Rapport ENSTA 2022 – ROB305 – Henrique MORAU RODRIGUES

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

Les codes associés à réalisation de ce TD sont présents dans le dossier TD1/src, et les fichiers main_td1X.cpp sont aussi dans le dossier TD1, soit le X pour les items a, b, c, d et e.

a) Gestion simplifiée du temps Posix

Pour cette question, les fonctions et opérateurs ont été créés de façon a simplifier l'utilisation des structures *timespec,* qui réprésent la mesure de temps dans l'API Posix, en utilisant la bibliothèque *<timer.h>*. Les codes sont présents dans les fichiers TimerLib.cpp et TimerLib.h du dossier TimerLib, qui seront aussi présents dans les prochaines questions et TDs.

La structure *timespec* est composé pas les valeurs *tv_sec* et *tv_nsec*, correspondant aux valeurs respectivement en secondes et en nanosecondes, avec une convention adoptée que les valeurs en nanosecondes sont toujours positives ou égale à zéro.

La fonction main présente dans le fichier *main_td1a.cpp* réalise un test avec chacun des méthodes et opérateurs crées dans le développement de cette question du TD.

b) Timers avec callback

Le but de cette question est l'implémentation d'un timer Posix périodique avec une fréquence égale à 2 Hz, qui incrément un compteur 15 fois et affiche sa valeur.

L'implémentation est réalisée avec la déclaration d'une fonction *myHandler* (présenté ci-dessous), qui dans la définition du Timer, est exécuté à chaque fin de la période déterminée. La définition du Timer a deux parties principales, qui sont les valeurs données aux variables *it_value* et *it_interval*. La première détermine le *delay* pour la première réalisation et la deuxième, détermine l'intervalle entre deux périodes. Donc, pour le cas de la question, *it_value.tv_sec* a été égale à 2 (pour attend 2 secondes avant de commencer à incrémenter) et *it_interval.tv_nsec* égale à 5e8 (pour correspondre à une fréquence de 2Hz), avec les autres paramètres égales à zéro.

Le fichier *main_td1b.cpp* présent la déclaration de la fonction *myHandler()*, ainsi que la fonction *main()*, où les définitions ont été réalisés.

```
void myHandler(int, siginfo_t* si, void*)
{
    volatile int& count_ref = *((int*) si→si_value.sival_ptr);
    count_ref += 1;
}
```

c) Fonction simple consommant du CPU

Une fonction simple, présente ci-dessous, que consomme du CPU a été code comme *void incr(unsigned int nLoops, double* pCounter)*, avec l'objectif de réaliser *nLoops* fois la boucle, qui incrément un compteur pointé par *pCounter*, à récupérer dans le *main()*.

La fonction main() a été construit avec la signature standard de passage de paramètres come argument au binaire pour exécution, donc, écrit comme *int main(int argc, char* argv[])*. Cette structure permet la vérification du nombre d'argument inséré, avec la variable *argc* et récupérer les arguments avec le vecteur *argv[]*, où le premier argument est indexé comme 1. Pour la prise de valeur dans le code, la fonction *atoi(argv[1])* a été utilisé comme forme de conversion de l'argument pour la variable du type *int*.

Comme forme de mesurer le temps, le fichier *TimerLib.h* (crée dans la question TD1-a) a été linké, donc, la fonction *timespec_now()*, qui fait appel à la fonction *clock_gettime()* est utilisé pour le marquage du temps et après, affichage du temps dans le terminal en seconds avec une petite conversion de la sortie de la fonction *timespec_to_ms()*, aussi de *TimerLib.h*.

Les codes ont été écrits dans le fichier *main_td1c.cpp*, où sont présentes la fonction *incr()* et la fonction *main()*.

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

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

Pour cette question, la fonction *incr()* précèdent a été modifié en l'ajoutant un paramètre *bool* pStop* initialisé a *false*, que permet l'incrémentation du compteur jusqu'au moment où *pStop* passe à *true*. Comme la valeur de cette variable peut être modifiable par une fonction *myHandler()* a un instant donné, *pStop* a été déclaré comme volatile. La fonction *myHandler()* est présent ci-dessous :

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

Une fonction *setIncrement(time_t sec)* a été codé, où sont initialisés les paramètres comme le *pStop* a false, le *counter* a *zero* et le *nLoops* (nombre de boucles) a *UINT_MAX*, représentant le nombre maximal possible, pour que l'incrément du compteur puisse être limité que pour la variable *pStop* dans le moment *time_t sec* défini dans le *timer* POSIX (*its.it_value.tv_sec = sec*).

Pour fin, une fonction *calib()* a été déclaré pour la calibration des coefficients d'une fonction affine *I(t) = a.x + b* en utilisant la fonction *setIncrement(time_t sec)* avec différentes valeurs pour sec, suivant d'une approximation directe comme présent dans la partie du code ci-dessous, où params est une struct avec deux doubles correspondantes à a et b.

```
 coef calib()
{
    coef params;
    double iLoop_4sec = setIncrement((time_t) 4);
    double iLoop_6sec = setIncrement((time_t) 6);
    params.a = (iLoop_6sec - iLoop_4sec)/(6 - 4);
    params.b = (iLoop_6sec - params.a*6);
    return params;
}
```

Les codes ont été écrits dans le fichier *main_td1d.cpp*, où sont présentes les fonctions citées et la fonction *main()*, où la performance de la détermination des coefficients a été testé en comparant le nombre de boucles utilisés pour une appel à la fonction *setIncrement()* et ce obtenu à partir de la fonction affine.

e) Amélioration des mesures

Comme forme d'amélioration des mesures, il a été développé une façon de moyenne des coefficients successivement calculées, mais à la fin, l'amélioration n'était pas significative.

Pour la section 3c du TD-3, cette approche de formulation de fonction est retournée, mais comme dans ce cas, on compte avec plusieurs pairs x et y pour la détermination des coefficientes de l'équation, la régression linéaire a été utilisé.

Le code avec l'amélioration est présent dans le fichier *main_td1e.cpp*.

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

Les codes associés à réalisation de ce TD sont présents dans le dossier TD2/src, et les fichiers main_td2X.cpp sont aussi dans le dossier TD2, soit le X pour les items a, b et c.

a) Exécution sur plusieurs tâches sans mutex

Pour cette question, une fonction *void* call_incr(void* thread_arg)* présent ci-dessous a été codée et fait appel à la fonction *void incr()* présenté dans le TD-1. Comme argument, la struct *thread_arg* correspond à un *int nLoops* pour le comptage de boucles et un volatile double *pCounter* pour la valeur du compteur. Cette fonction, ainsi que l'argument est utilisé dans la fonction *pthread_create()* pour le lancement de les *nTasks* threads (où nTasks est passé comme argument au binaire), suivant de l'appel à fonction *pthread_join()*.

```
void* call_incr(void* thread_arg)
{
    threadArg* pThread_arg = (threadArg*) thread_arg;
    incr(pThread_arg -> nLoops, (double*) &pThread_arg->pCounter);
    return thread_arg;
}
```

Les codes sont écrits dans le fichier *main_td2a.cpp.* Il est possible constater que la valeur finale trouvée pour le compteur est inférieure à la réponse correcte. Par exemple, si on a 5 comme nTasks et 1000000 boucles, on attend un compteur égal à 5000000, et en fait, on obtient une valeur inférieure parce que les threads modifient la valeur du compteur simultanément.

b) Mesure de temps d'exécution

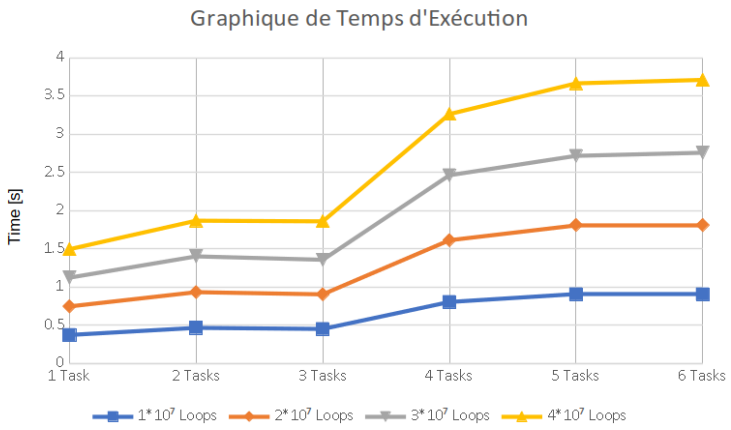
Une modification dans le code a été faite pour permet le passage d'un argument dans la ligne de command (comme *argv[3]* dans ce cas) que spécifie la politique d'ordonnancement, entre SCHED_RR, SCHED_FIFO ou SCHED_OTHER. Si la politique est SCHED_OTHER, la priorité donnée est 0, et pour toutes les politiques, la priorité du *main* est la maximale, garanti pour les lignes de code suivantes :

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

Le programme, présente dans le fichier *main_td2b.cpp* a été exécuté plusieurs fois avec la politique SCHED_RR, en variante le paramètre *nLoops* ($1 \times 10^7, 2 \times 10^7, 3 \times 10^7, 4 \times 10^7$) et le paramètre déterminant le nombre de tâche *nTask* (1, 2, 3, 4, 5, 6), toujours en mesurant le temps d'exécution avec les fonctions crées dans le TD-1 importé comme *TimerLib.h*. L'évolution des valeurs peut être observé dans le tableau ci-dessous, et plus visiblement dans le graphique qui suive.

nTasks	1*10⁷ loops	2*10⁷ loops	3*10⁷ loops	4*10⁷ loops
1	0.375169	0.748555	1.12514	1.49689
2	0.468215	0.935417	1.40312	1.86981
3	0.453548	0.906596	1.35869	1.86135
4	0.806904	1.61416	2.46406	3.26376
5	0.907823	1.81358	2.71774	3.66615
6	0.906885	1.81201	2.75885	3.71162

Il est possible constater un changement dans le temps d'exécution quand on passe de 3 à 4 Tasks, que permet de conclure que l'architecture processeur de la Raspberry utilisé compte avec 3 threads actives.



Selon la spécification, la Raspberry 2 Model B compte avec 4 processeurs (il n'est pas présent des threads virtuels, donc 1 thread pour chaque processeur et un totale de 4 threads). Alors, après une investigation plus détaillée avec les commandes que montre les processeurs actifs (présent ci-dessous), il a été observé que en fait cette cible compte avec 3 processeurs actifs, donc, 3 threads.

Ensuit il est possible confirmer l'observation obtenue à partir du tableau et du graphique, d'où l'évolution en temps d'exécution montre la présence de 3 threads dans la cible.

```
rob305-pi51# nproc
3
rob305-pi51# cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
[...]
processor       : 1
[...]
processor       : 2
[...]
[...]
Model           : Raspberry Pi 2 Model B Rev 1.1
```

c) Exécution sur plusieurs tâches avec mutex

Le code présenté dans la question a de ce TD a été modifié pour l'addition d'un troisième argument qui détermine si l'incrément du compteur sera fait avec l'utilisation de Mutex ou non, avec les fonctions *pthread_mutex_lock()* et *pthread_mutex_unlock()*. La struct *threadArg* a été aussi modifié pour le passage d'une *pthread_mutex_t* et un *bool* qui détermine si l'exécution est *protected* ou non. Le code de la fonction *call_incr()* modifié est présent ci-dessous :

```
void* call_incr(void* thread_arg)
{
    threadArg* pThread_arg = (threadArg*) thread_arg;
    if (pThread_arg->isProtected)
    {
        pthread_mutex_lock(&pThread_arg->mutex);
        incr(pThread_arg->nLoops, (double*) &pThread_arg->pCounter);
        pthread_mutex_unlock(&pThread_arg->mutex);
    } else {
        incr(pThread_arg->nLoops, (double*) &pThread_arg->pCounter);
    }
    return thread_arg;
}
```

La solution est présente dans le fichier *main_td2c.cpp*. Après l'exécution plusieurs fois, il est possible constater qu'avec le Mutex, le compter présente la valeur attendu (*nTasks*nLoop*) et aussi, que le temps d'exécution est plus grand quand comparé à l'exécution sans l'argument *protected*, où le Mutex n'est pas utilisé.

[TD-3] Classes pour la gestion du temps

Les codes associés à réalisation de ce TD sont présents dans le dossier TD3/src, et les fichiers main_td3X.cpp sont aussi dans le dossier TD3, soit le X pour les items a, b et c.

a) Classe Chrono

Une classe Chrono a été implémenté, qui compte avec des fonctionnalités pour la mesure de temps avec des fonctions *stop()*, *start()* et *lap()* en utilisant les fonctions crées dans le TD1, ici importé comme *TimerLib.h*. En plus les fonctions *startTime()*, *stopTime()* et *isActive()* retournent les valeurs de state de le timer chrono, comme la valeur du *startTime*, du *stopTime* et un *bool* qui retourne si le chrono est actif avec une comparaison entre les valeurs de *startTime* et *stopTime*.

Le fichier *main_td3a.cpp* réalise une série de tests dans les fonctions codées, en comparant avec des valeurs attendues.

b) Classe Timer

La classe Timer a été implémenté de façon à encapsuler les définitions d'une timer Posix. Après, une classe PeriodicTimer a été créé héritant de Timer avec une redéfinition de la fonction *start()* pour réaliser la périodisation (*its.it_interval = its.it_value*).

Dans le constructeur, les définitions du timer Posix ont été réalisé, avec les structs type *sigaction* et *sigevent*, suivi des fonctions *start()*, qui configure un *itimerspec* avec une duration passée comme argument (*its.it_value = timespec_from_ms(duration_ms)*) et la fonction stop() qui configure aussi un *itimerspec*, mais avec les valeurs nulles.

Ces fonctions sont publiques, une fois que doivent être accédé à l'extérieur de la classe (dans le main). La fonction Callback() déclaré comme virtuelle, et donc, non implémenté, est définit comme *protected*, une fois que doit être accessible que pour la classe elle-même, ou les classes dérivées, qui l'implémenteront. La même accessibilité comme *protected* est valable pour la variable *time_t tid.*

Pour finir, la fonction *call_callback()*, déclarée comme privée, une fois qu'elle doit être accédé que pour la classe Timer. Elle est utilisée de façon pareille au *call_incr()* du TD-2 mais avec l'implémentation pareil à un *myHandler()* comme dans le TD-1 pour être appelé dans le timer Posix, en appelant le *callback()* définie pour la classe héritée.

Une classe CountDown qui hérite de PeriodicTimer a été créé en définissant une fonction callback*()* que réalise le décrément du compter (et donc, il est ce callback qui est appelé dans la fonction *call_callback()*).

Le fichier *main_td3c.cpp* réalise la fonction main que prendre la valeur *nLoops* comme argument du binaire, instancie un CountDown avec cette valeur et réalise un *while(countDown.get_count() != 0){}* pour l'observation du décrément à 1Hz de *nLoops* à zero.

c) Calibration en temps d'une boucle

Basée sur la question 1d du TD-1, on s'intéresse en encapsuler des fonctions que réalisent une mesure des paramètres a et b d'une fonction affine du type *l(t) = a.t + b*. Pour ça, une classe *Calibrator*, dérivée de *PeriodicTimer* a été créé avec une méthode *callback()* qui prendre des valeurs d'une instance de la classe *Looper*, et est instanciée dans une classe *CpuLoop* (cette dérivée de *Looper*), de façon à faire l'appel à la méthode *nLoops(duration_ms).* En permettant alors la réalisation de l'incrément d'un compteur jusqu'à qu'un nombre déterminé de samples soit pris pour la réalisation de l'estimation des coefficients désirées. Cette estimation est faite dans le constructeur de la classe *Calibrator* à partir d'une régression linéaire.

Le fichier main_td3c.cpp présent les tests pour la calibration, où est passé comme argument pour le binaire, le période de réalisation du sampling, le numéro désiré de samples et la duration en millisecondes du test à être réalise avec les coefficients calculés.

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

Les codes associés à réalisation de ce TD sont présents dans le dossier TD4/src, et les fichiers main_td4X.cpp sont aussi dans le dossier TD4, soit le X pour les items a, b, c et d.

a) Classe Thread

Une classe *PosixThread* a été implémentée avec les définitions des paramètres d'un thread Posix, comme la politique d'ordonnancement à être suive, et la priorité donnée. Ella compte avec deux constructeurs, dans lequel, le deuxième sert au cas où le thread est déjà existant et une vérification de sa id est fait avec la fonction *pthread_getschedparam()*, en générant une exception dans le cas de mauvais id.

Une classe *Thread* a été créé comme héritée de cette première avec l'implémentation des fonctions de réalisation de mesure de temps et une fonction *call_run()* qu'appelle la méthode run() désiré d'une classe héritée.

Pour finir, une classe *Incrementer* a été créé comme hérité de *Thread*, avec la définition d'un méthode *run()* que réalise l'incrément d'un compteur pendant un nombre *nLoops* de boucles.

Le fichier *main_td4a.cpp* présent la fonction *main()* qui réalise des tests sur les fonctions créées, où ne nombre de loops, le nombre de tasks et la politique d'ordonnancement est passé comme argument. Il est possible constater le même que dans la question TD-2a, où la valeur du compteur est inférieure à laquelle attendue (*nTasks*nLoops*).

b) Classes Mutex et Mutex::Lock

Une classe *Mutex* et les classes héritées *Lock*, *TryLock* et *Monitor* ont été créées pour la gestion de Mutex dans l'application. *Lock* et *TryLock* s'occupent de l'obtention et de la libération du Mutex, et la classe *Monitor* s'occupe de la gérance des threads qui ont ou pas d'accès au mutex avec les fonctions *wait()*, *notify()* et *notifyAll().*

Pareil à question précédant, une classe IncrementerMutex a été créé comme hérité de *Thread*, avec la définition d'un méthode *run()* que réalise l'incrément d'un compteur mais dans cette fois, avec la présence d'un *Mutex* garanti pour la instance de la classe *Mutex::Lock* avant de la réalisation du incrément (*Mutex::Lock lock(mutex_)*).

Aussi pareil à question précédant, un fichier main_td4b.cpp présent la fonction *main()* qui test les méthodes crées. A cause de la présence du Mutex pour la réalisation de l'incrément, avec cette solution, la valeur du compteur est laquelle attendu (*nTask*nLoops*).

c) Classe Semaphore

Une classe Semaphore a été créé pour l'implémentation de semaphores, avec l'idée de la prise et don de jetons en utilisant un mutex. Une méthode *take()* est utilisé pour retirer un jeton, la méthode *give()* pour rajouter un jeton et une variation de *take()* reçoi un timeout en millisecondes que est utilisé dans la fonction déjà implémenté et appelé dans une instance de la classe *Mutex ::Lock*.

Deux classes dérivées de *Thread* et pointées à un même sémaphore dans le main ont été créés pour production et le consomme de jetons du sémaphore en implémentant une méthode run() différent chacune. La classe *SemProducer* donne des jetons au semaphore avec l'utilisation de la méthode *give()* de la classe *Semaphore.* La classe *SemConsumer* prend des jetons du semaphore avec l'utilisation de la méthode *take()* de la classe *Semaphore.*

Le fichier *main_td4c.cpp* teste les classes créées avec un semaphore partagé pour la classe consommatrice et la productrice. Le nombre de producteur (nProd), nombre de consommateur (nCons) et le nombre de items (nItems) sont passés comme argument dans l'exécution du binaire.

Un vecteur de consommateur et un de producteur ont été déclarés pour la création du nombre désirée de chacun. Pour les taches consommatrices, une division du nombre de items à être consommés pour chacune a été fait comme dans le code ci-dessous :

```
for (unsigned int i=0; i<nCons-1; i++)
    consumer.push_back(SemConsumer(&semaphore, (int) nProd*nItems/nCons));
consumer.push_back(SemConsumer(&semaphore, (nProd*nItems - (nCons-1)* ((int) nProd*nItems/nCons))));
```

Après l'exécution, est affiché de nombre de items créés et consommés, était possible la vérification de que les items produits ont été consommés.

d) Classe Fifo multitâches

Une classe template Fifo a été créé en utilisant le conteneur *std::queue* dans une fichier .hpp avec le template de <*typename T*> où le type de variable a été définie dans le main comme *int* pour ce cas. Dans la classe ont été créés les méthodes pop() et push() pour satisfaire la logique *First In First Out*. La méthode *push()* fait utilisation de la fonction *push()* de la bibliothèque <queue> importée, déjà la méthode *pop()* (et la variation avec le paramètre *timeout_ms*) utilise la fonction *pop()* et *front()* pour la prise du premier élément, toujours en utilisant un mutex pour la protection des opérations.

Les classes *FifoConsumer* et *FifoProducer* ont été aussi créés pour le test de la classe Fifo. La classe *FifoProducer* produit une série de nombres de 0 à un nombre *n* qui sont consommés par la classe *FifoConsumer*.

Le fichier main_td4d.cpp présent la fonction main() qui test les fonctionnalités de la classe Fifo. Pour la division de nombre à être consommés pour les taches consommatrices, un algorithme pareil à l'utilisé pour le cas du sémaphore a été utilisé. Pour la vérification en plus de la comparaison du nombre de items produites et consommés, on vérifie aussi si la somme totale des nombres produites pour toutes les tâches productrices est égale à la somme totale des nombre consommés pour les tâches consommatrices.

[TD-5] Inversion de priorité

Les méthodes créées dans les TD-3 et TD-4 ont été utilisés pour la réalisation de l'inversion de priorité. Le but est de créer 3 tâches A, B et C comme présenté dans le slide 23 de la deuxième diapositive de cours et observer le phénomène d'inversion de priorité.

L'inversion de priorité se produit lorsqu'une tâche qui possède le mutex a la plus haute priorité, par rapport aux autres taches, même si sa priorité initiale est inférieure. Ainsi, si une tâche C a une priorité inférieure à une tâche A, la tâche C, en ayant le mutex, aura une priorité supérieure.

La classe Mutex a été modifié de façon à ajouter un *bool* au constructeur, qui détermine s'il y a ou non une protection contre la Inversion de Priorité, avec le *code* présente ci-dessous :

```
    if(isInversionSafe)
    {
        pthread_mutexattr_setprotocol(&mutexAttribute, PTHREAD_PRIO_INHERIT);
    }
```

Une Classe, appelé *CpuMutexLoop* a été créé pour la définition de la fonction *run()* a être exécuté par le thread. Dans le cas, où la tâche a besoin du mutex après un nombre déterminé de cycles de *clock*, le code présenté ci-dessous réalise da division d'exécution sans et avec le Mutex, comme déterminé dans la diapo (où les paramètres de temps sont passés au constructeur de la classe).

```
            loop->runTime((double) beginMutex / CLOCKS_PER_SEC * 1e3);
/*!< Before getting the mutex*/
            Mutex::Lock lock(*mutex);
            std::cout << ">>> Thread with priority " << priority << " WITH Mutex" << std::endl;
            loop->runTime((double) durationMutex / CLOCKS_PER_SEC * 1e3);
/*!< With the mutex*/
            lock.~Lock();
            loop->runTime((double) (execTime - (durationMutex + beginMutex)) / CLOCKS_PER_SEC * 1e3);
/*!< After unlock of the mutex*/
```

Dans la fonction main(), présente dans le fichier main_td5.cpp implémente l'arrangement des méthodes pour l'observation désirée. Comme argument pour l'exécution du binaire, est attendu le passage de la mot 'InversionSafe' pour le cas avec protection, ou 'notInversionSafe' pour le cas sans la protection. Il st relevant que pour l'observation, il fout exécuter le program qu'avec un processeur, alors, *CPU affinity* a été utilisé pour cette configuration.