TP Threads n°2

Sommaire

Programmation concurrente **2**
Présentation . . . 2
Synchronisation de données . . . 2

Séquence n°1 : mise en évidence du problème de synchronisation de données **3**
Objectifs . . . 3
Étape n°0 : deux tâches qui ne s'entendent pas ! . . . 3
Étape n°1 : mise en oeuvre d'un mutex . . . 6
Conclusion . . . 9

Séquence n°2 : le mutex dans tous ses états ! **10**
Objectifs . . . 10
Étape n°1 : aie ! . . . 10
Conclusion . . . 10

Séquence n°3 : les mutex dans différents environnements **11**
Objectifs . . . 11
Étape n°1 : les mutex sous Qt . . . 11
Étape n°2 : les mutex sous Builder . . . 15
Étape n°3 : les mutex en C++ avec l'API commoncpp . . . 18
Étape n°4 : les mutex en Java . . . 22

Questions de révision **25**

Travail demandé **25**
Exercice 1 : . . . 25

Les objectifs de ce tp sont de découvrir la programmation concurrente de *threads* en utilisant des *mutex* dans différents environnements de développement.

Programmation concurrente

Présentation

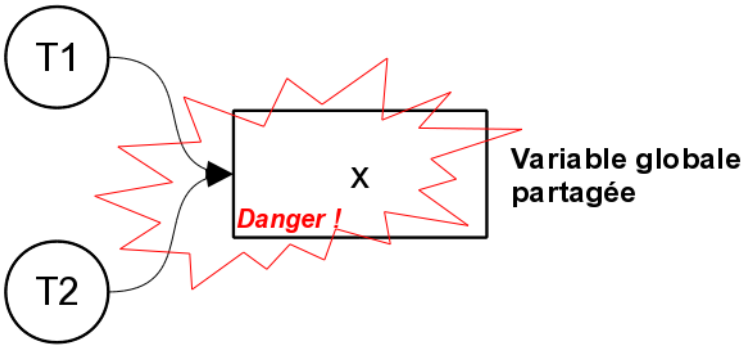
Dans la programmation concurrente (avec des processus lourds ou légers), le terme de **synchronisation** se réfère à deux concepts distincts (mais liés) :

- la synchronisation de tâches
- la **synchronisation de données**

Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

Synchronisation de données

La synchronisation de données est un mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.



Rappel : les *threads* sont englobés dans un processus lourd et partagent donc sa mémoire virtuelle. Cela permet aux *threads* de partager les données globales mais de disposer de leur propre pile pour implanter leurs variables locales.

Des processus lourds, donc séparés et indépendants, qui désirent partager des données devront utiliser un mécanisme fourni par le système pour communiquer tel que IPC (*Inter Processus Communication*).

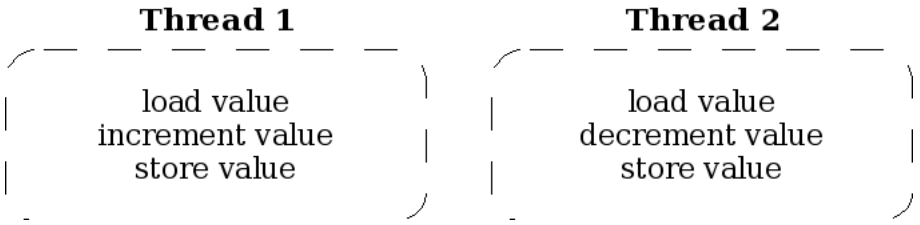
Séquence n°1 : mise en évidence du problème de synchronisation de données

Objectifs

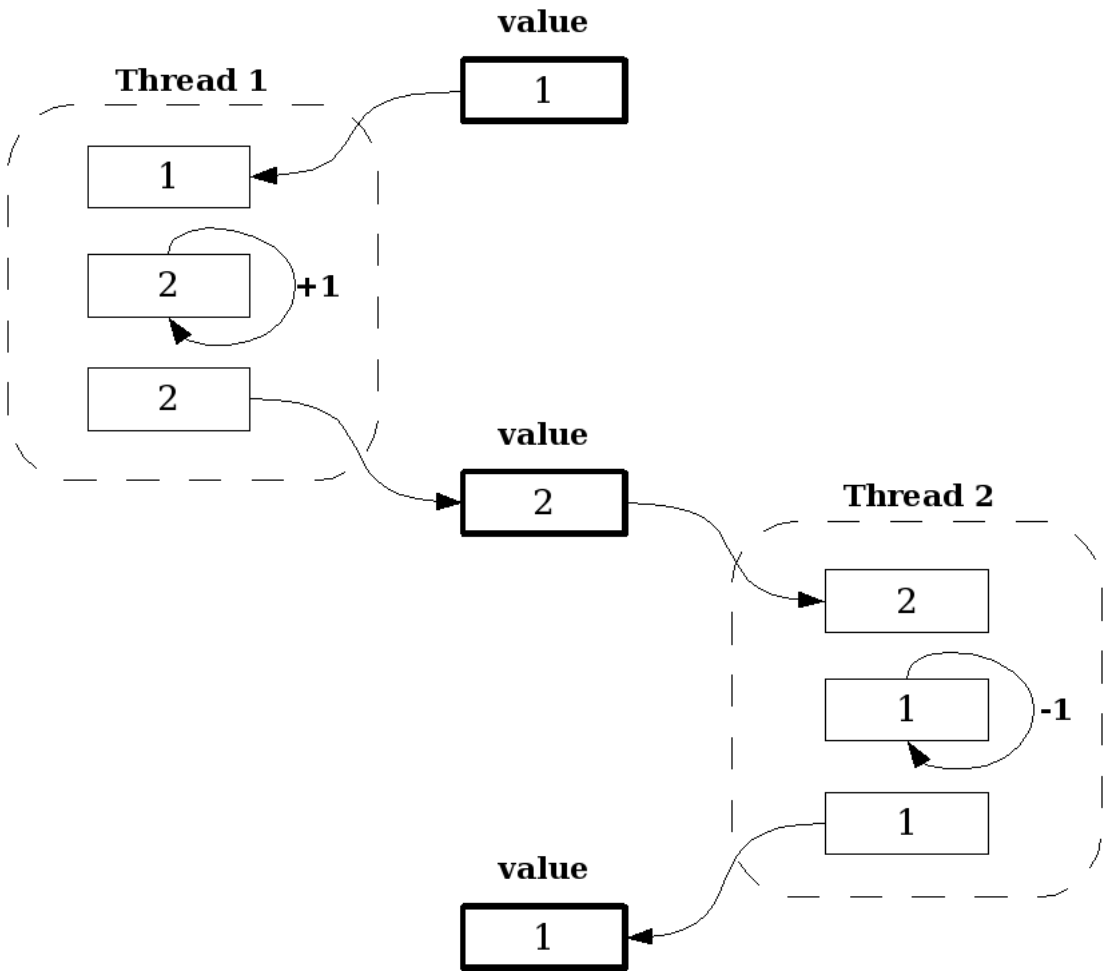
L'objectif de cette séquence est de mettre en évidence le problème classique de la synchronisation de données à base de *threads*.

Étape n°0 : deux tâches qui ne s'entendent pas !

On va créer deux tâches : une incrémente une variable partagée et l'autre la décrémente.



Un déroulement possible serait :



En réalité, le résultat n'est pas prévisible, du fait que vous ne pouvez pas savoir ni prévoir l'ordre d'exécution des instructions.

On va écrire un programme en C qui illustre ce problème. Les deux tâches réalisent le même nombre de traitement (`COUNT`). On suppose donc que la variable globale (`value_globale`) doit revenir à sa valeur initiale (`1`) puisqu'il y aura le même nombre d'incrémentation et de décrémentation.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Variable globale partagée
int value_globale = 1; // valeur initiale

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT 5
// Fonctions correspondant au corps d'un thread (tache)
void *increment(void *inutilise);
void *decrement(void *inutilise);

int main(void)
{
  pthread_t thread1, thread2;

  printf("Avant les threads : value = %d\n", value_globale);
  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, decrement, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Apres les threads : value = %d\n", value_globale);
  printf("Fin du thread principal\n");
  pthread_exit(NULL);
  return EXIT_SUCCESS;
}

void *increment(void *inutilise)
{
   int value;
   int count = 0;
   while(1) {
      value = value_globale;
      printf("Thread1 : load value (value = %d) ", value);
      value += 1;
      printf("Thread1 : increment value (value = %d) ", value);
      value_globale = value;
      printf("Thread1 : store value (value = %d) ", value_globale);
      count++;
      if(count >= COUNT) {
         printf("Le thread1 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```



```
void *decrement(void *inutilise)
{
   int value;
   int count = 0;
   while(1) {
      value = value_globale;
      printf("Thread2 : load value (value = %d) ", value);
      value -= 1;
      printf("Thread2 : decrement value (value = %d) ", value);
      value_globale = value;
      printf("Thread2 : store value (value = %d) ", value_globale);
      count++;
      if(count >= COUNT) {
         printf("Le thread2 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

threads.2a.c

L'exécution de ce programme montre que la variable globale ne revient pas à son état initial !

Vous pouvez tester plusieurs fois et même avec des valeurs différentes de `COUNT`.

```
$ ./threads.2a
Avant les threads : value = 1
Thread1 : load value (value = 1) Thread1 : increment value (value = 2) Thread1 : store value
     (value = 2) Thread1 : load value (value = 2) Thread1 : increment value (value = 3)
    Thread1 : store value (value = 3) Thread1 : load value (value = 3) Thread1 : increment
    value (value = 4) Thread1 : store value (value = 4) Thread1 : load value (value = 4)
    Thread1 : increment value (value = 5) Thread1 : store value (value = 5) Thread1 : load
    value (value = 5) Thread1 : increment value (value = 6) Thread1 : store value (value =
    6) Le thread1 a fait ses 5 boucles
Thread2 : load value (value = 2) Thread2 : decrement value (value = 1) Thread2 : store value
     (value = 1) Thread2 : load value (value = 1) Thread2 : decrement value (value = 0)
    Thread2 : store value (value = 0) Thread2 : load value (value = 0) Thread2 : decrement
    value (value = -1) Thread2 : store value (value = -1) Thread2 : load value (value = -1)
    Thread2 : decrement value (value = -2) Thread2 : store value (value = -2) Thread2 : load
     value (value = -2) Thread2 : decrement value (value = -3) Thread2 : store value (value
    = -3) Le thread2 a fait ses 5 boucles
Apres les threads : value = -3
Fin du thread principal
$
```

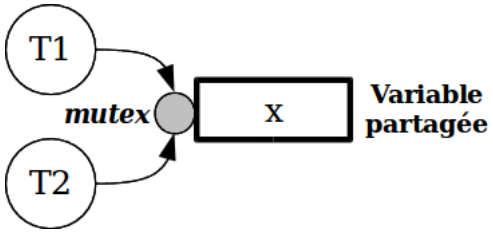
Conclusion : cette application n'assure pas une synchronisation des données entre les deux tâches.

Étape n°1 : mise en oeuvre d'un mutex

Pour résoudre ce genre de problème, le système doit permettre au programmeur d'utiliser un **verrou d'exclusion mutuelle**, c'est-à-dire de pouvoir bloquer, en une seule instruction (atomique), toutes les tâches tentant d'accéder à cette donnée, puis, que ces tâches puissent y accéder lorsque la variable est libérée.

Remarque : une instruction atomique est une instruction qui ne peut être divisée (donc interrompue).

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**.



Ce qu'il faut savoir et retenir :
- Un *mutex* peut êtte dans deux états : **déverrouillé** ou **verrouillé** (cela signifie qu'il est donc possédé par un *thread*).
- Un *mutex* est une ressource booléenne car il ne peut être pris que par un seul *thread* à la fois.
- Un *thread* qui tente de verrouiller un *mutex* déjà verrouillé est suspendu jusqu'à ce que le *mutex* soit déverrouillé.
- On peut donc faire deux opérations sur un *mutex* : **verrouiller** (*lock*) ou **déverrouiller** (*unlock*).

Remarque : il existe parfois l'opération *trylock*, équivalent à *lock*, mais qui en cas d'échec ne bloquera pas le *thread*.

```
// déclaration et initialisation d'un mutex
pthread_mutex_t globale_lock = PTHREAD_MUTEX_INITIALIZER;

int value_globale = 1; // la variable globale partagée

#define COUNT 5
...

pthread_mutex_lock(&globale_lock); // demande de verrouillage du mutex

... // je suis dans une zone d'exclusion mutuelle (section critique)
// je peux donc lire ou écrire dans la variable globale partagée en toute sécurité

pthread_mutex_unlock(&globale_lock); // déverrouillage du mutex
...
```

Exemple d'utilisation d'un mutex

Il faut lire la page *man* de `pthread_mutexattr_init(3)` pour plus d'informations sur les attributs de *mutex* et leur utilisation.

On va écrire maintenant le programme en C qui corrige le problème. Les deux tâches réalisent le même nombre de traitement (`COUNT`). On aura donc la variable globale (`value_globale`) qui revient à sa valeur initiale (`1`) puisqu'il y aura le même nombre d'incrémentation et de décrémentation.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define MUTEX

#ifdef MUTEX
pthread_mutex_t globale_lock = PTHREAD_MUTEX_INITIALIZER;
#endif

int value_globale = 1; // la variable globale partagée

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT 25

// Fonctions correspondant au corps d'un thread (tache)
void *increment(void *inutilise);
void *decrement(void *inutilise);

int main(void)
{
  pthread_t thread1, thread2;

  #ifdef MUTEX
  printf("Exemple avec le mutex\n");
  #endif

  printf("Avant les threads : value = %d\n", value_globale);
  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, decrement, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Apres les threads : value = %d\n", value_globale);
  printf("Fin du thread principal\n");
  pthread_exit(NULL);
  return EXIT_SUCCESS;
}

void *increment(void *inutilise)
{
   int value;
   int count = 0;

   while(1)
   {
      #ifdef MUTEX
      pthread_mutex_lock(&globale_lock);
      #endif
      value = value_globale;
```



```
      printf("Thread1 : load value (value = %d) ", value);
      value += 1;
      printf("Thread1 : increment value (value = %d) ", value);
      value_globale = value;
      printf("Thread1 : store value (value = %d) ", value_globale);
      #ifdef MUTEX
      pthread_mutex_unlock(&globale_lock);
      #endif
      count++;
      usleep(100000);
      if(count >= COUNT)
      {
         printf("Le thread1 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}

void *decrement(void *inutilise)
{
   int value;
   int count = 0;

   while(1)
   {
      #ifdef MUTEX
      pthread_mutex_lock(&globale_lock);
      #endif
      value = value_globale;
      printf("Thread2 : load value (value = %d) ", value);
      value -= 1;
      printf("Thread2 : decrement value (value = %d) ", value);
      value_globale = value;
      printf("Thread2 : store value (value = %d) ", value_globale);
      #ifdef MUTEX
      pthread_mutex_unlock(&globale_lock);
      #endif
      count++;
      usleep(100000);
      if(count >= COUNT)
      {
         printf("Le thread2 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

threads.2b.c

L'exécution de ce programme montre la **synchronisation effectuée grâce au *mutex***.

```
$ ./threads.2b
Exemple avec le mutex
```



```
Avant les threads : value = 1
Thread1 : load value (value = 1) Thread1 : increment value (value = 2) Thread1 : store value
     (value = 2) Thread2 : load value (value = 2) Thread2 : decrement value (value = 1)
    Thread2 : store value (value = 1) Thread1 : load value (value = 1) Thread1 : increment
    value (value = 2) Thread1 : store value (value = 2) Thread2 : load value (value = 2)
    Thread2 : decrement value (value = 1) Thread2 : store value (value = 1) Thread1 : load
    value (value = 1) Thread1 : increment value (value = 2) Thread1 : store value (value =
    2) Thread2 : load value (value = 2) Thread2 : decrement value (value = 1) Thread2 :
    store value (value = 1) Thread2 : load value (value = 1) Thread2 : decrement value (
    value = 0) Thread2 : store value (value = 0) Thread1 : load value (value = 0) Thread1 :
    increment value (value = 1) Thread1 : store value (value = 1) Thread2 : load value (
    value = 1) Thread2 : decrement value (value = 0) Thread2 : store value (value = 0)
    Thread1 : load value (value = 0) Thread1 : increment value (value = 1) Thread1 : store
    value (value = 1) Le thread2 a fait ses 5 boucles
Le thread1 a fait ses 5 boucles
Apres les threads : value = 1
Fin du thread principal
$
```

Conclusion

Les définitions à connaître :

- **Exclusion mutuelle** : Une ressource est en exclusion mutuelle si seul un *thread* peut utiliser la ressource à un instant donné.
- **Section Critique** : C'est une partie de code telle que deux *threads* ne peuvent s'y trouver au même instant.
- **Chien de garde (*watchdog*)** : Un chien de garde est une technique logicielle utilisée pour s'assurer qu'un programme ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti. Il s'agit en général d'un compteur qui est régulièrement remis à zéro. Si le compteur dépasse une valeur donnée (*timeout*) alors on procède à un redémarrage (*reset*) du système. Si une routine entre dans une boucle infinie, le compteur du chien de garde ne sera plus remis à zéro et un reset est ordonné.

Séquence n°2 : le mutex dans tous ses états !

Objectifs

L'objectif de cette séquence est de montrer les dangers liés à la synchronisation de données dans une application multi-tâches.

Étape n°1 : aie !

Si on reprend le programme en C précédent, que se passerait-il si un des deux *threads* ne dévérrouille pas le mutex ? Pour tester cela, il vous suffit de mettre en commentaire un des deux appels à `pthread_mutex_unlock(&globale_lock)`. On obtient alors une situation de **blocage** infini entre les deux *threads* :

```
$ ./threads.2b
Exemple avec le mutex
Avant les threads : value = 1
Ctrl-C
$
```

Conclusion

Les mécanismes de synchronisation peuvent conduire aux problèmes suivants :
- **Interblocage (*deadlocks*)** : Le phénomène d'interblocage est le problème le plus courant. L'interblocage se produit lorsque deux *threads* concurrents s'attendent mutuellement. Les *threads* bloqués dans cet état le sont définitivement.
- **Famine (*starvation*)** : Un processus léger ne pouvant jamais accéder à un verrou se trouve dans une situation de famine. Cette situation se produit, lorsqu'un processus léger, prêt à être exécuté, est toujours devancé par un autre processus léger plus prioritaire.
- **Endormissement (*dormancy*)** : cas d'un processus léger suspendu qui n'est jamais réveillé.

Séquence n°3 : les mutex dans différents environnements

Objectifs

L'objectif de cette séquence est la mise en oeuvre simple d'une synchronisation de données entre deux traitements multi-tâches mais avec différents langages (C++ et Java) et environnements de développement (*framework* Qt et Builder, API commoncpp et boost).

Étape n°1 : 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.

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

```
#ifndef MYTHREAD1_H
#define MYTHREAD1_H

#include <QThread>
#include <QMutex>

class MyDialog;

class MyThread1 : public QThread
{
   Q_OBJECT
   public:
      MyThread1(MyDialog *myDialog, QMutex *mutex);
      void run();
   private:
      int steps;
      MyDialog *myDialog;
      QMutex *mutex;
   signals:
      void newStep(int steps);
};

#endif
```

Déclarer son propre thread sous Qt

```
#include "mythread1.h"
#include "mydialog.h"

#include <QDebug>

MyThread1::MyThread1(MyDialog *myDialog, QMutex *mutex) : QThread()
{
   this->myDialog = myDialog;
   this->mutex = mutex;
}
```



```
void MyThread1::run()
{
   int i = 0;
   while(isRunning() && i < COUNT) {
      mutex->lock();
      steps = myDialog->getValue();
      qDebug() << "MyThread1 : load value (value = " << steps << ")";
      steps++;
      qDebug() << "MyThread1 : increment value (value = " << steps << ")";
      emit newStep(steps);
      qDebug() << "MyThread1 : store value (value = " << steps << ")";
      mutex->unlock();
      msleep(250);
      ++i;
   }
   //qDebug() << "MyThread1 is finish with " << myDialog->getValue();
}
```

Définir son propre thread sous Qt

```
#ifndef MYDIALOG_H
#define MYDIALOG_H

#include <QtGui>

class MyThread1; // ++
class MyThread2; // --
#define COUNT 10

class MyDialog : public QDialog
{
   Q_OBJECT
   public:
      MyDialog( QWidget *parent = 0 );
      int getValue();
      void setValue(int value);
   private:
      QProgressBar *progressBar; // l'affichage de la variable partagée
      QPushButton *bSimuler;
      QPushButton *bQuitter;
      MyThread1 *myThread1; // ++
      MyThread2 *myThread2; // --
      QMutex mutex; // le mutex
      int value; // la variable partagée
   private slots:
      void simuler();
      void arreter();
      void perform(int steps);
      void terminer();
};

#endif
```

Déclaration d'une boîte de dialogue utilisant les threads sous Qt

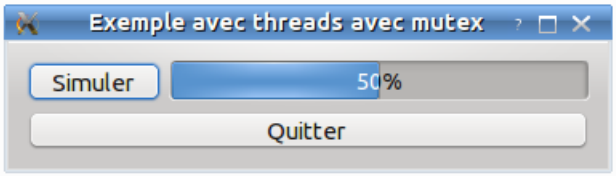

```
#include <QtGui>

#include "mydialog.h"
#include "mythread1.h"
#include "mythread2.h"

#include <QDebug>

MyDialog::MyDialog( QWidget *parent ) : QDialog( parent )
{
   progressBar = new QProgressBar(this);
   progressBar->setMaximum(100);
   this->value = 50;
   progressBar->setValue(this->value);

   bSimuler  = new QPushButton(QString::fromUtf8("Simuler"), this);
   bQuitter  = new QPushButton("Quitter", this);

   QVBoxLayout *vLayoutB = new QVBoxLayout;
   QHBoxLayout *hLayout1 = new QHBoxLayout;
   QHBoxLayout *hLayout2 = new QHBoxLayout;
   QVBoxLayout *mainLayout = new QVBoxLayout;
   vLayoutB->addWidget(bSimuler);
   hLayout1->addLayout(vLayoutB);
   hLayout1->addWidget(progressBar);
   hLayout2->addWidget(bQuitter);
   mainLayout->addLayout(hLayout1);
   mainLayout->addLayout(hLayout2);
   setLayout(mainLayout);

   setWindowTitle(QString::fromUtf8("Exemple avec threads avec mutex"));
   setFixedHeight(sizeHint().height());

   myThread1 = new MyThread1(this, &mutex);
   connect(myThread1, SIGNAL(newStep(int)), this, SLOT(perform(int)), Qt::
      BlockingQueuedConnection);
   myThread2 = new MyThread2(this, &mutex);
   connect(myThread2, SIGNAL(newStep(int)), this, SLOT(perform(int)), Qt::
      BlockingQueuedConnection);

   connect(bSimuler, SIGNAL(clicked()), this, SLOT(simuler()));
   connect(bQuitter, SIGNAL(clicked()), this, SLOT(terminer()));
}

void MyDialog::simuler()
{
   this->value = 50; // valeur initiale
   progressBar->setValue(this->value);
   qDebug() << "MyDialog is start with " << getValue();
   if (!myThread1->isRunning())
   {
      myThread1->start();
   }
```

```
   if (!myThread2->isRunning())
   {
      myThread2->start();
   }
}

void MyDialog::arreter()
{
   if (myThread1->isRunning())
   {
      myThread1->terminate();
      myThread1->wait();
   }
   if (myThread2->isRunning())
   {
      myThread2->terminate();
      myThread2->wait();
   }
   qDebug() << "MyDialog is stop with " << getValue();
}

void MyDialog::perform(int steps)
{
   setValue(steps);
}

void MyDialog::terminer()
{
   arreter();
   close();
}

int MyDialog::getValue()
{
   //qDebug() << "MyDialog::getValue() -> " << value;
   return value;
}

void MyDialog::setValue(int value)
{
   //qDebug() << "MyDialog::setValue() with " << value;
   if(value >= 0 && value <= progressBar->maximum())
   {
      this->value = value;
      progressBar->setValue(this->value);
   }
}
```

Définition d'une boîte de dialogue utilisant les threads sous Qt

Notez la présence de l'argument `Qt::BlockingQueuedConnection` dans l'appel à `connect()`. Ceci est nécessaire pour bloquer le *thread* qui émet un signal jusqu'à ce que la méthode *slot* soit terminée.

L'application obtenue :



Étape n°2 : les mutex sous Builder

La mise en place des *mutex* sous *Builder* revient à créer une section critique en instanciant un objet de la classe `TCriticalSection` fournie par l'API et à appeler la méthode `Acquire()` pour entrer dans la section critique et la méthode `Release()` pour en sortir.

```
//---------------------------------------------------------------------------
#ifndef TMyThread1H
#define TMyThread1H

#include <Classes.hpp>
#include <SyncObjs.hpp> /* pour TCriticalSection */

class TMyThread1 : public TThread
{
private:
  int steps;
  TCriticalSection *mutex;
  void __fastcall UpdateEtat();

protected:
    void __fastcall Execute();

public:
    __fastcall TMyThread1(bool CreateSuspended, TCriticalSection *mutex);
};
//---------------------------------------------------------------------------
#endif
```

Déclarer son propre thread sous Builder

```
#include <vcl.h>
#pragma hdrstop

#include "TMyThread1.h"
#include "ThreadIHM.h"
#pragma package(smart_init)

__fastcall TMyThread1::TMyThread1(bool CreateSuspended, TCriticalSection *mutex)
    : TThread(CreateSuspended)
{
  this->mutex = mutex;
  Priority = tpNormal;
  steps = Form1->getValue();
}
```



```
void __fastcall TMyThread1::Execute()
{
    try
    {
         for(int i = 0; (i < COUNT && !Terminated); i++)
         {
            mutex->Acquire();
            steps = Form1->getValue();
            steps++;
            Form1->setValue(steps);
            Synchronize(UpdateEtat);
            mutex->Release();
            Sleep(100); //pour voir dans l'ihm
         }
    }
    catch (...)
    {
      // faire quelque chose avec les exceptions
    }
}

void __fastcall TMyThread1::UpdateEtat()
{
  Form1->trackBar->Position = Form1->getValue();
}
```

Définir son propre thread sous Builder

```
#ifndef ThreadIHMH
#define ThreadIHMH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
#include <ComCtrls.hpp>

class TMyThread1;
class TMyThread2;
#define COUNT 20

class TForm1 : public TForm
{
__published: // Composants gérés par l'EDI
   TButton *buttonSimuler;
   TButton *buttonQuitter;
   TStaticText *StaticText2;
   TTrackBar *trackBar;
   void __fastcall buttonSimulerClick(TObject *Sender);
   void __fastcall buttonQuitterClick(TObject *Sender);
private: // Déclarations de l'utilisateur
   TMyThread1 *myThread1;
```



```
   TMyThread2 *myThread2;
   int value;
   TCriticalSection *mutex;

public: // Déclarations de l'utilisateur
    __fastcall TForm1(TComponent* Owner);
    __fastcall ~TForm1();
    int __fastcall getValue();
    void __fastcall setValue(int value);
};

extern PACKAGE TForm1 *Form1;
#endif
```

Déclaration d'une application utilisant les threads sous Builder

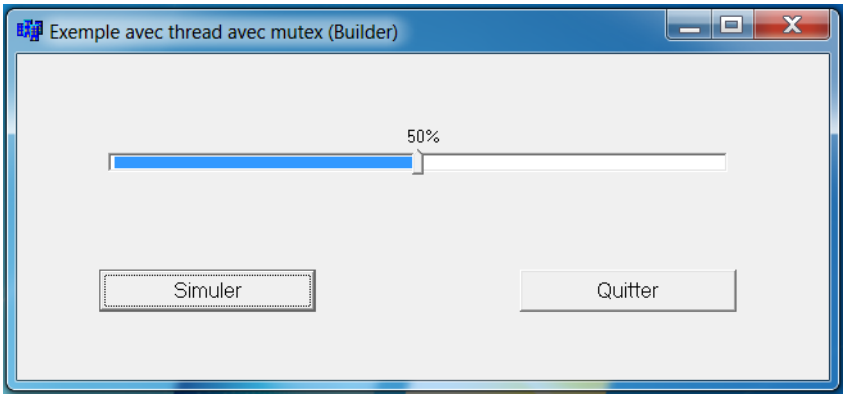
```
#include <vcl.h>
#pragma hdrstop

#include "ThreadIHM.h"
#include "TMyThread1.h"
#include "TMyThread2.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
  value = 50; //trackBar->Position
  buttonSimuler->Enabled = true;
  buttonQuitter->Enabled = true;
  myThread1 = NULL;
  myThread2 = NULL;
  mutex = new TCriticalSection;
}

__fastcall TForm1::~TForm1()
{
   delete mutex;
}

void __fastcall TForm1::buttonSimulerClick(TObject *Sender)
{
  myThread1 = new TMyThread1(true, mutex);
  myThread2 = new TMyThread2(true, mutex);
  myThread1->Resume();
  myThread2->Resume();
}

void __fastcall TForm1::buttonQuitterClick(TObject *Sender)
{
   if(myThread1 != NULL)
```



```
      myThread1->Terminate();
   if(myThread2 != NULL)
      myThread2->Terminate();
   Close();
}

int __fastcall TForm1::getValue()
{
   return value;
}

void __fastcall TForm1::setValue(int value)
{
   this->value = value;
}
```

Définition d'une application utilisant les threads sous Builder

L'application obtenue :



Étape n°3 : les mutex en C++ avec l'API commoncpp

La mise en place des *mutex* revient à instancier un objet de la classe `Mutex` fournie par l'API et à appeler la méthode `lock()` pour vérouiller le *mutex* et la méthode `unlock()` pour le dévérouiller.

```
#ifndef MYTHREAD1_H
#define MYTHREAD1_H

#include <cc++/thread.h>
#include <iostream>

#ifdef  CCXX_NAMESPACES
using namespace std;
using namespace ost;
#endif

// extern permet d'indiquer au compilateur que cette variable globale
// existe mais qu'elle est déclarée ailleurs
extern int value_globale;

// Chaque thread (tache) va faire ses COUNT boucles
```

```
#define COUNT 5

using namespace std;

class MyThread1 : public Thread
{
   private:
      bool stopped;
      int value;
      Mutex *mutex;
      void run();
      void final();

   public:
      MyThread1(int, int, Mutex *mutex);
      ~MyThread1();
      void stop();
};

#endif
```

Déclarer son propre thread avec commoncpp

```
#include "mythread1.h"

MyThread1::MyThread1(int prio, int stack, Mutex *mutex) : Thread(prio, stack)
{
   //setCancel(cancelDeferred);
   setCancel(cancelImmediate);
   stopped = false;
   this->mutex = mutex;
}

MyThread1::~MyThread1()
{
   // Rien à faire !
}

void MyThread1::run()
{
   stopped = false;
   cout << "\n";
   for(int i=1;(i<=COUNT && !stopped);i++)
   {
      mutex->enter();
      value = value_globale;
      cout << "Thread1 : load value (value = " << value << ") \n";
      value += 1;
      cout << "Thread1 : increment value (value = " << value << ") \n";
      value_globale = value;
      cout << "Thread1 : store value (value = " << value_globale << ") \n";
      mutex->leave();
      Thread::sleep(100); // 100 ms
   }
```



```
   cout << "\n";
   exit(); // appelle automatiquement final()
}

void MyThread1::stop()
{
   if(isRunning())
      stopped = true;
}

void MyThread1::final()
{
   // Rien à faire !
}
```

Définir son propre thread avec commoncpp

```
#include <iostream>

#include "mythread1.h"
#include "mythread2.h"

using namespace std;


int value_globale = 1;

int main(void)
{
   char choix;
   bool fini = false;
   MyThread1 *myThread1;
   MyThread2 *myThread2;
   Mutex mutex;

   myThread1 = new MyThread1(0, 0, &mutex);
   myThread2 = new MyThread2(0, 0, &mutex);

   cout << "Avant les threads : value = " << value_globale << endl;

   myThread1->start();
   myThread2->start();

   myThread1->join();
   myThread2->join();

   cout << "Apres les threads : value = " << value_globale << endl;

   delete myThread1;
   delete myThread2;

   return 0;
}
```

Exemple d'une application utilisant les threads et les mutex avec commoncpp

L'application obtenue :

```
[tv@alias commonc++-avec-thread-avec-mutex]$ ./main
Avant les threads : value = 1


Thread1 : load value (value = 1)
Thread1 : increment value (value = 2)
Thread1 : store value (value = 2)
Thread2 : load value (value = 2)
Thread2 : decrement value (value = 1)
Thread2 : store value (value = 1)
Thread1 : load value (value = 1)
Thread1 : increment value (value = 2)
Thread1 : store value (value = 2)
Thread2 : load value (value = 2)
Thread2 : decrement value (value = 1)
Thread2 : store value (value = 1)
Thread1 : load value (value = 1)
Thread1 : increment value (value = 2)
Thread1 : store value (value = 2)
Thread2 : load value (value = 2)
Thread2 : decrement value (value = 1)
Thread2 : store value (value = 1)
Thread1 : load value (value = 1)
Thread1 : increment value (value = 2)
Thread1 : store value (value = 2)
Thread2 : load value (value = 2)
Thread2 : decrement value (value = 1)
Thread2 : store value (value = 1)
Thread1 : load value (value = 1)
Thread1 : increment value (value = 2)
Thread1 : store value (value = 2)
Thread2 : load value (value = 2)
Thread2 : decrement value (value = 1)
Thread2 : store value (value = 1)


Apres les threads : value = 1
[tv@alias commonc++-avec-thread-avec-mutex]$
```

Vous trouverez aussi dans l'archive un exemple pour l'API *boost*.

Étape n°4 : les mutex en Java

La mise en place des *mutex* revient à instancier un objet de la classe `Lock` (ou `ReentrantLock`) fournie par l'API et à appeler la méthode `lock()` pour vérouiller le *mutex* et la méthode `unlock()` pour le dévérouiller.

```
import java.lang.*;
import java.util.*;
import java.io.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class MonMain
{
   public static void main(String[] args)
   {
      boolean fini = false;
      BufferedReader clavier = new BufferedReader(new InputStreamReader(System.in));
      String choix = null;

      MyThread1 myThread1 = null;
      MyThread2 myThread2 = null;
      Lock mutex = new ReentrantLock(true); // le mutex
      Value value = new Value(1);

      try
      {
         while(!fini)
         {
            System.out.println("Simuler ou Quitter ? (s|q) ");
            choix = clavier.readLine();

            if (choix.compareTo("q") == 0)
            {
               if(myThread1 != null)
                  myThread1.stop();
               if(myThread2 != null)
                  myThread2.stop();
               fini = true;
            }
            if (choix.compareTo("s") == 0)
            {
               myThread1 = new MyThread1(value, mutex);
               myThread2 = new MyThread2(value, mutex);
               myThread1.start();
               myThread2.start();
            }
         }

         myThread1.join();
         myThread2.join();
         System.out.println("final value = " + value.getValue() + "");
      }
      catch (IOException e)
```



```
      {
         System.out.println("IOException !");
      }
      catch(InterruptedException e)
      {
         System.out.println("InterruptedException !");
      }
   }
}

class Value
{
   private int value; // la variable partagée

   public int getValue()
   {
      return value;
   }

   public void setValue(int value)
   {
      this.value = value;
   }

   public Value(int value)
   {
      this.value = value;
   }
}

class MyThread1 extends Thread
{
   Value value;
   Lock mutex;
   private int valeur;
   private static final int COUNT = 10;

   public MyThread1(Value value, Lock mutex)
   {
      this.value = value;
      this.mutex = mutex;
   }

   public void run()
   {
      try
      {
         for(int i=0;i<COUNT;i++)
         {
            mutex.lock();
            valeur = value.getValue();
            System.out.println("Thread1 : load value (value = " + valeur + ")");
            ++valeur;
```



```
            System.out.println("Thread1 : increment value (value = " + valeur + ")");
            value.setValue(valeur);
            System.out.println("Thread1 : store value (value = " + valeur + ")");
            mutex.unlock();
            Thread.sleep(100); // 100 ms
         }
         System.out.println("");
      }
      catch(InterruptedException e)
      {
         System.out.println("InterruptedException !");
      }
      finally
      {
         // Rien à faire !
      }
   }
}

class MyThread2 extends Thread
{
   // TODO
}
```

Exemple d'une application utilisant les threads et les mutex en Java

Pour compiler cette application, il vous faudra faire pour générer les fichiers `.class` :

```
$ javac MonMain.java
$
```

Et pour l'exécuter :

```
$ java MonMain
$
```


Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés des séquences précédentes.

Question 1. Est-ce qu'un *thread* peut accéder aux variables globales de son processus ? Si oui, Quel est le problème que cela engendre ?

Question 2. Qu'est-ce qu'un *mutex* ?

Question 3. Dans quels états peut être un *mutex* ?

Question 4. Qu'est-ce qu'une ressource en exclusion mutuelle ?

Question 5. Qu'est-ce qu'une section critique ?

Question 6. Quels dangers entraînent l'utilisation de mécanismes de synchronisation comme les *mutex* ?

Travail demandé

Exercice 1 :

L'objectif de cet exercice est de mettre en oeuvre une synchronisation de données entre deux traitements multi-tâches.

Question 7. Coder le *thread* qui assure la décrémentation en utilisant un *mutex* pour la synchronisation et tester l'application dans le langage ou l'environnement de votre choix.

