Le multitâche sous Qt

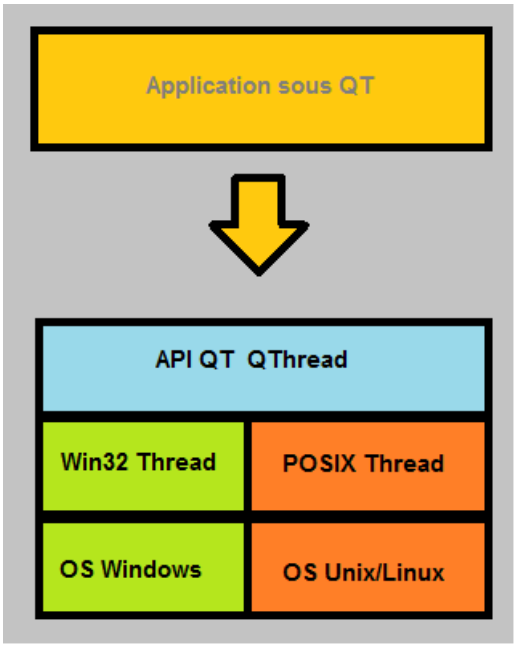
Sommaire

Les objectifs de ce tp sont de découvrir la programmation multitâche sous Qt.

Le multitâche sous Qt

Qt fournit plusieurs possibilités pour manipuler des threads :

- `QThread` : c'est une classe qui interface un thread. Elle va créer, stopper, faire exécuter sa méthode `run()` et autres opération sur un thread.
- `QThreadPool` : pour optimiser la création de threads, il est possible de manipuler un *pool* de thread avec `QThreadPool`. Ce *pool* de threads exécutera des classes héritant de `QRunnable` et réimplementant la méthode `run()`.
- `QtConcurrent` : un ensemble d'algorithmes simplifiant énormément l'utilisation des threads. Il partage un *pool* de threads qui s'adaptera à la machine d'exécution.



Les classes `QThreadPool` et `QtConcurrent` ne sont pas traitées dans ce document.

QThread

Qt fournit la classe `QThread` pour la création et manipulation d'un thread.

Pour les version de Qt inférieures à la 4.4, elle ne peut être instanciée, car la méthode `run()` est virtuelle pure (`QThread` était donc abstraite). Pour les versions plus récentes, la méthode `run()` est uniquement virtuelle et la classe peut donc être instanciée et être utilisée telle quelle.

Les principales fonctionnalités de base sont :

- `run()` : méthode exécutée par le thread ;
- `exec()` : fonction bloquante qui va exécuter une boucle d'événements (*event loop*) dans le thread, cette fonction ne peut être appelée que par la méthode `run()`
- `quit()` : slot qui stoppe la boucle d'événements du thread
- `start()` : slot qui lance un thread qui exécute la méthode `run()`, cette fonction peut prendre en paramètre la priorité donnée au thread
- `setPriority()` : modifie la priorité d'exécution du thread
- `wait()` : fonction bloquante qui attend la fin de l'exécution, il est possible de spécifier un *timeout*

- `sleep()`, `msleep()` et `usleep()` : ces méthodes statiques mettent en pause le thread respectivement pendant n secondes, n millisecondes ou n micro secondes
- `yieldCurrentThread()` : cette méthode statique permet de rendre la main du CPU à l'OS et se met en attente

Cette classe émet le signal `started()` lorsqu'un thread est lancé et `finished()` lorsque le thread est terminé.

➟ Documentation Qt 4.8 : QThread

La grande majorité des API graphiques des divers OS ne sont absolument pas *thread-safe*. Le traitement GUI dans différents threads peut être la source d'accès concurrents qui peuvent mener à des erreurs fatales de l'application. C'est pour cela que Qt oblige les traitements GUI dans le thread principal (celui exécutant le `main()` du programme). Il faut utiliser le système de connexion **signal/slot** pour manipuler l'affichage GUI à partir d'un thread. (cf. "Utilisation des threads Qt avec une GUI" page 14)

Avant d'utiliser un `QThread`, il faut savoir que : (cf. "Utilisation des objets QObject avec les threads sous Qt" page 8)

- `QThread` n'est pas un thread et n'appartient pas au thread. Ses *slots* ne s'exécutent pas dans le thread qu'elle interface. Les objets héritant de `QObject` appartiennent à `QThread` et non au thread.
- Seule la fonction `run()` appartient au thread. Cette fonction `run()` est similaire au `main()` du programme principal. Son exécution correspond au moment où le thread existe réellement. Seuls les objets héritant de `QObject` créés dans la fonction `run()` appartiennent au thread. Les événements pour ces objets sont alors gérés par la boucle événementielle du thread.

Il y a donc plusieurs approches possibles dans l'utilisation de `QThread`, en voici deux :

- on dérive une classe de `QThread` et on implémente la fonction `run()` qui contiendra le code du thread (cf. "Les threads sous Qt" page 6). Seuls des objets n'héritant pas de `QObject` peuvent être utilisés. Si le thread doit utiliser des objets héritant de `QObject`, la boucle d'événements doit être exécutée (appel `exec()`). Il est possible de transférer les `QObject` que l'on veut utiliser vers ce thread (`moveToThread()`).
- pour exécuter du code dans un nouveau thread, on instancie directement un `QThread` et on assigne les objets héritant de `QObject` à ce thread en utilisant la fonction `moveToThread()` (cf. "Le modèle producteur/consommateur" page 11). Depuis Qt 4.4, `QThread` exécute une boucle d'événements par défaut.

➟ FAQ : qt.developpez.com/faq

QMutex et QMutexLocker

Pour protéger des données partagées entre threads, Qt fournit la classe `QMutex` (cf. "Les mutex sous Qt" page 7). Cette classe fournit les méthodes de base suivantes :

- `lock()` : bloque le mutex
- `tryLock()` : essaie de bloquer le mutex (possibilité de mettre un *timeout*)
- `unlock()` : libère le mutex

Afin de simplifier sa manipulation, Qt fournit la classe `QMutexLocker` (basée sur le pattern RAII) qui permet de manipuler correctement le mutex et éviter certains problèmes (un thread qui essaye de bloquer deux fois un mutex, un mutex non débloqué suite à une exception ou à un oubli ou une erreur de

codage ...). **QMutexLocker va bloquer le mutex lors de sa création et le libérer lors de sa destruction.** Il permet aussi de libérer (`unlock()`) et de bloquer à nouveau (`relock()`) le mutex.

Qt fournit deux classes qui simplifient la manipulation du mutex `QMutexLocker` :

- `QReadLoker` : manipulation du mutex en lecture ;
- `QWriteLoker` : manipulation du mutex en écriture.

➟ Documentation Qt 4.8 : QMutex et QMutexLocker

QReadWriteLock

`QReadWriteLock` est un synchronisateur **à écriture exclusive et à lecture multiple** pour l'accès à des données protégées (cf. "Le modèle lecteurs/rédacteurs" page 18).

Lorsqu'une ressource est partagée entre plusieurs threads, ces threads ont le droit d'accéder parallèlement à la ressource uniquement s'ils ne font que des accès en lecture. Ainsi, pour optimiser les accès, Qt fournit `QReadWriteLock`, qui est un autre mutex, beaucoup plus adapté. `QReadWriteLock` va différencier les `lock()` en lecture et écriture :

- Mutex bloqué en lecture :
  - Si un thread essaye de bloquer le mutex en lecture : aucune attente, le thread peut accéder à la ressource ;
  - Si un thread essaye de bloquer le mutex en écriture : le thread attend la libération du mutex.
- Mutex bloqué en écriture :
  - Dans les deux cas, le thread attend la libération du mutex.

Cette classe fournit les méthodes de base suivantes :

- `lockForRead()` : bloque le mutex en lecture
- `tryLockForRead()` : essaie de bloquer le mutex en lecture (possibilité de mettre un *timeout*)
- `lockForWrite()` : bloque le mutex en écriture
- `tryLockForWrite()` : essaie de bloquer le mutex en écriture (possibilité de mettre un *timeout*)
- `unlock()` : libère le mutex

➟ Documentation Qt 4.8 : QReadWriteLock

QSemaphore

La classe `QSemaphore` fournit un sémaphore à comptage général (cf. "Utilisation des sémaphores sous Qt" page 12).

Un sémaphore est une généralisation d'un mutex . Alors qu'un mutex ne peut être verrouillé qu'une fois, il est possible d'acquérir un sémaphore à plusieurs reprises. Les sémaphores sont généralement utilisés pour protéger un certain nombre de ressources identiques.

`QSemaphore` dispose notamment des méthodes suivantes :

- `acquire(n)` : tente d'acquérir n ressources. S'il n'y a pas assez de ressources disponibles, l'appel bloquera jusqu'à ce que ce soit le cas
- `release(n)` : libère n ressources
- `tryAcquire(n)` : retourne immédiatement s'il ne peut pas acquérir les n ressources
- `available()` : renvoie le nombre de ressources disponibles à tout moment

➟ Documentation Qt 4.8 : QSemaphore

QWaitCondition

Les variables conditions `QWaitCondition` agissent comme des signaux visibles pour tous les threads.

Quand un thread termine une opération dont dépendent d'autres threads, il le signale en appellant `wakeAll()`. `wakeAll()` active le signal afin que les autres threads en attente (`wait()`) soit débloqués.

➟ Documentation Qt 4.8 : QWaitCondition

Les signaux et les slots

Qt offre le mécanisme *signal/slot* qui est utilisable entre les threads. Cela fournit une manière intéressante de communiquer (et donc passer des données) entre les threads (cf. "Le modèle producteur/consommateur" page 11).

Rappel : Si un thread interagit avec une GUI (*Graphical User Interface*) par un thread, on doit alors utiliser le système de connexion signal/slot (cf. "Utilisation des threads Qt avec une GUI" page 14 et "Le modèle lecteurs/rédacteurs" page 18).

Contrairement aux slots, les signaux sont *thread safe* et peuvent donc être appelés par n'importe quel thread.

Par défaut, la connexion entre threads est asynchrone, car le slot sera exécuté dans le thread qui possède l'objet receveur. Pour cette raison, les paramètres du signal doivent pouvoir être copiés. Ce qui implique quelques règles simples :

- Ne jamais utiliser un pointeur ou une "référence non const" dans les signatures des signaux/slots. Rien ne permet de certifier que la mémoire sera encore valide lors de l'exécution du slot
- S'il y a une référence `const`, l'objet sera copié
- Il est préférable d'utiliser des classes Qt car elles implémentent une optimisation de la copies

Il est possible d'utiliser ses propres classes dans le mécanisme signal/slot. Pour cela, il faut :

- que cette classe ait un constructeur, un constructeur de copie et un destructeur public
- l'enregistrer dans les métatypes par la méthode `qRegisterMetaType()` (cf. "Le modèle lecteurs/rédacteurs" page 18)

Les fonctions appelées entre les threads sont des slots qui ne retournent rien. Les slots sont appelés via un connect avec un signal. Le résultat d'une fonction sera donc retourné par un signal.

Le dernier paramètre de l'appel `connect()` indique le type de connexion et a son importance lorsqu'on utilise des threads :

```
bool QObject::connect(const QObject *sender, const char *signal,
                      const QObject *receiver, const char *method,
                      Qt::ConnectionType type = Qt::AutoConnection);
```

La valeur par défaut est `Qt::AutoConnection`, ce qui signifie que, si le signal est émis d'un thread différent de l'objet le recevant, le signal est mis dans la queue de gestion d'événements, un comportement semblable à `Qt::QueuedConnection`. Le type de connexion est déterminé quand le signal est émis.

➟ Les différents types de connexion (fr)

Les threads sous Qt

Sous **Qt**, il faut dériver la classe `QThread` pour créer son propre *thread* :

```
#include <QThread>

class MyThread1 : public QThread
{
   Q_OBJECT
   public:
      MyThread1(QObject * parent = 0);
      void run(); // le code du thread

   private:

   signals:
};
```

On écrit ensuite le **code du *thread*** dans la méthode `run()` :

```
// le code du thread
void MyThread1::run()
{
    int count = 0;
    char c1 = '*';

    while(isRunning() && count < 50)
    {
        write(1, &c1, 1);// écrit un caractère sur stdout (descripteur 1)
        count++;
    }
}
```

Pour finit, on crée et démarre le *thread* :

```
MyThread1 *myThread1; // *

myThread1 = new MyThread1;

myThread1->start();

myThread1->wait();
```

On appelera la méthode `start()` pour démarrer le *thread* et la méthode `wait()` pour attendre sa terminaison. C'est la méthode `start()` qui se charge d'exécuter la méthode `run()`.

Les mutex sous Qt

La mise en place des *mutex* revient à instancier un objet de la classe `QMutex` fournie par l'API et à appeler la méthode `lock()` pour vérouiller le *mutex* et la méthode `unlock()` pour le dévérouiller. Un *mutex* est un **objet d'exclusion mutuelle** (*MUTual EXclusion*), et est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des **sections critiques**.

Évidemment, l'objet *mutex* doit être partagé entre les *threads*.

Ici, la variable globale (`value_globale`) est une ressource critique puisque plusieurs tâches peuvent y accéder pour faire des modifications concurrentes. Il faut donc utiliser un **mutex** pour protéger l'accès à la variable globale (`value_globale`).

```
// la donnée partagée entre les threads
int value_globale = 1;
QMutex mutex;

// Le thread 1
void MyThread1::run()
{
    int count = 0;
    int value = 0;
    while(isRunning() && count < 50)
    {
        // entre dans une section critique
        mutex.lock();
        value = value_globale;
        qDebug("Thread1 : load value (value = %d) ", value);
        value += 1;
        qDebug("Thread1 : increment value (value = %d) ", value);
        value_globale = value;
        qDebug("Thread1 : store value (value = %d) ", value_globale);
        mutex.unlock();
        // sort de la section critique
        count++;
   }
}

// Le thread 2
void MyThread2::run()
{
    int count = 0;
    int value = 0;
    while(isRunning() && count < 50)
    {
        // entre dans une section critique
        mutex.lock();
        value = value_globale;
        qDebug("Thread2 : load value (value = %d) ", value);
        value -= 1;
        qDebug("Thread2 : decrement value (value = %d) ", value);
        value_globale = value;
        qDebug("Thread2 : store value (value = %d) ", value_globale);
```



```
        mutex.unlock();
        // sort de la section critique
        count++;
    }
}
```

Afin de simplifier la manipulation des *mutex*, Qt fournit la classe `QMutexLocker` (basée sur le pattern RAII) qui permet de manipuler correctement le mutex et éviter certains problèmes (un *thread* qui essaye de bloquer deux fois un mutex, un mutex non débloqué suite à une exception ou à un oubli ou une erreur de codage ...).

QMutexLocker va bloquer le mutex lors de sa création et le libérer lors de sa destruction. Il permet aussi de libérer (`unlock()`) et de bloquer à nouveau (`relock()`) le mutex :

```
QMutex mutex;

void MyThread1::run() {
  // ...
  executerSectionCritique()
}

void MyThread1::executerSectionCritique() {
    QMutexLocker verrou(&mutex);

    // je suis dans la section critique ...
}
```

Utilisation des objets QObject avec les threads sous Qt

Exemple : FAQ Qt du site developpez.com.

Il y a plusieurs approches possibles dans l'utilisation de `QThread`, en voici deux :

- on dérive une classe de `QThread` et on implémente la fonction `run()` qui contiendra le code du *thread* (cf. "Les threads sous Qt" page 6). Seuls des objets n'héritant pas de `QObject` peuvent être utilisés. Si le *thread* doit utiliser des objets héritant de `QObject`, la boucle d'événements doit être exécutée (appel `exec()`).

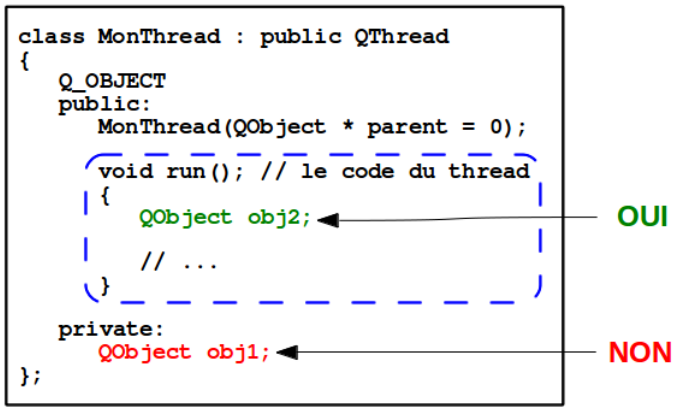


FIGURE 1 – `obj1` est créé par celui qui instancie le `QThread`. `obj2` est créé dans le *thread*. Ici, seul l'objet `obj2` appartient au *thread*.

Il est possible de transférer les `QObject` que l'on veut utiliser vers ce *thread* (`moveToThread()`).

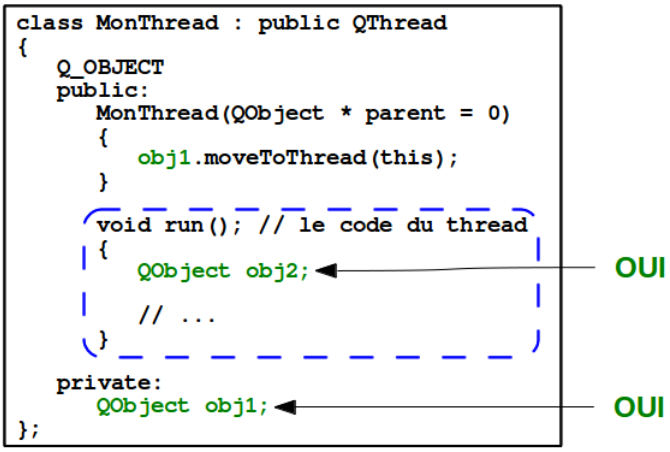


FIGURE 2 – `obj1` est créé par celui qui instancie le `QThread` puis celui-ci transfére son appartenance au *thread* grâce à la méthode `moveToThread()`. `obj2` est créé dans le *thread*. Ici, les deux objets `obj1` et `obj2` appartiennent au *thread*.

- pour exécuter du code dans un nouveau *thread*, on instancie directement un `QThread` et on assigne les objets héritant de `QObject` à ce *thread* en utilisant la fonction `moveToThread()` (cf. "Le modèle producteur/consommateur" page 11). Depuis Qt 4.4, `QThread` exécute une boucle d'événements par défaut.

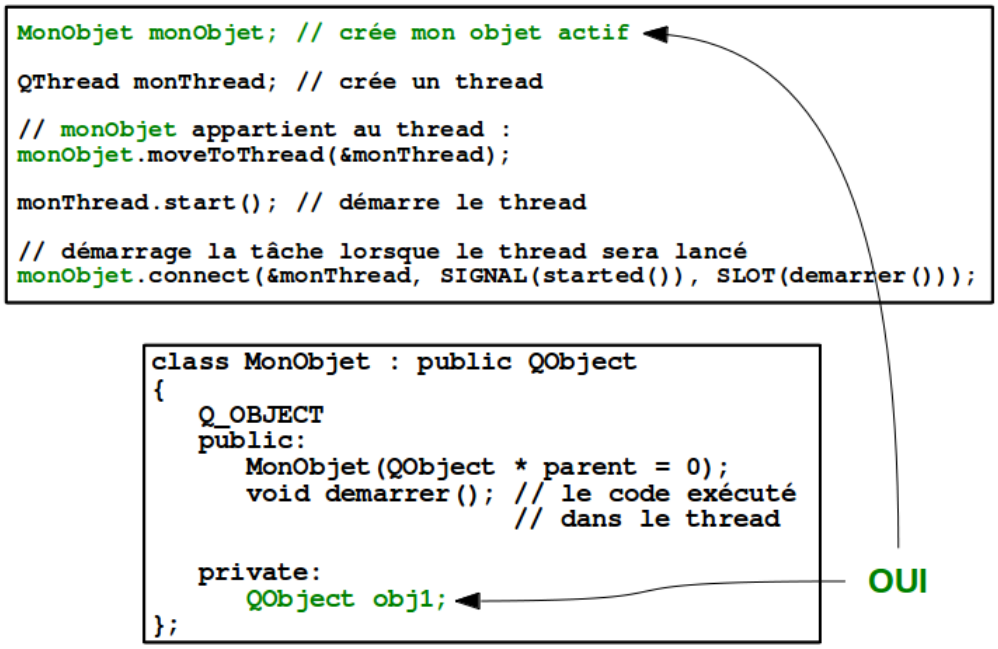


FIGURE 3 – Ici, les deux objets `monObjet` et `obj1` appartiennent au *thread*. Le code exécuté dans le *thread* sera la méthode `demarrer()` (en quelque sorte le `main()` du *thread*).

Seule la fonction `run()` (et les objets qu'elle crée) appartient au *thread*. Le constructeur (et les membres attributs qu'il initialise), les slots et les autres méthodes n'appartiennent pas au *thread* mais a celui qui l'instancie.

Qt offre le mécanisme *signal/slot* qui offre une manière intéressante de **communiquer** (et donc passer des données) entre les *threads* (cf. "Le modèle producteur/consommateur" page 11).

Par défaut, la connexion entre *threads* est **asynchrone**, car le slot sera exécuté dans le thread qui possède l'objet receveur.

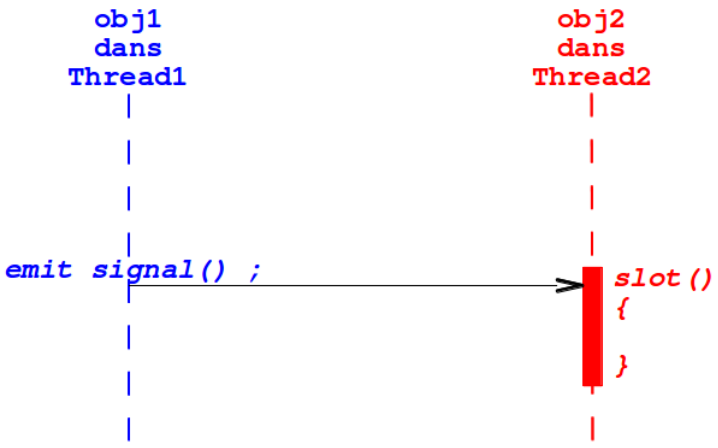


FIGURE 4 – **Message asynchrone** : `obj1` appartient au *thread1* et `obj2` appartient au *thread2*. Le slot de `obj2` sera exécuté dans le *thread2* et le *thread1* continue son exécution une fois le signal émis.

Attention à l'appartenance des objets quand on utilise des *threads* :

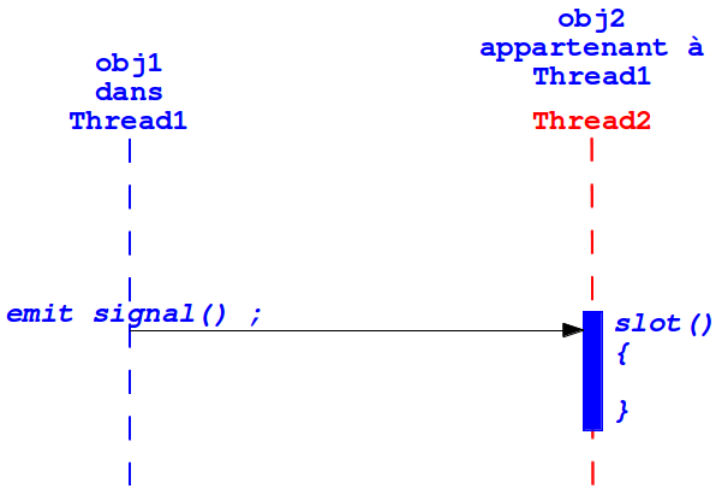
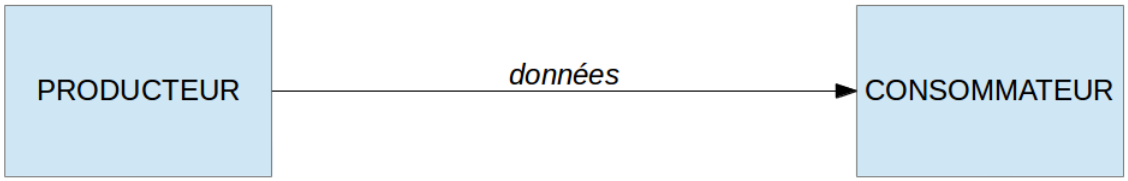


FIGURE 5 – **Message synchrone** : `obj1` et `obj2` appartiennent au *thread1*. Le slot de `obj2` sera donc exécuté dans le *thread1* qui attendra la fin de l'exécution de celui-ci. Et ce sera pareil avec un slot de *thread2* !

Le modèle producteur/consommateur

Exemple : "Les threads sans maux de tête" de Bradley T. Hughes

On va mettre en œuvre le modèle Producteur/Consommateur suivant :



Le modèle Producteur/Consommateur est un exemple de synchronisation de ressources. Il peut s'envisager dans différents contextes, notamment en environnement *multi-thread*. Ici, le principe mis en œuvre est le suivant : **un producteur produit des données et un consommateur consomme les données produites**.

Dans ce cas, on a juste besoin d'assurer une synchronisation entre le producteur et le consommateur car il n'y a pas de données partagées à protéger. Pour cela, on peut utiliser le **mécanisme signal/slot** propre à Qt. D'autre part, on n'a plus besoin de dériver `QThread` et d'implémenter la méthode `run()` depuis la version 4.4 de Qt (de plus la boucle d'évènements est démarré par défaut par un `exec()`).

```
enum {
    TailleMax = 123456,
    TailleBloc = 7890
};

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    // création du producteur et du consommateur
    Producteur producteur;
    Consommateur consommateur;

    // une simple synchronisation à base de signal/slot entre producteur et consommateur
    producteur.connect(&consommateur, SIGNAL(consomme()), SLOT(produire()));
    consommateur.connect(&producteur, SIGNAL(produit(QByteArray *)), SLOT(consommer(
        QByteArray *)));

    // chacun son thread : création du thread producteur et du thread consommateur
    QThread producteurThread;
    producteur.moveToThread(&producteurThread);

    QThread consommateurThread;
    consommateur.moveToThread(&consommateurThread);

    // démarrage de l'activité du producteur lorsque son thread sera lancé
    producteur.connect(&producteurThread, SIGNAL(started()), SLOT(produire()));

    // lorsque le consommateur a fini, on arrête son thread
    consommateurThread.connect(&consommateur, SIGNAL(fini()), SLOT(quit()));

    // lorsque le thread consommateur a fini, on stoppe le thread producteur
```



```
    producteurThread.connect(&consommateurThread, SIGNAL(finished()), SLOT(quit()));

    // lorsque le thread producteur a fini, on quitte l'application
    a.connect(&producteurThread, SIGNAL(finished()), SLOT(quit()));

    // on démarre les 2 threads
    producteurThread.start();
    consommateurThread.start();

    // on exécute la boucle d'évènements de l'application
    return a.exec();
}
```

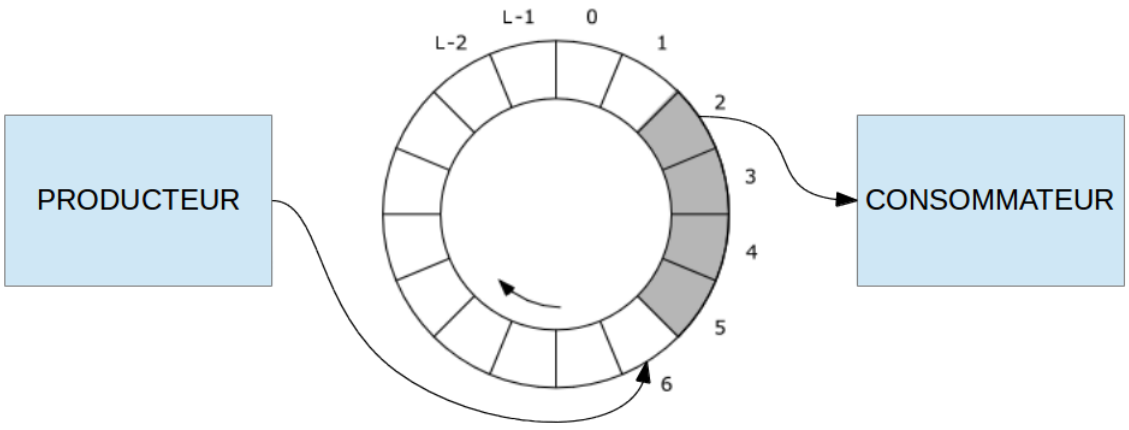
Conclusion (de Bradley T. Hughes) : Le résultat final est une solution au problème du Producteur/Consommateur sans la nécessité d'un verrou, d'une variable conditionnelle ou d'un sémaphore. Comment ? on n'a même pas eu besoin d'écrire un thread. On peut tout faire d'une manière orientée objet. Le code du Producteur va à un endroit, celui du Consommateur dans un autre, puis on déplace tout cela dans un merveilleux thread boîte noire qui fait ce que on attend.

Utilisation des sémaphores sous Qt

Exemple de la documentation Qt : qt-threads-semaphores-example.html

Le modèle Producteur/Consommateur est un exemple de synchronisation de ressources. Il peut s'envisager dans différents contextes, notamment en environnement *multi-thread*.

Le principe mis en œuvre est le suivant : **un producteur produit des informations et les place dans un tampon partagé. Un consommateur vide le tampon et consomme les informations.** Le tampon peut être une file, un buffer circulaire, une zone mémoire partagée, etc ... Ici on utilisera un **buffer circulaire de taille `L` (ici `TailleBuffer`)**.



Il faudra tenir compte des contraintes de synchronisation en définissant le comportement à avoir lorsqu'un consommateur souhaite lire depuis le buffer circulaire lorsque celui-ci est vide et lorsqu'un producteur souhaite écrire dans le buffer circulaire mais que celui-ci est plein. Et éventuellement, il faudra se protéger des accès concurrents sur les ressources critiques partagées. Ici, un seul processus produit des informations et un seul processus les consomme.

Donc :

- Le Producteur ne peut déposer que s'il existe une place libre dans le buffer circulaire.
- Le Consommateur ne peut prélever que si le buffer circulaire n'est pas vide.
- Le buffer circulaire est une ressource épuisable.

La synchronisation du Producteur et du Conducteur vis à vis du buffer circulaire est à mettre en place. On utilisera un **sémaphore `octetsDisponibles` initialisé à `TailleBuffer`**. La synchronisation du Consommateur vis à vis du Producteur est à mettre en place. On utilisera un **sémaphore `semBuffer` initialisé à 0**. Par contre, Le buffer circulaire n'est pas une ressource critique car le Producteur et le Consommateur n'en manipulent pas la même partie.

On a donc besoin des sémaphores suivants :

```
Sem semBuffer init 0

Sem octetsDisponibles init TailleBuffer
```

L'algorithme du Producteur :

```
Début
   // Éventuellement : Construire information
   P(octetsDisponibles)
   Déposer information
   V(semBuffer)
Fin
```

L'algorithme du Consommateur :

```
Début
   P(semBuffer)
   Retirer information
   V(octetsDisponibles)
   // Éventuellement : Traiter information
Fin
```

Utilisation des sémaphores dans le modèle Producteur/Consommateur :

```
enum
{
    TailleMax = 128,
    TailleBuffer = 8
};

// données partagées
char buffer[TailleBuffer];

QSemaphore octetsDisponibles(TailleBuffer);
QSemaphore semBuffer; // = 0

void Producteur::produire()
{
   qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
   for (int i = 0; i < TailleMax; ++i)
   {
      octetsDisponibles.acquire();
      buffer[i % TailleBuffer] = "ACGT"[(int)qrand() % 4]; // A ou C ou G ou T
```

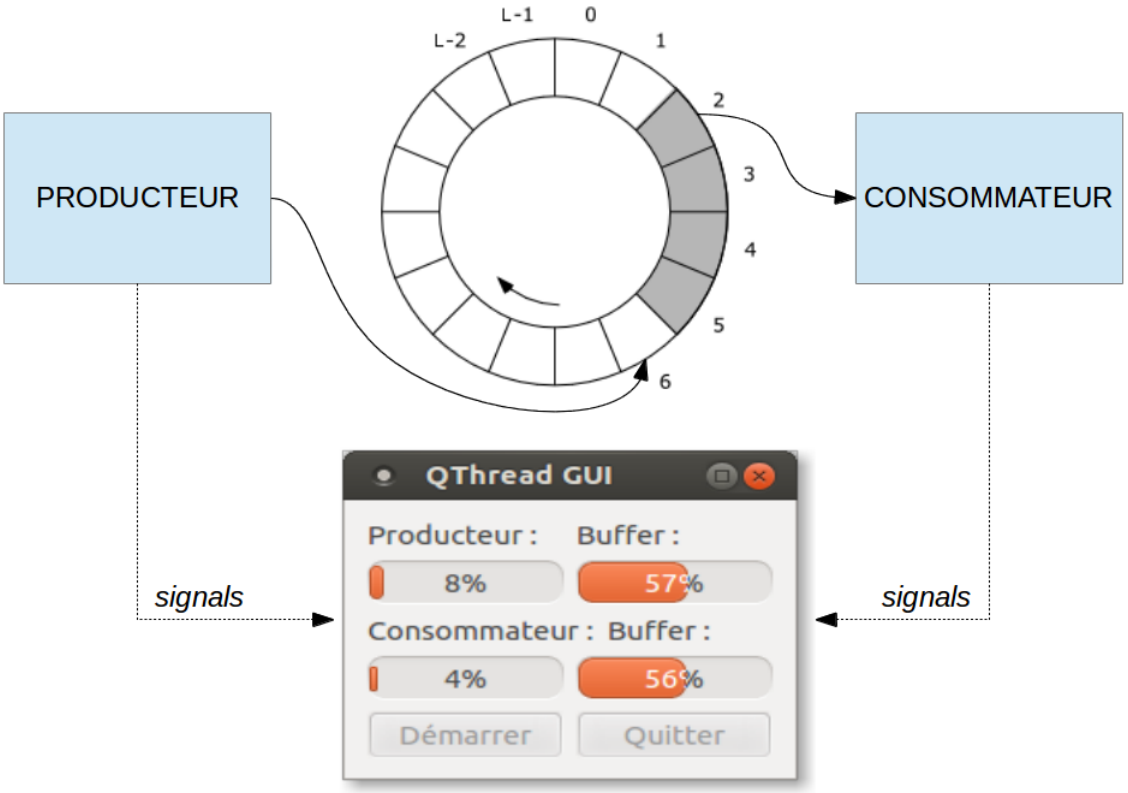


```
      semBuffer.release();
   }
}

void Consommateur::consommer()
{
   for (int i = 0; i < TailleMax; ++i)
   {
      semBuffer.acquire();
      fprintf(stderr, "<%c>", buffer[i % TailleBuffer]);
      octetsDisponibles.release();
   }
}
```

Utilisation des threads Qt avec une GUI

Cette séquence est basée sur la “Utilisation des sémaphores sous Qt” page 12.

On désire simplement ajouter l’affichage du pourcentage des données produites et consommées ainsi que le pourcentage d’utilisation du buffer circulaire par le producteur et le consommateur :



La classe pour la GUI :

```
class MyDialog : public QDialog
{
   Q_OBJECT
   public:
      MyDialog( QWidget *parent = 0 );

   private:
      // ...

      Producteur producteur;
      Consommateur consommateur;
      QThread producteurThread;
      QThread consommateurThread;

   private slots:
      void demarrer();
      void performProducteur(int steps);
      // ...
};
```

```
MyDialog::MyDialog( QWidget *parent ) : QDialog( parent )
{
   // ...

   // chacun son thread : création du thread producteur et du thread consommateur
   producteur.moveToThread(&producteurThread);
   consommateur.moveToThread(&consommateurThread);

   // démarrage des deux activités lorsque les 2 threads seront lancés
   producteur.connect(&producteurThread, SIGNAL(started()), SLOT(produire()));
   consommateur.connect(&consommateurThread, SIGNAL(started()), SLOT(consommer()));

   // la mise à jour de la GUI par signal/slot
   connect(&producteur, SIGNAL(evolutionProducteur(int)), this, SLOT(performProducteur(int))
      );
   connect(&consommateur, SIGNAL(evolutionConsommateur(int)), this, SLOT(performConsommateur
      (int)));
   connect(&producteur, SIGNAL(evolutionProducteurBuffer(int)), this, SLOT(
      performProducteurBuffer(int)));
   connect(&consommateur, SIGNAL(evolutionConsommateurBuffer(int)), this, SLOT(
      performConsommateurBuffer(int)));

   // lorsque le consommateur a fini, on arrête son thread
   consommateurThread.connect(&consommateur, SIGNAL(fini()), SLOT(quit()));

   // lorsque le thread consommateur a fini, on stoppe le thread producteur
   producteurThread.connect(&consommateurThread, SIGNAL(finished()), SLOT(quit()));

   // lorsque le thread producteur a fini, on réinitialise la GUI
   connect(&producteurThread, SIGNAL(finished()), this, SLOT(terminer()));
}
```



```
void MyDialog::demarrer()
{
   // on démarre les 2 threads
   if (!producteurThread.isRunning())
   {
      producteurThread.start();
   }
   if (!consommateurThread.isRunning())
   {
      consommateurThread.start();
   }
}

void MyDialog::performProducteur(int steps)
{
   if(steps <= progressBarProducteur->maximum())
      progressBarProducteur->setValue(steps);
}

// ...
```

Utilisation de la GUI dans le modèle Producteur/Consommateur :

```
void Producteur::produire()
{
   qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
   for (int i = 0; i < TailleMax; ++i)
   {
      // ...
      octetsDisponibles.acquire();
      buffer[i % TailleBuffer] = "ACGT"[(int)qrand() % 4]; // A ou C ou G ou T
      emit evolutionProducteur((int)(double(i)/double(TailleMax)*100.));
      emit evolutionProducteurBuffer((int)(double(TailleBuffer-octetsDisponibles.available()
         )/double(TailleBuffer)*100.));
      semBuffer.release();
   }
   emit fini();
}

void Consommateur::consommer()
{
   for (int i = 0; i < TailleMax; ++i)
   {
      semBuffer.acquire();
      fprintf(stderr, "<%c>", buffer[i % TailleBuffer]);
      emit evolutionConsommateur((int)(double(i)/double(TailleMax)*100.));
      emit evolutionConsommateurBuffer((int)(double(TailleBuffer-octetsDisponibles.available
         ())/double(TailleBuffer)*100.));
      octetsDisponibles.release();
      // ...
   }
   emit fini();
}
```


Temporisation dans un thread Qt

La classe `QThread` fournit des méthodes statiques `protected` pour mettre en pause le *thread* pendant n secondes (`sleep()`), n millisecondes (`msleep()`) ou n micro secondes (`usleep()`).

La méthode statique `yieldCurrentThread()` permet de rendre la main du CPU à l'OS et le *thread* se met en attente.

Ces méthodes sont seulement utilisables si on dérive son propre *thread* à partir de `QThread` ("Les threads sous Qt" page 6).

Malheureusement, on ne peut pas appeler à partir d'un objet `QObject` s'exécutant dans un *thread* les méthodes statiques `sleep()`, `msleep()` et `usleep()` de la classe `QThread` car elles sont déclarées `protected`.

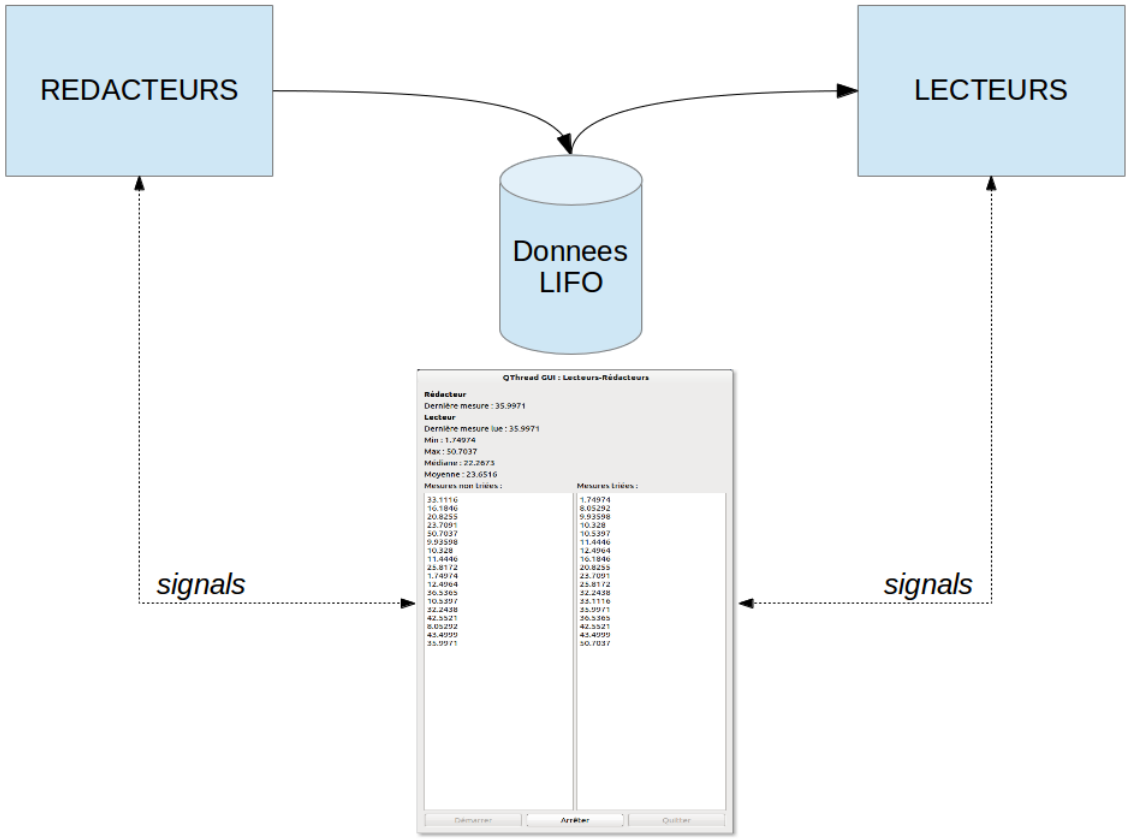
À la place, on va utiliser le *timeout* d'un `QWaitCondition` pour créer une méthode `msleep()` interne à l'objet :

```
class XXX : public QObject
{
    Q_OBJECT

private:
    QMutex localMutex;
    QWaitCondition sleepSimulator;

    void msleep(unsigned long sleepMS)
    {
        sleepSimulator.wait(&localMutex, sleepMS);
    }
    void cancelSleep()
    {
        sleepSimulator.wakeAll();
    }

public:
    XXX() { localMutex.lock(); }

...
};
```


Le modèle lecteurs/rédacteurs

Exemple : "Thread travailleur utilisant les signaux et les slots"

Pour cela, on va mettre en œuvre le modèle Lecteur/Rédacteur suivant :



N processus (ou tâches) répartis en 2 catégories (les Lecteurs et les Rédacteurs), se partagent une ressource commune :

- Les lecteurs peuvent à accéder simultanément en lecture.
- Les rédacteurs doivent avoir un accès exclusif à la ressouce : lorsqu'un rédacteur écrit, aucune Lecture ni aucune Ecriture n'est possible.

Sous Qt, le plus simple pour mettre en œuvre le modèle Lecteur/Rédacteur est d'utiliser un `QReadWriteLock` qui est un synchronisateur **à écriture exclusive et à lecture multiple** pour l'accès à des données protégées.

La classe pour la GUI crée les threads et interagit avec eux à partir du mécanisme *signal/slot* :

```
class MyDialog : public QDialog
{
   Q_OBJECT

   public:
      MyDialog( QWidget *parent = 0 );

   private:
      // ...
      QList<Redacteur *> redacteurs;
      QList<Lecteur *> lecteurs;
```



```
    QList<QThread *> redacteursThread;
    QList<QThread *> lecteursThread;

  private slots:
    // ...

  signals:
    // ...
};
```

```
MyDialog::MyDialog( QWidget *parent ) : QDialog( parent )
{
  // ...
  // chacun son thread : création des threads redacteur et des threads lecteur
  redacteurs.push_back(new Redacteur);
  redacteurs.push_back(new Redacteur);
  lecteurs.push_back(new Lecteur);
  lecteurs.push_back(new Lecteur);

  redacteursThread.push_back(new QThread);
  redacteursThread.push_back(new QThread);
  lecteursThread.push_back(new QThread);
  lecteursThread.push_back(new QThread);

  redacteurs.at(0)->moveToThread(redacteursThread.at(0));
  redacteurs.at(1)->moveToThread(redacteursThread.at(1));
  lecteurs.at(0)->moveToThread(lecteursThread.at(0));
  lecteurs.at(1)->moveToThread(lecteursThread.at(1));

  // démarrage des deux activités lorsque les 2 threads seront lancés
  connect(redacteursThread.at(0), SIGNAL(started()), redacteurs.at(0), SLOT(acquerir()));
  connect(redacteursThread.at(1), SIGNAL(started()), redacteurs.at(1), SLOT(acquerir()));

  connect(lecteursThread.at(0), SIGNAL(started()), lecteurs.at(0), SLOT(prelever()));
  connect(lecteursThread.at(1), SIGNAL(started()), lecteurs.at(1), SLOT(traiter()));

  // la mise à jour de la GUI par signal/slot
  connect(redacteurs.at(0), SIGNAL(evolutionRedacteur(double)), this, SLOT(performRedacteur
      (double)));
  connect(redacteurs.at(1), SIGNAL(evolutionRedacteur(double)), this, SLOT(performRedacteur
      (double)));
  qRegisterMetaType <QVector<double> >("QVector<double>");
  connect(lecteurs.at(0), SIGNAL(evolutionLecteur(double)), this, SLOT(performLecteur(
      double)));
  connect(lecteurs.at(0), SIGNAL(evolutionLecteur(const QVector<double> &, bool)), this,
      SLOT(performLecteur(const QVector<double> &, bool)));
  connect(lecteurs.at(1), SIGNAL(evolutionLecteur(const QVector<double> &, bool)), this,
      SLOT(performLecteur(const QVector<double> &, bool)));
  connect(lecteurs.at(1), SIGNAL(evolutionLecteurMinMax(double, double)), this, SLOT(
      performLecteurMinMax(double, double)));
  connect(lecteurs.at(1), SIGNAL(evolutionLecteurCalculs(double, double)), this, SLOT(
      performLecteurCalculs(double, double)));

  // lorsque les lecteurs ont fini, on arrête leur thread
```



```
   connect(lecteurs.at(0), SIGNAL(fini()), lecteursThread.at(0), SLOT(quit()));
   connect(lecteurs.at(1), SIGNAL(fini()), lecteursThread.at(1), SLOT(quit()));

   // lorsque les redacteurs ont fini, on arrête leur thread
   connect(redacteurs.at(0), SIGNAL(fini()), redacteursThread.at(0), SLOT(quit()));
   connect(redacteurs.at(1), SIGNAL(fini()), redacteursThread.at(1), SLOT(quit()));

   // lorsque les threads redacteurs ont fini, on stoppe les lecteurs
   connect(redacteursThread.at(0), SIGNAL(finished()), this, SLOT(arreterLecteurs()));
   connect(redacteursThread.at(1), SIGNAL(finished()), this, SLOT(arreterLecteurs()));

   // lorsque les threads lecteurs ont fini, on réinitialise la GUI
   connect(lecteursThread.at(0), SIGNAL(finished()), this, SLOT(terminer()));
   connect(lecteursThread.at(1), SIGNAL(finished()), this, SLOT(terminer()));

   // pour stopper les lecteurs
   connect(this, SIGNAL(stopLecteur()), lecteurs.at(0), SLOT(arreter()));
   connect(this, SIGNAL(stopLecteur()), lecteurs.at(1), SLOT(arreter()));

   // pour stopper les redacteurs
   connect(this, SIGNAL(stopRedacteur()), redacteurs.at(0), SLOT(arreter()));
   connect(this, SIGNAL(stopRedacteur()), redacteurs.at(1), SLOT(arreter()));
}

void MyDialog::demarrer()
{
   // on démarre les threads
   redacteursThread.at(0)->start();
   redacteursThread.at(1)->start();
   lecteursThread.at(0)->start();
   lecteursThread.at(1)->start();
   // ...
}

void MyDialog::arreter()
{
   // on stoppe les threads redacteurs
   if (redacteursThread.at(0)->isRunning() && redacteursThread.at(1)->isRunning())
   {
      emit stopRedacteur();
   }
}

void MyDialog::arreterLecteurs()
{
   // on stoppe les lecteurs lorsque les threads redacteurs sont terminés
   while(redacteursThread.at(0)->isRunning() || redacteursThread.at(1)->isRunning());

   emit stopLecteur();
}

// ...
```


Il est possible d'utiliser ses propres classes ou ses propres structures de données (ici `QVector<double>`) dans le mécanisme signal/slot. Pour cela, il faut l'enregistrer dans les métatypes par la méthode `qRegisterMetaType()`.

La classe `Redacteur` produit périodiquement une mesure stockée dans le `QStack` en utilisant le verrou `QReadWriteLock` pour un accès en **écriture**. La classe `Lecteur` assure deux traitements (`prelever()` et `traiter()`) à partir des mesures stockées dans le `QStack` en utilisant le verrou `QReadWriteLock` pour un accès en **lecture**.

```
// données partagées
QStack<double> donnees; // LIFO

QReadWriteLock verrou;

void Redacteur::acquerir()
{
   running = true;
   double donnee = 0.;

   qsrand(QTime(0,0,0).secsTo(QTime::currentTime())+QThread::currentThreadId());
   while(running)
   {
       // acquisition périodique
       donnee = mesurer(0., 50.);

       verrou.lockForWrite();
       donnees.push(donnee);
       verrou.unlock();

       emit evolutionRedacteur(donnee);

       qApp->processEvents(); // traite tous les événements en attente
   }
   emit fini();
}

void Lecteur::prelever()
{
   running = true;
   double donnee = 0.;

   while(running)
   {
       verrou.lockForRead();
       if(!donnees.isEmpty())
       {
          donnee = donnees.top();
       }
       verrou.unlock();

       emit evolutionLecteur(donnee);
       emit evolutionLecteur(donnees, false);

       qApp->processEvents(); // traite tous les événements en attente
```



```
    }
    emit fini();
}

void Lecteur::traiter()
{
    running = true;
    double donneeMin = 0., donneeMax = 0., mediane = 0., moyenne = 0., somme = 0.;

    while(running)
    {
        verrou.lockForRead();
        QVector<double> donneesLocales = donnees;
        verrou.unlock();

        qSort(donneesLocales.begin(), donneesLocales.end());
        donneeMin = donneesLocales.first();
        donneeMax = donneesLocales.last();
        if(donneesLocales.count() % 2 == 0) {
            mediane = static_cast<double>((donneesLocales[donneesLocales.count() / 2 - 1] +
                donneesLocales[donneesLocales.count() / 2])) / 2;
        } else {
            mediane = donneesLocales[donneesLocales.count() / 2];
        }
        somme = 0.;
        for(int i=0; i < donneesLocales.count(); i++) {
            somme += donneesLocales.at(i);
        }
        moyenne = somme / (double)donneesLocales.count();

        emit evolutionLecteur(donneesLocales, true);
        emit evolutionLecteurMinMax(donneeMin, donneeMax);
        emit evolutionLecteurCalculs(mediane, moyenne);

        qApp->processEvents(); // traite tous les événements en attente
    }
    emit fini();
}
```