

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:

```
#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:

```
+thread-gdb Debug Console (Ctrl+Shift+V)
GNU gdb (GDB) /x86_64:
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs>.
Warning: Debuggee TargetArchitecture not detected, assuming x86_64.
<cmd-param-changed,param="pagination",value="off">
<cmd-param-changed,param="args",value="--COR 1xCOR 4COR">
[New Thread 14364.64d68]
[New Thread 14364.642158]
[New Thread 14364.64d768]
[New Thread 14364.64d858]
Loaded "C:\WINDOWS\System32\kernel32.dll". Symbols loaded.
Loaded "C:\WINDOWS\System32\kernelbase.dll". Symbols loaded.
Loaded "C:\WINDOWS\System32\user32.dll". Symbols loaded.
Loaded "C:\WINDOWS\System32\ole32.dll". Symbols loaded.
Loaded "C:\WINDOWS\System32\oleaut32.dll". Symbols loaded.
Loaded "C:\WINDOWS\System32\GDI32.dll". Symbols loaded.
Execute debugger commands using "exec command", for example "exec info registers" will list registers in use (when GDB is the debugger).
The program "C:\Users\Alvaro\Downloads\CrackOffice2019\CrackOffice.exe" has entered into mode 3 (dependencies).
```

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:

```

1  #include <iostream>
2  #include <vector>
3  #include <cstdlib>
4  #include <thread>
5  #include <queue>
6  #include <mutex>
7  #include <condition_variable>
8  #include <semaphore>
9  #include <chrono>
10
11 using namespace std;
12
13 constexpr int BUFFER_SIZE = 10;
14 std::queue<int> buffer;
15 std::mutex mtx;
16 std::condition_variable cv_producer, cv_consumer;
17 std::counting_semaphore<BUFFER_SIZE> empty_slots(BUFFER_SIZE);
18 std::counting_semaphore<BUFFER_SIZE> full_slots(0);
19
20 void producer(int id, int num_tasks){
21     for (int i = 0; i < num_tasks; ++i) {
22         int item = id * 100 + i;
23         empty_slots.acquire();
24         {
25             std::lock_guard<std::mutex> lock(mtx);
26             buffer.push(item);
27             std::cout << "Producer " << id << " produced item" << item << std::endl;
28         }
29         full_slots.release();
30         cv_consumer.notify_one();
31     }
32 }
33
34 void consumer (int id) {
35     while (true) {
36         full_slots.acquire();
37         std::unique_lock<std::mutex> lock(mtx);
38         cv_consumer.wait(lock, [] {return !buffer.empty();});
39         int item = buffer.front();
40         buffer.pop();
41         std::cout << "consumer " << id << " consumed item" << item << std::endl;
42
43         lock.unlock();
44         empty_slots.release();
45         cv_producer.notify_one();
46         std::this_thread::sleep_for(std::chrono::milliseconds(100));
47     }
48 }
49
50 int main() {
51     const int num_producers = 2;
52     const int num_consumers = 3;
53     const int num_tasks_per_producer = 10;
54
55     std::vector<std::thread> producers, consumers;
56
57     for (int i = 0; i < num_producers; ++i) {
58         producers.emplace_back(producer, i, num_tasks_per_producer);
59     }
60     for (int i = 0; i < num_consumers; ++i) {
61         consumers.emplace_back(consumer, i);
62     }
63     for (auto& producer_thread : producers) {
64         producer_thread.join();
65     }
66     std::this_thread::sleep_for(std::chrono::seconds(2));
67     std::cout << "All producers have finished." << std::endl;
68     return 0;
69 }

```

Figura 3: Código de prueba de los casos

Este código entrega la la siguiente salida:

```
$ ./lab5.exe
Producer0produced item0
Producer1produced item100
Producer0produced item1
Producer0produced item2
Producer0produced item3
Producer0produced item4
Producer0produced item5
Producer0produced item6
Producer0produced item7
consumer 2consumed item0
consumer 0consumed item100
consumer 1consumed item1
Producer1produced item101
Producer1produced item102
Producer1produced item103
Producer0produced item8
consumer 1consumed item2
Producer1produced item104
consumer 2consumed item3
consumer 0consumed item4
Producer0produced item9
Producer1produced item105
consumer 0consumed item5
Producer1produced item106
consumer 2consumed item6
Producer1produced item107
consumer 1consumed item7
Producer1produced item108
consumer 1consumed item101
Producer1produced item109
consumer 0consumed item102
consumer 2consumed item103
consumer 2consumed item8
consumer 1consumed item104
consumer 0consumed item9
consumer 0consumed item105
consumer 1consumed item106
consumer 2consumed item107
consumer 0consumed item108
consumer 1consumed item109
All producers have finished.
terminate called without an active exception
```

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 líneas:

```
std::lock_guard<std::mutex> lock(mtx);
```

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 tráfico 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:

```
std::counting_semaphore<BUFFER_SIZE> empty_slots(BUFFER_SIZE);  
std::counting_semaphore<BUFFER_SIZE> full_slots(0);
```

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:

```
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();
```

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. Además 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.