Poznan University of Technology
Object Oriented Programming
Multithreading

Jakub Piotr Hamerliński, M.Eng.

https://www.linkedin.com/in/hamerlinski

https://github.com/hamerlinski

Object Oriented Programming

Agenda

01. Multithreading
02. Task

> "Well, I will not be blackmailed by some ineffectual, privileged, effete, debutante. You want to start a street fight with me bring it on but you're gonna be surprised by how ugly it gets, you don't even know my real name - I'm the lizard king!"

Robert California

Multithreading

Object Oriented Programming

Multithreading

What is multithreading?

Multithreading is a technique that allows a program to run multiple threads of execution concurrently within the same process.

A thread is the smallest unit of execution in a program and has its own call stack, program counter, and registers.

Threads within a process share the same memory space, including global variables and heap memory, which allows for easy communication between threads. However, this shared memory also introduces the potential for data races and other synchronization issues.

Object Oriented Programming

Multithreading

Why use multithreading?

01. Improved performance: Multithreading can be used to improve the performance of CPU-bound tasks by parallelizing their execution across multiple CPU cores.
02. Better resource utilization: By dividing work among multiple threads, a program can make more efficient use of available system resources, such as CPU, memory, and I/O.
03. Responsiveness: In user interfaces, multithreading can be used to offload time-consuming tasks to background threads, keeping the main thread responsive to user input. Similarly, server applications can use multithreading to handle multiple client requests simultaneously, improving responsiveness.

Object Oriented Programming

Multithreading

Challenges in multithreading

Correctness: Ensuring the correctness of a multithreaded program can be challenging due to the potential for data races, deadlocks, and other concurrency-related issues.

Portability: The behavior of multithreaded programs can vary across different platforms and hardware, which may require platform-specific tuning and optimization.

Complexity: Writing, debugging, and maintaining multithreaded code can be more complex than single-threaded code, due to the need for synchronization and the inherent nondeterminism of concurrent execution.

Object Oriented Programming

Multithreading

The C++ Standard Library

C++11 introduced native support for multithreading in the C++ Standard Library, providing a high-level and portable API for creating and managing threads, synchronizing data access, and performing inter-thread communication.

The key components for multithreading in C++ are available in the `<thread>`, `<mutex>`, `<atomic>`, and `<condition_variable>` headers.

Object Oriented Programming

Multithreading

Creating and managing threads

The `std::thread` class is used to create and manage threads. To create a new thread, simply pass a function or callable object to the `std::thread` constructor. When a `std::thread` object goes out of scope or is explicitly joined, it waits for the associated thread to complete its execution.

Object Oriented Programming

Multithreading

Simple thread creation and joining

```cpp
#include <iostream>
#include <thread>
void print_hello() {
  std::cout << "Hello from thread!" << std::endl;
}
int main() {
  // Create a new thread that executes the print_hello function
  std::thread t(print_hello);
  // Main thread waits for the created thread to finish
  t.join();
  std::cout << "Hello from main!" << std::endl;
  return 0;
}
```

Object Oriented Programming

Multithreading

Synchronization

To synchronize access to shared data, the C++ Standard Library provides several synchronization primitives, such as `std::mutex`, `std::recursive_mutex`, `std::lock_guard`, and `std::unique_lock`.

Object Oriented Programming

Multithreading

Synchronizing access to shared data using a mutex

```cpp
#include <iostream>
#include <thread>
#include <vector>
std::mutex mtx;
int counter = 0;
void increment() {
  for (int i = 0; i < 1000; ++i) {
    std::unique_lock<std::mutex> lock(mtx);
    ++counter;
    lock.unlock();
  }
}
int main() {
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) { threads.emplace_back(increment); }
  for (auto& t : threads) { t.join(); }
  std::cout << "Counter: " << counter << std::endl;
}
```

Object Oriented Programming

Multithreading

Atomic operations

For simple data types, C++ provides atomic operations through the `std::atomic` template class, which ensures that operations on the data are atomic and not interrupted by other threads.

These are types that encapsulate a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads

Object Oriented Programming

Multithreading

Using an atomic variable for simple data types

```cpp
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>
std::atomic<int> counter(0);
void increment() {
  for (int i = 0; i < 1000; ++i) { ++counter; }
}
int main() {
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) { threads.emplace_back(increment); }
  for (auto& t : threads) { t.join(); }
  std::cout << "Counter: " << counter << std::endl;
}
```

Object Oriented Programming

Multithreading

Condition variables

`std::condition_variable` and `std::condition_variable_any` can be used for inter-thread communication and signaling, allowing one or more threads to wait for a specific condition to be satisfied.

```
std::mutex mtx;
std::condition_variable cv;
bool ready = false;
void set_ready() {
  std::unique_lock<std::mutex> lock(mtx);
  ready = true;
  cv.notify_one();
}
void wait_for_ready() {
  std::unique_lock<std::mutex> lock(mtx);
  cv.wait(lock, [] { return ready; });
}
```

Object Oriented Programming

Multithreading

Using a condition variable for inter-thread communication part 1

```
#include <iostream>
#include <condition_variable>
#include <thread>
std::mutex mtx;
std::condition_variable cv;
bool ready = false;
void change_ready_status() {
  std::unique_lock<std::mutex> lock(mtx);
  ready = true;
  cv.notify_one();
}
```

Object Oriented Programming

Multithreading

Using a condition variable for inter-thread communication part 2

```
1   void wait_for_ready() {
2     std::unique_lock<std::mutex> lock(mtx);
3     cv.wait(lock, [] { return ready; });
4     std::cout << "Ready!" << std::endl;
5   }
6   int main() {
7     std::thread waiting_thread(wait_for_ready);
8     std::thread setter_thread(change_ready_status);
9     waiting_thread.join();
10    setter_thread.join();
11  }
```

Task

Object Oriented Programming

Task

Implement a program that calculates the frequency of each nucleotide (A, C, G, and T) in a large DNA sequence using multiple threads.

The main thread should divide the input sequence into smaller sections and assign each section to a separate worker thread. The worker threads should calculate the frequency of each nucleotide in their assigned sections, and then the main thread should combine the results from all worker threads to compute the total frequency of each nucleotide.

Object Oriented Programming

Task

Implement a program that calculates the frequency of each nucleotide (A, C, G, and T) in a large DNA sequence using multiple threads.

Hints:

01. Use std::string for the input DNA sequence.
02. Create a function for the worker threads that takes the input sequence, start index, and end index as arguments.
03. In the main function, create a vector of std::thread objects, spawn the worker threads, and store them in the vector.
04. Use join() to wait for all worker threads to finish.
05. Combine the results from all worker threads to get the total frequency of each nucleotide.
06. Don’t forget to protect shared data with a mutex, or use atomic variables to store the intermediate results.

Thank you

Feel free to reach me via LinkedIn

Fin