Multithreading in C++

Illustrating the fundamentals of multithreading in C++.

WE'LL COVER THE FOLLOWING

> *Multithreading* in C++ consists of threads, synchronization primitives for shared data, thread-local data, and tasks.

Threads

A `std::thread` represents an independent unit of program execution. The executable unit, which is started immediately, receives its work package as a callable unit. A callable unit can be a named function, a function object, or a lambda function.

The creator of a thread is responsible for its lifecycle. The executable unit of the new thread ends with the end of the callable. Either the creator waits until the created thread `t` is done (`t.join()`), or the creator detaches itself from the created thread: (`t.detach()`). A thread `t` is *joinable* if no operation

`t.join()` or `t.detach()` was performed on it. A *joinable* thread calls `std::terminate` in its destructor and the program terminates.

A thread that is detached from its creator is typically called a daemon thread because it runs in the background. A `std::thread` is a variadic template. This means that it can receive an arbitrary number of arguments by copy or reference; either the callable or the thread can get the arguments.

Shared Data

You have to coordinate access to a shared variable if more than one thread is using it at the same time and the variable is mutable (non-const). Reading and writing a shared variable at the same time is a data race, and therefore, undefined behavior. Coordinating access to a shared variable is achieved with mutexes and locks in C++.

Mutexes

A mutex (*mutual exclusion*) guarantees that only one thread can access a shared variable at any given time. A mutex locks and unlocks the critical section that the shared variable belongs to. C++ has five different mutexes; they can lock recursively, tentatively, and with or without time constraints. Even mutexes can share a lock at the same time.

Locks

You should encapsulate a mutex in a lock to release the mutex automatically. A lock implements the RAII idiom by binding a mutex's lifetime to its own. C++ has a `std::lock_guard` for the simple cases, and a `std::unique_lock` / `std::shared_lock` for the advanced use-cases, such as the explicit locking or unlocking of the mutex respectively.

Thread-safe Initialization of Data

If shared data is read-only, it's sufficient to initialize it in a *thread-safe* way. C++ offers various ways to achieve this including using *constant expression*, a *static variable with block scope*, or using the function `std::call_once` in combination with the flag `std::once_flag`.

Thread Local Data

Declaring a variable as *thread-local* ensures that each thread gets its own copy; therefore, there is no shared variable. The lifetime of a thread local data is bound to the lifetime of its thread.

Condition Variables

Condition variables enable threads to be synchronized via messages. One thread acts as the sender while another one acts as the receiver of the message, where the receiver blocks wait for the message from the sender. Typical use cases for condition variables are producer-consumer workflows. A condition variable can be either the sender or the receiver of the message. Using condition variables correctly is quite challenging; therefore, tasks are often the easier solution.

Tasks

Tasks have a lot in common with threads. While you explicitly create a thread, a task is simply a job you start. The C++ runtime will automatically handle, in the simple case of `std::async`, the lifetime of the task.

Tasks are like data channels between two communication endpoints. They enable thread-safe communication between threads: the promise at one endpoint puts data into the data channel, and the future at the other endpoint picks the value up. The data can be a value, an exception, or simply a notification. In addition to `std::async`, C++ has the class templates `std::promise` and `std::future` that give you more control over the task.