Concurrent and Distributed Programming

Carlos Bernalte García-Junco
carlos.bernalte@alu.uclm.es

University of Rzeszów — November 29, 2021

Contents

List of Figures

Introduction to critical section

The main propose of this project is how to solve some synchronization problems raised in the field of computer science. The problem of synchronization between few process or threads is when they have shared resources, this can lead to erroneous behavior. Our task is to have a control access for these processes and threads from this **critical section**.

1 Project decisions

First of all, the decisions made will be explained for developing this project. The programming language will be **C++** just because it provides tools to control threads and memory space. About how to distribute the files, as you can see in Figure 1, we will have the source code at folder *src* and the compiled files a the *bin* folder.

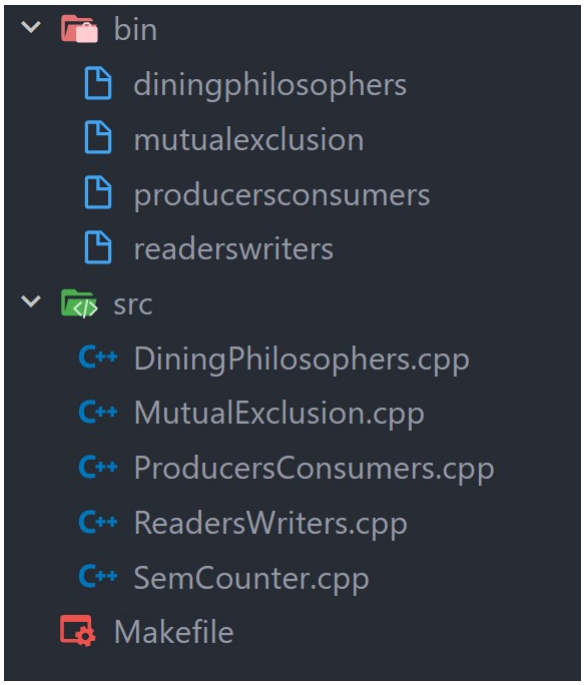

Figure 1: Folders with their files

On addition, there is also available a *Makefile* in order to compile and run much easier the programs. It is compulsory to add "*-pthread -std=c++11*" if you want to use Threads in your code.

```
DiningPhilosophers:
  g++ src/DiningPhilosophers.cpp -o bin/diningphilosophers -pthread -std=c++11
  ./bin/diningphilosophers
MutualExclusion:
  g++ src/MutualExclusion.cpp -o bin/mutualexclusion -pthread -std=c++11
  ./bin/mutualexclusion
ProducersConsumers:
  g++ src/ProducersConsumers.cpp -o bin/producersconsumers -pthread -std=c++11
  ./bin/producersconsumers
ReadersWriters:
  g++ src/ReadersWriters.cpp -o bin/readerswriters -pthread -std=c++11
  ./bin/readerswriters
```

Listing 1: Makefile

1.1 Semaphore Counter

To solve this problems, is necessary to have an auxiliary class. This class is called **Semaphore Counter**, and counts (with the variable **value**) the number of threads which can access to a critical section. This class is initialized with the number of threads we want to have access.

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

class SemCounter {
private:
  int value;
  std::mutex mutex_;
  std::mutex mutex_block;

  void block();
  void unblock();

public:
  SemCounter(int value);
  void wait();
  void signal();
  int getValue();
};

SemCounter::SemCounter(int v): value(v){};


void SemCounter::block(){mutex_block.lock();}
void SemCounter::unblock(){mutex_block.unlock();}

void SemCounter::wait() {

  mutex_.lock();
  if(--value <= 0){
    mutex_.unlock();
    block();
    mutex_.lock();
  }
  mutex_.unlock();

}


void SemCounter::signal() {

  mutex_.lock();
  if(++value <= 0){
       unblock();
       /*damos tiempo para que se realice el desbloqueo correctamente*/
       std::this_thread::sleep_for(std::chrono::milliseconds(200));

  }
   mutex_.unlock();
}

int SemCounter::getValue(){
  return value;
}
```

Listing 2: Semaphore Counter code

2 Mutual Exclusion

2.1 Description of the problem

In computer science, mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions. It is the requirement that one thread of execution never enters a critical section while a concurrent thread of execution is already accessing critical section, which refers to an interval of time during which a thread of execution accesses a shared resource, such as shared data objects, shared resources or shared memory.

The shared resource is a data object, which two or more concurrent threads are trying to modify (where two concurrent read operations are permitted but, no two concurrent write operations or one read and one write are permitted, since it leads to data inconsistency). Mutual exclusion algorithm ensures that if a process is already performing write operation on a data object (critical section) no other process/thread is allowed to access/modify the same object until the first process has finished writing upon the data object (critical section) and released the object for other processes to read and write upon.

2.2 Solution of the problem

For solving this problem, I prepared an example where we can compare how the final result of a sum is affected if you no not protect that critical section. At the begging, I calculate the sum without using threads just only to know which will be the final result, and I store the numbers (generated randomly) just to have the same numbers for the rest of the sums.

Then N threads with a protected area and other N threads with an unprotected area, the two types of threads have been assigned a range to access the vector where the stored numbers are found to perform the sum.

The unprotected threads they only access to the vector summing on the same variable (critical section) the numbers they have been assigned, on the other hand, we have the threads with protected that they are using a lockguard semaphore (s It will protect them from the rest of the threads until it reaches the end of the method). In both types of threads there is also a delay of 20 milliseconds just to add some complexity.

Then on the example of output we can see that there is a huge difference between bot types of threads.

```
#include <vector>
#include <thread>
#include <functional>
#include <mutex>
#define LIMIT 100

long normal_sum = 0;
long sum1 = 0;
long sum2 = 0;
std::mutex mutex;
std::vector<int> numbers;

void protected_sum(int begin, int end){
    for (int i = begin; i <= end; i++){
        std::lock_guard<std::mutex> guard(mutex);
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
        sum1 += numbers[i];
    }
}

void unprotected_sum(int begin, int end){
    for (int i = begin; i <= end; i++){
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
        sum2 += numbers[i];
    }
}

int main(int argc, char *argv[]){
    int nThreads = 10;
    std::vector<std::thread> vThreads;
    srand(time(NULL));
    int x = LIMIT / nThreads;
    int begin = 0;
    int end = x;
```

```
35
36      for (int i = 0; i < LIMIT; i++){
37          numbers.push_back(rand());
38          normal_sum += numbers[i];
39      }
40
41      for (int i = 0; i < nThreads; i++){
42          begin = i * x;
43          end = (begin + x) - 1;
44          vThreads.push_back(std::thread(protected_sum, begin, end));
45          vThreads.push_back(std::thread(unprotected_sum, begin, end));
46      }
47      std::for_each(vThreads.begin(), vThreads.end(), std::mem_fn(&std::thread::join));
48
49      std::cout << "Final sum without threads: " << normal_sum << std::endl;
50      std::cout << "Final sum with protected threads: " << sum1 << std::endl;
51      std::cout << "Final sum without protected threads: " << sum2 << std::endl;
52      return EXIT_SUCCESS;
53  }
```

Listing 3: Mutual Exclusion code

Command Line

```
$ make MutualExclusion
g++ src/MutualExclusion.cpp -o bin/mutualexclusion -pthread -std=c++11
./bin/mutualexclusion
  Final sum without threads: 103304140414
  Final sum with protected threads: 103304140414
  Final sum without protected threads: 100927005451
```

3 Dining Philosophers

3.1 Description of the problem

Philosophers find themselves eating or thinking. They all share a table round with five chairs, one for each philosopher. In the center of the table there is a fountain of rice and there are only five chopsticks on the table, so each philosopher has a chopstick to his left and another to his right.

When a philosopher thinks, then he abstracts from the world and does not relate to other philosophers. When he is hungry, he tries to access the chopsticks he has on your left and on your right (he need both). Naturally, a philosopher cannot take a toothpick from another philosopher and can only eat when he has taken both chopsticks. When a philosopher finishes eating, he leaves the chopsticks and thinks.

3.2 Solution of the problem

A possible variation of the original problem of the dinning philosophers consists of suppose that the philosophers will eat directly from a tray located in the center of the table. For this, the philosophers have N chopsticks that are arranged at the side of the tray. Thus, when a philosopher wants to eat, then he will have to take a pair of chopsticks, eat, and then put them down again so that another philosopher can eat them. Figure 2 graphically shows the problem raised.

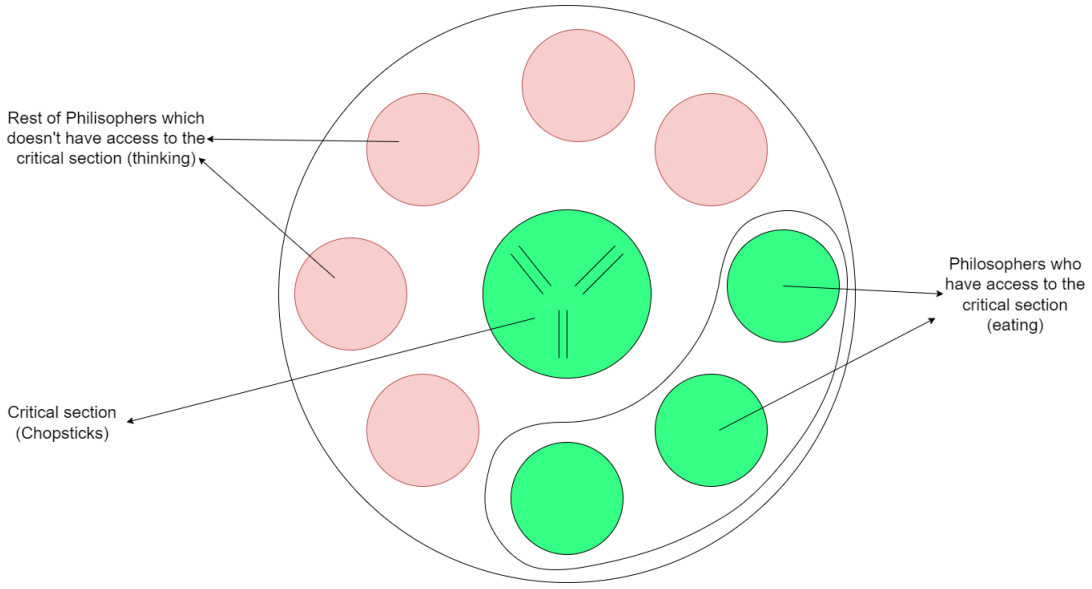

Figure 2: Dinning Philosophers schema

To solve it, you can use a counter semaphore initialized to N, which represents the pairs of chopsticks available to philosophers. This semaphore **sticks** manages access concurrent of the philosophers to the tray. To pick (lock the access) 2 contiguous chopsticks and that are in front of the philosopher, we apply the formula that appears in the lines 19-20 and 25-26 to put down the chopsticks. In each action of thinking and eating is added some time complexity to see better results.

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <thread>
5  #include <functional>
6  #include <mutex>
7  #include "SemCounter.cpp"
8
9  #define N 5
10
11 SemCounter counter(N);
12 std::mutex sticks[N];
13
14 void philosopher(int i){
15     while(1){
16         //thinking
17         counter.wait();
18         std::cout<<"Philosopher [" << i << "] is thinking "<<std::endl;
19         sticks[i].lock();
20         sticks[(i+1)%N].lock();
21         std::this_thread::sleep_for(std::chrono::milliseconds(1000));
22
23         //eating
24         std::cout<<"Philosopher [" << i << "] is eating "<< std::endl;
25         sticks[i].unlock();
26         sticks[(i+1)%N].unlock();
27         std::this_thread::sleep_for(std::chrono::milliseconds(2000));
28         counter.signal();
29     }
30 }
31
32 int main(int argc, char *argv[]){
33     std::vector<std::thread> philosophers;
34     for (int i = 0; i < 10; i++) {
35         philosophers.push_back(std::thread(philosopher, i));
36     }
37     std::for_each(philosophers.begin(), philosophers.end(),
       std::mem_fn(&std::thread::join));
38
```

```
39      return EXIT_SUCCESS;
40  }
```

Listing 4: Dining Philosophers code

Command Line

```
$ make DiningPhilosophers
g++ src/DiningPhilosophers.cpp -o bin/diningphilosophers -pthread -std=c++11
./bin/diningphilosophers
    Philosopher [0] is thinking
    Philosopher [2] is thinking
    Philosopher [1] is thinking
    Philosopher [4] is thinking
    Philosopher [3] is thinking
    Philosopher [0] is eating
    Philosopher [4] is eating
    Philosopher [3] is eating
    Philosopher [5] is thinking
    Philosopher [2] is eating
    Philosopher [6] is thinking
    Philosopher [1] is eating
    Philosopher [7] is thinking
    Philosopher [6] is eating
    Philosopher [5] is eating
    Philosopher [8] is thinking
    Philosopher [7] is eating
```

4 Producers and Consumers

4.1 Description of the problem

There is a space of limited common storage, that is, said space consists of a finite set of gaps that may or may not contain elements. On the one hand, producers will insert elements in the buffer. On the other hand, consumers will extract them.

The first issue to consider is that you have to control exclusive access to the critical section, that is, to the buffer itself. The second important issue is to control two situations:

- You cannot insert an element when the buffer is full
- An element cannot be extracted when the buffer is empty.

4.2 Solution of the problem

To control exclusive access to the critical section you can use a semaphore binary, so that the wait primitive allows such access or blocks a process you want to access the critical section. To control the two situations above mentioned, two independent semaphores can be used which will be updated when a buffer element is produced or consumed, respectively.

The proposed solution is based on the following elements:

- *mutex*, binary semaphore used to provide mutual exclusion for the access to the product buffer. This semaphore is initialized to 1.
- *empty*, counter semaphore used to control the number of empty spaces of the buffer. This semaphore is initialized to n, where n is the size of the buffer.
- *full*, counter semaphore that is used to control the number of gaps filled in the buffer. This semaphore is initialized to 0.

In essence, the *empty* and *full* semaphores ensure that no more items can be inserted when the buffer is full or extract more items when it is empty, respectively. For example, if the internal value of empty is 0, then a process that executes wait on this semaphore will be blocked until another process executes signal, that is that is, until another process produces a new item in the buffer.

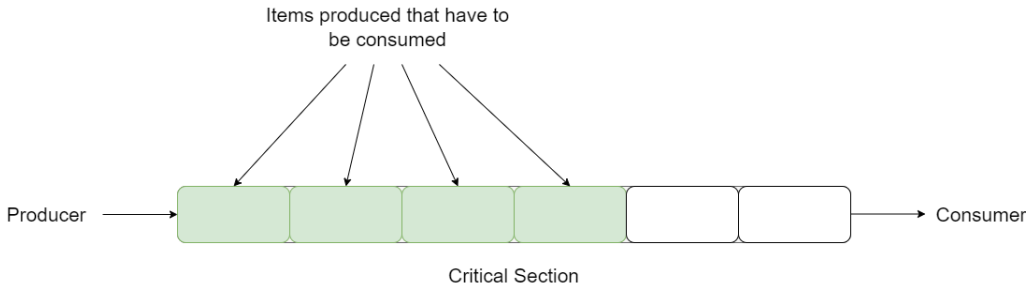

Figure 3: Producers Consumers schema

Listing 5, shows a possible solution of the buffer problem limited. Note how the producing process has to wait for some free gap before to produce a new element, using the wait(*empty*) operation. Access to the buffer It is controlled by wait(*mutex*), while release is done by signal(*mutex*). Finally, the producer indicates that there is a new element using signal(*empty*).

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <thread>
#include <functional>
#include <mutex>
#include "SemCounter.cpp"

#define N 5

SemCounter full(0);
SemCounter empty(N);
std::mutex access;
std::vector<int> buffer;
std::vector<std::thread> vThreads;

void productor(int i){
    while (1){
        empty.wait();
        access.lock();
        buffer.push_back(1);
        std::cout << "Producer ["<<i<<"]--Buffer Size: " << buffer.size() << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
        access.unlock();
        full.signal();
    }
}

void consumidor(int i){
    while (1){
        full.wait();
        access.lock();
        buffer.pop_back();
        std::cout << "Consumer ["<<i<<"]--Buffer Size: " << buffer.size() << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        access.unlock();
        empty.signal();
    }
}

int main(int argc, char *argv[]){
    for (int i = 0; i < 10; i++){
        if (i % 2 == 0){
            vThreads.push_back(std::thread(productor, i));
```

```
45          }else{
46              vThreads.push_back(std::thread(consumidor, i));
47          }
48      }
49
50      std::for_each(vThreads.begin(), vThreads.end(), std::mem_fn(&std::thread::join));
51      return EXIT_SUCCESS;
52  }
```

Listing 5: Producers and Consumers code

Command Line

```
$ make ProducersConsumers
g++ src/ProducersConsumers.cpp -o bin/producersconsumers -pthread -std=c++11
./bin/producersconsumers
    Producer [0]--Buffer Size: 1
    Consumer [1]--Buffer Size: 0
    Producer [2]--Buffer Size: 1
    Producer [4]--Buffer Size: 2
    Producer [6]--Buffer Size: 3
    Producer [8]--Buffer Size: 4
    Consumer [3]--Buffer Size: 3
    Producer [0]--Buffer Size: 4
    Consumer [5]--Buffer Size: 3
```

5 Readers and Writers

5.1 Description of the problem

Suppose a structure or a database shared by several concurrent processes. Some of these processes will simply have to read information, while that others will have to update it, that is, perform read and write operations. The first processes are called readers while the rest will be writers.

If two or more readers are trying to concurrently access the data, then no problem will be generated. However, a simultaneous access of a writer and any other process, whether reader or writer, would generate a race condition.

The problem of readers and writers has been used countless times to test different sync primitives and there are many variations on the original problem. The simplest version is based on the fact that no reader will have to wait unless there is already a writer in the critical section. That is, no reader has to wait for other readers to finish because a writing process is waiting.

5.2 Solution of the problem

A possible solution is to add a turn traffic light that governs access readers so that writers can acquire it in advance. In the Listing 6 there is possible solution.

If a writer hangs when waiting on said semaphore, then will force future readers to wait. When the last reader leaves the critical section, the next process to enter is guaranteed to be a writer.

As can be seen in the writing process, if a writer arrives while there are readers in the critical section, the first one will crash on the second statement. This implies that the turn traffic light is closed, generating a barrier that queues the rest of the readers while the writer is still waiting.

Regarding the reader, when the last one leaves and makes a signal on write_access, then the writer who was waiting is unlocked. Then the writer will enter your critical section because none of the other readers will be able to advance due to which turn it will block them.

When the writer finishes signaling about turn, then another reader or another writer will be able to keep moving forward. Thus, this solution ensures that at least one writer continue its execution, but it is possible that a reader enters while there are other writers waiting.

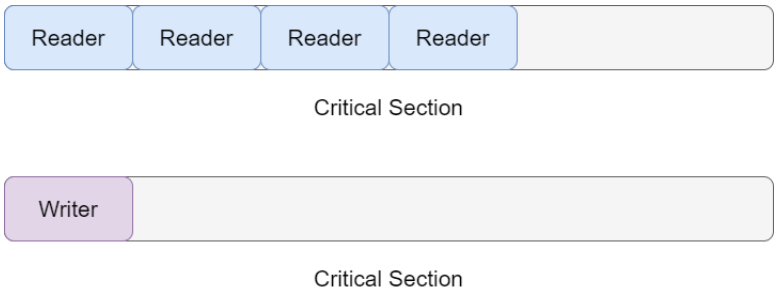

Figure 4: Reader Writers schema

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <thread>
#include <functional>
#include <mutex>
#include "SemCounter.cpp"

#define N 5

SemCounter turn(N);
SemCounter write_access(1);
std::mutex mutex;
std::vector<int> buffer;
std::vector<std::thread> vHilos;

void writer(int i){
    while (1){
        turn.wait();
        std::cout << "Writer [" << i << "] waiting...\n";
        write_access.wait();
        std::cout << "Writer [" << i << "] is editing the file.\n";
        std::this_thread::sleep_for(std::chrono::seconds(4));
        turn.signal();
        std::cout << "Writer [" << i << "] finished editing the file.\n";
        write_access.signal();
    }
}

void reader(int i){
    while (1){
        turn.wait();
        turn.signal();
        write_access.wait();
        std::cout << "Reader [" << i << "] is reading the file.\n";
        std::this_thread::sleep_for(std::chrono::seconds(2));
        write_access.signal();
    }
}

int main(int argc, char *argv[]){
    for (int i = 0; i < 4; i++){
        vHilos.push_back(std::thread(reader, i));
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }
    for (int i = 0; i < 1; i++){
        vHilos.push_back(std::thread(writer, i));
    }
    std::for_each(vHilos.begin(), vHilos.end(), std::mem_fn(&std::thread::join));

    return EXIT_SUCCESS;
    }
}
```

Listing 6: Readers and Writers code

Command Line

```
$ make ReadersWriters
g++ src/ReadersWriters.cpp -o bin/readerswriters -pthread -std=c++11
./bin/readerswriters
    Reader [0] is reading the file.
    Writer [0] waiting...
    Reader [1] is reading the file.
    Reader [2] is reading the file.
    Reader [3] is reading the file.
    Writer [0] is editing the file.
    Writer [0] finished editing the file.
    Writer [0] waiting...
    Reader [0] is reading the file.
    Reader [1] is reading the file.
    Reader [2] is reading the file.
    Reader [3] is reading the file.
    Writer [0] is editing the file.
    Writer [0] finished editing the file.
    Reader [0] is reading the file.
```