Rapport ENSTA 2020 – ROB305 – Julia de Carvalho F. Sula

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

Les applications en temps réel possèdent des contraintes en temps très stricts, par conséquent, la mesure et l'échantillonnage en temps précise sont essentiels. Le langage C++ a des différentes bibliothèques dédiées à ceci, on se concentrera dans les time.h et signal.h.

a) Gestion simplifiée du temps Posix

La structure timespec contient une valeur en secondes (tv_sec) et une en nanoseconds(tv_nsec) pour représenter le temps. La valeur tv_nsec doit être toujours positive. Cette structure était développée pour représenter le temps écoulé et elle est très intéressante pour les applications en temps réel.

Cependant, pour faire que son utilisation soit plus intuitive, on a créé fonctions qui permettent de transformer les données de type structure timespec à la valeur en milliseconds (en perdant précision) et vice versa, faire la somme, soustraction, incrément, décrément et comparaison, etc.

Un choix d'implémentation important a été l'utilisation de la fonction *clock_gettime(CLOCK_REALTIME, &timeNow)* pour obtenir le temps actuel, vu qu'il est compatible avec la raspberry et plus précise que d'autres manières d'obtenir le temps.

b) Timers avec callback

Les timers sont structures qui s'appuient sur *real-time application,* la bibliothèque time.h une structure timer est liée à un sigevent et une sigaction. Un sigevent est un signal envoyé dès l'expiration du timer, qui doit être reçu par la sigaction qui est une fonction handler/caller. La sigaction appellera la fonction qui doit être exécutée dès l'échéance du timer. C'est important de noter que la structure sigevent permet de transmettre données à la fonction appelée. Les caractéristiques du timer (intervalle de temps et temps manquant pour l'expiration) sont configurées à travers la structure itimerspec.

Pour implémenter un timer, on a écrit deux fonctions principales : la init_timer qui permet d'initialiser les valeurs de sigevent et sigaction et créer le timer et le handler qui augmentent un compteur à chaque échéance du timer. La fonction handler est représenté ci-dessous. Donc, à chaque expiration le sigevent envoie le compteur (un pointeur au compteur) à la sigaction/myHandler et le compteur est incrémenté.

```
void* myHandler (int, siginfo_t* si, void*)
{
    int* compt=(int*)si->si_value.sival_ptr;
    cout<<" [SigHandler] Counter: "<<*compt<< endl;
    *compt+=1;
}
```

c) Fonction simple consommant du CPU

La fonction incr est une fonction simple qui permet d'incrémenter un compteur un numéro déterminé de fois. Cependant il faut faire attention à la quantité de fois qu'on demande que le compteur est incrémenté, si cette valeur est plus grande que le UINTMAX (valeur maximale des unsigned int) , on observe des comportements non désirés sur la valeur du compteur.

C’est important noter aussi que quand on a besoin de récupérer le numéro de la ligne de commande d'exécution du programme, dans ce cas il faudrait récupérer le numéro des fois que le compteur sera incrémenté, il faut vérifier qui en fait le numéro a été écrit dans la ligne de commande, pour éviter une segmentation fault. Dans ce code, pour vérifier cette condition, on a analysé le argc, qui donne la quantité d'arguments fourni par la ligne de commande de l'exécution.

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

Ajouter un nouveau paramètre *p_stop* à la fonction incr pour qu’elle s’arrete n’est pas compliqué. Cependant, comme *p_stop* sera maintenant modifié par un timer, pour que la valeur finale du compteur soit cohérente avec le numéro de fois qu’il était incrémenté, il faut que *p_stop* soit une variable volatile.

Quand on déclare une variable comme volatile, c’est-à-dire, qu’on signale au compilateur que cette variable peut changer à quelque moment même si aucun code proche la change. Plus précisément cela veut dire que la compilation de cette variable ne peut pas être optimisé parce que chaque fois le compilateur doit récupérer sa valeur de la mémoire.

Ainsi au déclarer le compteur comme volatile, on empêche que le compilateur optimise l'exécution du code et empêche que tous les changements réalisés par l'échéance du timer ne soient pas faits.

Par ailleurs, dans cette implémentation on a créé des fonction permettant de calculer combien des fois il faut incrémenter pour dépenser n seconds et l’erreur entre la valeur estimée et la réelle. Ces fonctions sont lesquelles ont présenté des comportements aberrantes dans la vérification auprès de la Raspberry. Il a eu un problème de casting de variables qui n'étaient pas visible dans l’ordinateur, mais causé les problèmes de calcul dans la Raspberry, par conséquent, le calcul d’erreur a été divisé en plusieurs étapes en permettant de réaliser les bonnes casts.

e) Amélioration des mesures

Pour améliorer la calibration, c’est-à-dire, l’obtention de la relation entre l’incrémentation et le temps d'incrémentation, on a obtenu plusieurs fois les coeficientes de la droite qui représente cette relation et on a réalisé la moyenne de ces valeurs en permettant une estimation plus précise.

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

L’utilisation des multitâches est essentiel pour paralléliser fonction quand on a des ressources qui le permettent (plus d’une CPU, par exemple), mais elle permet aussi au système sans la multiplicité de ressources de diviser et coordonner l’accès à la ressource à travers politiques d'ordonnancement et priorités.

Dans ce TD, on a implémenté, et étudié, la création de threads, dès fonctions spécifiques fournies par les bibliothèques de C++ aux initialisations des vecteurs des thread soit par la structure vector ou à travers les fors.

a) Exécution sur plusieurs tâches sans mutex

La création d’un *pthread* est réalisé à travers la fonction pthread_create qui a comme arguments l’identifiant du thread, ses attributs, par exemple sa politique d'ordonnancement peut être passé à travers la structure *pthread_attr_t,* un pointeur à fonction ou routine du type void et une pointeur du type void à un argument.

Par conséquent, pour faire que le thread exécute une fonction déjá crée, disant la incr, systématiquement une fonction call_incr (fonction du type myHandler/caller) est créé permettant l'encapsulation de la incr. L'argument de la fonction incr est, donc transmis au thread à travers du pointeur à l'argument.Cela a été réalisé dans ce TD.

De plus, on permet que l'utilisateur choisissez les types de politique d'ordonnancement pour les threads. Pourtant, pour utiliser les politiques différentes de SCHED_OTHER (SCHED_RR et SCHED_FIFO) il faut faire tourner le code à travers la commande sudo.

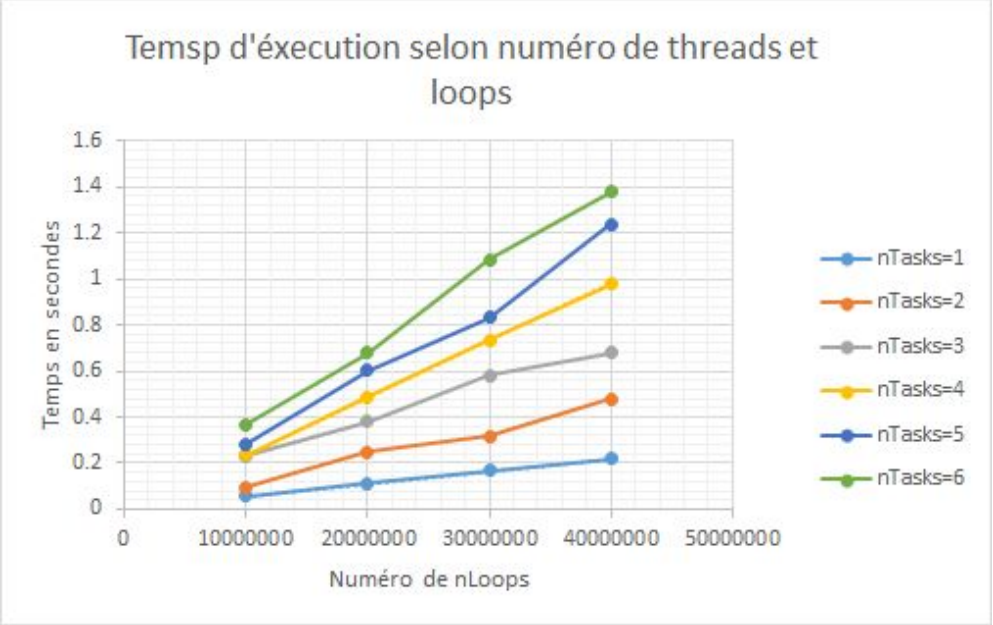
Par ailleurs, après réaliser le programme en utilisant le pthread_create, on a noté que la valeur du compteur changé à chaque exécution. Il s'agit à nouveau de la problématique du volatile maintenant accompagné de l'accès concurrent vu que plusieurs threads essaient d'accéder et changer la valeur du compteur au même temps. Par conséquent, pour garantir que la valeur du compteur est incrémenté, il faut le déclarer comme volatile. Cependant, pour garantir qu'il soit cohérent avec le numéro de threads et nLoops il faut réguler l'accès à la variable compteur.

b) Mesure de temps d'exécution

En mesurant le temps d'exécution du programme, selon des différents numéros de loops à réaliser et les différents numéros de threads, on obtient le graphique de la figure 1.

En analysant ce graphique on note que le plus grand le numéro de tâches lancées, la plus lente est l'exécution, cela est expliqué, car le plus grande le numéro de tâches, le plus grand est l'accès récurrent à la variable counter.

Par ailleurs, comme prévu, la plus importante la valeur de nLoops, la plus lente l'exécution du programme.



c) Exécution sur plusieurs tâches avec mutex

En protégeant l'incrémentation du counter des accès multiples de threads, on note que la valeur finale du compteur devient cohérent avec les valeurs demandées.

Cependant, si l'exécution n'est pas protégée, on observe le même comportement de la 2.a.

[TD-3] Classes pour la gestion du temps

a) Classe Chrono

La classe chrono implémente les fonctionnalités d'un chronomètre, c'est-à-dire, elle a un temps initial *(startTime*) et un temps final *(stopTime)* , à partir desquelles on peut avoir le temps écoulé.

b) Classe Timer

La classe Timer implémente les fonctionnalités d'un timer, par conséquent, elle a un constructor qui initialise le timer et configure le sigevent et le sigaction, un destructor du timer qui efface la structure timer, une fonction de start qui définit les interval du timer et une fonction de stop qui stop le timer. Ces quatres fonctions sont définies comme publiques, vu qu'elles sont nécessaires dans le main, c'est-à-dire, elles seront utilisées par l'utilisateur.

En contrepartie, la classe Timer contient aussi les fonctions callback et la variable identifiant du Timer qui sont protected. La fonction callback permet que chaque classe fille implémente sa propre fonction caractéristique dans le callback, comme elle sera utilisée par les classes dérivées il faut qu'elle soit protégée. Le même s'applique au *timer_Id*, vu que chaque classe dérivée será lié aussi à un timer.

Finalement, on a la fonction privée *call_callback*, cette fonction est privée parce qu'elle sera utilisée seulement pour la classe Timer. Elle permet d'accéder au callback à traves d'un pointer au objet qui fait le callback, ainsi on peut choisir le bon callback sans effort. Cette fonction est aussi de type static, c'est-à-dire, qu'on n'a pas besoin d'instancier un objet timer pour l'utiliser.

Autre partie intéressante de la classe timer sont ces fonctions virtuelles. Fonctions virtuelles sont en bref , fonctions définies par la classe mère, mais qui seront implémentés que dans la classe fille. Cela permet que d'une même signature de fonction ait des comportements différentes dépendant du type d'objet qui appelle la fonction.

c) Calibration en temps d'une boucle

La classe calibrator encapsule les fonctionnalités de la fonction calib() définit dans le TD1 lorsque la classe *CpuLoop* encapsulé les fonctionnalités de la fonction incr() .

Ainsi le *CpuLoop* exécute le nombre prévu de loops par le calibrator pour que son exécution soit de n seconds et la classe chrono permet de vérifier si le temps d'exécution était proche du prévue.

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

Les structures pour la programmation multitâches sont normalement complexes et demandent plusier d'étapes pour par exemple créer un thread, par conséquent, il est utile d'encapsuler ces structures à travers la programmation orientée aux objets.

a) Classe Thread

Pour encapsuler un thread dans une classe, ses fonctionnalités ont été divisées en deux classes, la classe *PosixThread* qui contient les fonctions proches à la structure pthread_t et la classe Thread qui contient les fonctions par rapport au temps d'exécution et *callrun*.

Dans la classe *PosixThread* les constructeurs sont devenus responsables de la configuration des identifiants et attributs (politique d'ordonnancement et priorité). Comme discuté pendant le déroulement du cours, on a deux constructors, un d'entre eux a pour objectif de créer un thread qui n'est pas activé (invalid), cependant la structure posix Thread n'a pas mal de contraintes pour la création d'un identifiant qui représente un thread non valide. En fait, pour qui un identifiant de thread soit invalide sans avoir le problème de segmentation fault, il faut que cet identifiant appartienne à un thread qui s'est déjà exécuté mais qui n'a pas encore réalisé le *join.* Par conséquent, pour obtenir cet identifiant une fonction dummy a été créé pour initialiser un thread bidon pour qui on puisse utiliser l'identifiant de ce thread pour le thread non valide

La classe *PosixThread* implémente aussi les fonctions basics du thread le *join* (qui fait que le thread soit mis en synchronisme avec le main), le *setScheduling* et le *getScheduling* qui configure et obtient les valeurs de paramètre du thread respectivement, entre autres.

Finalement, la classe Thread est responsable d'obtenir les temps d'exécution du thread et appeler les fonctions run qui seront caractéristiques de chaque classe dérivée de la classe thread.

b) Classes Mutex et Mutex::Lock

La classe Mutex, Mutex::Lock, Mutex::TryLock et Mutex::Monitor encapsulent les differents fonctionnalités de la structure Mutex.

La classe Mutex est responsable de toutes les étapes d'initialisation et configuration et contient aussi les fonctions de lock et trylock, cependant celles seront que appelées par les classes Mutex::Lock et Mutex::TryLock, vu qu'elles garantissent le traitement/envoi des exceptions pertinents pour les possibles erreurs des tentatives d'acquérir le mutex.

Finalmente, la classe Monitor est responsable du traitement des variables conditionnelles, quand un thread acquéri un mutex, c'est possible il a besoin d'une certaine condition pour résumer sa tâche, si cette condition n'est pas remplie, la classe Monitor permet que le thread libère ce mutex. Ainsi, le thread est bloqué dû à une variable conditionnel et il attendra qu'il aie la condition nécessaire pour resumir sa tâche. Le Monitor est aussi responsable des envoies le signaux en disant si la valeur de la variable conditionnelle a changé et la condition nécessaire pour qui le thread démarre a été rempli.

c) Classe Semaphore

La classe semaphore a été présenté comme une boîte de jetons et on peut utiliser cette analogie pour expliquer en bref ses fonctions: on a une boîte dont il y a *n* nombres de jetons (constructeur), on peut retirer les jetons de la boîte (fonction take) ou les mettre dedans (fonction give), s'il y a une fille d'attente pour retirer un jeton de la boîte, on attend pour un certain temps et après on abandonne la transaction(fonction take avec time-out) et finalement on a le droit de demander au responsable de la boîte combien de jetons il y a (function get_Counter). Ces 5 actions sont les fonctions implémentés pour le semaphore.

La fonction get_counter a été ajouté pour permettre de vérifier que le numéro de jetons dans le sémaphore est , en fait , zéro.

Pour représenter les personnes en retirant ou mettant les jetons dans la boite les GiveThread et le TakeThread ont été créés.

d) Classe Fifo multitâches

Une mémoire fifo fonctionne à partir de la logique first in, first out, c'est-à-dire, que le premier élément à entrer dans la fifo sera le premier à sortir.Pour représenter ce comportement, on a utilisé la variable std::queue que permet stocké différents types et que peut être utilisé avec les méthodes push et pop. La méthode push permet d'ajouter un élément à la queue et le pop libéré le dernier élément a entre dans la queue.

Comme la fifo a été créé à partir de templates, c'est possible d'enregistrer quelque type d'élément sur elle, ceci permet aussi que les threads TakeFifo et GiveFifo soient aussi si géneriques en pouvant retirer ou ajouter n'importe quel type de variable à la fifo.

[TD-5] Inversion de priorité

Dans ce TD, il y a eu trois implémentations principales: le changement de la fonction start des PosixThread pour qu'ils tournent que dans une CPU, le changement des constructeurs du Mutex pour permettre une protection contre l'inversion de priorité et la création d'une classe dérivée de la classe Thread qui permet de représenter les tâches du cours Multitâches-1.

Dans la fonction PosixThread::start, la fonction pthread_setaffinity_np a été ajouté pour garantir que l'exécution du programme n'utilisera pas les ressources multicore. Le code est reporté ci-dessous.

```
void PosixThread::start(void *(*threadFunc)(void *), void *threadArg)
{
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(0, &cpuset);
    pthread_setaffinity_np(posixId, sizeof(cpu_set_t), &cpuset);
    //Setting Threads for only one CPU
    pthread_create(&posixId, &posixAttr, threadFunc, threadArg)
}
```

Le nouveau constructor ajouté à la classe Mutex est le Mutex(bool isInversionSafe), quand on veut protéger le thread de l'inversion de priorité , la variable isInversionSafe à la valeur true et le protocol du thread est configuré pour être PTHREAD_PRIO_INHERIT. C'est-à-dire, que le thread qui a obtenu le mutex, aura la plus grande priorité.

Finalement, pour répliquer les tâches du cours Multitâches-1, une classe CpuLoopThread a été créé, elle utilise 3 paramètres pour définir le comportement du thread: le timeBefore_ms, timeAfter_ms et tics. Le timeBefore_ms représente le temps d'exécution du thread avant qu'il demande d'acquérir un Mutex, le timeAfter_ms le temps d'exécution après la libération du mutex et le tics le temps d'exécution pendant qu'il a le mutex. La variable laquelle il faudrait garantir l'accès mutex était un objet CpuLoop.