Rapport ENSTA 2020 – ROB305 – Juliette DRUPT

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

L'ensemble du code écrit dans le cadre de ce premier TD se trouve dans le répertoire TD1/src.

a) Gestion simplifiée du temps Posix

L'ensemble des fonctions écrites dans le cadre de cet exercice sont déclarées dans un fichier TimeSpec.h et implémentées dans un fichier TimeSpec.cpp. Ces fichiers se trouvent dans le répertoire TD1/src, comme les scripts du TD1. Comme ces fonctions sont réutilisées dans les autres TD, ces 2 fichiers ont été dupliqués et sont également présents dans les répertoires TD2/src, TD3/src, TD4/src et TD5/src, ceci pour faciliter les imports et les instructions de compilation. Ces fichiers sont toutefois tous identiques.

Le script td1a-main.cpp de TD1/src permet de tester l'ensemble des fonctions implémentées dans cet exercice.

b) Timers avec callback

Le timer Posix périodique de fréquence 2 Hz est implémenté dans le script td1b-main.cpp de TD1/src et appelle le handler suivant, qui affiche un message de type « coucou i », i augmentant d'une unité à chaque appel du handler et valant initialement 0. Le programme s'arrête quand après 15 incrémentations.

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

c) Fonction simple consommant du CPU

La fonction incr implémentée est la suivante :

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

Le code demandé figure dans le script td1c-main.cpp de TD1/src. On note qu'à la fin de l'exécution, la valeur du compteur est égale au nombre de boucles demandé par l'utilisateur, ce qui est normal puisqu'on effectue bien nLoops fois l'action d'incrémenter le compteur.

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

On modifie la fonction incr afin d'avoir une variable iLoop donnant l'indice de la boucle actuelle. La fonction devient donc :

```
unsigned int incr(unsigned int nLoops, volatile double* pCounter){
	unsigned int iLoop = 0;
	while (iLoop < nLoops && not *pStop)
	{
		*pCounter += 1.0;
		iLoop++;
	}
	return iLoop;}
```

Afin que la valeur de pStop puisse être modifiée par un callback à un instant donné et provoque l'arrêt de la boucle de la fonction incr, pStop doit être déclaré volatile. Pour vérifier que la calibration est correcte, il suffit de calculer le temps correspondant à un certain nombre d'itérations choisi, puis de le comparer au temps d'exécution réel.

e) Amélioration des mesures

Pour améliorer la précision, on pourrait utiliser non pas deux mesures seulement mais un nombre plus important de mesures, puis effectuer une régression linéaire. C'est d'ailleurs ce qu'on fera par la suite dans la classe Calibrator dans le TD3.

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

L'ensemble du code écrit dans le cadre de ce deuxième TD se trouve dans le répertoire TD2/src.

a) Exécution sur plusieurs tâches sans mutex

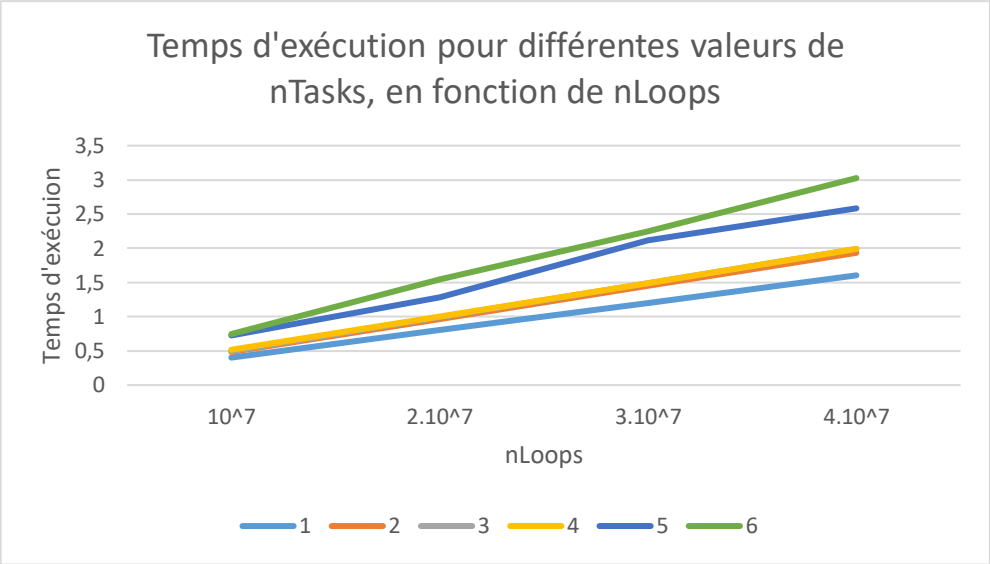
On utilise la fonction call_incr suivante :

```
void * call_incr(void* v_data)
{
	Data* p_data = (Data*) v_data;
	incr(p_data -> nLoops, &p_data -> counter);
	return v_data;
}
```

Le programme complet se trouve dans le script td2a-main.cpp de TD2/src. On constate que la valeur finale du compteur est un peu plus élevée que ce qu'elle aurait dû être lorsqu'on utilise plusieurs threads : en effet, les threads modifient en même temps la valeur du compteur, donc celle-ci n'est pas vraiment contrôlable (problème d'écritures concurrentes).

b) Mesure de temps d'exécution

On trouvera le programme complet dans le script td2b-main.cpp de TD2/src. On mesure le temps d'exécution pour différentes valeurs de nLoops et de nTasks. Le graphique suivant montre l'évolution du temps d'exécution en fonction de nLoops (abscisse) et de nTasks (couleur de la courbe). On constate une nette augmentation des temps de calcul pour nTasks > 4. On peut donc en conclure que le Raspberry PI utilisée a une architecture 4 cœurs, ce qui se vérifie bien dans la documentation de la Raspberry PI 2.



c) Exécution sur plusieurs tâches avec mutex

On ajoute une possibilité de protection par un mutex. Le code correspondant se trouve dans le script td2c-main.cpp de TD2/src. Cette fois, on constate que la valeur finale du compteur est égale à nLoops * nTasks lorsqu'on protège par un mutex. On constate également que dans ce cas le temps d'exécution est plus important.

[TD-3] Classes pour la gestion du temps

L'ensemble du code écrit dans le cadre de ce premier TD se trouve dans le répertoire TD3/src.

a) Classe Chrono

Le script td3a-main.cpp de TD3/src permet de tester les fonctionnalités de la classe Chrono.

b) Classe Timer

La méthode call_callback est privée car elle n'a aucune raison d'être appelée en dehors de la classe Timer. La méthode callback est protégée car comme elle est virtuelle il ne faut pas qu'elle puisse être appelée par « erreur » en dehors de Timer et de ses classes héritière, mais doit pouvoir être implémentée par les classes héritières de Timer. Les autres méthodes sont publiques puisqu'elles servent justement à contrôler de Timer et doivent donc pouvoir être appelées dans un script quelconque. La méthode de classe statique call_callback permet de spécifier le cadre de l'appel de la méthode callback, qui est dépendra des classes héritières de Timer. C'est pour cela que callback est virtuelle. Les classes Timer et PeriodicTimer sont testées grâce à la classe CountDown, qui hérite de PeriodicTimer. Le script correspondant est td3b-main.cpp, dans TD3/src.

c) Calibration en temps d'une boucle

Le programme td3c-main.cpp de TD3/src permet de tester les classes de l'exercice.

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

L'ensemble du code écrit dans le cadre de ce premier TD se trouve dans le répertoire TD4/src.

a) Classe Thread

Le programme td4a-main.cpp de TD4/src reproduit le TD2a avec une architecture implémentée objet et permet ainsi de tester les classes Thread et PosixThread au travers d'une classe héritière de Thread : IncrementThread, qui incrémente un compteur de manière similaire à ce qu'on a vu dans le TD2a. On se réfèrera au code et à ses commentaires pour plus de détails.

b) Classes Mutex et Mutex::Lock

Le programme td4b-main.cpp de TD4/src permet de tester la classe Mutex en utilisant la classe IncrementThreadWithMutex, qui est une classe héritière de la classe IncrementThread utilisant un objet Mutex. On se réfèrera au code et à ses commentaires pour plus de détails.

c) Classe Semaphore

Le programme td4c-main.cpp de TD4/src teste la classe Semaphore en utilisant deux classes héritières de le classe Thread : SemConsumerThread et SemProducerThread. Ces classes ont pour attribut une référence vers un Semaphore. La méthode run() de SemConsumerThread consiste à prendre un jeton à ce Semaphore, et celle de SemProducerThread à lui en donner. Lors de l'exécution de run(), le message « Giving ! », respectivement « Taking ! ». On teste ici avec 5 instances de chaque, qui utilisent le même Semaphore. On s'assure bien que tous les jetons produits ont été consommés en comparant les nombres de « Giving !» et de « Taking ! » qui s'affichent. Ce scénario reste possible puisqu'on a effectué des tests avec peu de tâches. On se réfèrera au code et à ses commentaires pour plus de détails.

d) Classe Fifo multitâche

Le programme td4d-main.cpp teste le template Fifo de manière similaire à ce qu'on a pu faire avec le Semaphore. On utilise ainsi les classes FifoConsumerThread et FifoProducerThread. FifoProducerThread produit une série d'entiers entre 0 et une valeur n (qu'on a pris égale à 5 dans le test) et les envoie dans la Fifo, tandis que FifoConsumerThread consomme n entiers de la Fifo. Les entiers consommés s'affichent lors de l'exécution, ce qui permet de vérifier en un coup d'œil qu'ils ont bien tous été produits et consommés correctement. On se réfèrera au code et à ses commentaires pour plus de détails.

[TD-5] Inversion de priorité

L'ensemble des sources utilisées pour ce TD ont été dupliquées et placées dans le dossier TD5/src afin de faciliter les imports et les instructions de compilation. Seule la classe Mutex est différente dans TD5/src par rapport à TD4/src, puisqu'on lui ajoute le paramètre déterminant la protection ou non contre l'inversion de priorité. Le programme td5-main.cpp teste le cas de figure présenté dans le cours. On a converti les tics d'horloges en temps en connaissant le nombre de tics par seconde. Les tâches utilisées exécutent des Loopers calibrés, ce qui permet de choisir le temps d'exécution de chaque tâche : ce temps est converti en nombre de boucles et un Looper est lancé. Pour ce faire on a créé une tâche ThreadWithCpuLoop qui hérite de Thread et possède un attribut CpuLoop. Les temps d'exécutions réels des tâches A, B et C sont affichés à la fin de l'exécution, en nombre de tics d'horloge. On se réfèrera au code et à ses commentaires pour plus de détails.

Remarques :

- il faut modifier une ligne de code pour choisir si on veut utiliser la protection contre l'inversion de priorité ou non – celle-ci est indiquée dans les commentaires dans les premières lignes du script.
- Comme le CPU est quadricore, on utilise la « CPU affinity » afin qu'un seul CPU soit utilisé.

On constate que les temps d'exécutions réels des 3 tâches varient beaucoup d'une exécution sur l'autre. On a donc effectué 20 mesures pour chaque cas (avec ou sans protection). Ce code a été testé sur PC, puis sur la Raspberry PI. Toutefois, il semble y avoir un souci avec l'exécution sur la Raspberry puisque les temps d'exécutions en tics d'horloge sont beaucoup trop élevés par rapport à ce qu'ils devraient être, comme on le voit dans le tableau ci-dessous. Cela pourrait venir d'une différence de fréquence d'horloge de la Raspberry, qui fausserait les conversions en nombre de tics.

Les résultats sont les suivants :

PC		Raspberry PI	
isInversionSafe == false	isInversionSafe == true	isInversionSafe == false	isInversionSafe == true
59,736	49,502	1174,22	1294,01
54,312	36,348	2554,17	1372,71
50,936	36,924	2557,81	1141,71
37,362	49,942	3397,03	1582,76
37,173	47,498	3209,38	1420,76
49,952	34,92	3446,2	3276,46
49,714	37,413	3039,38	3977,34
51,797	37,78	3703,65	2861,98
50,782	37,074	3567,55	3264,43
49,629	48,793	3910,16	4116,93
49,292	49,677	2959,69	3367,34
48,864	49,351	3297,14	3707,34
36,662	37,036	3152,4	3098,85
36,194	37,397	3260,05	3148,64
35,919	37,555	2755,16	3208,96
36,615	48,947	2733,28	3005,52
48,284	36,429	3320,62	3204,58
37,539	36,288	3286,77	3225,58
35,617	49,353	3493,7	2519,06
36,271	36,52	3309,06	3708,85
moyenne : 44,6325	moyenne : 41,73735	moyenne : 3106,371	moyenne : 2825,1905