# Rapport ENSTA 2020 – ROB305 – Davide L. Brambilla

Ce document represente le compte rendu de l'ensemble des TDs pour le cours ROB305 de l'ENSTA Paris.

Nous allons implémenter deux API POSIX :

- la première pour la *gestion du temps* qui s'occupe de mesurer la durée d'exécution avec timers où bloquer/débloquer un programme après un certain temps prédéfini.
- la deuxième pour la gestion d'un *programme multi-taches* qui s'occupera de créer, détruire une tache, gérer la concurrence données en les protégeant des accès concurrents, l'activation où le blocage des taches.

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

Nous allons utiliser les libraires *<time.h>* et *<signal.h>* afin de realiser les TD.

### a) Gestion simplifiée du temps Posix

Dans cette première partie, il nous est demandé d'implémenter les fonctions nécessaires pour pouvoir gérer facilement les opérations lié au temps pour la structure timespec.

La structure timespec est composée par deux champs: *tv_sec* qui indique les seconds du temps et le deuxième *tv_nsec* qu'indique les nanoseconds. Cette structure permet d'être précis dans la mesure du temps.

Ici je vais implementer des fonctions pour passer facilement entre un temps en milliseconds défini comme type *double*, par exemple 2,2 ms, et un temps définie comme *timespec*.

On pourrait aussi determiner à quelle instant de temps se trouve le programme grace à la fonction *clock_gettime(CLOCK_REALTIME, &now)*.

Nous devrons être capable aussi de pouvoir effectuer les opérations arithmetiques de bases: au travers la surcharge des opérateurs, il a été possible d'implémenter les opérations de base pour la structure *timespec*.

Une particulier attention je voudrais la donner à l'implementation de la fonction *timespec_wait(const timespec& delay_ts)* dans laquelle j'ai utilisé la fonction *nanosleep()*. Pour cette fonction il fallait gérer une éventuel erreur qui j'ai géré comme cela:

```
timespec timespec_wait(const timespec& delay_ts){
   struct timespec remaining;
   if(nanosleep(&delay_ts, &remaining) == -1){
      if(errno == EINTR){
         timespec_wait(remaining);


   return remaining;
```

Si *nanosleep()* retourne la valeur -1 je vais vérifier le valeur de *errno* qui, si égal à *EINTR*, me dis que il y a eu une interruption et la variable *remaining* me permettra de faire redémarrer l'attente et de faire passer le temps qui reste.

### b) Timers avec callback

Dans cette partie on a implémenté un timer *Posix* qui a le but d'incrémenter un compteur à la fréquence de 2Hz en imprimant son valeur. Cette valeur sera défini dans le main et il sera incrémenté chaque 0.5 seconds.

Pour un timer, la définition du temps est faite en choisissant deux valeurs différents du temps: le premier *it_value* qui indique la durée du premier période et le deuxième *it_interval* qui défini la durée des tous les périodes suivantes.

À la fin de chaque période le timer va exécuter un fonction *handling*: c'est ici que on incrementera la valeur de counter.

Nous avons implementé le timer en deux fonctions : une premiere pour la creation du *Timer* et une deuxieme pour l'implementation de la function de *handling* que nous avons reporté ici :

```
void myHandler (int, siginfo_t* si, void*)
{
   int* pCounter = (int*) si->si_value.sival_ptr;
   std::cout << "Iteration number " << *pCounter << std::endl;
   *pCounter += 1;
}
```

Nous allons donner les valeurs à la fonction de *handling* au travers le pointeur *si* qui contient les informations lié au signal.

Au moment de l'exécution, nous avons noté que, si on déclare la variable sans spécifiant la mot clé *volatile* nous pourrons avoir des problèmes dans le cas d'optimisation de compilation.

Si, en fait, nous utilisons l'option de compilation *-O3*, le compilateur ne considère pas la possibilité que la variable du compteur pourra être modifié par autres processus qui s'exécutent en parallèle au *main.*

En spécifiant la mot clé *volatile*, nous allons informer le compilateur que cette variable pourra être aussi modifié par autres processus en parallèle et nous aurons la possibilité de faire une compilation optimisé sans que le programme ait des problèmes d'exécution: la valeur du compteur sera incrémenté jusqu'à la valeur correcte et le programme s'arrêtera.

### c) Fonction simple consommant du CPU

Ici il nous est demandé d'implémenter une fonction qui occupe la CPU: nous avons implémenté une fonction très simple qui incrémente une compteur pour un nombre de boucles défini au moment de l'execution. Pour faire cela nous avons dû utiliser les arguments de la fonction principal *main*. La signature standard du main est *main(int argc, char* argv[])* où *argc* indique le nombre des chaînes données au programme et *argv[]* le tableau où ces chaînes sont sauvegardées. À l'exécution de notre programme, nous allons écrire *./td1c 100* et le numero de boucles qui le programme fera sera donc 100.

Afin de pouvoir afficher le temps d'execution du programme avec un nombre à virgule, j'ai utilisé les fonctions implementés lors du TD1a(*timespec_now()*) et affiché la valeur à l'ecran en utilisan la librerie iomanip et le mot clé setprecision(9) qui me fait avoir 9 chiffres significatives apres la virgule.

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

Dans ce cas, nous allons définir un timer *timer_t*, comme déjà fait dans la partie *b*, de façon que après un certain instant de temps, défini au moment de la création du timer, nous allons arrêter la fonction implémentée dans le point *TD1c* qui augmente la valeur du compteur.

Pour faire cela nous avons modifié la fonction i*ncr* 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*. L'activation de la valeur *pStop* est faite par le timer après que le temps d'attente choisi est passé.

Dans la deuxième partie, il était demandé de construire une fonction *calib* afin de pouvoir estimer le nombre des boucles effectués avec la formule *loops = a*t + b* et le but de la fonction *calib* était exactement de calculer les valeurs *a (pente)* et *b(constante)*. J'ai calculé les valeurs de *a* et de b en se basant sur deux mesures du temps le premier à 4 seconds et le deuxième à 6 seconds. Après avoir obtenu les valeurs de *a* et de *b*, nous avons testé le calcul pour un temps de 5 seconds.

Nous avons obtenu une bonne estimation de nombre des loops éant donnée que l'erreur en percentage es toujours inferieure au 2%.

### e) Amélioration des mesures

Afin d'améliorer la précision de la fonction j'ai choisi dans le main le numéro de fois que le programme devra faire la *calibration*: En particulier, il s'exécutera *numCalibs* fois de façon de pouvoir calibrer, avec une approximation meilleure, les valeurs de *a* e de *b*. Il démarrera avec l'étude pour 1 second et, pour chaque boucle, il incrémentera la période de 1 seconds. Cela nous permettra d'avoir plusieurs informations sur les temps d'exécution et les relatives numeros de boucles. Nous allons sauvegarder ces valeurs dans deux tableaux, afin de pouvoir, en suite, appliquer la formule de *régression linéaire* pour le calcul des valeurs *a* et *b*.

Les resultats sont ameliorés car nous allons obtenir un erreur en percentage plus petite que 1%.

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

### a) Exécution sur plusieurs tâches sans mutex

Dans ce point, nous avons eu un premier contact avec la création des threads (*pthread_create*) et nous en avons utilisé plusieurs  afin de pouvoir incrémente une variable compteur.

Cela nous a permis de observer un problème lié à l'accès aux données par plusieurs processus en parallèle.

En fait si on choisi de faire démarrer 30 threads chacun avec 2000 itérations, nous devrons obtenir nue valeur de compteur de 6000 mais cela n'est pas le cas.

Cette problème est du à l'acces par plusieurs processus à une meme zone memoire et sera résolu avec l'introduction de la *exclusion mutuelle*(*mutex*).

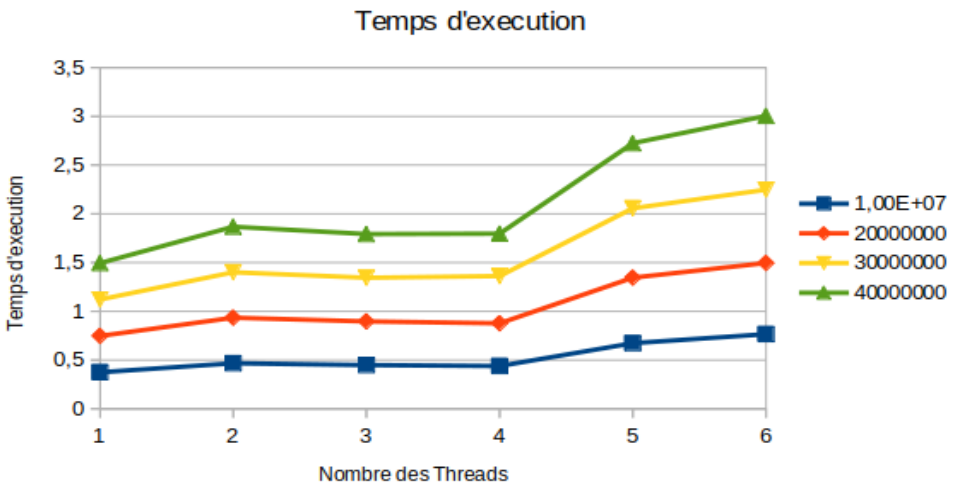
### b) Mesure de temps d'exécution

Dans ce point, nous allons étudier le différents politiques d'ordonnancement des threads: *SCHED_OTHER* qui est laquelle de default, *SCHED_RR* qui fait executer les taches avec la meme priorité à tour de rôle et *SCHED_FIFO* qui utilise une politique de premier arrivé, premier servi.

Nous allons donner, au travers de la fonction *sched_get_priority_max* la priorité maximale au *main* et nous allons donner une priorité mineure aux threads que nous allons créer. Si la politique est *SCHED_OTHER* ils auront une priorité 0, autrement une valeur aleatoire compris entre 0 et 98 étant donnée que la priorité maximale dans les deux autres cas est 99.

J'ai reporté dans le tableau suivante les valeurs du temps d'exécution en seconds exécute dans la carte Raspberry pour un nombre diffèrent des taches (colonnes) et de loop (lignes) avec la politique de reordonnancement *SCHED _RR* :

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 10e7 | 0.3742479 | 0.4673527 | 0.4485053 | 0.4387920 | 0.6728517 | 0.7655070 |
| 2*10e7 | 0.7470923 | 0.9343745 | 0.8965638 | 0.8766722 | 1.3449741 | 1.4959189 |
| 3*10e7 | 1.1209648 | 1.4009504 | 1.3442379 | 1.3618697 | 2.0567633 | 2.2481328 |
| 4*10e7 | 1.4942717 | 1.8675926 | 1.7927021 | 1.7970507 | 2.7245280 | 3.0051349 |




Ce que il a été mis en évidence c'est que jusqu'à quatre processus en parallèle nous avons plus ou moins le même temps d'exécution. Après le temps d'exécution augmente considérablement. Cela est du au fait que la *Raspberry* a *quatre coeurs* et les performances sont maintenu jusqu'à 4 processus en parallèle. Si on augmente le numéro de processus en parallèle nous allons ralentir l'exécution totale état donné que plusieurs processus devront fonctionner sur le même coeur.

### c) Exécution sur plusieurs tâches avec mutex

Dans cette section, nous allons résoudre le problème rencontré dans la partie *a*: nous allons utiliser l'*exclusion mutuelle* afin de gérer l'accès à la variable du compteur par plusieurs processus en parallèle.

Si l'option *protected* est spécifiée dans la ligne d'exécution du programme nous aurons que l'accès à la variable du compteur sera protégé par un mutex qu'assurera que seulement un processus pourra la modifier à la fois.

Pour faire cela nous avons dû ajouter une variable de type *pthread_mutex_t* dans les données que on donne aux processus lors de leur création et qui sera utilisée pour faire gagner le droit à un particulier processus de modifier la valeur de compteur.

On a pu assigner et enlever le mutex à un certain processus au travers les fonctions *pthread_mutex_lock* et *pthread_mutex_unlock* dans la fonction de *handling* exécute par les processus créés:

```
void* call_incr(void* data)
{
   Data* pData = (Data*) data;
   if(pData->protection){
      pthread_mutex_lock(&pData->mutex);

   incr(pData->nLoop, pData->counter);
   if(pData->protection){
      pthread_mutex_unlock(&pData->mutex);

   return data;
```

Si par contre, dans la ligne d'execution du programme nous ne specifions pas *protected* on aura le meme probleme que on a eu dans le point *a* car la protection avec *Mutex* sera desactivée.

Pour ce que concerne le temps d'exécution, nous pouvons noter que l'exécution de programme sans *mutex* est plus rapide: en executant sur la *Raspberry* 20 processus avec 20000 boucles a faire nous allons obtenir un temps d'exécution de 0.0189 seconds dans le cas *avec mutex* et de 0.0094 seconds dans le cas sans *mutex*.

## [TD-3] Classes pour la gestion du temps

### a) Classe Chrono

Dans cette section nous allons implémenter la classe *Chrono* qui implémente les fonctionnalités de mesures de temps d'un chronomètre. Les méthodes définies dans *TimeSpec.h* seront utilisées.

Cette classe nous permet de définir le temps de début, de fin et la durée d'un évènement, processus. Le *main* que a été écrit pour tester cette classe se base sur le fait de faire démarrer un objet de type Chrono et endormir la tache pour 6 seconds.

Enfin il a été possible de faire une comparaison entre la valeur du temps donné par *Chrono* et par les fonctions implementées dans la classe *TimeSpec* pour vérifier que ils sont effectivement bien utilisées à l'intérieur de la classe *Chrono*.

### b) Classe Timer

Ici il nous est demandé d'implémenter la classe *Timer* qui contiendra une méthode *virtual* afin qu'elle puisse être implémentée par une classe fille que hérite de elle.

Dans notre cas, la casse fille est la classe *PeriodicTimer* qui s'occupe d'implémenter la méthode virtuelle.

Je vais expliquer ci-dessous les raison pour lesquelles chaque fonction est publique, privée où protectée. Le constructeur, le destructeur, la méthode *start()* et la méthode *stop()* sont *publiques* car ils doivent être appelés depuis le main qui se trouve à l'extérieur de la classe et il n'a pas aucun lien avec la classe. La méthode *callback()* sera *protegé* pour le fait que devra être accessible par les classes filles qui dérivent de Timer et qui la implémentent. Enfin, la méthode *call_callback()* sera *privé* pour le fait qu'elle sera utilisé seulement à l'interieur de la classe. De plus cette méthode est definie comme *static* que signifie qu'elle n'est pas liée à l'objet mais à la class et donc elle peut etre appelée sans la création de l'objet.

La function qui doit être définie comme virtuelle est la fonction *callback* qui devra être implementé par les classes filles.

### c) Calibration en temps d'une boucle

La classe *Calibrator* est basé sur la fonction *calib* implémenté dans le TD1e. La classe *CpuLoop* encapsule les fonctionnalités de la fonction *Looper* qui permet de faire les boucles et contient un pointeur vers un objet *Calibrator*. Cela nous permet d'initialiser un objet *CpuLoop* qui va faire des boucles en ayant le comportement de *Calibrator.*

Finalement, le main nous permet de évaluer les performances de notre Calibrator et nous avons un erreur petit.

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

Le TD4 a l'objective d'implémenter la création des threads et la gestion des mutex en utilisant une programmation orienté objets.

### a) Classe Thread

Dans cette premiere partie il nous est demandé d'implémenter une classe *PosixThread* qui contient les paramètres basiques du thread: son identifiant et ses attributs et aussi les fonctions basiques pour sa création. Elle contient aussi le constructeur qui sera responsable de attribuer la priorité et la politique au thread créé. En particulier, la méthode *start* nous permettra de démarrer le thread et elle devra être publique afin qu'elle puisse être accédé dehors de la classe. Pour la même raison *join, set/getScheduling* seront publiques aussi.

Pour la création d'un Thread nous allons hériter une classe à partir de *PosixThread:* la classe *Thread* qui a l'objective de définir la fonction virtuelle *run* qui sera implementée et exécutée par les threads et des autres fonctions qui permettent de définir la durée du thread considéré.

Pour tester ces deux classes j'ai implémenté une fonction *Incr* fille de la classe *Thread* qui implémente la méthode virtuelle *run().* Cette méthode aura comme objective l'incrément d'un compteur pour un nombre de boucles prédéfinis.

Dans le *main* cets fonctions ont été testées: nous avons encore un problème d'accès données par plusieurs threads en parallèle. En fait, en démarrant plusieurs threads *Incr,* nous observons que la valeur du compteur, partagée entre eux, n'arrive pas à avoir la valeur finale espérée. Si on démarre 30 threads, chacun avec 1000 boucles a faire, la valeur de compteur n'arrivera pas à 30000 mais elle sera toujours une valeur inférieure. Pour résoudre cela nous avons dû implémenter une classe *Mutex* qui puisse garantir l'accès au compteur par un thread à la fois.

### b) Classes Mutex et Mutex::Lock

Dans cette section, il nous est demandé d'implémenter les classes pour la gestion d'un *Mutex:* les classes implémentés sont *Mutex, Mutex::Monitor, Mutex::Lock et Mutex::TryLock* qui permettent de gérer le blocage/déblocage d'un mutex pour garantir qu'un seul processus à la fois puisse accéder à une certaine donnée. On a choisi d'inserer les trois classes *Lock, TryLock and Monitor* à l'interièure d'une dossier appellé *Mutex* comme demandé dans les conventions de codage.

Les classes *Lock* et *TryLock* permettent d'obtenir (*bloquer*) un *mutex*. La classe *Monitor* permet de gérer les variables conditionnelles et l'envoie des signaux quand un thread est verrouillé afin de pouvoir arrêter un thread qui a l'access à un mutex, exécuter une autre thread pour un certain temps et en suite continuer avec l'exécution du première thread.

Ces classes ont été testées en implémentant la classe *IncrMutex* qu'est similaire à la classe *Incr* implementée dans le point précèdent mais qui contient une pointeur vers un objet *Mutex* qui permet d'avoir la valeur du compteur espérée: si on démarre 30 threads, chacun avec 1000 boucles la valeur du compteur sera 30000. Cela vient avec un temps d'exécution

plus grande car les threads doivent attendre que le mutex soit libéré pour pouvoir incrémenter compteur de façon ordonnée.

### c) Classe Semaphore

Dans cette exercice on a implémenté une classe *Semaphore* qui a l'objective de initialiser une *boîte à jetons* de façon que on puisse créer diffèrent threads *consommateurs* et *producteurs* qui, en accédant au même objet *Semaphore,* puissent augmenter(*give()*) où diminuer(*take()*) le numéro de jetons à l'intérieure de la boîte. Le numéro des jetons sera donné par une variable compteur à l'intérieure de la classe *Semaphore.*

Les classes *Consumer* et *Productor* ont été implementées, en les dérivant de la classe *Thread,* pour la création d'un main qui realise la situation précédement décrite. Dans le main on a aussi vérifié que le nombre de jetons créés sont effectivement consommés et nous avons que c'est le cas aussi quand le numéro de thread consommateurs n'est pas égal au numéro de threads producteurs.

### d) Classe Fifo multitâches

La classe template *Fifo* permet de créer une sorte de *Semaphore* pour une type quelconque. La *boite* sera implémenté par une queue d'éléments qui seront ajoutés à la fin de la queue (*push()*) où supprimés de la tête de la queue (*pop()*) pour satisfaire la logique de *First-In-First-Out*. Elle s'agit d'une classe template et, donc, aura la declaration et l'implementation des fonction dans le même fichier.

Dans ce cas les classes C*onsumer et Productor* sont substituées par les classes *FifoConsumer* et *FifoProductor.* Nous avons testé la classe *Fifo* pour des types *int*. Même dans ce cas, nous avons testé que le nombre des éléments créés correspond avec le nombre d'éléments détruits.

## [TD-5] Inversion de priorité

Dans cette dernière section, nous allons modifier la classe *Mutex* en introduisant un nouveau constructeur qui contient une variable booléenne *isInversionSafe*. Cette variable nous permet de activer et désactiver l'*inversion de priorité*. Pour tester cette classe on a implémente un thread avec mutex basé sur le fichier *CpuLoop* que on a appelé *CpuLoopMutex*.

L'*inversion de priorité* indique que la tache qui possede le mutex tournera à la priorité maximale de l'ensemble des tâches qui demandent le blocage du *mutex.* Par example, si nous avons que le mutex est bloqué par la tache *A* qui a une priorité basse, même si arrivera une tache *B* avec priorité plus elevé, elle va detener le *mutex* et elle obtenra la meme priorité de la tache *B.*

Étant donnée qu'on doit recréer les conditions décrites dans les diapositives, nous allons donner à chaque thread implémenté ses caractéristiques comme, par exemple, la durée d'exécution, le temps qui attend pour bloquer le mutex, le temps pour libérer le mutex. Pour ces valeurs on peut se refaire à la slide numero 23 des diapositives.

Pour tester l'algorithme nous avons dû spécifier que les taches doivent tourner sur un seul coeur de notre *Raspberry* avec le code suivante dans le *main* qui utilise *CPU Affiniy*:

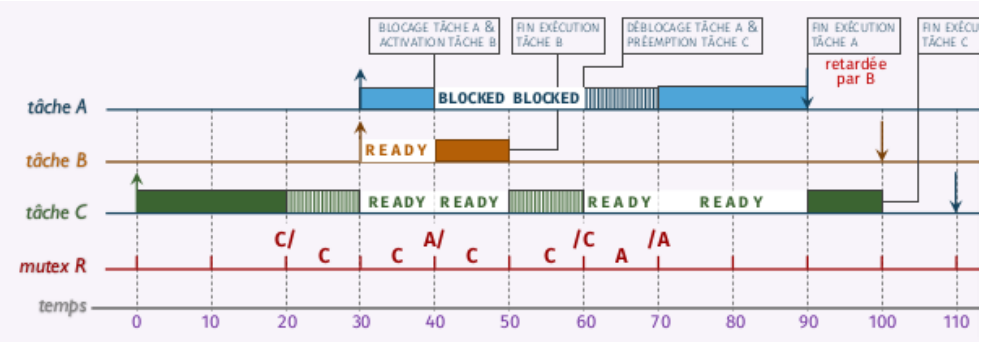
```
//set that we will work only on a single CPU
	cpu_set_t  cpuset;
	CPU_ZERO(&cpuset);
	CPU_SET(0, &cpuset);
	sched_setaffinity(0, sizeof(cpu_set_t), &cpuset);
```

J'ai verifié que on est en train d'utiliser un seul coeur de la *Raspberry* avec la fonction *top.*

En suite, pour pouvoir gérer l'exécution des differents parties de programme nous avons implementé la fonction suivante à l'interieur de la classe *CpuLoopMutex*:

```
void CpuLoopMutex::run()
{
	cout << "I am the thread with priority " << priority << endl;
	if(timeReqMutex != -1)
	{
		loop->runTime(timeReqMutex);
		Mutex::Lock lock(*mutex, 100000);
		cout << "I am the thread with priority " << priority << " and I have the Mutex" <<
endl;
		loop->runTime(timeLastMutex);
		lock.~Lock();
		loop->runTime(timeExecution - (timeLastMutex + timeReqMutex));
		cout << "---FINISHING--- I am the thread with priority " << priority << endl;
	}
	else
	{
		loop->runTime(timeExecution);
		cout << "I am the thread with priority " << priority << endl;
	}
}
```

Enfin, nous avons comparé les differents temps d'execution avec l'*inversion de priorité* et sans l'inversion de priorité pour le trois taches. Nous avons trouvé que, pour l'*inversion de priorité,* la durée de la tache *A* est de 3.9935 seconds et sans l'*inversion de priorité* la durée est de 3.9969 seconds. Cela on peut l'expliquer en regardant les deux figures: la première est laquelle sans *inversion de priorité* où *A* termine à 90 tics.




Dans la deuxième nous avons reporté la meme graphique (les points noirs sont les essayes de bloquer un thread) et on peut observer que le thread *A* termine a 80 tics. Nous avons donc que le thread *A* se termine avant dans le cas de *inversion de priorité.*