Rapport ENSTA 2022 – ROB305 – Yu WANG

Ceci est un rapport de tous les TDs du cours ROB305 à l'ENSTA Paris. On a mis en œuvre deux APIs POSIX étape par étape. Tous les codes présentés dans ce rapport peuvent être trouvés sur ce GitHub : https://github.com/Rescon/ROB305

- La première API POSIX est la gestion du temps, qui est chargée de mesurer le temps d'exécution à l'aide d'une Timer qui peut bloquer/débloquer les programmes après un certain temps prédéfini.
- La seconde API POSIX est la gestion des programmes multitâches, qui sera responsable de la création, de la destruction des tâches, de la gestion de la concurrence des données en les protégeant des accès concurrents, de l'activation ou du blocage des tâches.

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

Deux bibliothèques <ctime> et <csignal> sont utilisées pour implémenter ce TD.

a) Gestion simplifiée du temps Posix

Le but de ce premier exercice est d'implémenter des fonctionnalités permettant la gestion du temps Posix. Une série de fonctions a été créée pour permettre une transition facile entre le temps en millisecondes et le temps Posix dans la structure **timespec**.

Ensuite, pour mieux gérer les variables de type struct **timespec**, on a implémenté les opérateurs suivants : +, -(soustraction), -(négation), +=, -=, ==, ==, ! =, < et >.

La structure **timespec** est constituée de deux valeurs, *tv_sec* et *tv_nsec*, correspondant respectivement aux valeurs en secondes et en nanosecondes, la convention étant que la valeur en nanosecondes est toujours positive ou égale à zéro.

Dans la fonction main du fichier main_td1a.cpp, on a testé chacune des méthodes et des opérateurs créés.

b) Timers avec callback

Dans ce problème, on implémente un timer Posix périodique avec une fréquence de 2Hz dont le but est d'incrémenter périodiquement la valeur d'un compteur en l'imprimant. Le programme s'arrête après 15 incréments. L'implémentation se fait en déclarant une fonction myHandler (illustrée ci-dessous), qui est exécutée à la fin de chacune des périodes déterminées dans la définition du timer.

La définition du timer se compose de deux parties : *it_value* indique le délai du premier cycle et *it_interval* indique l'intervalle entre deux cycles. À la fin de chaque cycle, la fonction Myhandler sera exécutée et le compteur, dont la valeur initiale est 0, sera incrémenté d'une unité chaque fois que le gestionnaire sera appelé. Ainsi, dans ce cas, *it_interval.tv_nsec* est fixé à 5e8 (correspondant à une fréquence de 2Hz) et les autres paramètres sont fixés à 0. On donne la valeur à la fonction Myhandler via le pointeur si contenant les informations relatives au signal.

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

Au moment de l'exécution, on a remarqué que si on ne spécifie pas le mot-clé volatile lors de la déclaration des variables, des problèmes peuvent survenir dans le cas d'optimisations de compilation.

En effet, si on utilise l'option de compilation -O3, le compilateur ne tiendra pas compte de la possibilité que la variable *counter* soit modifiée par d'autres processus s'exécutant en parallèle avec main.

En spécifiant le mot-clé volatile, on informe le compilateur que cette variable peut également être modifiée par d'autres processus s'exécutant en parallèle et on aura la possibilité d'optimiser la compilation sans problème d'exécution du programme : la valeur du compteur sera augmentée à la valeur correcte et le programme s'arrêtera.

c) Fonction simple consommant du CPU

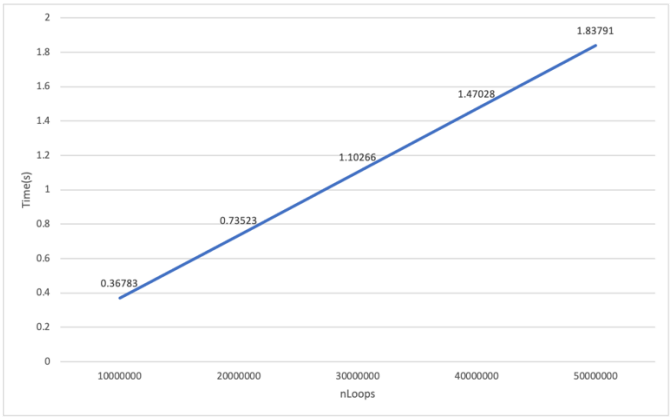
Dans cette section, on impléme une fonction de signature simple qui consomme le CPU : *void incr(unsigned int nLoops, double* pCounter)* (illustré ci-dessous). Cette fonction parcourt *nLoops* en boucle un certain nombre de fois. Et dans chaque boucle, il incrémente de 1,0 la valeur du compteur pointé par *pCounter*. La fonction *timespec_now()* implémentée dans timespec.cpp est utilisée pour mesurer la valeur du temps d'exécution.

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

La signature standard de main est *main(int argc, char* argv[])*, où *argc* indique le nombre de chaînes données au programme et *argv[]* est le tableau qui les contient. On peut passer des arguments au programme avec son aide (comme indiqué ci-dessous).

```
if (argc != 2)
{
    std::cerr << "Error: number of arguments. 1 expected : <nLoops>" <<
std::endl;
    exit(-1);
}
std::istringstream iss(argv[1]) ;
loop_t nLoops = 0 ;
iss >> nLoops ;
```

On a testé le programme en utilisant différents paramètres et les résultats sont présentés ci-dessous. On peut constater que le temps d'exécution du programme est approximativement linéaire par rapport à *nLoops*.



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

Dans ce problème, on utilise la fonction *incr* précédente avec quelques modifications. On change la condition d'arrêt : il s'arrête si l'une des conditions suivantes est remplie:

- Le nombre maximal de boucles est atteint.
- Un paramètre booléen comme référence à stop est mis à true.

La nouvelle implémentation de *incr* est la suivante:

```
using loop_t = unsigned long int;
loop_t incr(loop_t nLoops, double* pCounter, const bool* pStop)
{
    loop_t iLoop;
    for(iLoop = 0; iLoop < nLoops; ++iLoop)
    {
        if (*pStop) break;
        *pCounter += 1.0;
    }
    return iLoop;
}
```

On veut ensuite gérer l'arrêt de l'incrémentation via un timer : lorsqu'il expire, sa fonction callback mis la variable *pStop* à true, donc *incr* s'arrête. Ainsi, ici, la variable *pStop* est accessible à la fois par *incr* et *myHandler*. Cela nécessite que *pStop* soit déclaré comme une variable volatile. La nouvelle implémentation de *myHandler* est la suivante:

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

Soit $l(t)$ le nombre de boucles effectuées par la fonction incr durant l'intervalle de temps t ; on suppose que cette fonction est affine : $l(t)=a*t+b$. On a donc construit une fonction *calib* pour calculer les valeurs a(pente) et b(constante).

On calcule ces deux paramètres en obtenant le nombre d'exécutions de boucles pour des temps d'exécution de 4 et 6 secondes respectivement. Ceci est ensuite vérifié en utilisant un temps d'exécution de 10 secondes et le résultat final est montré ci-dessous.

```
Calculate the parameters of the calibration
Results : a = 2.13857e+07, b = -35841
```

```
Test the parameters of the calibration (for 10 seconds):
The actual number of cycles executed: 2.13792e+08
The calculated number of cycles executed: 2.13821e+08
Accuracy : 99.9862%
```

e) Amélioration des mesures

Afin d'améliorer la précision de la fonction, j'ai augmenté le nombre de calibrations dans le programme principal pour pouvoir calibrer les valeurs de a et b avec de meilleures approximations. Cela nous permet d'obtenir plus d'informations sur le temps d'exécution et le nombre de cycles exécutés. Cela nous permet d'appliquer la formule de régression linéaire pour calculer les valeurs de a et b (comme indiqué ci-dessous).

```
std::vector<double> x(num), y(num);
for (int i = 0; i<num; ++i)
{
    std::cout << "Calibrator task : " << i+1 << std::endl;
    x[i] = i+1;
    y[i] = (double)measure(i+1, 0);
}

double sum_x = 0, sum_xx = 0, sum_y = 0, sum_xy = 0;
for (int i = 0; i<num; ++i)
{
    sum_x += x[i];
    sum_y += y[i];
    sum_xx += x[i]*x[i];
    sum_xy += x[i]*y[i];
}

para.a = (num * sum_xy - sum_x * sum_y) / (num * sum_xx - sum_x * sum_x);
para.b = (sum_y - para.a * sum_x) / num;
```

Le résultat final est montré ci-dessous et l'estimation est plutôt satisfaisante.

```
Test the parameters of the calibration (for 10 seconds):
The actual number of cycles executed: 2.13794e+08
The calculated number of cycles executed: 2.13815e+08
Accuracy : 99.99%
```

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

Dans cet exercice, on se familiarise progressivement avec l'utilisation de l'API multitâche pthread.

a) Exécution sur plusieurs tâches sans mutex

Dans cette section, on essaye de créer plusieurs threads pour implémenter une variable de compteur incrémentielle. Cela nous permet d'observer un problème lié à l'accès aux données par plusieurs processus en parallèle.

Pour cela, on doit d'abord créer la fonction *void* call_incr(void* v_data)* (ndiqué ci-dessous). Comme arguments, la structure *v_data* correspond à un *unsigned long int nLoops* pour le nombre de boucles et un *double counter* pour la valeur du compteur.

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

Cette fonction et les arguments sont utilisés dans la fonction *pthread_create()* pour démarrer le thread *nTasks* (où *nTasks* est passé comme argument).

```
// Create nTasks threads
std::vector<pthread_t> incrementThread(nTasks);
for (unsigned int i=0; i<nTasks; ++i)
{
    pthread_create(&incrementThread[i], NULL, call_incr, (void*)&data);
}

// Join nTasks threads
for (unsigned int i=0; i<nTasks; ++i) {
    pthread_join(incrementThread[i], NULL);
}
```

En fait, si on choisi de lancer 5 threads, chacun itérant 10 000 000 de fois, on doit obtenir une valeur de compteur de 50 000 000, mais ce n'est pas le cas (Les résultats sont présentés ci-dessous.). Ce problème est causé par plusieurs processus accédant à la même zone de mémoire. Le MUTEX va être utilisé dans les sections suivantes pour résoudre ce type de problème.

```
nLoops : 10000000
nTasks : 5
Counter : 1.50998e+07
```

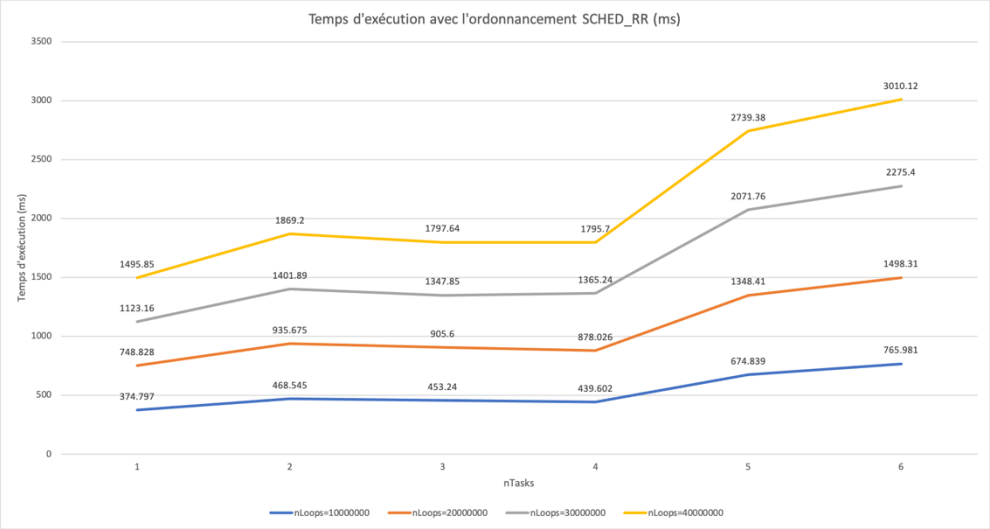
b) Mesure de temps d'exécution

Dans cette section, on va exécuter le code développé dans la section précédente en ajoutant un planificateur en temps réel, qui sera un argument de ligne de commande. La politique d'ordonnancement doit être spécifiée dans SCHED_RR, SCHED_FIFO et SCHED_OTHER. Si aucune politique d'ordonnancement n'est spécifiée, le programme s'exécutera avec SCHED_OTHER par défaut (comme ci-dessous).

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

En exécutant le programme avec l'ordonnancement SCHED_RR et en utilisant la fonction timespec_now() dans td_1 pour des valeurs de $nLoops \in \{10^7, 2*10^7, 3*10^7, 4*10^7\}$ et $nTasks \in \{1, 2, 3, 4, 5, 6\}$, on obtiens les résultats présentés ci-dessous.

Temps d'exécution avec l'ordonnancement SCHED_RR (ms)				
	10^7	$2*10^7$	$3*10^7$	$4*10^7$
1	374.797	748.828	1123.16	1495.85
2	468.545	935.675	1401.89	1869.2
3	453.24	905.6	1347.85	1797.64
4	439.602	878.026	1365.24	1795.7
5	674.839	1348.41	2071.76	2739.38
6	765.981	1498.31	2275.4	3010.12



On peut observer qu'avec jusqu'à quatre threads en parallèle, il n'y a pas d'augmentation significative du temps d'exécution. Cependant, si l'on continue à augmenter le nombre de threads, le temps d'exécution augmentera de manière significative. Cela est dû au fait que Raspberry ne possède que quatre cœurs. Si on augmente le nombre de threads, plusieurs threads devront tourner sur le même cœur.

c) Exécution sur plusieurs tâches avec mutex

Dans cette section, on va résoudre le problème rencontré dans la partie td_2a : on va utiliser Mutex pour gérer les accès parallèles à la variable compteur par plusieurs processus. Si l'option *protect* est spécifiée dans la ligne d'exécution du programme, alors l'accès à la variable compteur sera protégé par un mutex, ce qui garantira qu'un seul processus pourra la modifier à la fois.

Pour ce faire, on doit ajouter une variable de type *pthread_mutex_t* aux processus lors de leur création, qui sera utilisée pour donner à un processus particulier le droit de modifier la valeur du compteur. La fonction *pthread_mutex_lock* et *pthread_mutex_unlock* sont utilisées (comme ci-dessous).

```
void* call_incr(void* v_data)
{
    Data* p_data = (Data*) v_data;
    if (p_data->protect) {
        pthread_mutex_lock(&p_data->mutex);
        incr(p_data -> nLoops, &p_data -> counter);
        pthread_mutex_unlock(&p_data->mutex);
    } else {
        incr(p_data -> nLoops, &p_data -> counter);
    }
    return v_data;
}
```

On peut voir que sous la protection du mutex, la valeur finale du compteur est égale à *nLoops * nTasks*, ce qui signifie que le problème est bien résolu. En ce qui concerne le temps d'exécution, on peut noter que l'exécution du programme sans mutex est plus rapide.

[TD-3] Classes pour la gestion du temps

Dans cet exercice, on programme quelques classes pour la gestion du temps. On va traiter les problématiques du premier TD concernant la gestion du temps en utilisant un modèle orienté objet.

a) Classe Chrono

Dans cette section, on implémente la classe Chrono, qui met en œuvre la fonction de mesure du temps d'un chronomètre. Il peut être arrêté, démarré, redémarré et ses heures de début et de fin peuvent être déterminées.

Un objet Chrono est caractérisé par un *startTime_* et un *stopTime_*, qui sont tous deux des temps Posix. Ici, on prend soin de respecter la logique de la programmation orientée objet : toutes les propriétés sont déclarées privées et accessibles à l'aide d'accesseurs déclarés publics (dans ce cas, *startTime()* et *stopTime()*).

La classe est testée dans le fichier main_td3a.cpp et les valeurs de temps données par Chrono et timespec_now sont comparées pour vérifier la fonctionnalité de la classe.

b) Classe Timer

L'étape suivante consiste à traduire le traitement du timer Posix avec les td_1b et c en une version orientée objet. Pour ce faire, on crée d'abord une classe abstraite (la classe Timer) qui encapsule les fonctions du timer Posix et établit le polymorphisme. À partir de cette classe de Timer, on va créer des classes filles, chacune se spécialisant dans la tâche précise spécifiée par sa fonction de gestion.

Dans cette classe, le constructeur, le destructeur, la méthode *start()* et la méthode *stop()* sont Public, car ils doivent être appelés dans main (en dehors de la classe). L'opération *callback()* est Protected, car elle est virtuelle et sera implémentée par les sous-classes qui en héritent. Elle ne doit pas être utilisée en dehors de Timer ou des classes héritées. La méthode *call_callback()* est Private, car elle n'est utilisée que dans la classe Timer. Elle est définie comme statique car elle peut être utilisée sans instancier un objet de cette classe.

Comme on vient de le mentionner, la classe Timer est une classe abstraite générique qui définit des méthodes communes à tous les timers. En revanche, la plus grande différence entre un timer et un autre est le traitement effectué au niveau de la fonction *callback()* : chaque timer effectue son propre traitement à son expiration. La fonction *callback()* doit donc être déclarée comme purement virtuelle.

Enfin, afin de pouvoir accéder à la fonction *callback()* par référence dans une sous-classe du timer, on implémente une fonction spécifique pour chaque timer interne qui permet d'invoquer la fonction callback en utilisant une référence à l'objet timer. C'est pourquoi on ajoute la fonction de classe *call_callback()*, qui appelle la fonction callback(). L'implémentation de cette fonction est la suivante.

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

On crée ensuite une classe fille PeriodicTimer qui hérite de la classe Timer. En fait, comme son nom l'indique, PeriodicTimer est un minuteur périodique qui se répète. Cette classe n'implémente pas non plus la fonction virtuelle *callback()*, elle surcharge simplement la fonction *start()* pour ajuster son *its.it_interval*.

Enfin, afin de pouvoir tester ces deux classes, on ajoute une troisième classe qui implémente la fonction callback(). Pour ce faire, on crée la classe CountDown, qui hérite de la classe PeriodicTimer, pour imprimer à l'écran un compte à rebours de 1Hz du nombre n à 0. Ainsi, l'objet CountDown est un timer périodique avec un compteur qui se décrémente périodiquement à chaque expiration.

c) Calibration en temps d'une boucle

Dans ce problème, on veut refaire le travail du problème d du td_1 en utilisant un modèle orienté objet. Par conséquent, on exécute une boucle qui incrémente un compteur pendant une durée donnée. Pour ce faire, on doit calibrer cette fonction incrémentale pour trouver le nombre de cycles correspondant à la durée requise.

Tout d'abord, on crée la classe Looper. Un Looper démarre une boucle de type incr (td_1d) jusqu'à ce que sa propriété *doStop* devienne true. Il fournit ensuite le nombre de boucles exécutées depuis qu'il a commencé à s'arrêter à partir de sa propriété privée *iLoop*. La première consiste à ajouter un accesseur public à iLoop appelé *getSample()*.

Ensuite on crée classe Calibrator, qui est une classe fille de PeriodicTimer qui stocke le nombre de boucles exécutées pendant Calibrator par une fonction de type incr à son expiration successive. Par conséquent, chaque objet Calibrator doit avoir son propre objet Looper. Ensuite, une fois que le Calibrator a stocké le nombre requis de mesures, il effectue le calibrage en fixant les valeurs de ses propriétés privées a et b (en utilisant de la régression linéaire).

Enfin, pour compléter l'architecture, on ajoute la classe CpuLoop. C'est un Looper qui boucle pendant un certain temps. Pour ce faire, il doit calibrer son temps, il a donc une référence au Calibrator du programme pour accéder à a et b.

Pour tester ces trois classes, on écrit un script C++ (main_td3c.cpp). Le résultat final est montré ci-dessous et le résultat est parfait.

```
Calculate the parameters of the calibration (for 1 seconds)
nSamples: 10
Results : a = 18143.3, b = -3836.4
Loops used for calculation : 1.81394e+07
Time passed for calculation : 10000.8ms

Test the parameters of the calibration (for 5 seconds)
The actual number of cycles executed: 9.07125e+07
The calculated number of cycles executed: 9.07125e+07
Accuracy : 100%
Time passed for test : 4999.66ms
```

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

Dans cet exercice, on programme quelques classes pour la programmation multitâche. On encapsule la gestion des tâches Posix dans les classes PosixThread, Thread et Mutex.

a) Classe Thread

L'objectif de cette section est de revisiter le concept de td_2a avec une version orientée objet. On commence par créer une classe PosixThread qui contient les paramètres de base d'un thread : son identifiant et ses propriétés, ainsi que les fonctions de base pour créer des threads. Il contient également le constructeur qui sera chargé d'attribuer des priorités et des politiques aux threads créés. En particulier, la méthode start nous permettra de démarrer le thread, et elle doit être Public afin d'être accessible en dehors de la classe. Pour la même raison, *join()*, *setScheduling()* et *getScheduling()* sont également Public.

Ensuite, on ajoute une sous-classe abstraite Thread, qui hérite de PosixThread de manière publique. Entre autres choses, un thread a un timespec *startTime*, un timespec *stopTime*. On peut également déterminer son temps d'exécution, et on peut le faire dormir pendant un certain temps. Un thread possède une fonction purement virtuelle *run()*, qui doit être implémentée par ses sous-classes pour spécifier son traitement. D'autre part,

comme on l'a fait avec la classe timer, on ajoute une fonction *call_run()* qui est automatiquement appelée dans le thread afin qu'il appelle la fonction *run()*.

Enfin, pour pouvoir tester le fonctionnement de ces deux classes, on a implémenté une troisième classe, IncrThread, qui hérite de Thread et implémente la fonction *run()*. Un IncrThread est un thread qui augmente un compteur, qui lui est communiqué par référence.

Dans main_td4a.cpp on teste ces classes : on a toujours le problème d'avoir plusieurs threads accédant aux données en parallèle (comme ci-dessous). En effet, lorsque plusieurs threads sont lancés en même temps, on observe que la valeur du compteur partagé entre eux n'atteint pas la valeur finale attendue. Pour résoudre ce problème, on doit implémenter une classe Mutex qui garantit qu'un seul thread peut compter à la fois.

```
nLoops : 10000000
nTasks : 5
schedPolicy : 0
Counter : 1.35154e+07
Time passed by all the threads : 18562.3 ms
```

b) Classes Mutex et Mutex::Lock

Dans cette section, on implémente des classes qui gèrent les mutex : les classes implémentées sont Mutex, Mutex::Monitor, Mutex::Lock et Mutex::TryLock, qui nous permettent de gérer le blocage/déblocage des mutex pour s'assurer qu'un seul processus peut accéder à une certaine donnée à la fois.

Les classes Mutex::Lock et Mutex::TryLock permettent d'obtenir (bloquer) un mutex. La classe Mutex::Monitor permet de gérer des variables de condition et d'envoyer un signal lorsqu'un thread est bloqué afin d'arrêter un thread qui peut accéder au mutex, d'exécuter un autre thread pendant un certain temps puis de continuer avec le premier thread.

On a testé ces classes dans main_td4b.cpp avec la classe IncrMutex (comme ci-dessous), qui est similaire à la classe IncrThread utilisée dans td_4a. Un pointeur vers un Mutex est utilisé pour protéger l'accès aux données. Comme prévu. L'exécution prend un peu plus de temps que td_4a.

```
nLoops : 10000000
nTasks : 5
schedPolicy : 0
Counter : 5e+07
Time passed by all the threads : 286883 ms
```

c) Classe Semaphore

Dans le contexte multitâche, un sémaphore est une « boite à jetons » à accès concurrent. Dans cet exercice, on implémente une classe Semaphore dont le but est d'initialiser une boîte à jetons afin de pouvoir créer différents threads consommateurs et producteurs qui accèdent au même objet Semaphore et peuvent augmenter ou diminuer le nombre de jetons dans la boîte. Le nombre de jetons sera donné par une variable unsigned int *counter* dans la classe Semaphore.

On a également créé les classes SemaphoreCons et SemaphoreProd, qui ont chacune un pointeur vers le même objet Semaphore, et on crée différents threads où le producteur produit le jeton, c'est-à-dire augmente le nombre de jetons dans la boîte, et le consommateur consomme le jeton, c'est-à-dire le diminue.

Dans main_td4c.cpp, on vérifie également que le nombre de jetons créés est consommé en totalité par le consommateur, et aussi lorsque le nombre de threads du consommateur n'est pas égal au nombre de threads du producteur (comme ci-dessous).

```
Thread (consume) 0 has consumed 10589 tokens.
Thread (consume) 1 has consumed 7627 tokens.
Thread (consume) 2 has consumed 6784 tokens.

Total number of tokens produced : 25000
Total number of tokens consumed : 25000
All the tokens generated were consumed.

Time taken by all the threads that produce tokens : 192.237 ms.
Time taken by all the threads that consume tokens : 163.173 ms.
```

d) Classe Fifo multitâches

Dans cette section, on défini une classe Fifo qui représente la manière dont les threads communiquent entre eux. La classe modèle Fifo permet la création d'une sorte de sémaphore pour n'importe quel type et la boîte sera implémentée par une file d'éléments. La classe contient la méthode *push()*, qui permet d'ajouter un élément à la pile. La classe contient également la méthode *pop()*, qui supprime et renvoie le premier élément de la pile. C'est une classe modèle, il y aura donc des déclarations de fonctions et des implémentations dans le même fichier.

```
template<typename T>
void Fifo<T>::push(T element)
{
    Mutex::Lock lock(mutex);
    elements.push(element);
    lock.notifyAll();
    lock.~Lock();
}

template<typename T>
T Fifo<T>::pop()
{
    Mutex::Lock lock(mutex);
    if (elements.empty())
    {
        lock.wait();
        lock.notifyAll();
        lock.~Lock();
    }
    else
    {
        T popped = elements.front();
        elements.pop();
        lock.notifyAll();
        lock.~Lock();
        return popped;
    }
}
```

Dans ce cas, les classes SemaphoreCons et SemaphoreProd sont remplacées par les classes FifoCons et FifoProd. Dans main_td4d.cpp, on vérifie également que dans ce cas le nombre d'éléments créés est égal au nombre d'éléments détruits.

```
Thread (consume) 0 has consumed 8139 tokens.
Thread (consume) 1 has consumed 9536 tokens.
Thread (consume) 2 has consumed 7325 tokens.

Total number of tokens produced : 25000
Total number of tokens consumed : 25000
All the tokens generated were consumed.

Time taken by all the threads that produce tokens : 407.532 ms.
Time taken by all the threads that consume tokens : 344.226 ms.
```

[TD-5] Inversion de priorité

Dans cette section, on ajoute une option à la classe Mutex pour empêcher les inversions de priorité en héritant des priorités. Cette option est initialisée dans le constructeur et permet de modifier le protocole du thread (comme ci-dessous).

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

On crée la classe CpuLoopMutex pour définir la fonction *run()* qui sera exécutée par le thread. Dans ce cas, si la tâche nécessite un Mutex après un certain nombre de cycles d'horloge, le code présenté ci-dessous effectuera la division du travail entre l'absence et la présence d'un Mutex, comme déterminé dans la diapositive (où le paramètre temps est passé au constructeur de la classe).

```
void CpuLoopMutex::run()
{
    if(timeBeginMutex != -1) // with mutex
    {
        cpuLoop.runTime(tick_to_ms(timeBeginMutex));

        Mutex::Lock lock(mutex, 1000);
        cpuLoop.runTime(tick_to_ms(durationMutex));
        lock.~Lock();

        cpuLoop.runTime(tick_to_ms(timeExecution - (timeBeginMutex +
durationMutex)));
    }
    else // without mutex
    {
        cpuLoop.runTime(tick_to_ms(timeExecution));
    }
}
```

L'inversion de priorité indique que la tache avec le mutex tournera à la priorité maximale de l'ensemble des tâches qui demandent le blocage du mutex. Autrement dit, lorsque la tâche A avec une priorité inférieure empêche le mutex, même si une autre tâche B avec une priorité plus élevée arrive, A a toujours le mutex, et la priorité de la tâche A deviendra la priorité de B.

Étant donné qu'on doit recréer les conditions décrites dans les diapositives, on donne 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 23 des diapositives.

Le processeur étant multi-cœur, pour mettre en évidence l'inversion de priorité, il est nécessaire d'occuper tous les cœurs à 100% sauf un et faire tourner votre TD sur cet unique cœur laissé libre. A l'aide de CPU Affiniy (comme ci-dessus), on a réussi à exécuter notre code en un seul cœur.

```
// CPU setting
cpu_set_t  cpuSetting;
CPU_ZERO(&cpuSetting);
CPU_SET(0, &cpuSetting);
sched_setaffinity(0, sizeof(cpu_set_t), &cpuSetting);
```

Le test a été effectué dans le main_td5.cpp. On peut constater qu'avec isInversionSafe true, le thread A se termine plus tôt (par rapport au cas où isInversionSafe est false).