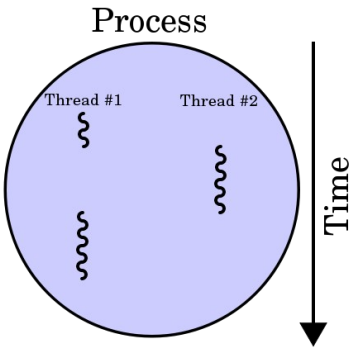
Les threads (que l'on peut traduire par « fil d'exécution ») peuvent être mis en place sur une machine équipée d'un système d'exploitation multitâches



Un thread permet au sein d'un programme d'exécuter des codes différents en parallèle tout en partageant la mémoire du programme.

Le code exécuté dans la fonction main sera appelé le « main thread » ou thread principal, les autres threads, threads secondaires.

Comme le code est exécuté en parallèle, les fonctions bloquantes (ex while(1)) ne sont plus un problème, seul le thread qui les exécute sera bloqué, le reste du programme pouvant continuer son exécution.

Soit le programme de test suivant **qui n'utilise pas de threads:**

```
#include <iostream>

using namespace std;

// affN affiche des nombres de 0 a maxi separes par une virgule
void affN(int maxi)
{
    int i=0;
    while(i<=maxi)
    {
        cout << i++ << ",";  // ici deux ecritures sur la console
    }
}

// affC affiche des caracteres ASCII de debut a fin separes par un tiret
void affC(char debut, char fin)
{
    char c=debut;
    while(c<=fin)
    {
        cout << c++ << "-"; // ici deux ecritures sur la console
    }
}

int main()
{
    affN(9);
    affC('A','Z');
    return 0 ;
}
```

Le résultat est :

0,1,2,3,4,5,6,7,8,9,A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P-Q-R-S-T-U-V-W-X-Y-Z-

On constate que les deux séries sont consécutives

1 Execution parallèle : les threads

Exemple d'utilisation de threads : thread1.cpp
Voici un exemple de mise en œuvre de threads, les fonctions affN et affC seront appelées alternativement par le système d'exploitation..
Les threads sont définis dans C++11, l'option du compilateur gcc : **-std=c++11 doit être activée.**
(dans CODE::BLOCK , onglet Settings, Compiler , Compiler Flags)

La bibliothèque <thread> doit également être ajoutée.

```
#include <thread>
#include <iostream>

using namespace std;

// affN affiche des nombres de 0 a maxi separes par une virgule
void affN(int maxi)
{
    int i=0;
    while(i<=maxi)
    {
        cout << i++ << ",";  // ici deux ecritures sur la console
    }
    cout<<"\nFin du thread t1\n";
}

// affC affiche des caracteres ASCII de debut a fin separes par un tiret
void affC(char debut, char fin)
{
    char c=debut;
    while(c<=fin)
    {
        cout << c++ << "-"; // ici deux ecritures sur la console
    }
    cout<<"\nFin du thread t2\n";
}

int main()
{
    thread t1(affN,9);          // creation d'un thread t1 qui execute affN avec
le parametre 9
    thread t2(affC,'A','Z');    // creation d'un thread t2 qui execute affC avec
les parametres 'A' et 'Z'
    t1.join();                  // execution des threads
    t2.join();
    cout<<endl<<"Les threads sont termines"<<endl;
    return 0 ;
}
```

Le résultat est aléatoire, deux exécutions ne donnent pas le même résultat

Contrairement à l'exécution séquentielle, les deux fonctions affN et affC sont appelées alternativement par le système d'exploitation.
L'accès aux ressources (ici la sortie par cout) est aléatoire, les moments ou le système d'exploitation donne la main à un thread n'est pas maîtrisé.

Vous expliquerez les lignes :
```
thread t1(affN,9);
t1.join();
```
Que se passe t il si l'on retire les lignes avec join ?

2 Contrôler l'exécution des threads : Les mutex

Les threads se terminent automatiquement à la fin de la fonction appelée

Il est parfois nécessaire de contrôler les alternances effectuées par le système d'exploitation.
Les mutex permettent de verrouiller l'accès à une ressource utilisée dans le thread avec les méthodes lock et unlock.

Voici le même programme que précédemment avec un mutex : threads2.cpp

```
 1  #include <thread>
 2  #include <iostream>
 3  #include <mutex>
 4
 5  using namespace std;
 6  mutex monmutex;  // un mutex
 7  // affN affiche des nombres de 0 a maxi separes par une virgule
 8  void affN(int maxi)
 9  {
10      monmutex.lock();
11      int i=0;
12      while(i<=maxi)
13      {
14          cout << i++ << ",";  // ici deux ecritures sur la console
15      }
16      cout<<"\nFin du thread t1\n";
17      monmutex.unlock();
18  }
19
20  // affC affiche des caracteres ASCII de debut a fin separes par un tiret
21  void affC(char debut, char fin)
22  {
23      monmutex.lock();
24      char c=debut;
25      while(c<=fin)
26      {
27          cout << c++ << "-"; // ici deux ecritures sur la console
28      }
29      cout<<"\nFin du thread t2\n";
30      monmutex.unlock();
31  }
32
33  int main()
34  {
35      thread t1(affN,9);          // creation d'un thread t1 qui execute affN
avec le parametre 9
36      thread t2(affC,'A','Z');    // creation d'un thread t2 qui execute affC
avec les parametres 'A' et 'Z'
37      t1.join();                  // execution des threads
38      t2.join();
39      cout<<endl<<"Les threads sont termines"<<endl;
```

```
40      return 0 ;
41  }
```

Lancer plusieurs fois le programme les threads ne sont plus interrompus:

```
A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P-Q-R-S-T-U-V-W-X-Y-Z-
Fin du thread t2
0,1,2,3,4,5,6,7,8,9,
Fin du thread t1
Les threads sont termines
```

On retrouve les séquences de premier programme sans thread

Threads et fonctions « lambda »

Une fonction lambda est une fonction particulière. Elle n'a pas de nom (elle est dite anonyme) et peut être déclarée dans le corps d'une autre fonction. A part cela, elle se comporte comme une fonction classique et peut recevoir des arguments, retourner une valeur.
Les fonctions lambdas sont « locales »

La définition d'une fonction lambda se décompose en trois parties :
[capture](paramètres){corps de la fonction}

La capture permet d'accéder à des variables en dehors de la fonction et les paramètres permettent de passer des informations dans la fonction.

Exemple : lambda_demo.cpp:

```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       int a = 1;
5       auto p = [a](int b,int c) {
6                                     int d;
7                                     d=a+b+c;
8                                     return d;
9                                   };
10      cout<<p(2,3); // Appel de la fonction
11      return 0;
12  }
```

auto : Permet de créer une variable automatique, le type est déduit de l'initialisation.
[ ] : Dit au compilateur qu'il s'agit d'un type lambda et peut comporter des spécifications concernant le passage des paramètres qui vont suivre.
() : On décrit ici les paramètres de la fonction lambda.
{ … }: Le code de la fonction lambda.

Le programme lambda_demo crée une fonction lambda p , avec accès à la variable a et deux paramètre b et c.
On remarque que la fonction lambda est « à l'intérieur » de main, elle n'est accessible que dans main.

Une fonction lambda sans accès aux variables, sans paramètres et même sans corps, s'écrit :
[](){}
Bien entendu une telle fonction lambda ne sert à rien.

Les threads sont souvent déclaré avec des fonctions lambda (ce n'est pas obligatoire)
Exemple : threadsLambdas.cpp

```
1   #include <thread>
```

```
 2  #include <iostream>
 3  using namespace std;
 4  int main() {
 5      thread t1([]() {
 6          for (int i = 0; i < 10; ++i)
 7          {
 8              cout << (i * 3) << " ";
 9          }
10      });
11      thread t2([]() {
12          for (int i = 0; i < 10; ++i)
13          {
14              cout << (i * 3) + 1 << " ";
15          }
16      });
17      thread t3([]() {
18          for (int i = 0; i < 10; ++i)
19          {
20              cout << (i * 3) + 2 << " ";
21          }
22      });
23      t1.join();
24      t2.join();
25      t3.join();
26      return 0;
27  }
```

Tester et analyser ce programme.

Ajouter les mutex permettant de verrouiller les threads pendant leur exécution

1 Exercice 1

Réaliser un programme exo1threads.cpp avec deux threads.
T1 affiche A si l'on appuie sur A, et un . dans tous les cas
T2 affiche Z si l'on appuie sur Z et un + dans tous les cas

Vous utiliserez la fonction getch() pour lire un caractère saisit sur le clavier
Les threads posséderont une boucle infinie. Commenter !!!
Résultat attendu : .+.+.+.+.+A.+A.Z+A.Z+A.+A.Z+.Z+

Squelette proposé :

```
#include <thread>
#include <mutex>
#include <iostream>
#include <conio.h>

using namespace std;
mutex monmutex;

void appA(void)
{
    char c;
    while(1)
    {
        // A COMPLETER
    }
}

void appZ(void)
{
    char c;
    while(1)
    {
        // A COMPLETER
    }
}

int main()
{
    cout<<"Appuyer sur A ou Z"<<endl;
    thread t1(appA);
    thread t2(appZ);
    t1.join();
    t2.join();
    return 0 ;
}
```

2 Exercice 2

Réaliser un programme exo2threads.cpp qui partage d'une variable globale
- Créer une variable globale : string message
- un thread qui rend cette variable égale à « AAA »
- un thread qui rend cette variable égale à « BBB»
- un thread qui affiche « message » seulement s'il a été modifié.

Ce dernier thread contiendra une temporisation d'environ 0,5s entre deux affichage

Résultat attendu :
BBB - AAA - BBB - AAA -