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

Dans ce TD, il s'agit d'encapsuler la gestion des tâches Posix dans les classes `PosixThread`, `Thread`, `Mutex` et `Lock` en s'inspirant du modèle multitâches du langage Java.

a) Classe Thread

Programmez la classe `PosixThread` en vous référant à l'interface de la **figure 1a** ainsi qu'aux appels Posix du cours ; référez-vous aux manpages des appels Posix suivants pour programmer le 2e constructeur ainsi que les méthodes `getScheduling()` et `setScheduling()` :

- `pthread_getschedparam()`
- `pthread_setschedparam()`
- `pthread_attr_getschedparam()`
- `pthread_attr_setschedparam()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_setschedpolicy()`

En particulier, on peut utiliser les codes d'erreur renvoyés par la fonction `pthread_getschedparam()` pour tester la validité d'un identifiant de thread.

Pour le 2e constructeur, si la tâche Posix n'existe pas, une exception `PosixThread::Exception` devra être produite. Pour les méthodes `getScheduling()` et `setScheduling()`, il faut distinguer les deux cas où le thread est déjà lancé ou pas et les deux méthodes doivent renvoyer `true` si la tâche est active, et `false` si la tâche n'est pas active.

- Le 2e constructeur `PosixThread(posixId : pthread_t)` s'applique à un pthread déjà existant.
- `setScheduling()` applique l'ordonnancement et la priorité spécifiée à la tâche (à l'attribut de tâche et à la tâche elle-même si elle est déjà lancée).
- `getScheduling()` renvoie l'ordonnancement et la priorité de la tâche.
- pour la méthode `join()` avec timeout, utilisez la fonction `pthread_timedjoin_np()` qui n'appartient pas au standard Posix ;

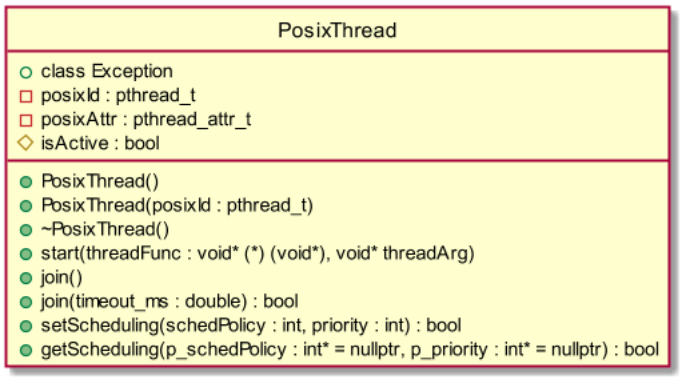

Figure 1a : Classe PosixThread permettant d'encapsuler les fonctions d'une tâche Posix.

Programmez la classe `Thread` dérivant de `PosixThread` en vous référant à l'interface de la **figure 1b** ainsi qu'aux éléments du cours. Les méthodes `startTime_ms()`, `stopTime_ms()` et `execTime_ms()` doivent renvoyer respectivement les temps absolus de début et de fin d'exécution et la durée d'exécution de la tâche (en millisecondes).
En ce qui concerne la méthode statique `sleep_ms()`, il s'agit juste d'une fonction permettant d'endormir le **thread appelant** durant le temps spécifié en millisecondes ; testez votre classe en refaisant le **TD-2a** dans un contexte orienté objets. Imaginez un programme simple permettant de tester tous les aspects de la classe.

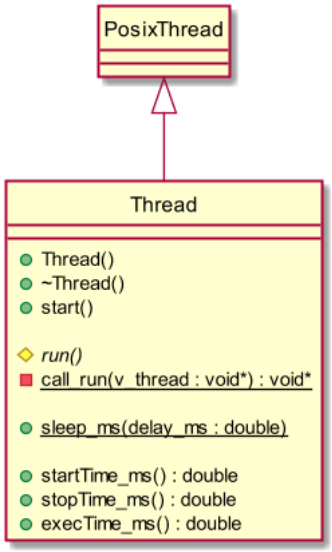

Figure 1b : Classe Thread.

b) Classes Mutex et Mutex::Lock

Programmez les classes `Mutex`, `Mutex::Lock`, `Mutex::TryLock` en reprenant les spécifications de la **figure 2** et les explications du cours. Prenez note, en particulier, que la classe `Mutex` contient l'identifiant de condition Posix et que la classe intermédiaire `Mutex::Monitor` dont héritent `Mutex::Lock` et `Mutex::TryLock` porte également les opérations relatives à l'utilisation d'une condition.
Comme type de mutex Posix, choisissez le type récursif.
Testez vos classes en protégeant l'accès au compteur de la question précédente par un mutex.

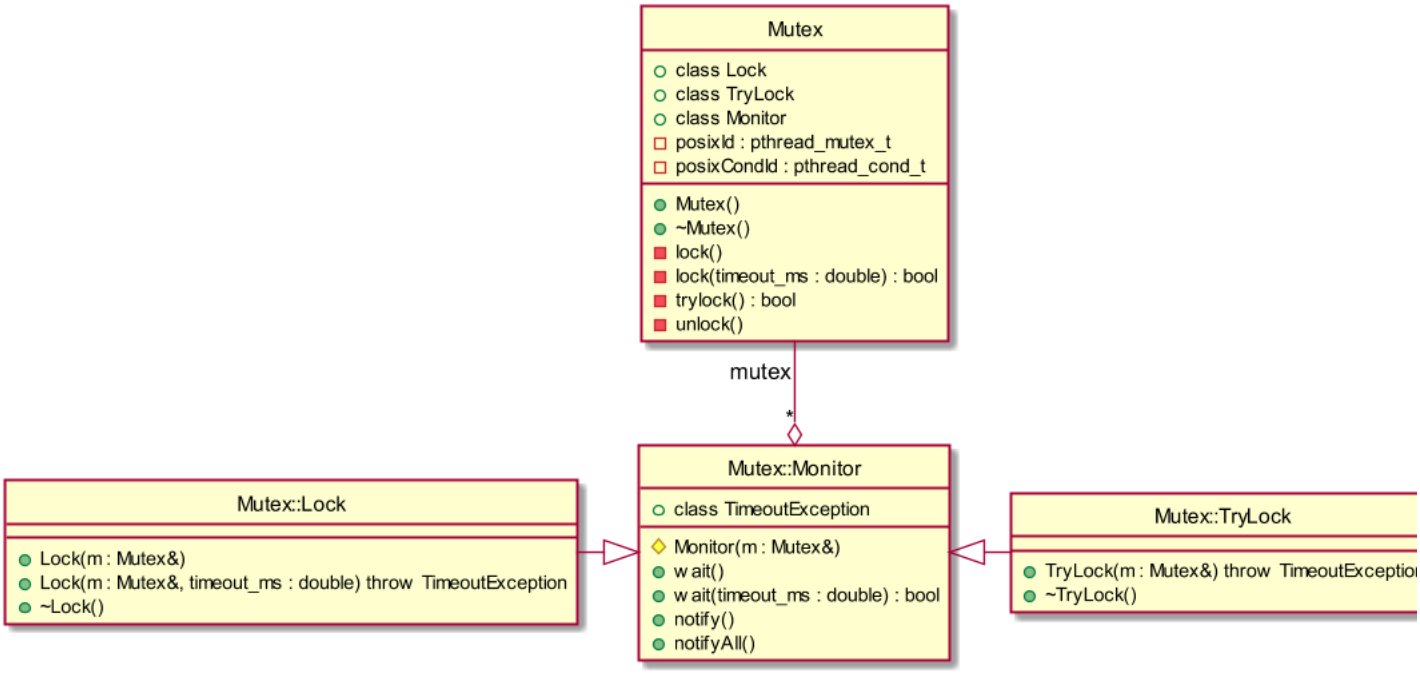

Figure 2 : Spécification des classes Mutex et Mutex::Lock.

Complétez la classe `Thread` avec un champ booléen `started` représentant l'état actif de la tâche ; modifiez votre code de sorte qu'on ne puisse pas relancer une tâche tant que son exécution précédente n'est pas terminée. L'appel de `start()` doit renvoyer un booléen : `true` si la tâche est effectivement démarrée par l'appel de `start()`, `false` si la tâche était déjà démarrée au moment de l'appel.

c) Classe Semaphore

Dans le contexte multitâches, un sémaphore est une « boite à jetons » à accès concurrent :

- en appelant sa méthode `give()`, on lui rajoute un jeton ;
- en appelant sa méthode `take()` on lui retire un jeton ;
- si le compteur de jetons du sémaphore est à zéro, l'appel de `take()` est bloquant, avec ou sans timeout ;
- à sa création, le sémaphore peut être vide ou contenir un nombre quelconque de jetons ;
- on peut définir un nombre maximal de jetons au delà duquel le sémaphore « sature », c'est-à-dire que l'appel de `give()` ne modifie pas son compteur ;
- un sémaphore est qualifié de « binaire » si sa valeur maximale est 1, c'est-à-dire qu'il ne peut avoir que 2 états : vide (`0`) ou plein (`1`).

Programmez la classe `Semaphore` en reprenant l'interface proposée dans le cours et reprise sur la **figure 3**.

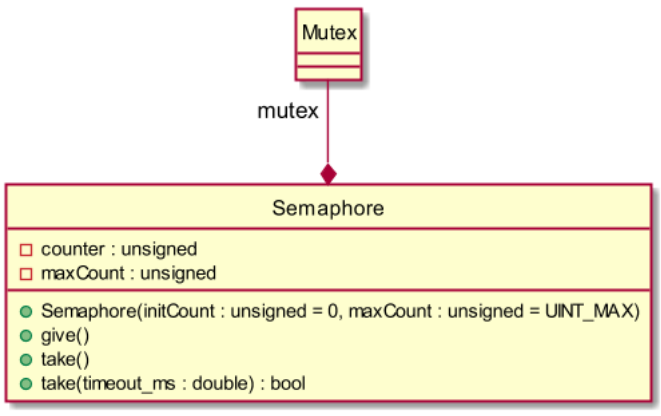

Figure 3 : Structure de la classe Semaphore.

Pour tester votre classe, instanciez un sémaphore initialement vide partagé par 2 types de tâches :
– une tâche productrice qui « donne » des jetons au sémaphore ;

> – une tâche consommatrice qui « prend » des jetons au sémaphore ;
> dans votre `main`, faites tourner `nCons` tâches consommatrices et `nProd` tâches productrices et vérifiez que tous les jetons crées ont bien été consommés.

d) Classe Fifo multitâches

La **figure 4** spécifie l'interface d'une classe template `Fifo`. L'appel à `pop()` doit être bloquant si la fifo est vide ; l'appel bloquant doit comprendre une version avec timeout.

> Programmez la classe `Fifo` en utilisant le conteneur C++ `std::queue`. Comme il s'agit d'un **template**, les déclarations et implémentations doivent être dans un seul fichier `Fifo.hpp`.
> Testez la classe en y accédant de manière concurrente par de multiples tâches productrices et consommatrices. Pour cela, utilisez une fifo de nombres entiers `Fifo<int>` et faites produire par chaque tâche productrice une série d'entiers de `0` à `n`. Mettez en place un mécanisme pour vérifier que tous les entiers produits par les tâches productrices ont bien été reçus par les tâches consommatrices.

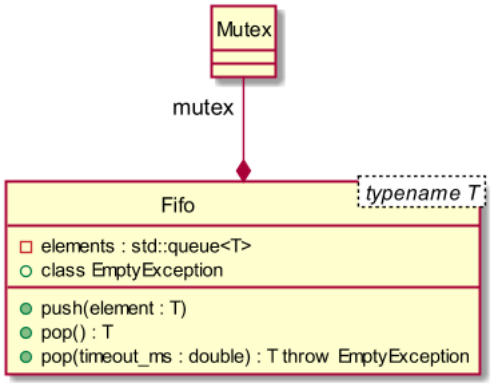

Figure 4 : Classe Fifo à accès concurrent.