Laboratorio 6: Programación paralela

Ricardo Hidalgo Campos B63464

June 2025

1. Condiciones de carrera

Para exponer un caso de condición de carrera se corrió el siguiente código:

```cpp
#include <iostream>
#include <cstdlib>
#include <vector>

int main() {
    int counter = 0;

    for (int i = 0; i < 1000; ++i) {
        ++counter;
    }

    std::vector<int> myList;
    myList.push_back(1);
    myList.push_back(2);

    return 0;
}
```

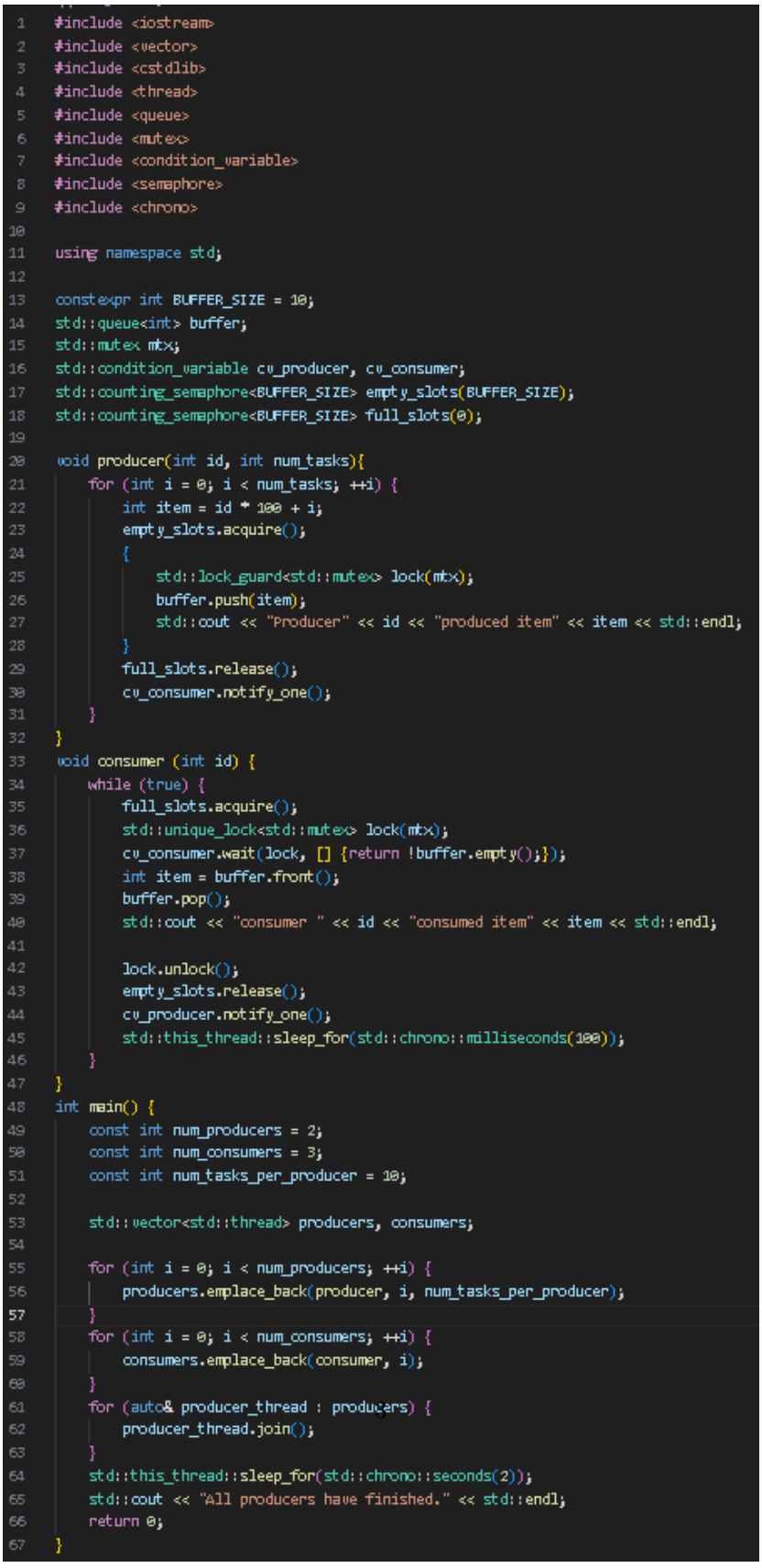
Figura 1: Código de prueba de condición de carrera

Al correr dicho código se obtuvo el siguiente comportamiento:

Figura 2: Salida del código prueba

Aquí se puede observar como los threads se comportan de manera errática debido a que intentan simultáneamente ingresar al mismo espacio de memoria, guardando valores mal. La otra parte del código lo que sucede es que hay un conteo incorrecto de counter porque al haber múltiples hilos, estos leen su valor antes de actualizar.

Seguidamente, para hacer un estudio de cada método para solucionar este tipo, se realizo un código ejemplo con diferentes métodos y sera explicado cada uno de ellos mas adelante:



```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <semaphore>
#include <chrono>

using namespace std;

constexpr int BUFFER_SIZE = 10;
std::queue<int> buffer;
std::mutex mtx;
std::condition_variable cv_producer, cv_consumer;
std::counting_semaphore<BUFFER_SIZE> empty_slots(BUFFER_SIZE);
std::counting_semaphore<BUFFER_SIZE> full_slots(0);

void producer(int id, int num_tasks){
    for (int i = 0; i < num_tasks; ++i) {
        int item = id * 100 + i;
        empty_slots.acquire();
        {
            std::lock_guard<std::mutex> lock(mtx);
            buffer.push(item);
            std::cout << "Producer" << id << "produced item" << item << std::endl;
        }
        full_slots.release();
        cv_consumer.notify_one();
    }
}
void consumer (int id) {
    while (true) {
        full_slots.acquire();
        std::unique_lock<std::mutex> lock(mtx);
        cv_consumer.wait(lock, [] {return !buffer.empty();});
        int item = buffer.front();
        buffer.pop();
        std::cout << "consumer " << id << "consumed item" << item << std::endl;

        lock.unlock();
        empty_slots.release();
        cv_producer.notify_one();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
int main() {
    const int num_producers = 2;
    const int num_consumers = 3;
    const int num_tasks_per_producer = 10;

    std::vector<std::thread> producers, consumers;

    for (int i = 0; i < num_producers; ++i) {
        producers.emplace_back(producer, i, num_tasks_per_producer);
    }
    for (int i = 0; i < num_consumers; ++i) {
        consumers.emplace_back(consumer, i);
    }
    for (auto& producer_thread : producers) {
        producer_thread.join();
    }
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "All producers have finished." << std::endl;
    return 0;
}
```

Figura 3: Código de prueba de los casos

Este código entrego la la siguiente salida:

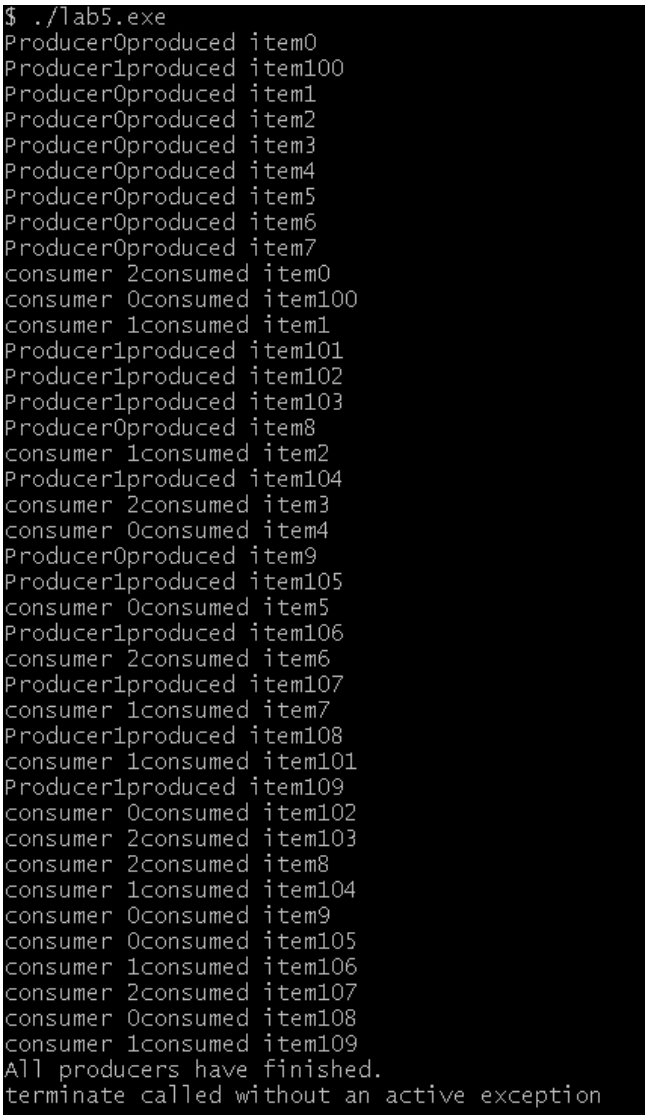


Figura 4: Salida del código prueba

ahora, para solucionar y evitar problemas de threads se usaron mecanismos de sincronización los cuales, de diferentes maneras, hacen un manejo de cada thread para que no ocurran errores. El primero que hablaremos es del mutex el cual hace que los threads tomen turnos y así evitar los errores. En el código aparece en las siguientes lineas:

Figura 5: Sincronización por mutex

Otro método de sincronización utilizado es el de semaphore, el cual existe desde las versiones de C++20 y permite crear un semáforo que mediante un contador permite controlar el trafico de threads de cuantos threads pueden acceder a la información al mismo tiempo y cada cuanto tiempo. de ejemplo de su uso esta en el código ejemplo en la siguiente sección:

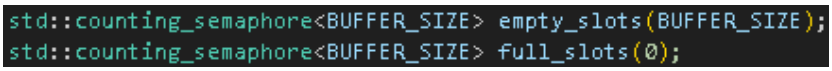


Figura 6: Sincronización por semaphore

El siguiente mecanismo es un mecanismo llamado "variables de condición."[el] cual funciona de forma que un thread esta bloqueado hasta que otro thread de la señal que uno define. Esta implementación se ve de la siguiente manera:

Figura 7: Sincronización por variable de condición

Se observa que se evitaron los deadlock o locks donde un thread no avanza ni el otro tampoco por esperar al anterior por lo que ninguno accede y se bloquean mutuamente. Ademas no fue necesario usar otro mecanismo mas de sincronización como lo es las barries que su comportamiento se basa en que toma un grupo de threads y los hace esperar hasta que todo el grupo alcancen cierto nivel de sincronización para dejarlos avanzar.