IN204
Programmation Orientée Objet – Le parallélisme

Séance de Travaux Dirigés du 13 novembre 2019

B. Monsuez

Partie I – Création & Manipulation de Processus léger

Références

`std::thread` voir http://en.cppreference.com/w/cpp/thread/thread

Question n°1

Créer un processus léger qui est associé à une fonction simple.

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

void simple_method()
{
          int i = 5;
          int x = 10;
          int result = i * x;
          std::cout << "This code calculated the value "
                    << result << " from thread ID: "
                    << std::this_thread::get_id() << "\n";
}


int main()
{
          std::thread simpleThread(&simple_method);
          std::cout << "Main thread is executing and waiting\n";
          simpleThread.join();
          std::cout << "Alternate thread has terminated.\n";
          return 0;
}
```

Exécuter le code et analyser la sortie. Commenter celle-ci, notamment au regard de la documentation de la classe `std::tread` présente en annexe.

Question n°2.0

Ecrire un programme qui lance les deux calculs suivants en parallèle, le premier dans un processus léger secondaire, le premier dans le processus léger principal :

```
void worker_process(int numberOfIterations)
{
	for (int i = 1; i < numberOfIterations; i++)
	{
		std::cout << "Worker Thread: " <<  i << "\n";
	}
}
```

et

```
void main_process()
{
	for (int i = 1; i < 1000; i++)
	{
		std::cout << "Primary Thread: " << i << "\n";
	}
}
```

Tester le code.

Partie II – Sections critiques

Références

`std::unique_lock` voir http://en.cppreference.com/w/cpp/thread/unique_lock

`std::mutex` voir http://en.cppreference.com/w/cpp/thread/mutex

Question n°1

Nous supposons que nous avons deux fonctions, une première fonction calcule la valeur maximal d'un tableau numérique :

```
void array_find_max(const std::vector<int>& theArray, int* theMaxValue)
{
        int result = theArray[0];
        for (int i = 1; i < (int)theArray.size(); i++)
        {
                if (theArray[i] > result)
                        result = theArray[i];
        }
        *theMaxValue = result;
}
```

la seconde fonction modifie le tableau en multipliant chaque valeur numérique du tableau par 2 :

```
void array_multiply_by(std::vector<int>& theArray, int theValue)
{
        for (int i = 1; i < theArray.size(); i++)
                theArray[i] *= theValue;
}
```

Question 1 : Ecrire chacune des fonctions.

Question 2 : Nous souhaitons exécuter en parallèle les deux fonctions.

Exécuter plusieurs fois ces fonctions sur un tableau de grande taille que vous créez soit de manière aléatoire soit de manière cycliques. Ci-dessous quelques exemples de fonctions générant des tableaux de taille *n* et ayant comme valeur des nombres entre *0* et *theMaxValue*.

```
void initialize_array(std::vector<int>& theArray, int theSize,
        int theMaxValue)
{
        theArray.clear();
        int step = theMaxValue / 3;
        theArray[0] = 0;
        for (int i = 1; i < theSize; i++)
```

```
                        theArray[i] = (theArray[i-1] + step) % (theMaxValue + 1);
}
```

Ou

```
void initialize_random_array(std::vector<int>& theArray, int theSize,
        int theMaxValue)
{
        theArray.clear();
        for (int i = 0; i < theSize; i++)
                theArray[i] = std::rand() % (theMaxValue + 1);
}
```

Expliquer pourquoi les résultats ne sont pas toujours cohérents ? ie. que la valeur maximale retournée est supérieure à la valeur maximale du tableau spécifiée au montant de sa création.

Question n°3

Il faut utiliser un verrou qui permet à une méthode de se garantir l'exclusivité de l'usage du tableau. Pour ce faire, nous pouvons utiliser un objet `std::mutex` qui encapsule un mécanisme d'exclusion mutuelle.

L'objet `std::unique_lock` permet d'obtenir un accès exclusif sur un object `std::mutex`. Quand nous écrivons le code suivant :

```
#include <iostream>        // std::cout
#include <thread>          // std::thread
#include <mutex>           // std::mutex, std::unique_lock

std::mutex mtx;            // mutex fournissant le mécanisme d'exclusion mutuelle.

void my_function (int n, char c) {
        // section critique, tant que l'objet lck existe, personne ne pourra
        // accéder à mtx.
        std::unique_lock<std::mutex> lck(mtx);
        // Code s'exécutant.
}
```

Question n°3.1

Modifier les fonctions

```
int array_find_max(std::vector<int>& theArray, int* theMaxValue)
```

et

```
void array_multiply_by(std::vector<int>& theArray, int theValue)
```

pour implanter le mécanisme d'exclusion mutuelle propose.

Question n°3.2

Exécuter le code et vérifier que les résultats sont dorénavant cohérents.

Partie III – Exécution asynchrone

Références

`std::async` voir http://en.cppreference.com/w/cpp/thread/async

`std::future` voir http://en.cppreference.com/w/cpp/thread/future

Nous considérons la fonction suivante qui calcule les décimales de « e ».

```
std::string computeE(int numberOfDigits)
{
	int sizeOfTable = numberOfDigits + 9;
	int* table = (int*)_alloca(sizeOfTable * sizeof(numberOfDigits));
	table[0] = 0;
	table[1] = 2;
	for (int i = sizeOfTable - 1; i > 0; i--) {
		table[i] = 1;
	}

	std::ostringstream output;
	int x = 0;
	table[1] = 2;
	for (; sizeOfTable > 9; sizeOfTable -- )
	{
		for (int i = sizeOfTable - 1; i > 0; i--)
		{
			table[i] = x % i;
			x = 10 * table[i - 1] + x / i;
		}
		output << x;
	}
	return output.str();
}
```

Question n°1 :

Implanter la fonction et vérifier que celle-ci fonctionne correctement.

Question n°2 :

Nous constatons que calculer 10000 ou 20000 décimales de e, cela prend du temps. Nous souhaitons transformer cela en une fonction asynchrone à l'aide de la fonction `std::async`.

Ecrire le code transformant la précédente fonction en une fonction asynchrone en utilisant la fonction `std::async.`

Question n°3 :

Lancer deux calculs asynchrones, l'un calculant les 1000 premières décimales, l'autre les 10000 premières décimales et affichés les résultats dès que ceux-ci sont disponibles.