ROB305 – Hanin Hamdi et Ahmed Yassine Hammami

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

a) Gestion simplifiée du temps Posix

L'objectif de ce premier exercice est d'implémenter des fonctions qui permettent de gérer le temps Posix. L'idée est de créer des alternatives pour faire une transition facile entre le temps en millisecondes et le temps Posix de la structure **timespec.**

Ainsi, les fonctions implémentée sont les suivantes :

- `double timespec_to_ms(const timespec& time_ts)` et `timespec timespec_from_ms(double time_ms)` pour le passage du temps Posix au temps en millisecondes et vice versa.
- `timespec timespec_now()` pour déterminer le temps courant sous la forme d'une variable timespec.
- `timespec timespec_negate(const timespec& time_ts)` pour avoir l'opposé d'un temps Posix.
- `timespec timespec_add(const timespec& time1_ts, const timespec& time2_ts)` pour faire l'addition de deux temps Posix.
- `timespec timespec_subtract(const timespec& time1_ts, const timespec& time2_ts)` pour faire la soustraction de deux temps Posix.
- `timespec timespec_wait(const timespec& delay_ts)` pour arrêter le thread pendant une certaine durée donnée sous la forme d'un temps Posix.

Ensuite, pour une meilleure manipulation des variables du type struct timespec, on a implémenté les opérateurs suivants : +, - (soustraction), - (négation), +=, -=, ==, !=, < et >.
Nous réalisé, en outre tous les tests unitaires pour s'assurer du bon fonctionnement de ces implémentation. Référez-vous au script TD1a_main.cpp pour vérifier le test.

Enfin, nous avons construit avec l'ensemble de ces opérateurs et de ces fonctions une nouvelle library qu'on a nommée **time_library** et que nous avons utilisée tout au long de ce travail.

b) Timers avec Callback

L'objectif de cette deuxième question est de se familiariser avec les timers Posix en développant un programme qui permet d'incrémenter et d'imprimer sur l'écran un compteur initialisé à 0 jusqu'à la valeur 15 d'une façon périodique et avec une fréquence égale à 2 Hz soit une période égale 0.5 secondes. Pour ce faire, l'idée est de créer un timer période avec un délai et une période égaux à 0.5 secondes. On associe au timer une fonction callback (myHandler) qui incrémente un compteur qui lui est passé par référence et qui affiche sur l'écran sa nouvelle valeur, et ceci suite à chaque expiration du timer.

Référez-vous au script TD1b_main.cpp pour voir l'implémentation de ce petit programme.

On donne dans la figure suivante un exemple d'output donnée par ce code :

```
rob305-pi48# ./TD1b_main_test
-----------------------------------------------
------------------ ROB305 TD1B ----------------
--------- Hanin Hamdi & Ahmed Yassine Hammami
-----------------------------------------------
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

c) Fonction simple consommant du CPU

Pour cette partie, on souhaite évaluer le temps d'exécution d'une fonction simple mais qui est toujours utilisée et qui consomme du CPU ; une boucle d'incrémentation d'un compteur. L'idée est de passe un argument en ligne de commande qui détermine le nombre de boucles qu'on souhaite réaliser. L'implémentation de cette fonction est la suivante :

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

Ensuite, pour évaluer le temps d'exécution de la fonction **incr**, on utilise CLOCK_REALTIME comme suit :

```
Clock_gettime(CLOCK_REALTIME, &starttime) ;
Incr(nLoops, pCounter);
Clock_gettime(CLOCK_REALTIME, &endtime) ;
Duration_ts = endtime – starttime ;
```

Un exemple d'output pour ce code est le suivant :

```
rob305-pi48# ./TD1c_main_test 10000000
-----------------------------------------------
------------------ ROB305 TD1C ----------------
--------- Hanin Hamdi & Ahmed Yassine Hammami ------
-----------------------------------------------
Final value of the counter = 1e+07
Duration of the function incr = 0.370388 seconds
```

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

Ici, on reprend la fonction **incr** précédente avec quelques modification. Sa fonctionnalité globale reste la même : elle incrémente un compteur double. Par contre, on change sa condition d'arrêt : elle s'arrête si l'une des conditions suivantes sont vérifiées :

- On atteint le nombre de boucle maximal.
- On paramètre booléen stop passé en paramètre par référence est mis à la valeur true.

La nouvelle implémentation de inc est comme suit :

```
Unsigned int incr(unsigned int nLoops, double* pCounter, volatile bool*
pStop)
{
    unsigned int iLoops = 0 ;
    while (iLoops < nLoops and not(*pStop))
    {
        *pCounter += 1.0 ;
        iLoops += 1 ;
    }
    return iLoops ;
}
```

Ensuite, on souhaite gérer l'arrêt de l'incrémentation par un timer : une fois expiré, sa fonction callback mais la variable stop à true et donc incr s'arrête. Ainsi, la variable stop ici est exposée à un accès concurrent par les deux fonctions incr et myHandler, le callback du timer. Ce qui impose que stop soit déclarée comme une variable **volatile**.

Donc, pour résumer, on modifie incr comme ci-dessous. On crée un timer non périodique avec une durée d'expiration donnée en ligne de commande. On initialise le timer et on déclenche en parallèle la fonction incr. Une fois le timer arrive à échéance, sa fonction callback myHandler met la variable stop à true ce qui arrête le fonctionnement de incr. En conclusion, de cette façon on peut incrémenter un compteur pour une certaine durée.

Un output typique de ce script (TD1d_main.cpp) est le suivant :

```
rob305-pi48# ./TD1d_main_test 1500
-------------------------------------------------------
-------------------- ROB305 TD1D ----------------------
--------- Hanin Hamdi & Ahmed Yassine Hammami ------
-------------------------------------------------------
Done Loops = 24756470
```

Ainsi pour une durée de 1.5 secondes, on réalise environ 25.10^6 boucles d'incrémentation.

L'objectif suivant est de trouver une calibration pour relier entre le temps d'exécution de la fonction incr et du nombres des boucles effectuées durant cette durée d'exécution. Pour ce faire, on suppose tout d'abord qu'il y a une relation affine entre ces deux paramètres :

$$L(t) = a*t + b$$

Ensuite, pour trouver a et b, il faut exécuter incr comme précédemment pendant deux durées différentes T1 et T2. On aura alors :

$$a = \frac{L_2 - L_1}{T_2 - T_1}$$

$$b = \frac{L_1 + L_2 - a(T_1 + T_2)}{2}$$

Avec cette calibration, on obtient le output suivant qui donne un résultat de a et b déterminé à partir de nombre de boucles exécutées par incr pour 4 et 6 secondes :

```
rob305-pi48# ./TD1d_main_bis_test 1000
-------------------------------------------------------
-------------- ROB305 TD1D_bis : Calibration ----------
--------- Hanin Hamdi & Ahmed Yassine Hammami ----------
-------------------------------------------------------
Calibration using timers : l(t) = 1.65081e+07*t + -11864   ( t(s) )
---- Test Calibration : real duration = 0.999483 seconds
```

Le test de la calibration a été effectué pour une durée d'exécution de incr égale à 1 seconde, et on voit que la vérification avec lo CLOCK_REALTIME donne une durée réellement écoulée égale à 0.9994 secondes -> la calibration est alors valide.

e) Fonction simple consommant du CPU

Enfin, nous voulons améliorer la précision de la calibration précédente. Une solution qu'on a adopté est de considérer plus de mesures et ensuite de faire une sorte de régression linéaire pour déterminer a et b.

On suppose qu'on utiliser n mesures (n nombre paire). On va alors déterminer a et b en utilisant les moyenne des n/2 premières mesures et n/2 mesures suivantes :

$$Pour\ chaque\ i \geq 1\ ,\ on\ a\ T_i = i * T, avec\ T\ la\ période\ d'échantillonnage.\ Donc\ :$$

$$a * T * \sum_{1}^{n} i + n * b = \sum_{1}^{n} L_i$$

$$\rightarrow\ a * T * \sum_{1}^{\frac{n}{2}} i + \left(\frac{n}{2}\right) * b = \sum_{1}^{\frac{n}{2}} L_i$$

$$et\ a * T * \sum_{\frac{n}{2}+1}^{n} i + \left(\frac{n}{2}\right) * b = \sum_{\frac{n}{2}+1}^{n} L_i$$

$Donc\ a * T * \left(\sum_1^n i - \sum_1^{\frac{n}{2}} i\right) = \sum_{\frac{n}{2}+1}^{n} L_i - \sum_1^{\frac{n}{2}} L_i \rightarrow ce\ qui\ donne\ la\ valeur\ de\ a$ et on pourra par suite déduire b.

Le résultat de cette deuxième calibration est le suivant :

```
rob305-pi48# ./TD1e_main_test 1000 4 1000
-------------------------------------------------------------
------  ROB305 TD1e : Calibration with more samples --------
--------- Hanin Hamdi & Ahmed Yassine Hammami ---------------
-------------------------------------------------------------
Calibration using timers : l(t) = 1.65073e+07*t + -4727.5   ( t(s) )
---- Test Calibration : real duration = 0.999833 seconds
```

On remarque un légère amélioration par rapport à la calibration précédente.

On double encore le nombre de mesures utilisé ; on teste avec 8 mesures espacées chacune par une seconde, on obtient :

```
rob305-pi48# ./TD1e_main_test 1000 8 1000
-------------------------------------------------------------
------  ROB305 TD1e : Calibration with more samples --------
--------- Hanin Hamdi & Ahmed Yassine Hammami ---------------
-------------------------------------------------------------
Calibration using timers : l(t) = 1.65066e+07*t + -2345.72   ( t(s) )
---- Test Calibration : real duration = 0.999959 seconds
```

On voit bien que la précision a encore augmenté ; cette solution apporte bien une amélioration.

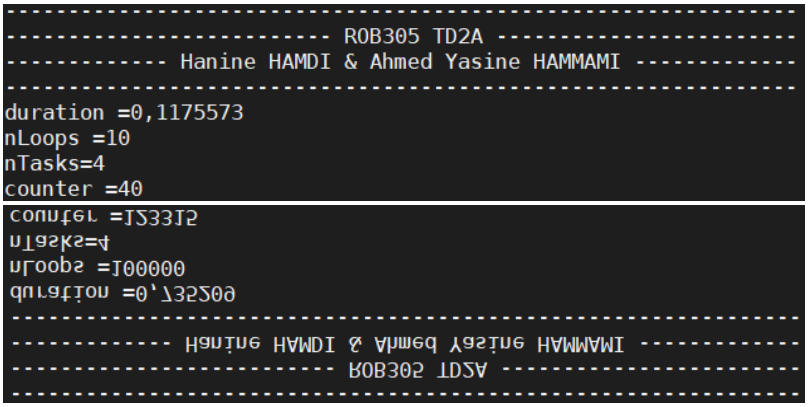
[TD-2] Familiarisation avec l'API multitâches *pthread*

a) Exécution sur plusieurs tâches sans mutex

L'idée de cet exercice est d'ajouter un argument donné en input `nTasks` qui représente le nombre de tâche à exécuter. La fonction `main` doit lancer de tâches qu'indiqué par

`nTasks` par l'exécution de la fonction `call_incr`. Elle doit aussi attendre la fin de l'exécution de toutes les tâches à l'aide de la fonction `pthread_join` et afficher ensuite la durée de l'exécution en millisecondes, la valeur de `nLoops` qui représente le nombre de boucle effectué par les threads, le nombre de taches `nTasks` et la valeur finale de `counter`.

On a lancé ce programme pour différentes valeurs de `nLoops`. On remarque qu'à partir d'une valeur assez grande, la valeur de `counter` qui doit être `nLoops*nTasks` n'est plus correctes. Les figures suivantes montrent ce résultat.



Ceci est dû à l'utilisation des variables partagées en même temps par les threads et par conséquent la variables `counter` est modifiée par les threads simultanément et donc sa valeur de sortie est fausse.

b) Mesure de temps d'exécution

Dans cette partie, on va exécuter le code développé dans la partie précédente en ajoutant un ordonnancement en temps réel qui sera un paramètre en ligne de commande. On doit donc mentionner la politique d'ordonnancement parmi SCHED_RR, SCHED_FIFO et SCHED_OTHER, si on ne spécifie pas la politique d'ordonnancement, le programme s'exécute par défaut avec SCHED_OTHER.

```
int schedPolicy = SCHED_OTHER;

if (argc > 3)
{
if (std::string(argv[3]) == "SCHED_RR")
 {
     schedPolicy = SCHED_RR ;
 }

else if (std::string(argv[3]) == "SCHED_FIFO")
 {
     schedPolicy = SCHED_FIFO ;
 }

else
{
  std::cout << "Unknown scheduling policy : " << argv[3] << std::endl;
  return 1;
}
}
```

On va appliquer à la fonction main un ordonnancement maximal dans le cas il est de type temps réel, sinon SCHED_OTHER va prendre une valeur d'ordonnancement égale à 0.

```
if(schedPolicy == SCHED_OTHER) {
    schedParam.sched_priority = 0;
}
else {
    schedParam.sched_priority = 9;
}
```

Le résultat de l'exécution est le suivant :

```
------------------------------------------------------------------
------------------------- ROB305 TD2B ----------------------------
------------- Hanine HAMDI & Ahmed Yasine HAMMAMI -------------
------------------------------------------------------------------
Scheduling type : SCHED_FIFO
Duration =0,715886
nLoops =10000000
counter =1.1753e+07
rob305-pi48#
```

```
------------------------------------------------------------------
------------------------- ROB305 TD2B ----------------------------
------------- Hanine HAMDI & Ahmed Yasine HAMMAMI -------------
------------------------------------------------------------------
Scheduling policy : SCHED_OTHER
Duration =0,705989
nLoops =10000000
counter =1.14969e+07
rob305-pi48#
```

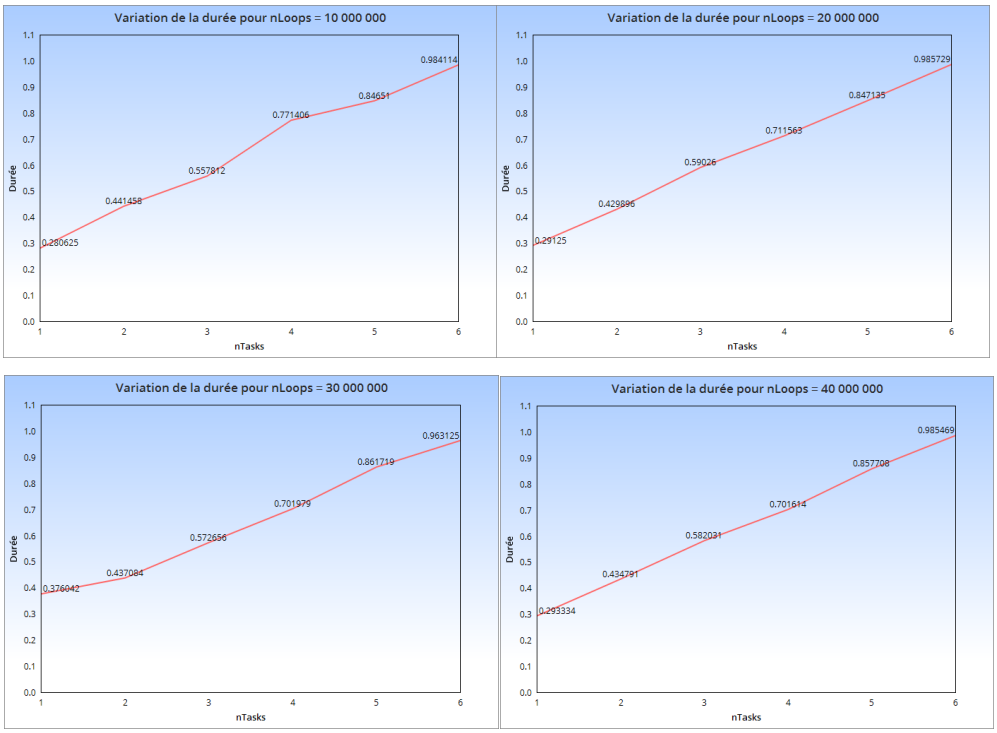
```
------------------------------------------------------------------
------------------------- ROB305 TD2B ----------------------------
------------- Hanine HAMDI & Ahmed Yasine HAMMAMI -------------
------------------------------------------------------------------
Scheduling type : SCHED_RR
Duration =0,713750
nLoops =10000000
counter =1.11739e+07
rob305-pi48#
```

On va ensuite lancer le programme avec la politique d'ordonnancement SCHED_RR pour des valeurs de nLoops $\in \{10^7, 2\times 10^7, 3\times 10^7, 4\times 10^7\}$ et nTasks $\in \{1,2,3,4,5,6\}$.

Le temps d'exécution est représenté par le tableau suivant :

nTasks / nLoops	1	2	3	4	5	6
10^7	0,280625	0,441458	0,557812	0,771406	0,846510	0,984114
2×10^7	0,291250	0,429896	0,590260	0,711563	0,847135	0,985729
3×10^7	0,376042	0,437084	0,572656	0,701979	0,861719	0,963125
4×10^7	0,293334	0,434791	0,582031	0,701614	0,857708	0,985469

Variation du temps d'exécution en fonction du nTasks :



On remarque que le temps d'exécution augment d'une façon presque linéaire pour le même nombre d'itération et en faisant varier le nombre de tâches. Par contre, on remarque que pour chaque nTasks, le temps d'exécution est presque le même en modifiant le nombre d'itération.

Exécution sur plusieurs tâches avec mutex

Dans cette partie, on va ajouter un argument `protected` au programme pour spécifier si l'incrémentation se fait d'une manière protégée par un mutex ou non. Pour ce faire, on a ajouté à la structure `incrData` deux variables `protected_test` qui est initialisée à `false` par défaut, et un pointeur vers `pthread_mutex_t *mutex`.

```
struct incrData {
  // protected mode or not
  bool protected_test = false;
  unsigned int nLoops;
  double counter;
  unsigned nTasks;
  pthread_mutex_t *mutex ;
};
```

La fonction `call_inc` est modifiée comme suit pour implémenter le mutex et protéger la variable partagée :

```
void* call_incr(void* v_data) {
  incrData* p_data = (incrData*) v_data;
  if (p_data->protected_test == false)
  {
      incr(p_data->nLoops, &p_data->counter);
  }
  else if (p_data->protected_test == true)
  {
      pthread_mutex_lock(p_data->mutex);
      incr(p_data->nLoops, &p_data->counter);
      pthread_mutex_unlock(p_data->mutex);
  }
```

On a exécuté le programme avec le mode protégé et le mode normal et on a eu ces résultats :

```
------------------------------------------------------------
------------------------- ROB305 TD2C ----------------------
------------ Hanine HAMDI & Ahmed Yasine HAMMAMI ------------
------------------------ Normal Mode -----------------------
------------------------------------------------------------
duration =0,801719
nLoops =10000000
counter =1.13413e+07
rob305-pi48#
```

```
rob305-pi48# ./test_td2c 10000000 4 true
------------------------------------------------------------
------------------------- ROB305 TD2C ----------------------
------------ Hanine HAMDI & Ahmed Yasine HAMMAMI ------------
------------------ Mutex Protected Mode --------------------
------------------------------------------------------------
duration =0,709740
nLoops =10000000
counter =4e+07
rob305-pi48#
```

On remarque que la valeur finale de `counter` est correcte dans le `Mutex Protected Mode` est égale à `nLoops*nTasks`. La durée d'exécution est inférieure à celle dans le mode normal. On peut conclure que les threads prennent plus du temps à accéder à une variable partagée non protégée. La protection des variables par le mutex accélère le programme et donnent un résultat correct.

[TD-3] Classes pour la gestion du temps

a) Classe Chrono

Pour ce troisième TD, on a voulu traiter les problématiques du premier TD concernant la gestion du temps en utilisant un modèle orienté objet. On a commencé par créer la classe Chrono qui permet de repérer le temps écoulé. Son layout est le suivant :

```
class Chrono
{
    private :
       timespec startTime_ ;
       timespec stopTime_ ;
    public :
       Chrono() ;
       void stop() ;
       void restart() ;
       bool isActive() const ;
       double startTime() const ;
       double stopTime() const ;
       double lap() const ;
};
```

Un objet Chrono est caractérisé par un startTime_ et un stopTime_ qui sont tout les deux des temps Posix. Il est un chronomètre proprement dit : on peut l'arrêter, le lancer, le relancer et déterminer l'instant de son début et de sa fin.

Ici, on veille à respecter la logique de la programmation orientée objet : tous les attributs sont déclarés comme privés et en y accède en utilisant des accesseurs déclarés comme publics (ici startTime() et stopTime()).

Afin de tester la validité de cette classe, on écrit un programme qui prend comme argument un temps d'attente, on crée un objet Chrono, on effectue un délai d'attente (sleep(waitTime_ms)) puis on arrête le chrono et en affiche la durée qu'il a repéré. On obtient le résultat suivant :

```
rob305-pi48# ./TD3a_main_test 1
-----------------------------------------------
---------------------- ROB305 TD3A --------------
--------- Hanin Hamdi & Ahmed Yassine Hammami -----
-----------------------------------------------
waiting for  1  seconds
Done! Chrono time =  1000.16  millisecondes
```

On trouve que la chrono respecte quasiment le même temps d'attente assigné.

b) Classe Timer

L'étape suivante est de transformer les traitements avec les timers Posix du TD1 b et c en une version orientée objet. Pour cela, on commence par construire une classe abstraite afin de créer un polymorphisme : à partir de cette classe Timer, on va créer des classes filles spécialisées chacune dans une tâche précise spécifiée par leur fonction handler. Le layout de la classe abstraite Timer est le suivant :

```
class Timer
{
    public :
       Timer() ;
       ~Timer() ;
       void start(double duration_ms) ;
       void stop() ;
    protected :
       virtual void callback() = 0 ;
    private :
       static void call_callback(int, siginfo_t* sig, void*) ;
    protected :
    timer_t tid ;
    private :
       struct sigaction sa ;
       struct sigevent sev ;
};
```

Protected : Ici l'identifiant du timer **tid** et la fonction virtuelle pure **callback()** doivent être accessibles par les classes filles qui hérite de la classe Timer d'une façon publique. Au même temps, ils ne doivent pas être accessibles par les classes qui n'héritent pas de Timer ou qui ne sont pas ses amies. Ceci explique le choix de les mettre en protected.

Public : Les méthodes déclarée comme publiques sont accessibles depuis n'importe quelle classe ou fonction de l'environnement. Les constructeur et destructeurs **Timer()** et **~Timer()** doivent être visibles depuis typiquement la fonction main, donc elles doivent être déclarées comme publiques. De même on a besoin d'accéder aux méthodes **stop()** et **start()** depuis le main() pour pouvoir gérer les objets de type Timer.

Private : Les attributs **sa** et **sev** ne sont utilisé qu'à l'intérieur de la classe Timer à l'intérieur du constructeur, donc elles sont propres à l'objet Timer et ne doivent jamais être modifiées depuis l'extérieurs, du coup, on les déclare comme étant privées. C'est la même raison pour laquelle on déclare la méthode call_callback() comme privée : on ne doit jamais l'appeler depuis l'extérieur, elle est appelée automatiquement depuis le constructeur pour qu'elle appelle le callback() après l'expiration du Timer.

Comme on vient de le mentionner, la classe Timer est une classe générique abstraite qui définit les méthodes communes à tous les timers. Par contre, la grande différence entre un timer et un autre est le traitement effectué au niveau de la fonction callback() : chaque timer effectue un traitement propre à lui suite à son expiration. Ainsi, on devra déclarer la fonction callback() comme étant virtuelle pure.

Enfin, pour pouvoir accéder par référence à la fonction callback() à l'intérieur des classe filles de Timer, il faut implémenter une fonction propre à chaque timer interne qui permet d'appeler la fonction callback et ceci en utilisant une référence sur l'objet timer. Pour cette raison, on a ajouté la fonction de classe call_callback() qui appelle la fonction callback(). Une implémentation de cette fonction est comme suit :

```
void Timer::call_callback (int, siginfo_t* si, void*)
{
    Timer* pTimer = static_cast <Timer*>(si->si_value.sival_ptr) ;
     pTimer->callback() ;
}
```

L'idée est d'extraire un pointer sur l'objet timer pour pouvoir accéder à sa fonction callback().

Ensuite, on construit une classe fille PeriodicTimer qui hérite d'une façon publique de la classe Timer. En fait, comme le nom le dit, un PeriodicTimer est un Timer périodique qui se répète. Cette classe n'implémente pas non plus la fonction virtuelle callback(), elle réécrit seulement la fonction start() pour ajuster son its.it_interval :

```
class PeriodicTimer : public Timer
{
    public :
       void start(double duration_ms) ;

};
```

Enfin, pour pouvoir tester ces deux classes, on a dû ajouter une troisième classe qui implémente la fonction callback(). Pour cela, on a créer la fonction CountDown qui hérite d'une façon publique de la fonction PeriodicTimer. Donc, un objet CountDown est un timer périodique qui possède un compteur qu'il décrémente d'une façon périodique au bout de chacune de ses expiration.

Ainsi, le layout de la classe CountDown est le suivant :

```
class CountDown : public PeriodicTimer
{
    public :
      CountDown(unsigned int n_count) ;
      void s_counter(unsigned int n_count) ;
      unsigned int a_counter() ;
    private :
      void callback() override ;
    private :
      unsigned int counter_ ;
};
```

Donc, un objet CountDown possède un attribut privé counter_ qu'il doit décrémenter pour chaque expiration de son horloge, un constructeur qui est le même que le constructeur Timer() avec une initialisation de counter_, une implémentation de la fonction callback() (une simple décrémentation de counter_) et un accesseur et un setter de counter_.

Pour tester le fonctionnement de ces trois classes, on écrit un script C++ qui crée un objet CountDown, initialise son counter_ et sa fréquence depuis la ligne de commande et puis affiche le compteur depuis sa valeur initiale jusqu'à 0 avec la fréquence donnée. On obtient l'affichage suivant : (référez-vous au binaire TD3b_main_test pour voir l'affichage avec le temps d'attente) :

```
rob305-pi48# ./TD3b_main_test 10 1
------------------------------------------------
---------------------- ROB305 TD3B -------------
--------- Hanin Hamdi & Ahmed Yassine Hammami ---
------------------------------------------------
timer created
periodic timer started
counter = 10
counter = 9
counter = 8
counter = 7
counter = 6
counter = 5
counter = 4
counter = 3
counter = 2
counter = 1
counter = 0
timer deleted
```

c) Calibration en temps d'une boucle

Dans cette question, on souhaite refaire le boulot de la question d du TD1 avec un modèle orienté objet. Donc, on veut exécuter une boucle qui incrémente un compteur pendant une durée bien déterminée. Pour ce faire, il faut calibrer cette fonction d'incrémentation en trouvant le nombre de boucles correspondant à la durée demandée. L'idée alors est de créer une classe Calibrator pour réaliser la calibration, une classe Looper pour lancer la boucle d'incrémentation jusqu'à ce qu'elle reçoit un signal d'arrêt et une classe finale CpuLoop qui assimile le fonctionnement demandé : lancer le Looper pour une durée donnée.

On commence par la classe **Calibrator** : un Calibrator est un PeriodicTimer qui stocke au bout de ses expirations successives le nombre de boucles effectuée par une fonction type incr du TD1d durant la période du Calibrator. Donc, chaque objet Calibrator doit avoir un objet Looper propre à lui. Ensuite, une fois le Calibrator arrive à stocker le nombre de mesures souhaité, il réalise la calibration en fixant les valeurs de ses attributs privés a et b. Pour pouvoir accéder à a et b depuis l'extérieur, on a ajouté des accesseurs a_A() et a_B().

Enfin, le layout de la classe Calibrator est le suivant :

```
class Calibrator : public PeriodicTimer
{
     private :
        double a ;
        double b ;
        vector<unsigned int> samples ;
        Looper looper ;
        unsigned int nSamples_ ;
     public :
        Calibrator(double samplingPeriod_ms, unsigned int nSamples) ;
        unsigned int nLoops(double duration_ms) ;
        double a_A() ; // Accessor to private parameter a
        double a_B() ; // Accessor to private parameter b
     protected :
        virtual void callback()  ;
};
```

Ensuite, on la classe **Looper**. Un Looper lance une boucle type incr (TD1d) jusqu'à ce que son attribut doStop devient true. Ensuite, il fournit le nombre de boucles qu'il a effectué depuis son lancement jusqu'à son arrêt à partir de son attribut privé iLoop. Un ajoute un accesseur public à iLoop dit getSample().

Le layout de la classe Looper est le suivant :

```
class Looper
{
    public :
       void runLoop(unsigned int nLoops = UINT_MAX) ;
       unsigned int getSample() ;
       void stopLoop() ;
       void s_doStop(bool stop) ;
    private :
       bool doStop ;
       unsigned int iLoop ;
};
```

Enfin, pour finir l'architecture on ajouter la classe **CpuLoop**. C'est un Looper qui boucle pendant une durée donnée. Pour cela il a besoin de calibrer sont temps, et donc il possède une référence sur le Calibrator du programme pour accéder à a et b.

Le layout de la classe est le suivant :

```
class CpuLoop : public Looper
{
    public :
       CpuLoop(Calibrator& calibrator) ;
       void runTime(double duration_ms) ;
    private :
       Calibrator& calibrator_ ;
};
```

Enfin, pour tester ces trois classes, on écrit un script C++ (TD3c_main.cpp) dans lequel on crée un objet Calibrator pour déterminer a et b en utilisant la durée d'échantillonnage et le nombres de mesures données en ligne de commande. Ensuite, on crée un objet CpuLoop recevant comme référence le calibrator déjà créé. Ensuite, on lance le cpuLoop pendant la durée communiquée lors de l'exécution et on repère le temps réel d'exécution. On obtient le output suivant :

```
rob305-pi48# ./TD3c_main_test 1000 1000 4
------------------------------------------------------------
---------------------- ROB305 TD3C -------------------------
--------- Hanin Hamdi & Ahmed Yassine Hammami --------------
------------------------------------------------------------
timer created
periodic timer started
Measure 1 : 26028470
Measure 2 : 52052953
Measure 3 : 78080264
Measure 4 : 104109782
nLoops 26027133
Measure 5 : 104109783
TEST of the calibration : RealTime Duration = 999.954
timer deleted
```

On trouve bien que grâce à la calibration, le cpuLoop a pu tourné pour presque 1 seconde, la durée donnée en ligne de commande.

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

a) Classe Thread

L'objectif de cette section est de revisiter les notions du TD2a avec une version orientée objets. On commence alors par créer une classe **PosixThread** qui redéfinie les attributs propres aux threads Posix ainsi que leurs fonctionnalités génériques. Cette classe possède deux constructeurs : un constructeurs par défaut, et un constructeur qui permet de créer un thread à partir d'un autre déjà existant. On peut aussi lancé un PosixThread, le joindre avec ou sans timeout à la fin de son exécution, définir sa priorité et sa politique « policy » et on peut vérifier s'il est actif ou non. Bref, le layout de cette classe PosixThread est le suivant :

```
class PosixThread
{
  public :
    class Exception ;
  private :
    pthread_t posixId ;
    pthread_attr_t posixAttr ;
  protected :
    bool isActive ;
  public :
    PosixThread() ;
    PosixThread(pthread_t posixId) ;
    ~PosixThread() ;
    void start(void* (*threadFunc) (void*), void* threadArg) ;
    void join() ;
    bool join(double timeout_ms) ;
    bool setScheduling(int schedPolicy , int priority) ;
    bool getScheduling(int* p_schedPolicy = nullptr , int* p_priority = nullptr) ;
};
```

Ensuite, on ajouter une classe fille abstraite **Thread** qui hérite d'une façon publique de PosixThread. Entre autres, un thread possède un startTime_ms_, un stopTime_ms_, un attribut started_ pour savoir s'il a déjà été lancé ou pas. On peut aussi déterminer son temps d'exécution et on peut le faire dormir pendant une certaine durée. Un Thread possède une fonction virtuelle pure run() qui devra être implémentée par ses classes filles pour spécifier leurs traitement. Par contre, comme on a fait avec la classe timer, on a ajouté une fonction de classe call_run appelée automatiquement à l'intérieur d'un Thread pour qu'elle appelle à son tour la fonction run().

Le layout de cette classe Thread est alors le suivant :

```
class Thread : public PosixThread
{
   private :
     double startTime_ms_ ;
     double stopTime_ms_ ;
     bool started_ ;
     pthread_t posixId ;
     pthread_attr_t posixAttr ;
   public :
     Thread() ;
     ~Thread() ;
     bool start() ;
     double startTime_ms() ;
     double stopTime_ms() ;
     void stopTime() ;
     double execTime_ms() ;
     static void sleep_ms(double delay_ms) ;
   protected :
     virtual void run() = 0 ;
   private :
        static void* call_run(void* v_thread) ;
};
```

Enfin, pour pouvoir tester le fonctionnement de ces deux classes, on implémente une troisième classe **IncrementorThread** qui hérite de Thread et qui implémente la fonction run(). Un IncrementorThread est un Thread qui incrémente un compteur qui lui est communiqué par référence. Pour vérifier les priorité appliquées, on ajoute à chaque objet IncrementorThread un compteur local localCounter_ propre à lui qui permet de déterminer les tentatives d'incrémentation que le thread a réellement été autorisé de faire du point de vue priorité.

Le layout de la classe est le suivant :

```
class IncrementorThread : public Thread
{
  private :
    volatile unsigned int* counter_ ;
    unsigned int localCounter_ ;
    volatile bool* pStop_ ;
  private :
    void run() override ;
  public :
    IncrementorThread(volatile unsigned int* counter, volatile bool* pStop) ;
    unsigned int a_localCounter() ;

};
```

Enfin, pour tester ces trois classes, on crée un programme dans lequel on crée trois IncrementorThreads qui reçoivent tous la même référence d'un compteur. Ensuite, on fixe les priorité tel que :

- Le thread 1 a une priorité = 10
- Le thread 2 a une priorité = 15
- Le thread 3 a une priorité = 20

Ensuite, on lance les trois threads et on affiche à la fin de l'exécution, la valeur finale du compteur commun, les valeurs finales des trois compteurs locaux propres à chaque thread et le temps d'exécution de chaque thread. On obtient le output suivant :

```
----Init thread -----
----- First thread initiated ------
----Init thread -----
----- Second thread initiated ------
----Init thread -----
----- Third thread initiated ------
---- First thread isActive = 0
---- Scond thread isActive = 0
---- Third thread isActive = 0
---Starting Thread ---
Waiting ...
----Thread started ----
----- First thread started -----
---Starting Thread ---
Waiting ...
----Thread started ----
----- Second thread started ----
---Starting Thread ---
Waiting ...
----Thread started ----
----- Third thread started -----
Type 's' to stop the threads s
---- First thread joined -----
---- Second thread joined ----
---- Third thread joined -----
---- Execution time of Thread 1 = 9641.24
---- Execution time of Thread 2 = 9640.15
---- Execution time of Thread 3 = 9629.06
----------------- TEST PRIORITY ------------------------
---- Incrementations done by THREAD 1 = 26659933
---- Incrementations done by THREAD 2 = 38974545
---- Incrementations done by THREAD 3 = 40135703
---- Final value of the commun counter = 37658448
---PosixThread destroyed----
---PosixThread destroyed----
---PosixThread destroyed----
rob305-pi48#
```

Enfin, on retrouve l'effet des priorités imposées ; Le compteur local du thread3 contient la plus grande valeur et celui du thread 1 contient le plus petite valeur.

b) Classes Mutex et Mutex::Lock

Dans cette partie, on va créer une classe Mutex qui simule la gestion multitâche en s'inspirant du Java. Cette classe possède les méthodes qui permettent de protéger et synchroniser les variables partagées entre les threads. La définition de la classe Mutex est la suivante :

```
class Mutex
{
    public :
       Mutex() ;
       ~Mutex() ;
       class Monitor ;
       class Lock;
       class TryLock;
    protected :
       pthread_mutex_t posixId ;
       pthread_cond_t posixCondId;
       bool lock();
       bool lock(double timeout_ms);
       bool trylock();
       bool unlock();
};
```

On va définir aussi des classes Mutex::Monitor, Mutex::Lock et Mutex::TryLock.

La classe Mutex::Monitor permet de gérer des opérations relatives à l'utilisation d'une condition :

```
class Mutex::Monitor
{
    public :
      class TimeoutException;
      void wait();
      bool wait(double timeout_ms);
      void notify();
      void notifyAll();
      Monitor(Mutex& m);
      Mutex& mutex_ ;
};
```

Les méthodes wait() et wait(double timeout_ms) utilisent les fonctions pthread_cond_timedwait() et pthread_cond_wait() qui bloquent le thread en tenant compte d'une condition. Elles sont appelées avec le mutex verrouillé par le thread.

Les méthodes notify() et notifyAll() utilisent les fonctions pthread_cond_signal(pthread_cond_t *cond) et pthread_cond_broadcast(pthread_cond_t *cond) qui permettent de débloquer les threads bloqués sur une variable de condition.

L'appel de notify() débloque au moins un des threads qui sont bloqués sur la variable de condition.

L'appel de notifyAll() débloque tous les threads actuellement bloqués sur la variable de condition.

La classe Mutex ::Lock permet de verrouiller le mutex en appelant la fonction pthread_mutex_lock().

Cette classe implémente la fonction pthread_mutex_timedlock() en passant la variable timeout_ms dans le constructeur de cette classe. Cette fonction permet de verrouiller l'objet mutex. Si le mutex est déjà verrouillé, le thread appelant se bloque jusqu'à ce que le mutex devienne disponible comme dans la fonction pthread_mutex_lock ().

Si on dépasse le délai timeout_ms, on interrompe le déverrouillage et on retourne une exception de type TimeoutException.

```
class Mutex::Lock : Mutex::Monitor
{
    public :
        Lock(Mutex &m);
        Lock(Mutex &m, double timeout_ms) ;
       ~Lock();
};
```

La classe Mutex ::TryLock utilise la fonction pthread_mutex_trylock () qui est équivalente à pthread_mutex_lock (), sauf que si le type de mutex est PTHREAD_MUTEX_RECURSIVE et que le mutex appartient actuellement au thread appelant, le nombre de verrous de mutex doit être incrémenté de un et la fonction pthread_mutex_trylock () va renvoyer zéro en cas de réussite de verrouillage. Sinon, une exception TimeoutException est retournée.

```
class Mutex::TryLock : public Mutex::Monitor
{
   public :
       TryLock( Mutex &m) ;
       ~TryLock();
};
```

La classe Monitor ::TimeoutException est définie comme suit :

```
class Mutex::Monitor::TimeoutException
{
   public :
     char* throwException() ;
} ;
```

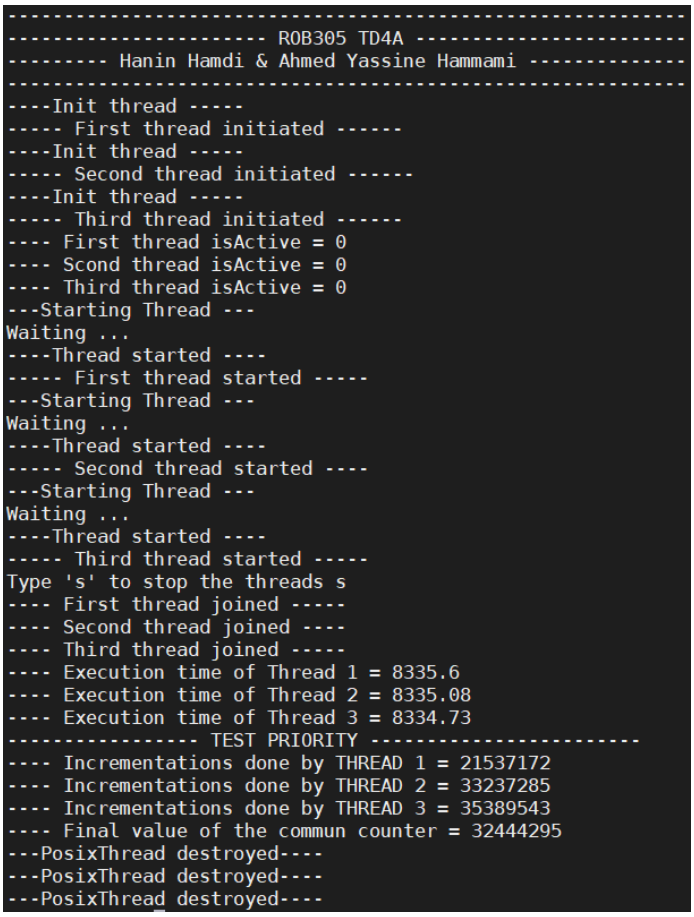
Test de la classe :

Pour tester la classe Mutex, on va se baser dans une première étape sur le code développé dans la partie TD2c. On a changé la variable pthread_mutex_t *mutex par un objet instancié de la classe Mutex. Les fonctions de lock et unlock utilisées sont celle implémenté dans la classe Mutex. Le résultat du programme est le suivant : (le fichier binaire équivalent est test_td4b_bis se trouvant dans le dossier TD4)

```
rob305-pi48# ./test_td4b 10000000 4 true
------------------------------------------------------------
------------------------- ROB305 TD4B ----------------------
------------ Hanine HAMDI & Ahmed Yasine HAMMAMI ------------
------------------- Mutex Protected Mode ---------------------
------------------------------------------------------------
duration =0,718177
nLoops =10000000
counter =4e+07
rob305-pi48#
------------------------------------------------------------
------------------------- ROB305 TD4B ----------------------
------------ Hanine HAMDI & Ahmed Yasine HAMMAMI ------------
------------------------ Normal Mode -------------------------
------------------------------------------------------------
duration =0,807865
nLoops =10000000
counter =1.13789e+07
rob305-pi48#
```

On remarque bien que l'utilisation de l'objet Mutex définit précédemment permet de protéger la variable partagée.

Dans une deuxième étape, on va tester la classe Mutex en protègent l'accès au compteur du programme développé dans la partie TD4a. Le résultat du test est le suivant (fichier binaire test_td4b) :



Deuxième partie : Référez-vous au script de la classe Thread.

c) Classe Semaphore

Les sémaphores sont utilisés pour résoudre de nombreux problèmes de coordination. On va définir dans cette partie une classe sémaphore qui contient la méthode give() permettant d'ajouter un jeton et la méthode take() permettant de lui retirer un jeton. La définition de cette classe est la suivante :

L'attribut maxCount_ représente le nombre maximal de jetons au-delà duquel le sémaphore sature, c'est-à-dire que l'appel de give() ne modifie pas son compteur.

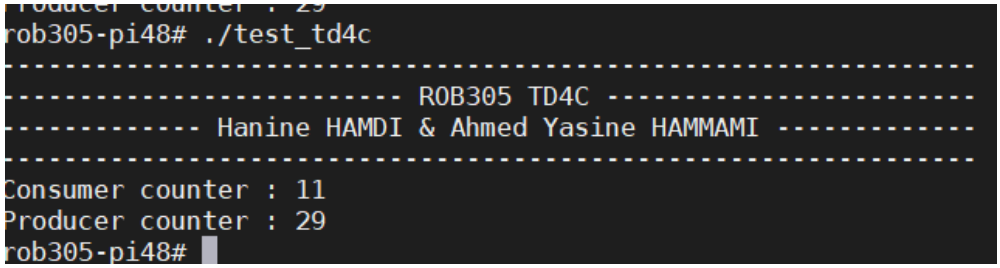
```
class Semaphore
{
    public :
       Semaphore() ;
       Semaphore(unsigned int initCount,unsigned int maxCount) ;
       ~Semaphore() ;
       void take();
       bool take(double timeout_ms);
       void give();
       int getCounter();

    private :
        unsigned int counter_ ;
        unsigned int maxCount_ ;
        Mutex mutex_ ;

};
```

Test de la classe Semaphore :

Pour tester cette classe, on va instancier un sémaphore partagé par deux tâches : consumer et producer. Le compteur est initialisé à 20 avec nCons = 9 et nProd = 9.



d) Classe Fifo multitâches

Dans cette partie, on va définir une classe Fifo qui représente un moyen de communication entre les threads. Cette classe contient la méthode push() qui permet d'ajouter une élément à la pile. Avant de procéder à la tâche d'ajout, on doit appeler les méthodes Mutex ::Monitor ::wait() et Mutex ::unlock pour déverrouiller le mutex et on finit par le verrouiller et appeler la fonction Monitor ::notify() pour débloquer au moins un des threads.

```
    void push(const T& elem) {
        Mutex::Monitor monitor(m_mutex);

        monitor.wait();
        m_mutex.unlock();
        elements.push_back(elem);

        m_mutex.lock();
        monitor.notify();
   }
```

Cette classe contient aussi la méthode pop() qui retire et renvoie le dernier élément de la pile.

```
  // remove next element from the queue and return its value
    T pop() {

        Mutex::Monitor monitor(m_mutex);
        monitor.wait();
        m_mutex.unlock();
        T elem(elements.front());
        elements.pop_front();

        m_mutex.lock();
        monitor.notif();
        return elem;
  }
```

La méthode pop(double timeout_ms) renvoie une ReadEmptyQueue si la pile est vide :

```
T pop(double timeout_ms) {

        Mutex::Monitor monitor(m_mutex);
        monitor.wait(timeout_ms);
        m_mutex.unlock();

        if (elements.empty()) {
            throw ReadEmptyQueue();
        }
        T elem(elements.front());
        elements.pop_front();
        m_mutex.lock();
        monitor.notifyAll();
        return elem;
  }
};
```

Test de la classe :

Pour tester cette classe, on a créé la classe Producer qui permet d'ajouter une série d'entier de 0 à n à l'objet Fifo<int> et la classe Consumer qui permet de lire ces entiers de l'objet Fifo<int>. Ces deux classes sont définis comme suit :

```
class Producer {
public:
  Producer(Fifo<int>& bq, int id):m_bq(bq), m_id(id) {
  }

  void operator()() {
    for (int i = 0; i < 9; i++) {
      m_bq.push(i);
    }
  }
private:
  Fifo<int> &m_bq;
  int m_id;
};

class Consumer {
public:
  Consumer(Fifo <int>& bq, int id):m_bq(bq), m_id(id) {
  }

  void operator()() {
    std::cout << "Reading from queue: \n";
    for (int i = 0; i < 10; i++) {
      int value;
      value = m_bq.pop();
      std::cout << "int Value : "<< value << " \n";
    }
    std::cout << std::endl;
  }
private:
  Fifo<int> &m_bq;
  int m_id;
};
```

On remarque bien que les entiers produits par la tâche productrice ont bien été reçus par la tâche consommatrice :

```
rob305-pi48# ./test_td4d
------------------------------------------------------------
------------------------- ROB305 TD4D ----------------------
------------ Hanine HAMDI & Ahmed Yasine HAMMAMI ------------
------------------------------------------------------------
Waiting on condition variable cond
Waiting on condition variable cond
Reading from queue:
Waiting on condition variable cond
int Value : 0
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 1
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 2
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 3
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 4
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 5
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 6
Waiting on condition variable cond
Waiting on condition variable cond
int Value : 7
Waiting on condition variable cond
```

[TD-5] Inversion de priorité

Dans cette partie, on a ajouté à la classe Mutex une option de protection contre l'inversion de priorité par héritage de priorité. Cette option est initialisée avec le constructeur et permet de changer le Protocol du thread comme suit :

```
Mutex::Mutex(bool isInversionSafe)
{
    posixId = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
    posixCondId =  PTHREAD_COND_INITIALIZER;
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    //Protected from Inversion
    if(isInversionSafe)
    {
        pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_PROTECT);
    }
    else {
        pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_NONE);
    }
    pthread_mutex_init(&posixId, &attr);
    pthread_mutexattr_destroy(&attr);
}
```