💡 Critical Sections and Race Conditions

Overview

Concept	Demo Takeaway
Nondeterminism	Students see that even simple math breaks down when shared.
Race Condition	Two or more threads accessing/modifying shared data without coordination.
Synchronization	`Lock()` enforces mutual exclusion, restoring deterministic behavior.
Scalability trade-off	Locking fixes correctness but reduces parallel performance.

🧠 Race Condition Demo (Python Version)

```python
# race_demo.py
import threading

counter = 0

def increment():
    global counter
    for _ in range(100000):
        # Not atomic: read, increment, write
        temp = counter
        temp += 1
        counter = temp

threads = [threading.Thread(target=increment) for _ in range(10)]

for t in threads:
    t.start()
for t in threads:
    t.join()

print("Expected:", 5 * 100000)
print("Actual:", counter)
```

🧩 Run it a few times:

```
python3 race_demo.py
```

You’ll get something like:

```
Expected: 500000
Actual: 500000
```

...and it doesn't change every run like we thought.

That’s pythons GIL slapping you in the face.

🧰 Unnecessarily Fixed Version (with Lock)

```python
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

threads = [threading.Thread(target=increment) for _ in range(5)]

for t in threads:
    t.start()
for t in threads:
    t.join()

print("Expected:", 5 * 100000)
print("Actual:", counter)
```

Python MultiProcessing

```python
#race_multi_demo.py
from multiprocessing import Process, Value

def increment(counter):
    for _ in range(10000):
        counter.value += 1  # Not atomic across processes!

if __name__ == "__main__":
    from ctypes import c_int
    counter = Value(c_int, 0)
    procs = [Process(target=increment, args=(counter,)) for _ in range(5)]

    for p in procs: p.start()
    for p in procs: p.join()

    print("Expected:", 5 * 10000)
    print("Actual:", counter.value)
```

```
Expected: 50000
```

C++ Examples

🧩 C++ Race Condition Demo

```cpp
// race_demo.cpp
#include <iostream>
#include <thread>
#include <vector>

long counter = 0;

void increment() {
    for (int i = 0; i < 100000; ++i) {
        // Not atomic: read -> modify -> write
        long temp = counter;
        temp += 1;
        counter = temp;
    }
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 5; ++i)
        threads.emplace_back(increment);

    for (auto& t : threads)
        t.join();

    std::cout << "Expected: " << 5 * 100000 << "\n";
    std::cout << "Actual:   " << counter << "\n";

    return 0;
}
```

⚙️ Compile & Run

```
g++ -std=c++17 -pthread race_demo.cpp -o race_demo
./race_demo
```

💥 Output Example

```
Expected: 500000
Actual:   278416 (ish)
```

…and it'll change every time you run it —

a live demonstration of **nondeterminism** and **race conditions**.

🧰 Fixed Version: Using `std::mutex`

```cpp
// race_demo_mutex.cpp
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

long counter = 0;
std::mutex mtx;

void increment() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        counter++;
    }
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 5; ++i)
        threads.emplace_back(increment);

    for (auto& t : threads)
        t.join();

    std::cout << "Expected: " << 5 * 100000 << "\n";
    std::cout << "Actual:   " << counter << "\n";

    return 0;
}
```

🧩 Discussion

Concept	C++ Example Insight
Data Race	Threads read/write shared variable simultaneously.
Nondeterminism	Different runs yield different results.
Synchronization	`std::mutex` ensures only one thread updates at a time.
RAII Safety	`std::lock_guard` automatically releases lock even on exceptions.
Performance Trade-off	Correctness over speed — necessary evil in OS design.

🎓 Extension Ideas

For students or labs:

1. **Add `std::this_thread::sleep_for()`** to exaggerate the race condition.
2. Replace `std::mutex` with `std::atomic<long>` and compare performance.
3. Visualize per-thread progress bars (using `std::cout` + `flush`).
4. Discuss which OS primitive (`futex`, spinlock, etc.) sits under `std::mutex`.