**Scenario - Multi-Threading Race Condition Debugging in C++**

**Scenario:**  
Detected and fixed a race condition in a multi-threaded counter program using Helgrind to identify data races and std::mutex to synchronize access.

**Problem Description:**  
The program attempted to increment a shared counter across multiple threads. Without proper synchronization, threads overwrote each other’s updates, leading to incorrect final values. This type of bug is common in multi-threaded applications and can cause inconsistent behavior in production.

**Tools & Languages:**

* **Language:** C++
* **Debugging Tools:** Valgrind Helgrind
* **Libraries:** <thread>, <mutex>, <vector>

**Initial Buggy Code:**

#include <iostream>

#include <thread>

#include <vector>

int counter = 0;

void incrementCounter(int n) {

for (int i = 0; i < n; ++i) {

++counter; // unsynchronized increment → race condition

}

}

int main() {

int numThreads = 2;

int increments = 1000000;

std::vector<std::thread> threads;

for (int i = 0; i < numThreads; ++i) {

threads.emplace\_back(incrementCounter, increments);

}

for (auto &t : threads) {

t.join();

}

std::cout << "Final counter value: " << counter << std::endl;

return 0;

}

**Issue Detected:**

* Helgrind reported **possible data races** on the counter variable.
* Output did not consistently match the expected value (numThreads \* increments).

**Fixed Code:**

#include <iostream>

#include <thread>

#include <vector>

#include <mutex>

int counter = 0;

std::mutex mtx;

void incrementCounter(int n) {

for (int i = 0; i < n; ++i) {

std::lock\_guard<std::mutex> lock(mtx); // protect critical section

++counter;

}

}

int main() {

int numThreads = 2;

int increments = 1000000;

std::vector<std::thread> threads;

for (int i = 0; i < numThreads; ++i) {

threads.emplace\_back(incrementCounter, increments);

}

for (auto &t : threads) {

t.join();

}

std::cout << "Final counter value: " << counter << std::endl;

return 0;

}

**Outcome:**

* Final counter consistently matches expected value:

Final counter value: 2000000

* Helgrind reports **0 data races**.

**Lessons Learned:**

* Race conditions occur when multiple threads access a shared resource without synchronization.
* Using std::mutex or atomic variables ensures thread-safe operations.
* Helgrind is invaluable for detecting subtle threading bugs that are hard to catch by inspection alone.
* Always validate multi-threaded fixes with both tool-assisted checks and repeated program runs.