilisé dans les sections suivantes pour résoudre ce type de problème.

b) Mesure de temps d'exécution

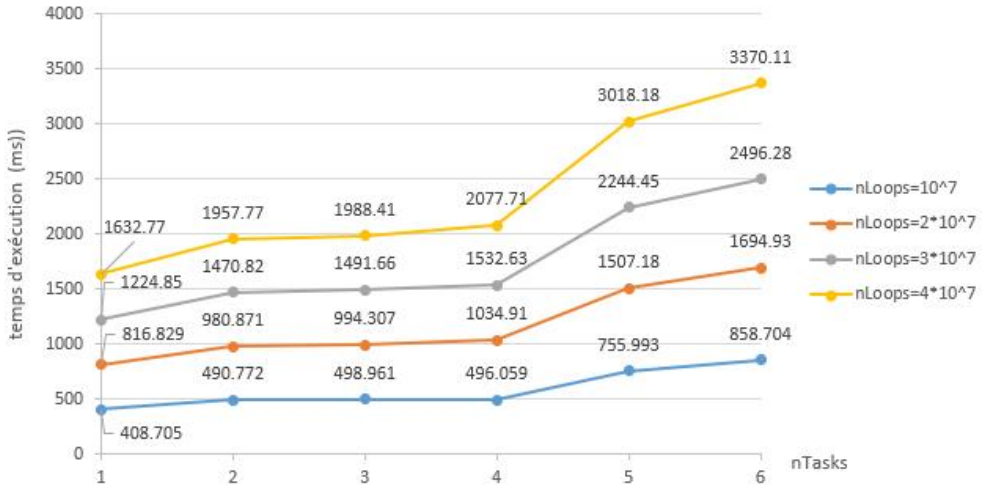
Dans cette section, on a rajouté un paramètre en ligne de commande pour spécifier et fixer le politique d'ordonnancement (*schedPolicy*).

La priorité maximale est donnée à la fonction *main* et une priorité inférieure est donnée. Pour la politique SCHED_OTHER, la priorité 0 va être donnée et pour les deux autres politiques, une valeur aleatoire entre 0 et 98 est donnée.

On mesure le temps d'exécution pour différentes valeurs de nLoops et de nTasks . L'évolution du temps d'exécution en fonction de nLoops (abscisse) et de nTasks (couleur de la courbe) sur le Raspberry PI est présenté dans la figure ci-dessous.

Temps d'exécution avec l'ordonnacement SCHED_RR(ms)

nTasks\nLoops	10^7	2×10^7	3×10^7	4×10^7
1	408.705	816.829	1224.85	1632.77
2	490.772	980.871	1470.82	1957.77
3	498.961	994.307	1491.66	1988.41
4	496.059	1034.91	1532.63	2077.71
5	755.993	1507.18	2244.45	3018.18
6	858.704	1694.93	2496.28	3370.11



c) Exécution sur plusieurs tâches avec mutex

Dans cette section, on a ajouté un autre argument de protection pour résoudre le problème existant dans la section TD2a. La fonction *pthread_mutex_lock* et *pthread_mutex_unlock* sont utlisées.

```
void* call_incr(void* v_data)
{
   Data* p_data = (Data*) v_data;
   if(p_data->protection)
   {
        pthread_mutex_lock(&p_data->mutex);
        incr(p_data->nLoops, &p_data->counter);
        pthread_mutex_unlock(&p_data->mutex);
   }
   else incr(p_data->nLoops, &p_data->counter);
   return v_data;
}
```

On peut constater que la valeur finale du compteur est égale à nLoops * nTasks avec la protection d'un mutex, qui signifie que le problème est bien résolu. On peut constater aussi que le temps d'exécution avec un mutex est plus important que celui sans mutex.

[TD-3] Classes pour la gestion du temps

a) Classe Chrono

Dans cette section, on a implémenté la classe Chrono qui implémente les fonctionnalités de mesures de temps d'un chronomètre. A l'aide de cette classe, on peut définir le temps de début, celui de fin et la durée entre le démarrage et la fin.

On teste cette classe dans le fichier *main_td3a.cpp* et une comparaison entre la valeur du temps donné par *Chrono* et par *timespec_now* est faite pour vérifier la fonctionnalité de cette classe.

b) Classe Timer

Dans cette section, on a implémenté une classe *Timer* qui encapsule les fonctionnalités d'un timer Posix.

Dans cette classe, le constructeur, le destructeur, la méthode start() et la méthode stop() sont publiques car ils doivent être appelés dans le main (à l'extérieur de la classe). L'opération *callback()* est protégée car elle est virtuelle et elle va être implémentée par une classe fille que hérite d'elle. Il ne faut pas l'utiliser en dehors de Timer ou les

classes héritées. La méthode de *call_callback()* est privée car elle est utilisée seulement dans la classe *Timer.* Elle est définie static car elle peut être utilisée sans instancier un objet de cette classe.

La classe *CountDown*, qui hérite de *PeriodicTimer* héritant de *TImer* est créée pour imprimer à l'écran un compte à rebours à 1 Hz depuis un nombre n jusqu'à 0.

c) Calibration en temps d'une boucle

Dans cette section, basé sur TD1e, on a créé une classe *Calibrator* qui uilise la méthode de regression linaire, une classe *Looper* qui démarre une boucle et augmente le compteur et une classe *CpuLoop* héritant de *Looper* où il y a un pointeur vers l'objet Calibrator. Cela nous permet d'initialiser un objet CpuLoop qui va faire des boucles en ayant le comportement de Calibrator.

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

Dans TD4, on a encapsulé la gestion des tâches Posix dans les classes *PosixThread*, Thread, Mutex et Lock.

a) Classe Thread

Dans cette section, on a tout d'abord implémenté une classe *PosixThread* qui contient les paramères basiques dont un thread a besoin, incluant l'ordonnancement et la propriété spécifiée à la tâche. La classe *Thread* dérivant de *PosixThread* est ensuite implémentée afin de réaliser la mesure du temps. Afin de tester ces deux classes mentionnées ci-dessus, on a créé une autre classe *IncrThread*, dont la méthode *run* a la même fonctionnalité comme ce que l'on a déjà fait dans le TD-2a : un compteur est incrémenté pour un certain nombre de boucles.

On teste ces classes dans main et on trouve le même problème que celui de TD-2a. Si on démarre 50 threads et que chaque thread effectue 1000 itérations, une valeur de compteur inférieure de 50, 000 (50,000 est la valeur attendue). Afin de résoudre ce problème, *MUTEX va* être utilisé dans les sections suivantes.

b) Classes Mutex et Mutex::Lock

Dans cette section, on a implémenté les classes qui s'occupent de la gestion de Mutex : *Mutex, Mutex::Monitor, Mutex::Lock* et *Mutex::TryLock. Mutex::Lock* et *Mutex::TryLock* sont utilisées pour obtenir et libérer un mutex, *Mutex::Monitor* est utilisé pour surveiller si un thread a l'accès à un mutex, sinon le thread va attendre pour un temps prédéfini ou jusqu'à l'obtention d'un signal.

On a testé ces classes dans main en utilisant une classe *IncrMutex*, qui est similaire à la classe *IncrThread* utilisée dans TD-4a. Le pointeur vers un Mutex permet de protéger l'accès aux données : on démarre 50 threads avec chaque thread roulant 1000 boucles, la valeur du compteur obtenue est justement 50,000 comme prévu. Le temps d'execution est un peu plus long que le cas de TD-4a.

c) Classe Semaphore

Dans cette section, on a implémenté une classe *Semaphore* qui initialise une boîte à jetons. On utilise la méthode *take* pour lui retire un jeton et la méthode *give* nous permet de lui rajouter un jeton.

En utilisant les classes *SemaConsumer* et *SemaProducer* qui ont un pointeur respectivement vers le même objet de *Semaphore*, on peut créer diffèrent threads, avec les producteurs produisant les jetons, c'est à dire d'augmenter le numéro de jetons dans boîte. et avec les consommateurs consumant les jetons, c'est à dire de le diminuer.

Dans *main_td4d* on a vérifié que le nombre de jetons créés sont effectivement consommés même si le nombre de consommateurs est différent que celui de producteurs.

d) Classe Fifo multitâches

Dans cette section, on a créé une classe template *Fifo* en utilisant le conteneur C++ *std::queue*. La logique de First-In-First-Out est satisfaite dans cette classe, les éléments sont ajoutés à la fin de la queue par la méthode *push* et éliminés à la tête par la méthode *pop*.

Les classes associées *FifoConsumer* et *FifoProducer* sont ensuite créées afin de tester le template *Fifo*. *FifoProducer* produit une série d'entiers entre 0 et une valeur n que *FifoConsumer* va consommer. Comme le résultat de TD-4c, le nombre des éléments créés est égal au nombre d'éléments détruits même si le nombre de consommateurs et celui de producteurs sont différents.

[TD-5] Inversion de priorité

Dans cette section, on a utilisé ce que l'on a créé dans TD-3 et Td-4 afin de réaliser l'inversion de priorité. L'inversion de priorité signifie que la tache posant le mutex a toujours la priorité maximale parmi tous les tâches qui demandent le blocage du mutex. Autrement dit, lorsque la tâche A avec une priorité inférieure empêche le mutex, même si une autre tâche B avec une priorité plus élevée arrive, A a toujours le mutex, et la priorité de la tâche A deviendra la priorité de B.

On a ajouté une option protection (une variable booléenne *isInversionSafe*) contre l'inversion de priorité dans le constructeur de la classe *Mutex*, qui nous permet d' activer et de désactiver l'inversion.

Étant donné que le processeur est multicore, on a besoin d'occuper tous les autres cœurs à 100% sauf l'un qu'on va utilisé. A l'aide de *CPU Affiniy*, on a réussi à exécuter notre code en un seul coeur.

```
// Work on a single CPU
cpu_set_t  cpuset;
CPU_ZERO(&cpuset);
CPU_SET(0, &cpuset);
sched_setaffinity(0, sizeof(cpu_set_t), &cpuset);}
```

On a implémenté notre code en fonction de la page 23 de diapositive 2, dont il s'agit des caractères que l'on doit fournir aux threads, comme le temps, le temps attendu avant d'obtenir un mutex, etc.

Ticks taken by thread A				
	PC		**Raspberry PI**	
order	***isInversionSafe* = true**	***isInversionSafe* = false**	***isInversionSafe* = true**	***isInversionSafe* = false**
1	136.38	111.042	914.219	842.968
2	111.119	110.559	853.594	906.927
3	121.833	73.991	836.198	806.041
4	132.259	110.703	828.75	848.802

5	112.092	110.586	808.229	841.563
6	111.207	110.441	901.459	905.417
7	110.101	109.719	889.844	837.552
8	114.422	128.156	840.313	864.636
9	112.351	110.256	840.937	820.989
10	128.536	126.475	836.25	850
11	111.511	99.578	872.24	849.271
12	131.681	110.861	942.187	784.688
13	100.22	118.567	836.459	831.771
14	111.662	118.797	880.364	861.197
15	109.98	110.68	806.771	894.792
16	102.522	110.222	895.989	889.115
17	106.792	103.864	848.698	828.229
18	109.589	131.889	833.698	911.302
19	111.785	112.196	856.198	829.063
20	102.52	111.01	851.042	894.896
Average	114.4281	111.4796	856.0157	852.8571