Programmation Multi-Thread

(Librairie pThread)

Introduction : Du FORK au THREAD

Jusqu'ici, nous avons géré des processus (programmes en cours d'exécution, avec leurs propres environnements, et leurs propres espaces mémoires). Cela nous a amené à étudier la création (*fork*) de processus « enfants », et la communication par les signaux (*kill*) et les outils des bibliothèques IPC (Inter Processus Communication) : *pipe*, *pipes* nommés (files de messages), mémoire partagée.

Travailler avec plusieurs processus pour une même application présente des avantages :

1. La séparation des espaces mémoires garantit que les processus ne mélangent pas leurs données,
2. Chaque processus peut avoir sa propre priorité d'exécution du point de vue de l'ordonnanceur
3. Possibilité de supprimer un des processus de l'application et ainsi libérer des ressources machine.
4. Un processus peut « planter » sans gêner les autres processus de l'application.

Avant de parler des inconvénients, nous allons nous intéresser à une autre technique du travail en multitâche pour une application : le MULTI-THREADING.

On peut définir un THREAD comme : une unité d'exécution. En fait, tous les processus que nous avons créés jusqu'ici contenaient 1 thread unique. Comme il était seul, nous l'avons « confondu » avec le processus.

Mais maintenant, il faudra dire : un processus contient un Thread, le thread principal, qui correspond à l'unité d'exécution du programme (*main*).

Le MULTI-THREADING consiste à créer d'autres threads à l'intérieur du processus. Chaque Thread étant géré par l'ordonnanceur, on peut faire du multi-tâches à l'intérieur d'1 seul processus.

L'avantage que nous allons mettre en évidence : Comme nous sommes dans un même espace mémoire, celui du processus, tous les threads utilisent les mêmes variables mémoires, ce qui évite le recours aux IPC pour échanger des informations.

Mais cet avantage cache un danger : que ce passe-t-il si plusieurs threads accèdent en même temps à une variable ? Il y a risque d'incohérence du contenu (par exemple si la lecture se fait avant que l'écriture soit terminée). L'outil MUTEX apportera la solution.

Partager le même espace mémoire ne veut pas forcément dire que toutes les variables sont communes : les règles de « portées des variables » s'appliquent normalement. Une variable déclarée dans un bloc {} n'est connue que de ce bloc, même s'il est exécuté par plusieurs threads.

Du point de vue de l'ordonnanceur, il y a avantage à travailler en multi-thread plutôt qu'en multi-processus. Le temps de commutation d'un processus à l'autre est plus grand que le temps de commutation d'un thread à l'autre dans un même processus.

Par contre, il n'est pas possible de donner des priorités différentes à des threads d'un même processus. En fait, on perd tous les avantages des processus énumérés ci-dessus.

Cependant l'utilisation des Thread reste très populaire. D'ailleurs, certains systèmes d'exploitation (Windows pour ne pas le citer) ne nous laissent que cette possibilité de programmation.

Nous étudierons ici les threads linux Posix, disponible en C et compatible avec les bibliothèques du C++.

Ont peut également utiliser d'autres bibliothèques, comme la QThread de Qt, qui permet de travailler en vrai C++.

Travail sur les THREADs

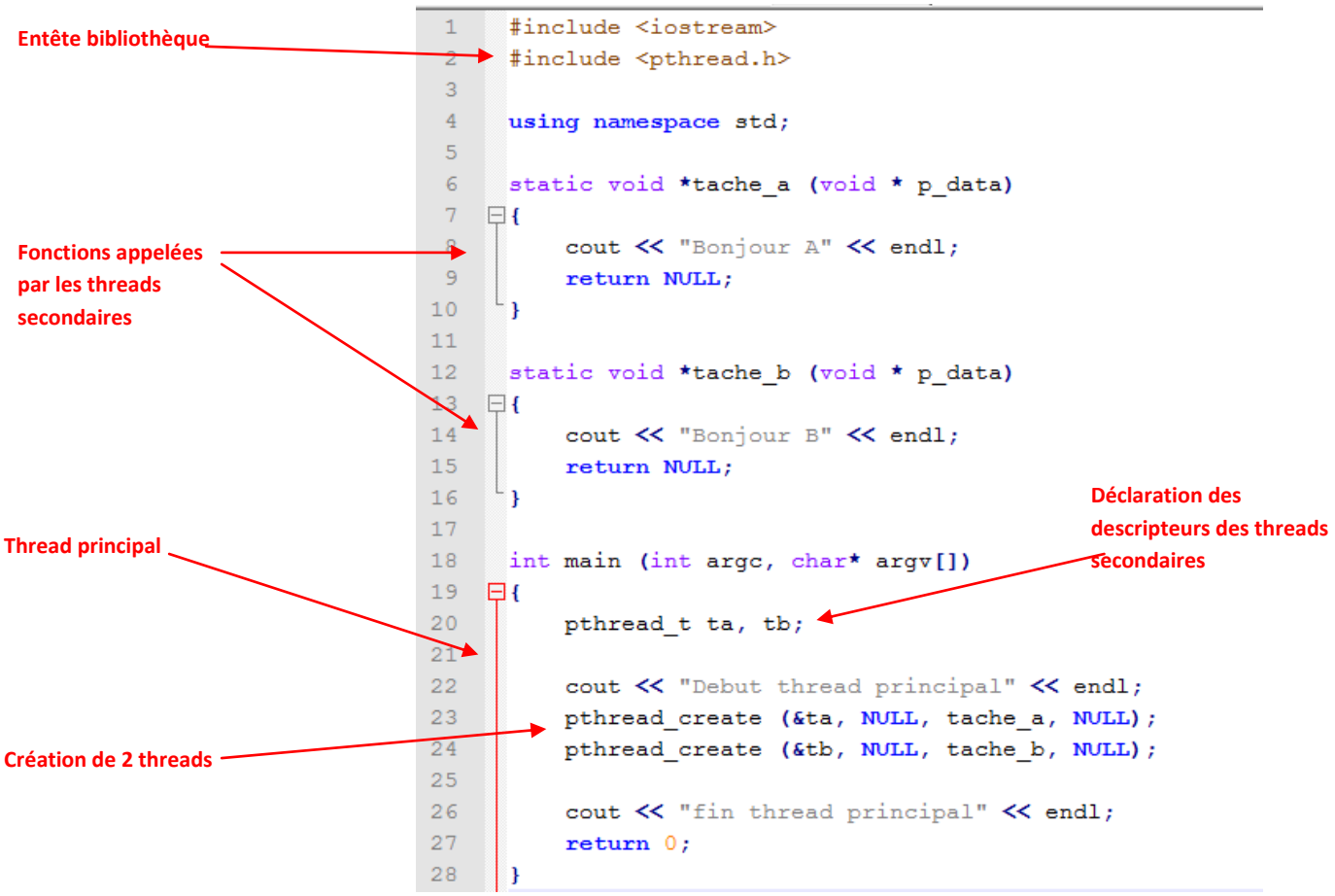
Bien compiler :

Les Threads Posix sont désormais installés en même temps que les dernières versions de librairie C (Glibc). Néanmoins, il reste nécessaire de préciser que l'on veut utiliser la librairie libpthread . De plus, le code exécutable que l'on veut produire doit être réentrant. C'est une notion un peu vague pour vous. Disons que le code doit pouvoir être exécuté par plusieurs threads sans que l'exécution de l'un n'interfère sur l'exécution d'un autre. Autrement dit, votre code ne doit pas faire référence de manière explicite à un fil d'exécution (thread). Pour tester si votre code est réentrant, on passe la constante symbolique D_REENTRANT à g++

```
g++ -D_REENTRANT -lpthread -o threads threads.cpp
```

Exercice 1 :

Saisir le code suivant :




```
1  #include <iostream>
2  #include <pthread.h>
3
4  using namespace std;
5
6  static void *tache_a (void * p_data)
7  {
8      cout << "Bonjour A" << endl;
9      return NULL;
10 }
11
12 static void *tache_b (void * p_data)
13 {
14     cout << "Bonjour B" << endl;
15     return NULL;
16 }
17
18 int main (int argc, char* argv[])
19 {
20     pthread_t ta, tb;
21
22     cout << "Debut thread principal" << endl;
23     pthread_create (&ta, NULL, tache_a, NULL);
24     pthread_create (&tb, NULL, tache_b, NULL);
25
26     cout << "fin thread principal" << endl;
27     return 0;
28 }
```

Note : les fonctions appelées par les threads sont de type *static void*. C'est logique puisque ces fonctions sont exécutées « en arrière plan » et donc le thread principal continue son exécution sans attendre une valeur de retour (*return*).

Si on lance plusieurs fois le programme créé, on s'aperçoit que l'affichage ne correspond pas toujours. On peut même avoir l'impression que les threads ne se lancent pas si la machine de test est trop rapide.

C'est parce que le thread principal meurt juste après la création de ses threads ; ils n'ont pas le temps de s'exécuter ; ils meurent avec le thread principal.

En ajoutant l'appel à la fonction `pthread_join()` à la fin du thread principal, on règle le problème.

Modifiez votre code vérifiez avec plusieurs essais que l'anomalie a disparu.

Faites contrôler votre travail.

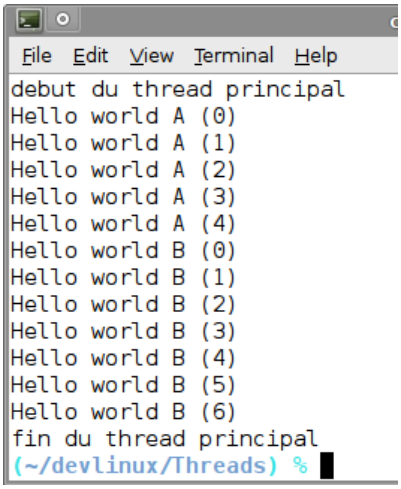
`pthread_join()` vous fera certainement penser au `wait()` dans la programmation multiprocessus (avec fork()).

Exercice 2 : Passage de paramètres à un thread

Le quatrième argument de `pthread_create()` permet un passage d'argument à la fonction thread. Cet argument est de type `void *` Cela implique des contraintes :

- On ne peut indiquer qu'une seule adresse. Donc si on souhaite transmettre plusieurs informations, il faudra utiliser une structure de données et envoyer l'adresse de cette structure.
- Il s'agit d'une adresse vers un type non défini (*void**). Il va donc falloir le « caster » (transtyper) vers le bon type (évidemment vers un type pointeur!)

On veut obtenir le résultat suivant : on désire avoir 2 threads affichant un certain nombre de fois un caractère :
A est affiché par le thread A et B par le thread B.



On va utiliser pour cela qu'une seule fonction qui sera exécutée par chacun des threads.

Pour lancer l'exécution de la même fonction par 2 threads on écrira :

```
pthread_create(&a, NULL, tache, &ma_struct1);
pthread_create(&b NULL, tache, &ma_struct2);
```

Il faut auparavant définir le contenu de la structure qui contiendra le caractère à afficher et le nombre de fois qu'il faut afficher ce caractère :

```
struct ma_structure
{
   char car;  // Caractère à afficher.
   int count; // Nombre de fois où il doit être affiché.
};
```

Reste ensuite la question du « cast » ou du « transtypage » de l'argument. Il faut définir le type *void* imposé par le système des threads Posix. La fonction exécutée par chaque thread pourrait ressembler à cela :

```
void* tache(void* p_sur_ma_structure)
{
 /* Effectue un transtypage du pointeur void* vers le bon type. */
 struct ma_structure* s = (struct ma_structure*) p_sur_ma_structure;
   ….
   /* on peut utiliser la variable s->car et s->count */
   ….
   return NULL;
}
```

Maintenant vous pouvez réaliser l'exercice pour produire l'affichage ci-dessus.

Exercice 3 : Partage de variables

Ci-dessous en annexe un programme composé deux tâches d'exécution task_d et task_i.
La première décrémente N fois une variable et la seconde incrémente cette même variable le même nombre de fois N.

Bref, logiquement, à la fin de l'exécution, la variable aura la même valeur qu'au début. On va s'apercevoir avec le programme suivant qu'en lançant plusieurs fois l'exécution, on a de temps à autre un résultat différent. Selon la machine, le nombre de logiciels qui s'exécutent à l'instant t, etc..

En plus d'un aspect aléatoire, ce programme est assez imprévisible.

1. Saisir le code source ci-dessous, compiler, linker et lancer plusieurs fois de façon à mettre en évidence le caractère erratique de ce dernier.

2. Modifier le code précédent de façon à définir une section critique autour de l'accès à la variable partagée. Pour cela, vous utiliserez les fonctions `pthread_mutex_lock()` et `pthread_mutex_unlock()`.

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

#define TINCNBMAX   100000
#define TDECNBMAX   100000

using namespace std;

// Déclaration globale
       struct ma_data
       {
          int data;
       };

void *task_i (void *p)
{
   if (p != NULL)
   {
      /* tache d'incrementation du champ data */
      struct ma_data *p_data = (struct ma_data *)p;
      int i;

      for (i = 0; i < TINCNBMAX; i++)
      {
         int x = p_data->data;
         x++;
         p_data->data = x;

         cout << "INC data : " << p_data->data << endl;
      }
   }
   return NULL;
}

void *task_d (void *p)
{
   if (p != NULL)
   {
      /* tache de decrementation du champ data */
      struct ma_data *p_data = (struct ma_data *)p;
      int i;

      for (i = 0; i < TDECNBMAX; i++)
      {
         int x = p_data->data;
         x--;
         p_data->data = x;

         cout << "DEC data : " << p_data->data << endl;
      }
   }
   return NULL;
}

int main (int argc, char* argv[])
{
   pthread_t ta;
   pthread_t tb;

   struct ma_data sh;
   sh.data = 0;

   cout << "debut du thread principal" << endl;

   pthread_create (&ta, NULL, task_i, &sh);
   pthread_create (&tb, NULL, task_d, &sh);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   cout << "fin du thread principal" << endl;

   return 0;
}
```