

1 TP 1

1.1 Création

- Créer trois threads avec l'API que vous voulez (POSIX, Win32, C++11, ...)
- Chaque thread doit afficher "Hello thread" suivi d'un numéro propre à chaque thread
- Joindre les threads dans le thread *main*
- Compiler et tester le programme

1.2 Partage de données

- Reprendre le projet précédent
- Ajouter une variable *int* initialisé à 0 et partagée par tous les threads (variable globale ou passée en paramètre)
- Chaque fonction de thread doit faire une boucle *for* de 100 000 itérations et incrémenter la variable
- Compiler puis tester plusieurs fois le programme
- Est-ce que la variable vaut 300000 ? Est-ce que le résultat est constant ? Pourquoi ?

1.3 Verrou

- Reprendre le programme précédent
- Modifier le code pour garantir un résultat correct avec un mutex

1.4 Atomique

- Reprendre le programme précédent
- Modifier le code pour garantir un résultat correct avec une variable atomique

1.5 Bonus : performance mutex vs atomique

- Comparer le temps d'exécution de la version mutex et atomique
- Que peut-on en déduire ?

2 TP 2

2.1 Condition

- Créer un thread *work* avec l'API de votre choix qui va s'endormir pendant 5 secondes
- Puis notifier le thread *main* avec une condition
- Le thread *main* va bloquer en attendant la notification de la condition
- Joindre le thread *work* dans le thread *main*

2.2 Condition 2

- Créer 10 threads *work* qui se mettent en attente d'une condition
- Si la condition est notifié, écrire sur *stdout* l'identifiant du thread et endormir le thread pendant 1 seconde
- Toutes les secondes, le thread *main* va notifier la condition
- Arrêter le programme après 10 notifications
 - Tips: utiliser une variable globale ou passée par paramètre au thread *work*
- Joindre les threads *work*

2.3 Barrière

- Créer une barrière avec un nombre de 11
- Créer 10 threads *work* qui vont attendre pendant 5 secondes et va attendre une barrière
- Dès que la barrière est franchi, afficher un message
- Le thread *main* attend la barrière puis affiche un message
- Joindre les threads *work*

3 TP 3

3.1 Péage

- Simuler un péage d'autoroute en respectant les contraintes suivantes :
 - il y a un nombre défini de voiture (10 par exemple)
 - il y a un nombre maximum de voiture qui peuvent entrer à un instant t au péage (2 par exemple)
 - chaque voiture reste 5 secondes bloquée au péage
- Le programme s'arrête quand toutes les voitures sont sorties du péage
- Logger les actions des voitures (entrée péage, sortie péage) avec l'identifiant du thread et éventuellement l'horodatage (14:10:44 [th1] voiture1 enter toll)

3.2 Triage

- Définir une structure *myitem* avec deux membres : id (entier) et value (entier)
- Définir une liste nommée *input* qui contient des structures *myitem*
- Définir 4 threads qui vont insérer des éléments *myitem* dans la liste *input*
- Définir 2 threads qui vont extraire les éléments de la liste *input* et les placer dans une liste *output* (propre au thread)
- Contraintes :
 - il faut insérer 100 000 éléments dans la liste *input*
 - il n'y a que 3 threads actifs à un instant t
 - tous les threads doivent démarrer en même temps
 - les threads d'insertion s'arrêtent quand tous les éléments ont été insérés
 - les threads d'extraction s'arrêtent quand tous les éléments (100 000) ont été extraits
 - le programme s'arrête quand tous les threads d'insertion et d'extraction ont terminé leur travail
- Note : il faut raisonner en terme de programmation parallèle et non en terme de performance

3.3 Comptes bancaires

- Définir une structure *bankaccount* avec deux membres : id (entier) et balance (entier)
- Créer quatre comptes (A, B, C, D)
- A l'initialisation, chaque compte bancaire dispose de 100 €
- Une transaction bancaire entre deux comptes doit se faire dans un thread spécifique
- Réaliser les transactions suivantes en parallèle :
 - Un compte A fait un virement de 50 € sur le compte B
 - Un compte B fait un virement de 130 € sur le compte A
 - Un compte C fait un virement de 70 € sur le compte A
 - Un compte D fait un virement de 20 € sur le compte C

- Contraintes :
 - pour des raisons de performances, un virement de A vers B ne doit pas bloquer un virement de C vers D (donc la solution d'utiliser un verrou **unique** n'est pas valide pour cet exercice !)
- Tips : attention à l'ordre de verrouillage des verrous !
- Bonus : empêcher un compte d'être négatif et bien s'assurer de n'avoir ni création ni destruction d'argent à la fin

4 TP 4

4.1 Visualisation de processus (Windows)

- Installer Process Explorer depuis <https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer> ou via Chocolatey (choco install procexp)
- Créer et lancer une application "gourmande" en CPU (boucle infinie, ...) et regarder les informations données par Process Explorer
- Faites de même avec Perfmon

4.2 Visualisation de processus (GNU/Linux)

- Installer htop (apt / yum install htop)
- Créer et lancer une application "gourmande" en CPU (boucle infinie, ...) et regarder les informations données par htop
- Regarder les options de tris et filtres de htop

4.3 Debug code parallèle

- Compiler le code source suivant sous GNU/Linux et / ou Windows
- Utiliser *valgrind* / Visual Studio pour déterminer où se trouve le problème
- Corriger le problème

Listing 1: bug.cpp

```
#include <cstdlib>

#include <iostream>
#include <atomic>
#include <thread>

void thread_func(int* val)
{
    for(int i = 0 ; i < 1000000 ; i++)
    {
        (*val)++;
    }
}

int main(int argc, char** argv)
{
    std::array<std::thread, 4> th;
    size_t nb = th.size();
    int val = 0;

    for(size_t i = 0 ; i < nb ; i++)
    {
        th[i] = std::thread(thread_func, &val);
    }

    for(size_t i = 0 ; i < nb ; i++)
    {

```

```

    /* wait threads to terminate */
    th[i].join();
}

std::cout << "val = ";
std::cout << val << std::endl;

std::cout << "Exit program" << std::endl;
return EXIT_SUCCESS;
}

```

4.4 Debug code parallèle 2

- Même exercice avec le code suivant

Listing 2: bug2.cpp

```

#include <cstdlib>

#include <iostream>
#include <thread>
#include <mutex>

int a = 0;
std::mutex mtx;

void thread_func()
{
    //std::cout << "Thread function" << std::endl;

    for(int i = 0 ; i < 10000 ; i++)
    {
        mtx.lock();
        // std::cout << std::this_thread::get_id() << " " << a << std::endl;
        std::this_thread::sleep_for(std::chrono::microseconds(10));
        mtx.unlock();
    }
}

int main(int argc, char** argv)
{
    std::thread thread(thread_func);
    std::thread thread2(thread_func);

    mtx.lock();
    for(int i = 0 ; i < 250 ; i++)
    {
        a++;
        std::this_thread::sleep_for(std::chrono::microseconds(100));
        mtx.unlock();
        mtx.lock();
    }

    thread.join();
    thread2.join();

    std::cout << "a = " << a << std::endl;

    std::cout << "Exit program" << std::endl;
    return EXIT_SUCCESS;
}

```

4.5 Debug code parallèle 3

- Même exercice avec le code suivant

Listing 3: bug3.cp

```
#include <cstdlib>

#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;
std::mutex mtx2;

void thread_func()
{
    std::this_thread::sleep_for(std::chrono::microseconds(10));

    mtx.lock();
    std::this_thread::sleep_for(std::chrono::microseconds(10));
    mtx2.lock();
    std::this_thread::sleep_for(std::chrono::microseconds(10));
    mtx2.unlock();
    mtx.unlock();
}

void thread_func2()
{
    std::this_thread::sleep_for(std::chrono::microseconds(10));

    mtx2.lock();
    std::this_thread::sleep_for(std::chrono::microseconds(2));
    mtx.lock();
    std::this_thread::sleep_for(std::chrono::microseconds(10));
    mtx.unlock();
    mtx2.unlock();
}

int main(int argc, char** argv)
{
    std::thread thread(thread_func);
    std::thread thread2(thread_func2);

    thread.join();
    thread2.join();

    std::cout << "Exit program" << std::endl;
    return EXIT_SUCCESS;
}
```

