TP Threads n°1

Sommaire

Les objectifs de ce tp sont de découvrir la programmation multi-tâches à base de *threads* dans différents environnements de développement.

Les threads

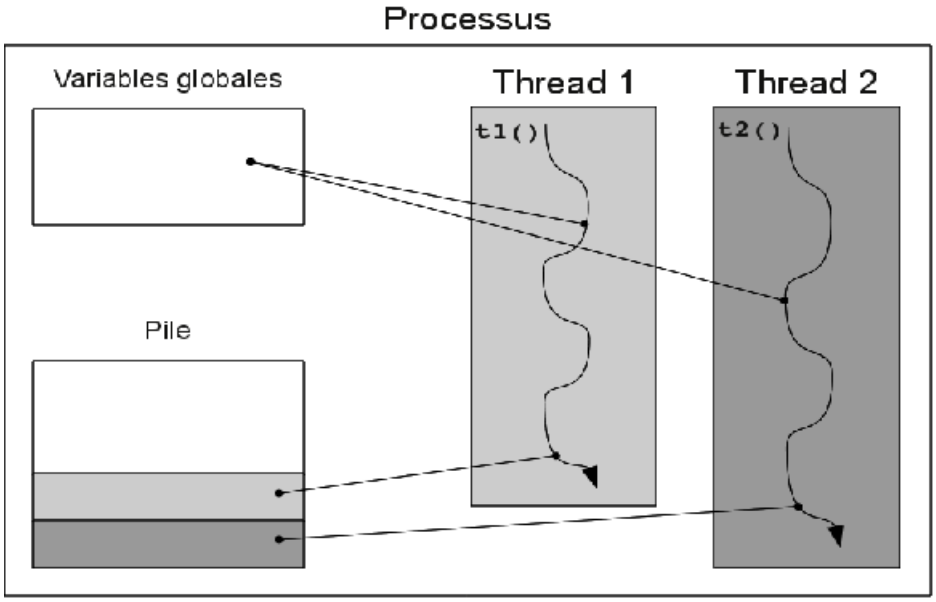
Définition

Il existe plusieurs traduction du terme *thread* :
– **fil** d'exécution,
– activité, tâche
– *lightweight process* (*lwp*) ou processus léger

Par opposition, un processus créé par `fork()` sera qualifié de processus lourd.

Les *threads* permettent de dérouler plusieurs suites d'instructions, en **PARALLELE**, à l'intérieur d'un même processus. Un *thread* exécutera donc une **fonction**.

Besoin : on aura besoin de thread(s) dans une application lorsqu'on devra paralléliser des traitements.



Donc, un *thread* :
– est englobé par un processus
– dispose de sa propre pile pour implanter ses variables locales,
– partage les données globales avec les autres threads.

Il ne faut pas confondre la technologie *Hyperthreading* incluse dans certains processeurs Intel avec les *threads*. Cette technologie permet en effet aussi bien l'exécution simultanée de processus lourds que de *threads*.

Manuel du pogrammeur

Les systèmes d'exploitation mettent en oeuvre généralement les *threads*. C'est le cas des systèmes Unix/Linux et Microsoft ©Windows.

La plupart des langages (Java sur la plupart des systèmes d'exploitation, C++, C# .NET, ...) utilisent des extensions du langage ou des bibliothèques pour utiliser directement les services de *multithreading* du système d'exploitation.

En 1995, un standard d'interface pour l'utilisation des processus légers a vu le jour, ce standard est connu sous le nom de *pthread* (ou POSIX thread 1003.1c-1995).

Le développeur utilisera donc concrètement une interface pour programmer une application multi-tâche grâce par exemple :
- au **standard POSIX `pthread`** largement mis en oeuvre sur les systèmes UNIX/Linux
- à l'API **WIN32 threads** fournie par Microsoft ©Windows pour les processus légers

Les pages man sous Unix/Linux décrivent les appels de l'API `pthread` :
- pthread_create(3) : crée (et exécute) un nouveau *thread*
- ...
- pthreads(7) : standard POSIX `pthread`

L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 3 pthread_create`

L'utilisation des *threads* POSIX est tout à fait possible en C++. Toutefois, il sera conseillé d'utiliser un *framework* C++ (Qt, Builder, commoncpp, boost, ACE, ...).

Fonctionnalités des threads

Les fonctionnalités suivantes sont présentes dans la norme **POSIX thread 1003.1c-1995** :
- gestion des processus légers (*thread management*) : initialisation, création, destruction ... et l'annulation (*cancellation*) ;
- gestion de la synchronisation : exclusion mutuelle (*mutex*), variables conditions ;
- données privées (*thread-specific data*) ;
- ordonnancement (*thread priority scheduling*) : gestion des priorités, ordonnancement préemptif ;
- signaux : traitant des signaux (*signal handler*), attente asynchrone, masquage des signaux, saut dans un programme (*long jumps*) ;

Les processus légers WIN32 sont implantés au niveau du noyau. L'unité d'exécution finale est le processus léger. L'ordonnancement est réalisé selon l'algorithme du tourniquet (*round-robin scheduling*). Tous les processus légers accèdent aux mêmes ressources du processus lourd, en particulier à son espace mémoire. Un processus lourd accède à une espace mémoire de 4 Gigaoctets (Go), découpé en 2 Go pour l'utilisateur et 2 Go pour le système d'exploitation. Chaque processus léger peut accéder à cette zone de 2 Go. Un processus léger d'un processus lourd ne peut pas accéder aux ressources d'un autre processus lourd.

Les fonctionnalités suivantes sont fournies avec les processus légers **WIN 32 threads** :
- gestion des processus légers : création, terminaison, priorités, suspension.
- gestion de la synchronisation : sections critiques, mutex, sémaphores, événements.

– communication : une file de message associée à chaque processus léger permet d'envoyer des messages à des processus légers du système.

La synchronisation sur un objet noyau est réalisée en utilisant une fonction commune (`WaitForSingleObject`). Une autre fonction permet également l'attente multiple sur une synchronisation (`WaitForMultipleObject`).

Séquence n°1 : mise en évidence du multi-tâche

Objectifs

L'objectif de cette séquence est la mise en oeuvre simple d'une application à base de *threads*.

Étape n°0 : création de deux fonctions

On crée deux fonctions : une affichera des étoiles '*' et l'autre des dièses '#'.

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

// Fonctions :
void etoile(void);
void diese(void);

int main(void)
{
  setbuf(stdout, NULL); // pas de tampon sur stdout
  printf("Je vais lancer les 2 fonctions\n");
  etoile();
  diese();

  return EXIT_SUCCESS;
}

void etoile(void)
{
  int i;
  char c1 = '*';

  for(i=1;i<=200;i++)
  {
     write(1, &c1, 1);// écrit un caractère sur stdout (descripteur 1)
  }
  return;
}


void diese(void)
{
  int i;
  char c1 = '#';
```



```
  for(i=1;i<=200;i++)
  {
      write(1, &c1, 1);
  }
  return;
}
```

threads.0.c

L'exécution de ce programme montre la **séquentialité des traitements effectués** : tout d'abord la fonction `etoile()` s'exécute entièrement puis la fonction `diese()` est lancée et s'exécute à son tour.

```
$ ./threads.0
Je vais lancer les 2 fonctions
**********************************************************
**********************************************************
**********************************************************
**************************##########################################
##########################################################
##########################################################
#####################################################
$
```

Cette application n'est donc pas multi-tâche car les traitements ne se sont pas effectués en parallèle mais l'un après l'autre.

Étape n°1 : création de deux *threads*

On crée maintenant deux tâches (*threads*) pour exécuter chacune des deux fonctions : une affichera des étoiles '*' et l'autre des dièses '#'.

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

// Fonctions correspondant au corps d'un thread (tache)
void *etoile(void *inutilise);
void *diese(void *inutilise);
// Remarque : le prototype d'une tâche doit être : void *(*start_routine)(void *)

int main(void)
{
  pthread_t thrEtoile, thrDiese; // les ID des de 2 threads

  setbuf(stdout, NULL); // pas de tampon sur stdout

  printf("Je vais creer et lancer 2 threads\n");
  pthread_create(&thrEtoile, NULL, etoile, NULL);
  pthread_create(&thrDiese, NULL, diese, NULL);

  //printf("J'attends la fin des 2 threads\n");
  pthread_join(thrEtoile, NULL);
```



```
  pthread_join(thrDiese, NULL);
  printf("\nLes 2 threads se sont termines\n");

  printf("Fin du thread principal\n");
  pthread_exit(NULL);

  return EXIT_SUCCESS;
}

void *etoile(void *inutilise)
{
  int i;
  char c1 = '*';

  for(i=1;i<=200;i++)
  {
     write(1, &c1, 1);// écrit un caractère sur stdout (descripteur 1)
  }

  return NULL;
}

void *diese(void *inutilise)
{
  int i;
  char c1 = '#';

  for(i=1;i<=200;i++)
  {
      write(1, &c1, 1);
  }

  return NULL;
}
```

threads.1a.c

Ici le paramètre `inutilise` n'est pas utilisé mais requis pour respecter le prototype de `pthread_create()`.

Étape n°2 : compilation et édition de liens

Il faut compiler et faire l'édition des liens avec l'option `-pthread`.

```
$ gcc -o threads.1a threads.1a.c -D_REENTRANT -pthread
$
```


Étape n°3 : exécution

L'exécution de ce programme montre la **parallélisation des traitements effectués** : la fonction `etoile()` s'exécute en parallèle avec la fonction `diese()`.

```
$ ./threads.1
Je vais creer et lancer 2 threads
*##################################################*
*##################################################*
*##################################################*
*##################################################*
*##################################################*
*##################################################*
**************************************************
Les 2 threads se sont termines
Fin du thread principal
$
```

La version 1b fournie permet de voir le gestionnaire de processus pendant l'exécution du programme :

```
$ ./threads.1b
Je vais creer et lancer 2 threads
UID        PID  PPID   LWP  C NLWP STIME TTY        TIME CMD
tv       11111 10961 11111 0   1 14:08 pts/5  00:00:00 bash
tv       30556 11111 30556 0   3 22:18 pts/5  00:00:00 ./threads.1b
tv       30556 11111 30557 0   3 22:18 pts/5  00:00:00 ./threads.1b
tv       30556 11111 30558 0   3 22:18 pts/5  00:00:00 ./threads.1b
tv       30559 30556 30559 0   1 22:18 pts/5  00:00:00 ps -Lf
**************************************************
#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*
**************************************************
#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
**************************************************
#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
**********************************************
Les 2 threads se sont termines
Fin du thread principal
$
```

Les *threads* ont le même `PID` que le processus lourd qui les englobe. Par contre, ils possèdent un TID (*Thread IDentifier*) affichée ici dans la colonne `LWP`. La colonne `NLWP` indique le nombre de threads sachant que le processus lourd (la fonction `main()` du programme) est considérée comme le *thread* principal (son TID est son `PID`).

Séquence n°2 : les threads en C++

Objectifs

L'objectif de cette séquence est d'expliquer la mise en oeuvre des *threads* POSIX en C++.

Étape n°0 : adresse d'une méthode

Le premier problème que l'on rencontre en C++ est de pouvoir passer l'adresse d'une méthode à la fonction `pthread_create()` comme on le faisait en C pour une fonction. Ce n'est pas possible car les méthodes d'une classe n'ont pas d'adresse au moment de l'édition de liens.

La solution consiste à déclarer la méthode comme **statique**. Cette méthode `static` sera donc disponible en dehors de toute instance de la classe et pourra donc être passée en argument de `pthread_create()`. Par contre elle ne pourra pas accéder aux attributs de sa classe en dehors des autres membres statiques.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;

class foo
{
  public:
  static void *go(void *arg)
  {
      char c1 = '#';
      for(int i=1;i<=100;i++)
      {
         write(1, &c1, 1);
      }
  }
};

int main(void)
{
  pthread_t leThread;
  if(pthread_create(&leThread, NULL, foo::go, NULL))
  {
      cout << "Echec de la creation du thread !" << endl;
      return 1;
  }
  cout << "Creation et démarrage du thread" << endl;
  sleep(1);
  return 0;
}
```

Utilisation d'une méthode statique en C++

On obtient bien une exécution multi-tâche :

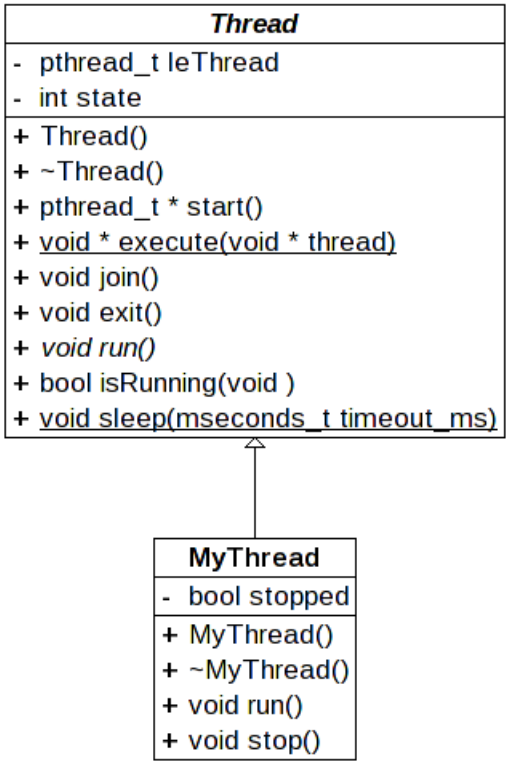
```
$ ./main
####################Creation et démarrage du thread
################################################################################
$
```


Étape n°1 : une classe Thread

On va maintenant réaliser le squelette simpliste d'une véritable classe `Thread` (libre à vous pour la terminer). Le principe décrit dans cette partie est celui que l'on retrouve le plus souvent dans les APIs C++.

Notre classe `Thread` est une classe **abstraite** qui contiendra :
– une méthode virtuelle pure `run()`
– et évidemment une méthode statique `execute()`

Pour utiliser cette classe, il faudra donc obligatoirement **créer une classe dérivée** en écrivant le code de la méthode **`run()`** qui représentera notre "fonction" *thread*. Notre classe dérivée pourra en outre contenir d'autres membres.



L'exemple ci-dessous est volontairement simplifiée pour montrer seulement le principe de l'implémentation de cette classe.

```
#ifndef MYTHREAD_H
#define MYTHREAD_H

#include <pthread.h>
#include <iostream>

using namespace std;

typedef unsigned long mseconds_t;

class Thread
{
   private:
      pthread_t leThread;
      int state;
```

```
   public:
      Thread();
      ~Thread();

      pthread_t* start();
      static void* execute(void* thread);
      void join();
      void exit();
      virtual void run() = 0;
      bool isRunning(void) const;
      static void sleep(mseconds_t timeout_ms);
};

class MyThread : public Thread
{
   private:
      bool stopped;

   public:
      MyThread();
      ~MyThread();

      void run();
      void stop();
};

#endif
```

Le fichier header

```
#include "mythread.h"

Thread::Thread()
{
   state = 0;
   leThread = 0;
}

Thread::~Thread()
{
   if(state == 1)
      pthread_cancel(leThread);
   pthread_detach(leThread);
}

pthread_t* Thread::start()
{
   if(!pthread_create(&(this->leThread), NULL, Thread::execute, static_cast<void *>(this)))
   {
      state = 1;
      return &(this->leThread);
   }
   else
```

```
   {
      state = 0;
      return NULL;
   }
}

void* Thread::execute(void* thread)
{
   static_cast<Thread *>(thread)->run();

   pthread_exit(0);
}

void Thread::join()
{
   void *value = 0;

   pthread_join(leThread, &value);
}

void Thread::exit()
{
   pthread_exit(NULL);
}

bool Thread::isRunning(void) const
{
    if(state == 0)
        return false;

    return (leThread != 0) ? true : false;
}

void Thread::sleep(mseconds_t timeout_ms)
{
    usleep(timeout_ms * 1000);
}


MyThread::MyThread():Thread()
{
   stopped = false;
}

MyThread::~MyThread()
{
}

void MyThread::run()
{
   char c1 = '#';

   stopped = false;
```



```
   cout << "\n";
   for(int i=1;(i<=100 && !stopped);i++)
   {
      write(1, &c1, 1);
      Thread::sleep(100); // 100 ms
   }
   cout << "\n";
   exit();
}

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

Le fichier source

Après avoir instancié un objet de type `MyThread`, il suffit d'appeler la méthode `start()` pour créer et démarrer le *thread* (en fait sa méthode `run()` sera exécutée).

```
#include <iostream>
#include <cstdlib>
#include <unistd.h>
#include "mythread.h"

using namespace std;

int main(void)
{
   char choix;
   bool fini = false;
   MyThread *myThread = new MyThread;

   while(!fini)
   {
      cout << "Simuler, Arrêter ou Quitter ? (s|a|q) ";
      cin >> choix;

      switch(tolower(choix))
      {
         case 'a' : myThread->stop();
                    break;
         case 'q' : myThread->stop();
                    fini = true;
                    break;
         case 's' : myThread->start();
                    break;
      }
   }
   myThread->join();
   delete myThread;
   return 0;
}
```

Exemple d'utilisation de la classe MyThread

Séquence n°3 : les threads dans différents environnements

Objectifs

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

Étape n°0 : besoin

On veut intégrer une IHM (Interface Homme-Machine) à notre exemple de la séquence n°1. L'utilisateur pourra donc cliquer (ou saisir) sur :
- '`Simuler`' pour simuler un traitement multi-tâche (création d'un *thread*) avec un affichage dans l'IHM d'une barre de progression (de 0 à 100%)
- '`Arrêter`' pour arrêter le traitement en cours du *thread*
- '`Quitter`' pour arrêter le traitement en cours du *thread* et quitter

Si l'application n'utilise pas les *threads*, l'IHM risquerait d'être bloquée jusqu'à la fin de son traitement avant de pouvoir reprendre son contrôle. Des exemples (pour Qt et Builder) sans *thread* met en évidence ce problème !

Étape n°1 : les threads sous Qt

La mise en place des *threads* revient à dériver une classe de base fournie par l'API et à écrire le code du *thread* dans une méthode spécifique. Sous **Qt**, on dérive une classe `QThread` et on écrit le code du *thread* dans la méthode `run()`. Ensuite, on appelera la méthode `start()` pour démarrer le *thread* et la méthode `stop()` pour l'arrêter.

```
#ifndef MYTHREAD_H
#define MYTHREAD_H

#include <QThread>

class MyThread : public QThread
{
   Q_OBJECT
   public:
      MyThread();
      void run();

   private:
      int steps;

   signals:
      void newStep(int steps);
};

#endif
```

Déclarer son propre thread sous Qt


```
#include "mythread.h"

MyThread::MyThread() : QThread()
{
   steps = 0;
}

void MyThread::run()
{
   steps = 0;
   while(isRunning())
   {
      steps++;
      msleep(250);
      emit newStep(steps);
   }
}
```

Définir son propre thread sous Qt

```
#ifndef MYDIALOG_H
#define MYDIALOG_H

#include <QtGui>

#include "mythread.h"

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

   private:
      QProgressBar *progressBar;
      QPushButton *bSimuler;
      QPushButton *bArreter;
      QPushButton *bQuitter;
      MyThread *myThread;

   private slots:
      void simuler();
      void arreter();
      void perform(int steps);
};

#endif
```

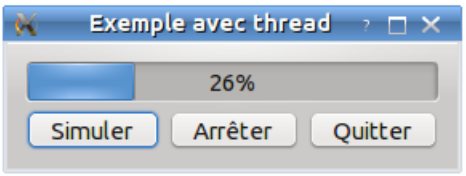
Déclaration d'une boîte de dialogue utilisant les threads sous Qt


```
#include <QtGui>

#include "mydialog.h"

MyDialog::MyDialog( QWidget *parent ) : QDialog( parent )
{
   progressBar = new QProgressBar(this);
   progressBar->setMaximum(100);
   progressBar->setValue(0);

   bSimuler  = new QPushButton(QString::fromUtf8("Simuler"), this);
   bArreter = new QPushButton(QString::fromUtf8("Arrêter"), this);
   bQuitter  = new QPushButton("Quitter", this);
   connect(bSimuler, SIGNAL(clicked()), this, SLOT(simuler()));
   connect(bArreter, SIGNAL(clicked()), this, SLOT(arreter()));
   connect(bQuitter, SIGNAL(clicked()), this, SLOT(close()));

   QVBoxLayout *vLayout1 = new QVBoxLayout;
   QHBoxLayout *hLayout1 = new QHBoxLayout;
   QVBoxLayout *mainLayout = new QVBoxLayout;
   vLayout1->addWidget(progressBar);
   hLayout1->addWidget(bSimuler);
   hLayout1->addWidget(bArreter);
   hLayout1->addWidget(bQuitter);
   mainLayout->addLayout(vLayout1);
   mainLayout->addLayout(hLayout1);
   setLayout(mainLayout);

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

   myThread = new MyThread;
   connect(myThread, SIGNAL(newStep(int)), this, SLOT(perform(int)));
}

void MyDialog::simuler()
{
   if (!myThread->isRunning())
   {
      myThread->start();
   }
}

void MyDialog::arreter()
{
   if (myThread->isRunning())
   {
      myThread->terminate();
      myThread->wait();
   }
   progressBar->setValue(0);
}
```

```
void MyDialog::perform(int steps)
{
   if(steps <= progressBar->maximum())
      progressBar->setValue(steps);
}
```

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

L'application obtenue :



Étape n°2 : les threads sous Builder

La mise en place des *threads* revient à dériver une classe de base fournie par l'API et à écrire le code du *thread* dans une méthode spécifique. Sous **Builder**, on dérive une classe `TThread` et on écrit le code du *thread* dans la méthode `Execute()`. Ensuite, on appelera la méthode `Resume()` pour démarrer le *thread* et la méthode `Terminate()` pour l'arrêter.

```
#ifndef TMyThreadH
#define TMyThreadH

#include <Classes.hpp>

class TMyThread : public TThread
{
   private:
      int steps;
      void __fastcall UpdateEtat();

   protected:
      void __fastcall Execute();

   public:
      __fastcall TMyThread(bool CreateSuspended);
      void __fastcall DoTerminate(void);
};

#endif
```

Déclarer son propre thread sous Builder

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

#include "TMyThread.h"
#include "ThreadIHM.h"
#pragma package(smart_init)
```

```
__fastcall TMyThread::TMyThread(bool CreateSuspended) : TThread(CreateSuspended)
{
  Priority = tpNormal;
  steps = 0;
}

void __fastcall TMyThread::Execute()
{
    try
    {
         steps = 0;
         for(int i = 0; (i < 100 && !Terminated); i++)
         {
            steps++;
            Synchronize(UpdateEtat);
            Sleep(100); //pour voir dans l'ihm
         }
    }
    catch (...)
    {
      // faire quelque chose avec les exceptions
    }
}

void __fastcall TMyThread::UpdateEtat()
{
  Form1->progressBar->Position = steps;
}

void __fastcall TMyThread::DoTerminate(void)
{
}
```

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 TMyThread;

class TForm1 : public TForm
{
   __published:  // Composants gérés par l'EDI
      TButton *buttonSimuler;
      TButton *buttonQuitter;
      TButton *buttonArreter;
```



```
      TProgressBar *progressBar;
      void __fastcall buttonSimulerClick(TObject *Sender);
      void __fastcall buttonArreterClick(TObject *Sender);
      void __fastcall buttonQuitterClick(TObject *Sender);
   private: // Déclarations de l'utilisateur
      TMyThread *myThread;

   public:    // Déclarations de l'utilisateur
      __fastcall TForm1(TComponent* Owner);
      __fastcall ~TForm1();
};

extern PACKAGE TForm1 *Form1;

#endif
```

Déclaration d'une application utilisant les threads sous Builder

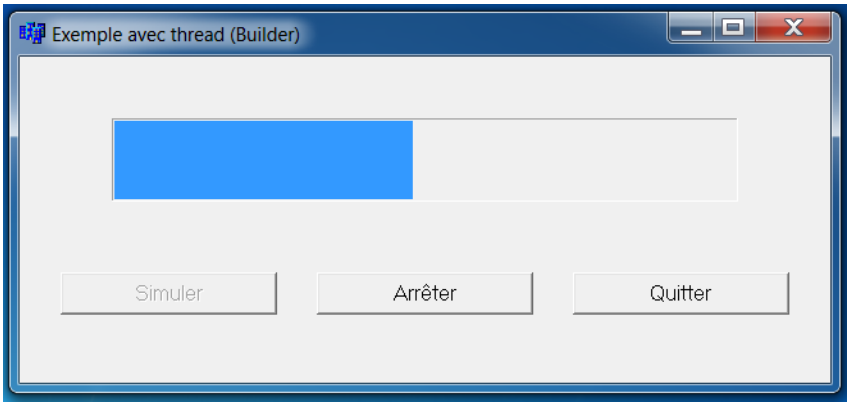
```
#include <vcl.h>
#pragma hdrstop

#include "ThreadIHM.h"
#include "TMyThread.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
  buttonSimuler->Enabled = true;
  buttonArreter->Enabled = false;
  buttonQuitter->Enabled = true;
}

__fastcall TForm1::~TForm1()
{
}
void __fastcall TForm1::buttonSimulerClick(TObject *Sender)
{
  myThread = new TMyThread(true);
  myThread->Resume();
  buttonSimuler->Enabled = false;
  buttonArreter->Enabled = true;
}

void __fastcall TForm1::buttonArreterClick(TObject *Sender)
{
  //myThread->Suspend();
  myThread->Terminate();
  buttonSimuler->Enabled = true;
  buttonArreter->Enabled = false;
}
```



```
void __fastcall TForm1::buttonQuitterClick(TObject *Sender)
{
   myThread->Terminate();
   Close();
}
```

Définition d'une application utilisant les threads sous Builder

L'application obtenue :



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

La mise en place des *threads* revient à dériver une classe de base fournie par l'API et à écrire le code du *thread* dans une méthode spécifique. Avec **commoncpp**, on dérive une classe `Thread` et on écrit le code du *thread* dans la méthode `run()`. Ensuite, on appelera la méthode `start()` pour démarrer le *thread* et la méthode `stop()` pour l'arrêter.

```
#ifndef MYTHREAD_H
#define MYTHREAD_H
#include <cc++/thread.h>
#include <iostream>
#ifdef  CCXX_NAMESPACES
using namespace std;
using namespace ost;
#endif

class MyThread : public Thread {
   private:
      bool stopped;
      void run();
      void final();

   public:
      MyThread(int, int);
      ~MyThread();
      void stop();
};

#endif
```

Déclarer son propre thread avec commoncpp


```
#include "mythread.h"

MyThread::MyThread(int prio, int stack) : Thread(prio, stack)
{
   //setCancel(cancelDeferred);
   setCancel(cancelImmediate);
   stopped = false;
}

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

void MyThread::run()
{
   char c1 = '#';

   stopped = false;
   cout << "\n";
   for(int i=1;(i<=100 && !stopped);i++)
   {
      write(1, &c1, 1);
      Thread::sleep(100); // 100 ms
   }
   cout << "\n";
   exit(); // appelle automatiquement final()
}

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

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

Définir son propre thread avec commoncpp

```
#include <iostream>

#include "mythread.h"

using namespace std;

int main(void)
{
   char choix;
   bool fini = false;
   MyThread *myThread;
```

```
   myThread = new MyThread(0, 0);

   while(!fini)
   {
      cout << "Simuler, Arrêter ou Quitter ? (s|a|q) ";
      cin >> choix;

      switch(tolower(choix))
      {
         case 'a' : myThread->stop();
                    break;
         case 'q' : myThread->stop();
                    fini = true;
                    break;
         case 's' : myThread->start();
                    break;
      }
   }

   myThread->join();
   delete myThread;

   return 0;
}
```

Exemple d'une application utilisant les threads avec commoncpp

L'application obtenue :

```
[tv@alias commonc++-avec-thread]$ ./main
Simuler, Arrêter ou Quitter ? (s|a|q) s
Simuler, Arrêter ou Quitter ? (s|a|q)
######a#
Simuler, Arrêter ou Quitter ? (s|a|q)
s
Simuler, Arrêter ou Quitter ? (s|a|q)
############q#
[tv@alias commonc++-avec-thread]$
```

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

Étape n°4 : les threads en Java

La mise en place des *threads* revient à dériver une classe de base fournie par l'API et à écrire le code du *thread* dans une méthode spécifique. En **Java**, on dérive une classe `Thread` et on écrit le code du *thread* dans la méthode `run()`. Ensuite, on appelera la méthode `start()` pour démarrer le *thread* et la méthode `stop()` pour l'arrêter.

```
import java.lang.*;
import java.util.*;
import java.io.*;

/* Lire http://alwin.developpez.com/tutorial/JavaThread/ */

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

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

            if (choix.compareTo("a") == 0)
            {
               if(myThread != null)
                  myThread.stop();
            }
            if (choix.compareTo("q") == 0)
            {
               if(myThread != null)
                  myThread.stop();
               fini = true;
            }
            if (choix.compareTo("s") == 0)
            {
               myThread = new MyThread();
               myThread.start();
            }
         }

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



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

class MyThread extends Thread
{
   public MyThread()
   {
   }

   public void run()
   {
      try
      {
         for(int i=1;i<=100;i++)
         {
            System.out.print("#");
            Thread.sleep(100); // 100 ms
         }
         System.out.println("");
      }
      catch(InterruptedException e)
      {
         System.out.println("InterruptedException !");
      }
   }
}
```

Exemple d'une application utilisant les threads en Java

Pour compiler cette application, il vous faudra faire pour générer un fichier `.class` :

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

Et pour l'exécuter :

```
$ java MonMain
$
```

```
[tv@alias src-java]$ javac MonMain.java
Note: MonMain.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
[tv@alias src-java]$ java MonMain
Simuler, Arrêter ou Quitter ? (s|a|q)
s
Simuler, Arrêter ou Quitter ? (s|a|q)
#######q#
[tv@alias src-java]$
```


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. Qu'est-ce qu'un *thread* ?

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

Question 3. Est-ce qu'un *thread* peut accéder aux variables globales de son processus ?

Question 4. Est-ce qu'un *thread* peut accéder aux variables locales d'un autre thread du processus ?

Question 5. Pourquoi est-il incorrect de dire qu'une classe ou un objet est un *thread* ?

Question 6. Les threads d'un même processus ont-ils un PID différent ? Si non, pourquoi ?

Travail demandé

Exercice 1 :

L'objectif de cet exercice est de gérer deux traitements multi-tâches à partir d'une IHM. Ces deux traitements distincts gèreront deux barres de progressions différentes pour la simulation. On doit pouvoir démarrer et arrêter les tâches indépendamment.

Dans le cas d'une application dans la console, on simulera les barres de progression avec des '*' pour l'une et des '#' pour l'autre.

Question 7. Réaliser l'application dans le langage ou l'environnement de votre choix.

