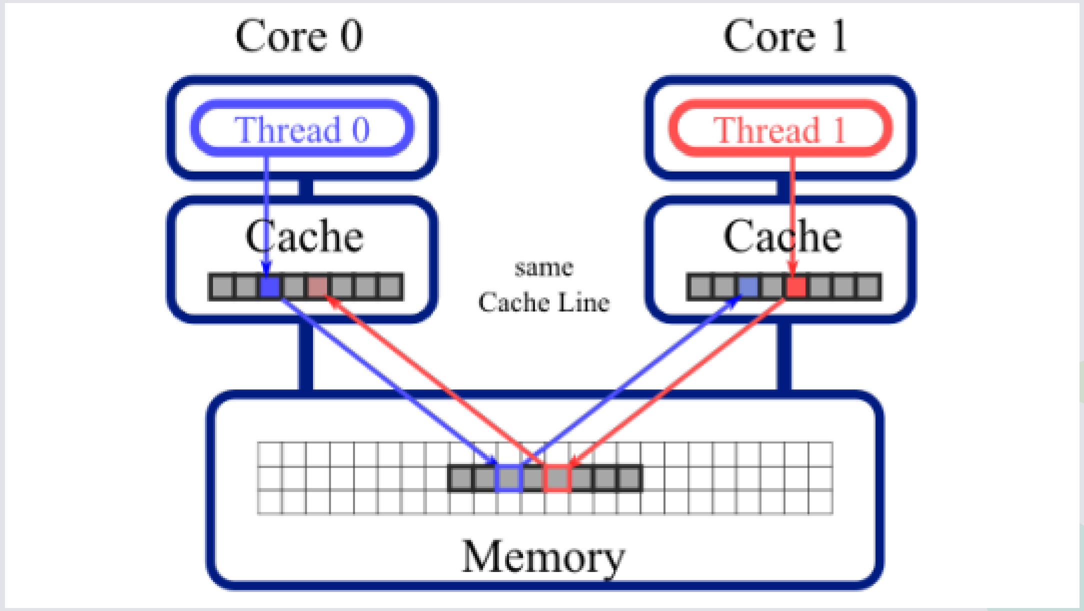
Analyzing False Sharing

Team Members: Alon Raigorodetsky, Puja Girishkumar Kakani

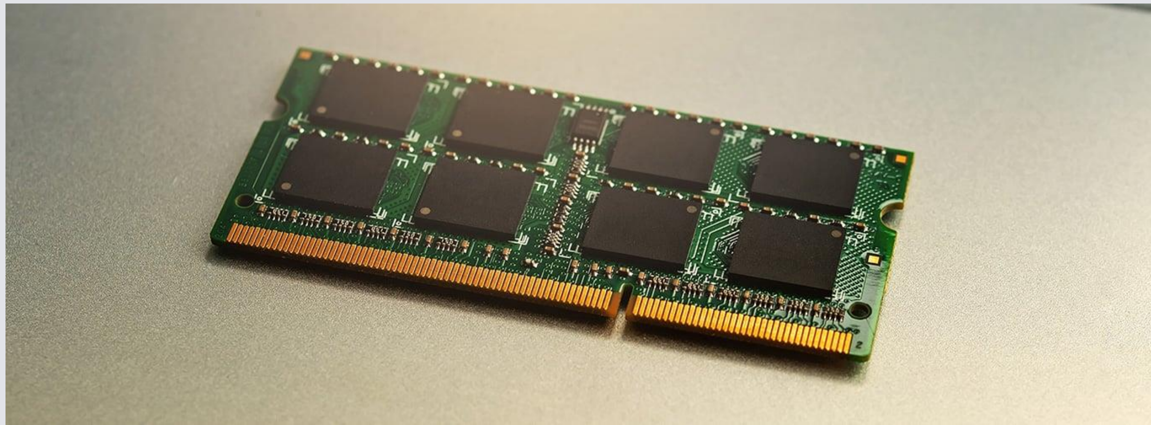
What is False Sharing?

- 2 threads accessing the same data
- Modification of the data that is being accessed.



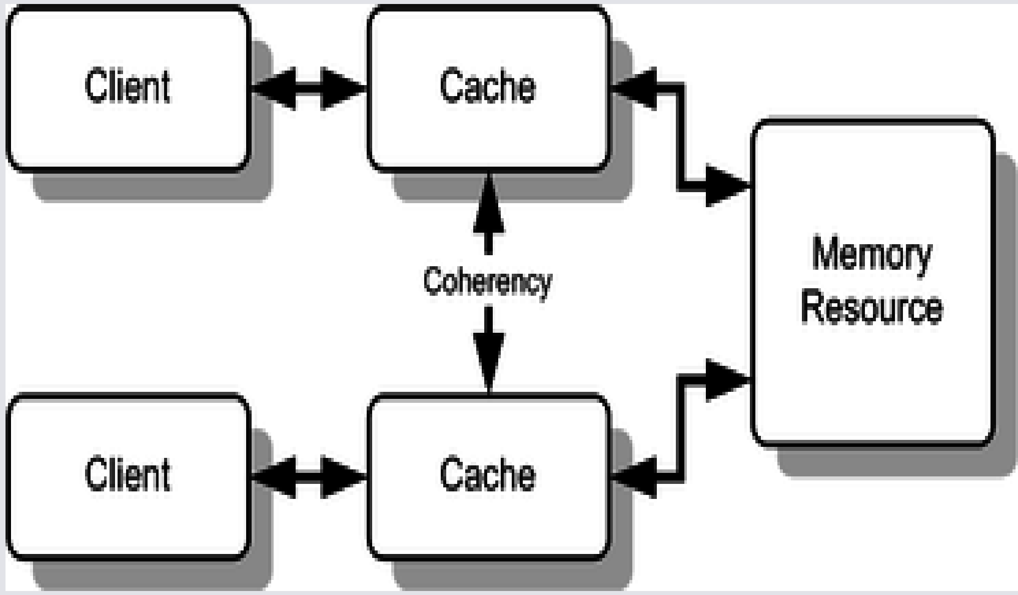
What is Cache?

- A cache is a high speed data storage layer that stores frequently accessed data which is used by the operating system
- Cache can reduce time or resources that take to perform and action but sometimes it can cause a stale if not updated properly



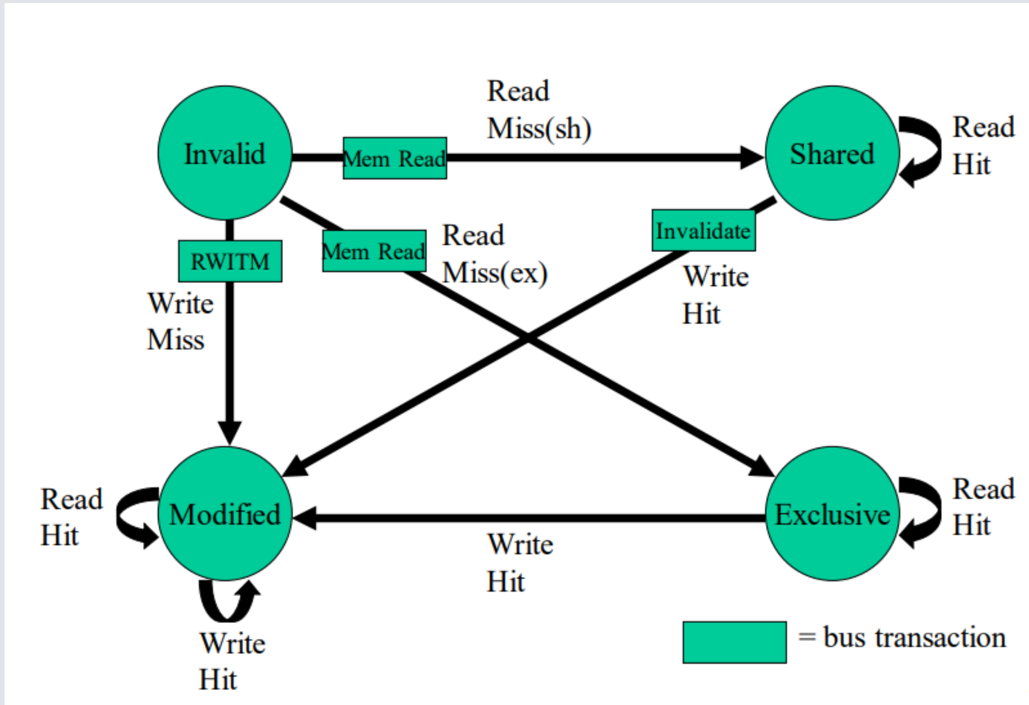
What is Cache Consistency?

- Cache consistency refers to the synchronization of data between different caches in a computer system, ensuring that all caches have the most up-to-date version of data.
- Without Cache Consistency, different processors may have different versions of the same data, which can lead to data inconsistencies and errors in the system.

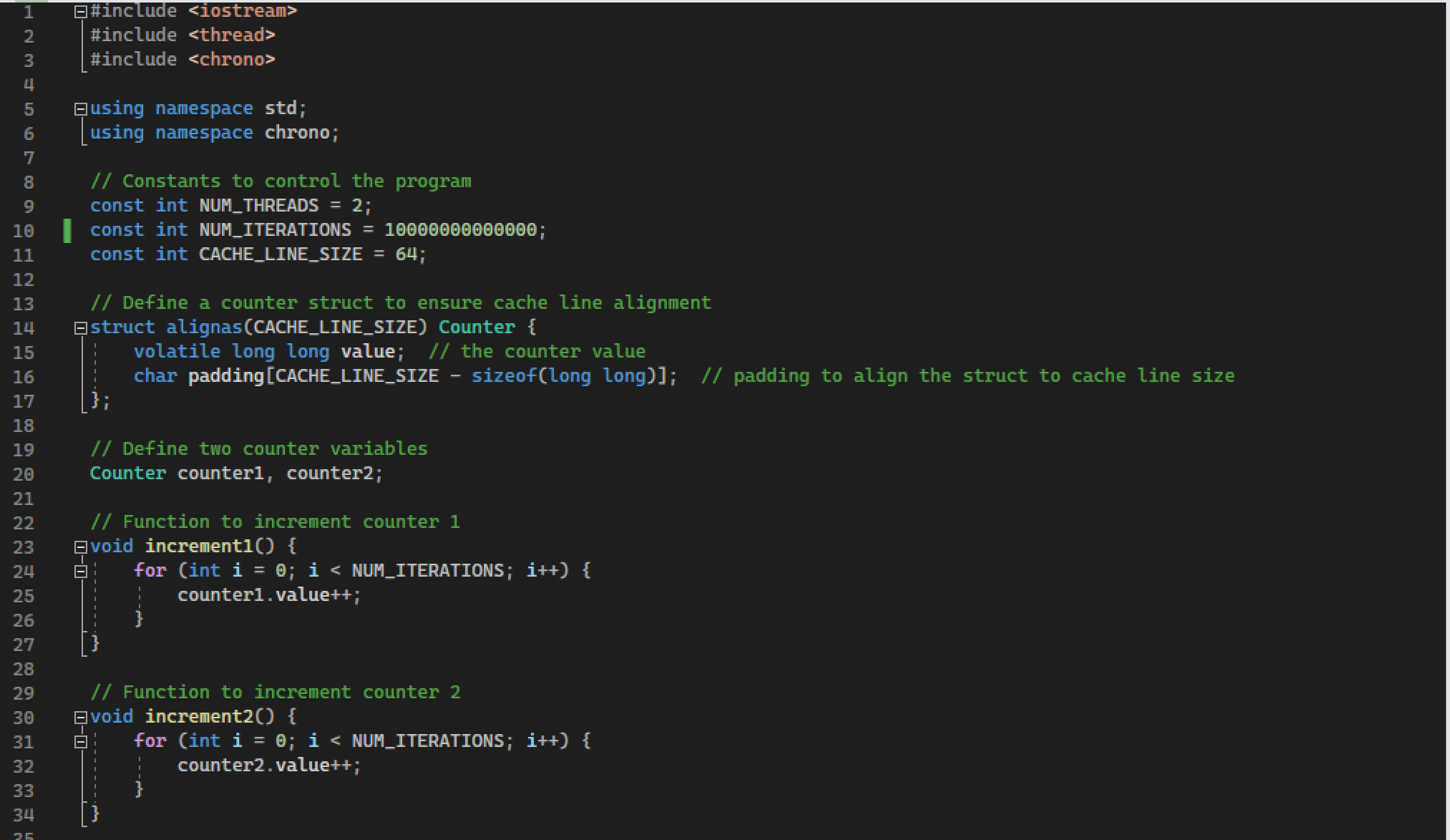


The MESI Protocol

- **Modified** - The only cached duplicate is the modified, distinct from main memory cache line.
- **Exclusive** - The modified, separate from main memory cache line is the only cached copy.
- **Shared** - identical to main memory, but there might be duplicates in other caches.
- **Invalid** - Line data is not valid.

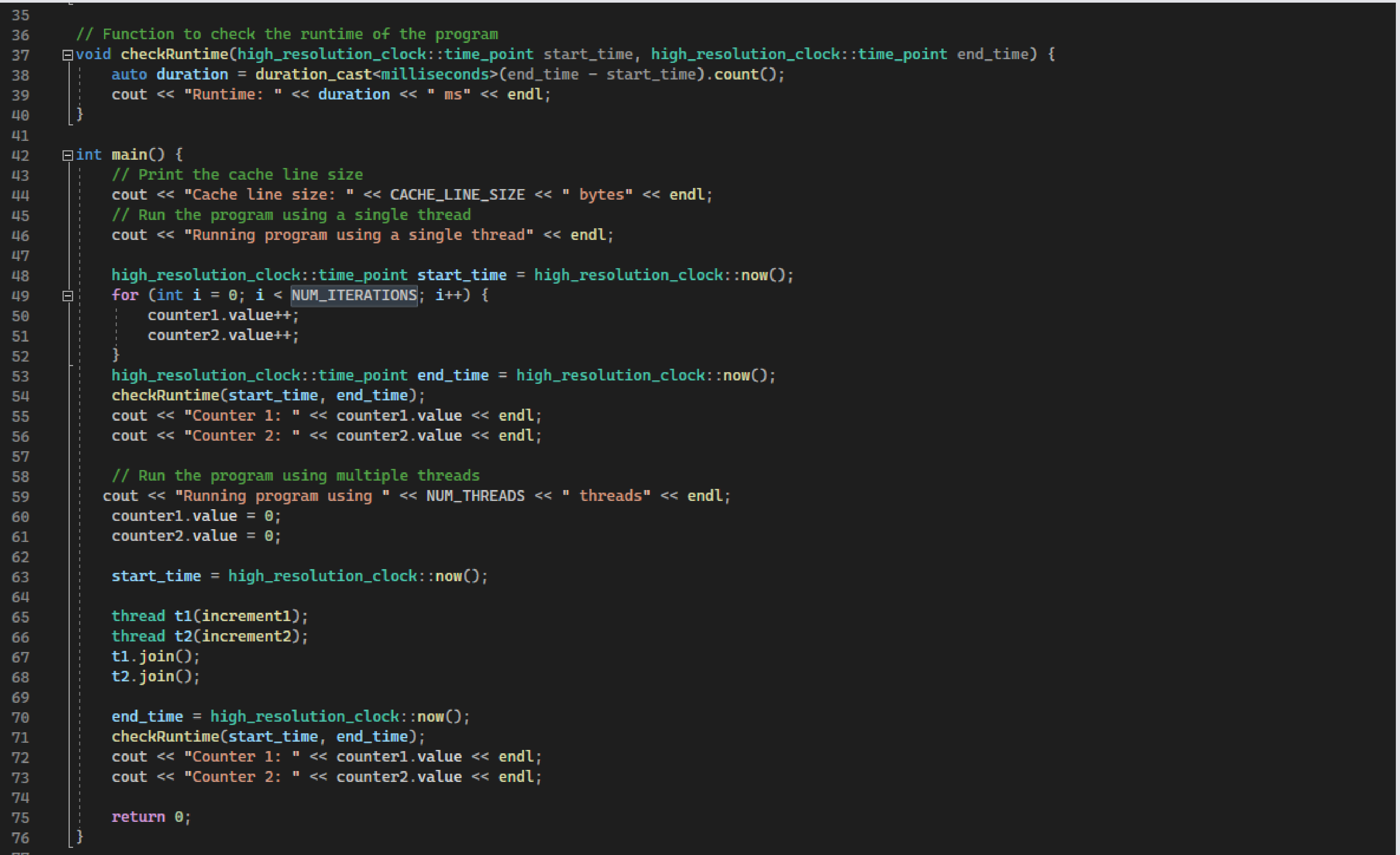


How the code works?



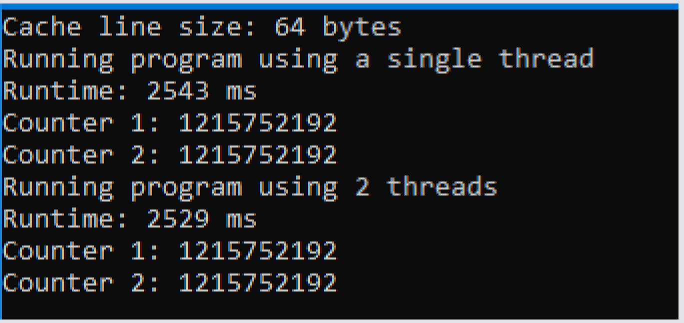
```
#include <iostream>
#include <thread>
#include <chrono>

using namespace std;
using namespace chrono;

// Constants to control the program
const int NUM_THREADS = 2;
const int NUM_ITERATIONS = 10000000000000;
const int CACHE_LINE_SIZE = 64;

// Define a counter struct to ensure cache line alignment
struct alignas(CACHE_LINE_SIZE) Counter {
    volatile long long value;  // the counter value
    char padding[CACHE_LINE_SIZE - sizeof(long long)];  // padding to align the struct to cache line size
};

// Define two counter variables
Counter counter1, counter2;

// Function to increment counter 1
void increment1() {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        counter1.value++;
    }
}

// Function to increment counter 2
void increment2() {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        counter2.value++;
    }
}
```

How the Code Works?

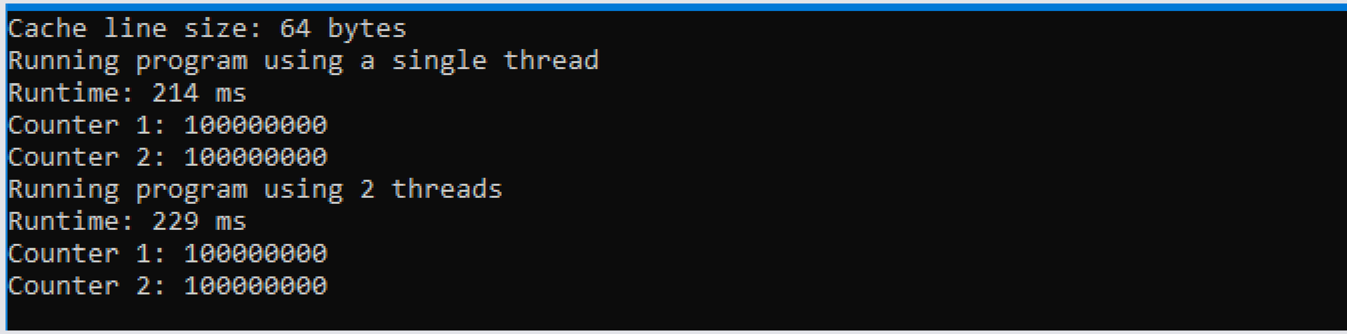


```cpp

// Function to check the runtime of the program
void checkRuntime(high_resolution_clock::time_point start_time, high_resolution_clock::time_point end_time) {
    auto duration = duration_cast<milliseconds>(end_time - start_time).count();
    cout << "Runtime: " << duration << " ms" << endl;
}

int main() {
    // Print the cache line size
    cout << "Cache line size: " << CACHE_LINE_SIZE << " bytes" << endl;
    // Run the program using a single thread
    cout << "Running program using a single thread" << endl;

    high_resolution_clock::time_point start_time = high_resolution_clock::now();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        counter1.value++;
        counter2.value++;
    }
    high_resolution_clock::time_point end_time = high_resolution_clock::now();
    checkRuntime(start_time, end_time);
    cout << "Counter 1: " << counter1.value << endl;
    cout << "Counter 2: " << counter2.value << endl;

    // Run the program using multiple threads
   cout << "Running program using " << NUM_THREADS << " threads" << endl;
    counter1.value = 0;
    counter2.value = 0;

    start_time = high_resolution_clock::now();

    thread t1(increment1);
    thread t2(increment2);
    t1.join();
    t2.join();

    end_time = high_resolution_clock::now();
    checkRuntime(start_time, end_time);
    cout << "Counter 1: " << counter1.value << endl;
    cout << "Counter 2: " << counter2.value << endl;

    return 0;
}
```

Output



```
Cache line size: 64 bytes
Running program using a single thread
Runtime: 2543 ms
Counter 1: 1215752192
Counter 2: 1215752192
Running program using 2 threads
Runtime: 2529 ms
Counter 1: 1215752192
Counter 2: 1215752192
```

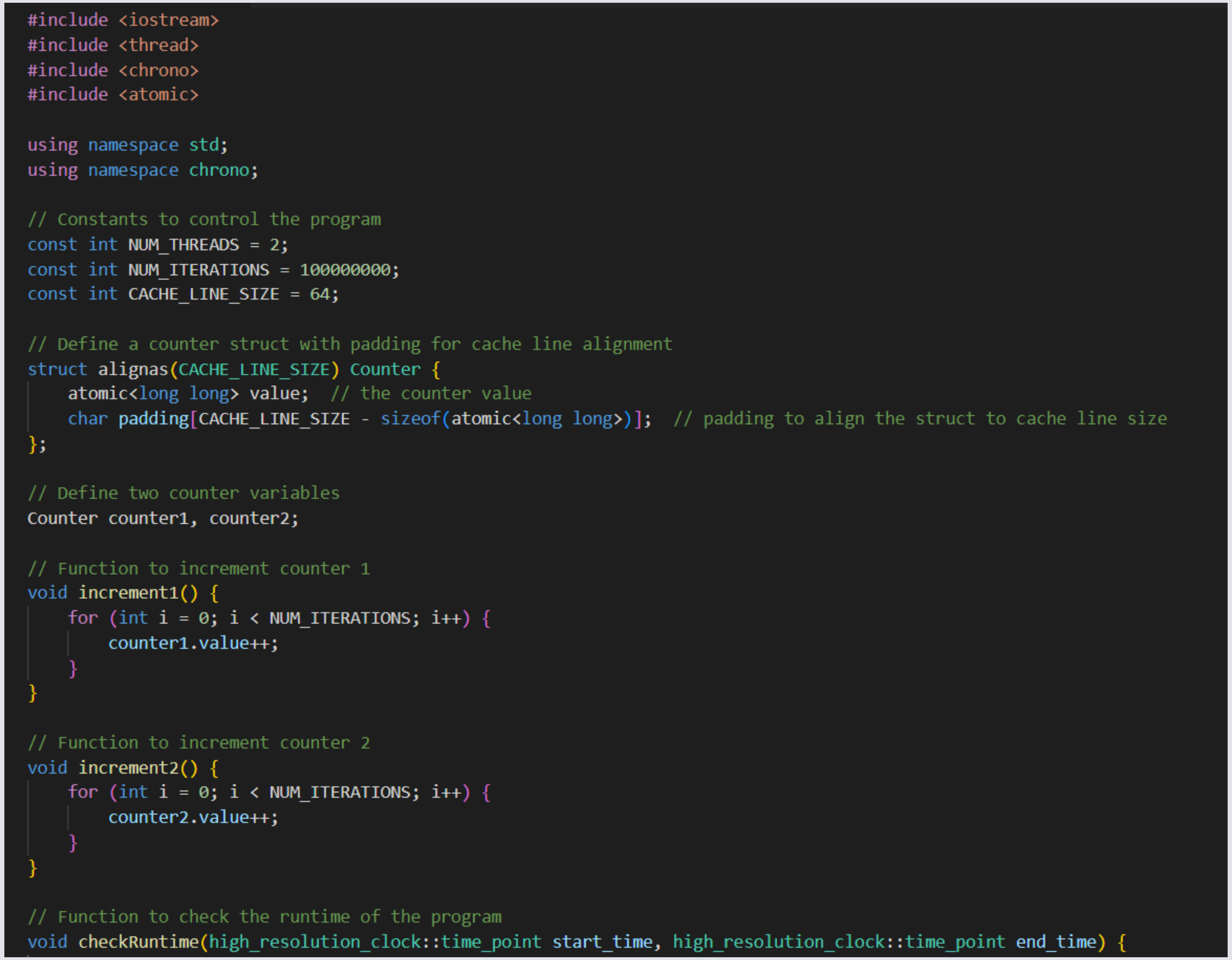


```
Cache line size: 64 bytes
Running program using a single thread
Runtime: 214 ms
Counter 1: 100000000
Counter 2: 100000000
Running program using 2 threads
Runtime: 229 ms
Counter 1: 100000000
Counter 2: 100000000
```

Solution

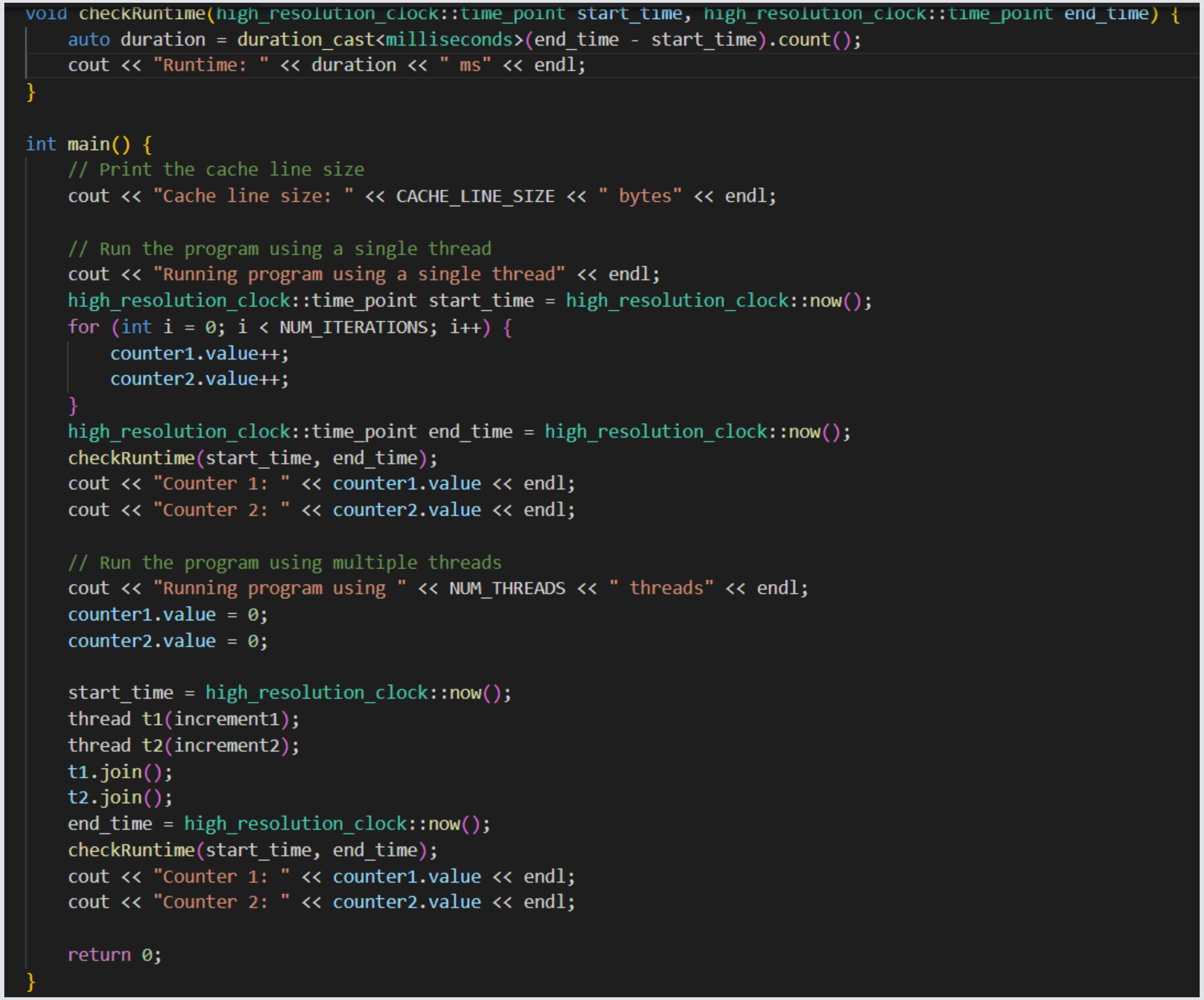
- A performance hit results as a result of each thread invalidating the cache line for the other thread. In order to ensure that each variable is aligned to its own cache line, we can fix this issue by adding padding to the Counter struct.
- By ensuring that each thread updates a different cache line, false sharing is prevented.

Code for the Solution



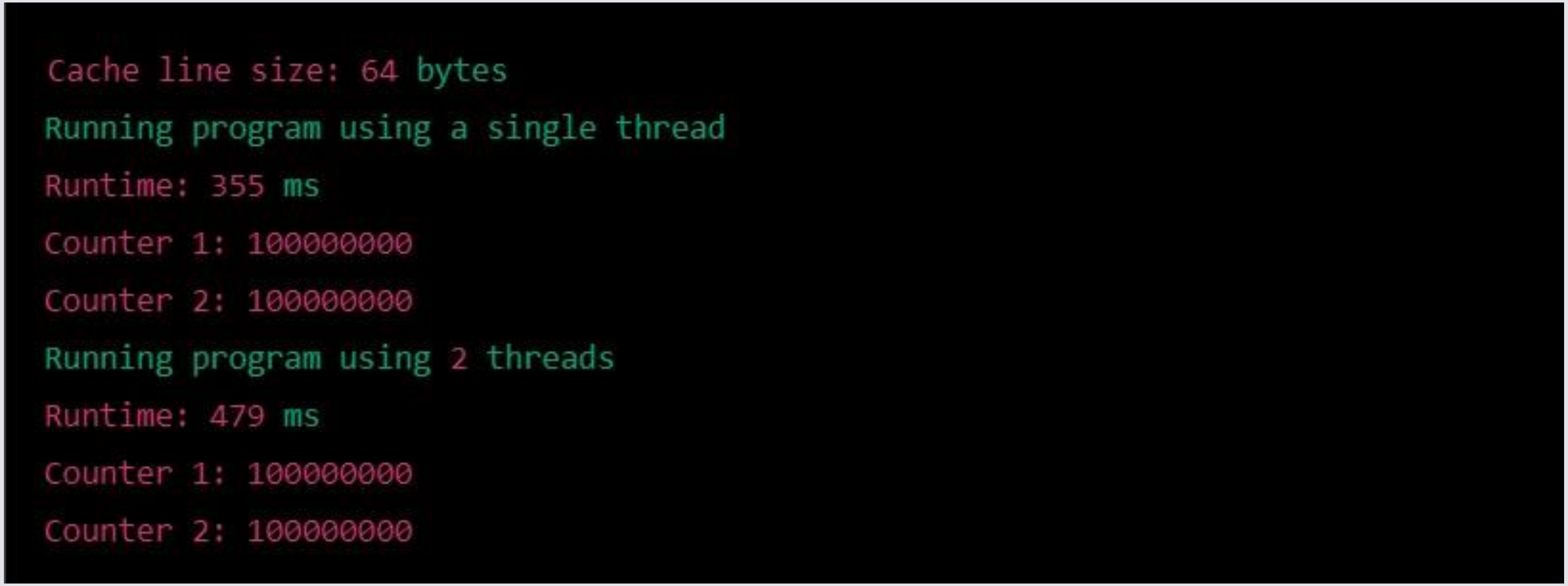
```cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <atomic>

using namespace std;
using namespace chrono;

// Constants to control the program
const int NUM_THREADS = 2;
const int NUM_ITERATIONS = 100000000;
const int CACHE_LINE_SIZE = 64;

// Define a counter struct with padding for cache line alignment
struct alignas(CACHE_LINE_SIZE) Counter {
    atomic<long long> value;  // the counter value
    char padding[CACHE_LINE_SIZE - sizeof(atomic<long long>)];  // padding to align the struct to cache line size
};

// Define two counter variables
Counter counter1, counter2;

// Function to increment counter 1
void increment1() {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        counter1.value++;
    }
}

// Function to increment counter 2
void increment2() {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        counter2.value++;
    }
}

// Function to check the runtime of the program
void checkRuntime(high_resolution_clock::time_point start_time, high_resolution_clock::time_point end_time) {
```

Code for the Solution



```
void checkRuntime(high_resolution_clock::time_point start_time, high_resolution_clock::time_point end_time) {
    auto duration = duration_cast<milliseconds>(end_time - start_time).count();
    cout << "Runtime: " << duration << " ms" << endl;
}

int main() {
    // Print the cache line size
    cout << "Cache line size: " << CACHE_LINE_SIZE << " bytes" << endl;

    // Run the program using a single thread
    cout << "Running program using a single thread" << endl;
    high_resolution_clock::time_point start_time = high_resolution_clock::now();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        counter1.value++;
        counter2.value++;
    }
    high_resolution_clock::time_point end_time = high_resolution_clock::now();
    checkRuntime(start_time, end_time);
    cout << "Counter 1: " << counter1.value << endl;
    cout << "Counter 2: " << counter2.value << endl;

    // Run the program using multiple threads
    cout << "Running program using " << NUM_THREADS << " threads" << endl;
    counter1.value = 0;
    counter2.value = 0;

    start_time = high_resolution_clock::now();
    thread t1(increment1);
    thread t2(increment2);
    t1.join();
    t2.join();
    end_time = high_resolution_clock::now();
    checkRuntime(start_time, end_time);
    cout << "Counter 1: " << counter1.value << endl;
    cout << "Counter 2: " << counter2.value << endl;

    return 0;
}
```

Code Output



```
Cache line size: 64 bytes
Running program using a single thread
Runtime: 355 ms
Counter 1: 100000000
Counter 2: 100000000
Running program using 2 threads
Runtime: 479 ms
Counter 1: 100000000
Counter 2: 100000000
```

Summary

- False sharing is a performance problem that can happen in multi-threaded applications when different variables on the same cache line are accessed by different threads at the same time.
- Finding the cache lines that are shared by several threads and then identifying the variables that cause the false sharing are the first steps in false sharing analysis.
- In conclusion, watch out for false sharing because it kills scalability.
- Prevent false sharing by lowering the frequency of updates to the variables that are shared.

Do you have any questions?