B4 - Concurrent Programming

B-CCP-400

Threads & Mutexes

Plazza Bootstrap

2.0

{EPITECH.}

This subject is rather long with a lot of nice and interesting reading at the end.
Many steps can be done easily by copying code, but we hope you understand why that would be a waste of your time if you don't fully understand the discussed concepts.
Moreover, although threads may seem like a simple API, they are actually a whole new paradigm.

THREADS AND MUTEXES

Throughout this first part, we expect you to use the `pthread` library and not the `std::thread` class from the `STL`. You can use those for the project, but this bootstrap aims to help you understand what those classes do!

+ EXERCISE 1

To get things started, you have to implement a typical use of mutexes.
Create an `incrementCounter` function that takes a reference to an `int` as parameter and increments it `n` times (`n` being a value of your choice).

Create a `main` function that spawns `t` threads, each of them calling `incrementCounter` on the same `int`.
After execution, the counter's value should be `t * n`.
If you try and run it however, you can see it is not. You should already know why that is.

Add a mutex to protect your counter. The total should be coherent now.

+ EXERCISE 2

Create an object-oriented encapsulation for mutexes, with an interface similar to the following:

```
class IMutex {
public:
        virtual ~Imutex() = default;
        virtual void lock() = 0;
        virtual void unlock() = 0;
        virtual void trylock() = 0;
};
```

Use your new class to improve your previous code.

+ EXERCISE 3

Create a `ScopedLock` class, which fills the role of RAII for a mutex lock (the lock is the resource to be acquired and released).
It should be built using a mutex as parameter, lock it during its construction and unlock it upon destruction.

Pay attention to the semantics…
* a `Mutex` is an object that can be locked, used for synchronization,
* a `Lock` is an object that locks and unlocks a `Mutex`.
Use your new `ScopedLock` to improve your previous code.

+ EXERCISE 4

Create a `Thread` class that encapsulates the concept of a thread. Take some time to think about its interface. An interesting starting point would be to encapsulate the following points:

- Status of the thread (started, running, dead)
- Start of the thread
- Waiting for the thread death

Use your new class to improve your previous code, which should now look like real C++.

Producer-Consumer

+ Exercise 5

Create a `SafeQueue` object.
This object encapsulates a queue within which several threads can add or remove elements.
To do so, you must use the `std::mutex` and `std::unique_lock` classes from the `STL`.

For now, your `SafeQueue` will only contain `ints`.
You will later modify it so that it offers a more generic solution.

Your class must conform to the following interface:

```
class ISafeQueue {
public:
        virtual ~ISafeQueue() = default;
        virtual void push(int value) = 0;
        virtual bool tryPop(int &value) = 0;
};
```

The `tryPop` method unstacks an element from the queue if it is not empty, and stores it in `value`.
It returns `true` if it succeeded, and `false` if the queue is empty.

Remember to run intensive tests on your `SafeQueue`!
That's the only way to bring up race conditions or deadlocks!

+ Exercise 6

Implement a producer-consumer pattern using your `SafeQueue`.

The `Consumer` must be in an active waiting state: if the queue is empty when it attempts to unstack an value from it, it sleeps for a while (c.f. `std::this_thread::sleep_for`), then tries again.

The `Producer` should be a thread adding random numbers to the queue, while the `Consumer` should be a thread that unstacks value from the queue and displays them.

Create a `main` function that starts a customizable number of producers and consumers.
#hnt(We strongly recommend testing your program exhaustively.)

+ Exercise 7

Use `std::condition_variable` to add an `int pop()` method to your `SafeQueue`.
It should block until it can unstack an `int` from the queue and return it.

GOING FURTHER

+ EXERCISE 8

Your `SafeQueue` isn't really useful currently: you generally don't work exclusively with `ints`.
Use templates to turn your `SafeQueue` into a generic container.

+ EXERCISE 9

Use templates to improve your `Thread` class so that it can take as parameter any callable object.

You may want to take a look at Day 9 of the C++ Pool.

+ EXERCISE 10

Implement a **thread pool**.

A thread pool is a list of tasks to be accomplished, along with a fixed number of threads.
Those threads unstack actions from the queue and execute them.

Recommended reading

Unfortunately, it is impossible for us to cover more than the basics of concurrent computing during a short session.
However, we do recommend reading through some of these, and going back to this list in the future:

- An overall presentation of thread pools
- A presentation of the "Active Objects", or "Actors" concept
- Another technique for mutexing data, and read-write locks
- A presentation of the recursive mutex
- An advanced discussion on data sharing for imperative languages
- "The practical of Parallel programming", a complete book dedicated to parallel and concurrent programming
- Why parallel programming is important, by Herb Sutter
- A tutorial for OpenMP, a multi-threading solution from the compiler
- A presentation of the transactional memory mechanism, another way to share data and control access within multi-threaded code
- Helgrind documentation, a tool for common error detection in multi-threaded programs