

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Course Work**

**Speciality:Computer Engineering**

**Faculty:Aerokosmic**

**Group: 1459i**

**Course:3**

**Subject: System programming and operating systems**

**Teacher: Sahmaliyev Mehemmed**

**Student: Karimova Zakhra**

## Topic: Multi threaded programming c++

**Baku 2022**

**Contents**

Introduction...............................................................................................1

## §1.Wha is Thread?.....................................................................................2

§2.Thread Returning a Value…….............................................................6

## §3. Communication Between Threads……………………………….…..7

## §4.The thread\_local Specifier…................................................................8

## §5.Sequences, Synchronous, Asynchronous, Parallel, Concurrent, Order

## Atomic Operations.....................................................................................9

## §6.Blocking a Thread……………….......................................................11

## §6.Mutex Types........................................................................................13

## §7.Conclusion...........................................................................................20

§8.**Literature**...........................................................................................20

**Introduction**

**A process is a program that is running on the computer. In modern computers, many processes run at the same time. A program can be broken down into sub-processes for the sub-processes to run at the same time. These sub-processes are called threads. Threads must run as parts of one program.**

**Some programs require more than one input simultaneously. Such a program needs threads. If threads run in parallel, then the overall speed of the program is increased. Threads also share data among themselves. This data sharing leads to conflicts on which result is valid and when the result is valid. This conflict is a data race and can be resolved.**

**Since threads have similarities to processes, a program of threads is compiled by the g++ compiler as follows:**

**g++ -std=c++17 temp.cc -lpthread -o temp**

**Where temp. cc is the source code file, and the temp is the executable file.**

**A program that uses threads, is begun as follows:**

**#include <iostream>**

**#include <thread>**

**Using namespace std;**

**Note the use of “#include <thread>”.**

## What is Thread?

The flow of control of a program can be single or multiple. When it is single, it is a thread of execution or simply, thread. A simple program is one thread. This thread has the main() function as its top-level function. This thread can be called the main thread. In simple terms, a thread is a top-level function, with possible calls to other functions.

Any function defined in the global scope is a top-level function. A program has the main() function and can have other top-level functions. Each of these top-level functions can be made into a thread by encapsulating it into a thread object. A thread object is a code that turns a function into a thread and manages the thread. A thread object is instantiated from the thread class.

So, to create a thread, a top-level function should already exist. This function is the effective thread. Then a thread object is instantiated. The ID of the thread object without the encapsulated function is different from the ID of the thread object with the encapsulated function. The ID is also an instantiated object, though its string value can be obtained.

If a second thread is needed beyond the main thread, a top-level function should be defined. If a third thread is needed, another top-level function should be defined for that, and so on.

## Creating a Thread

The main thread is already there, and it does not have to be recreated. To create another thread, its top-level function should already exist. If the top-level function does not already exist, it should be defined. A thread object is then instantiated, with or without the function. The function is the effective thread (or the effective thread of execution). The following code creates a thread object with a thread (with a function):

*#include <iostream>*  
*#include <thread>*  
using namespace std;  
  
void thrdFn() {  
        cout << "seen" << '\n';  
    }    
  
int main()  
{  
    thread thr(&thrdFn);  
  
    return 0;  
}

The name of the thread is thr, instantiated from the thread class, thread. Remember: to compile and run a thread, use a command similar to the one given above.

The constructor function of the thread class takes a reference to the function as an argument.

This program now has two threads: the main thread and the thr object thread. The output of this program should be “seen” from the thread function. This program as it is has no syntax error; it is well-typed. This program, as it is, compiles successfully. However, if this program is run, the thread (function, thrdFn) may not display any output; an error message might be displayed. This is because the thread, thrdFn() and the main() thread, have not been made to work together. In C++, all threads should be made to work together, using the join() method of the thread – see below.

## Thread Object Members

The important members of the thread class are the “join()”, “detach()” and “id get\_id()” functions;

**void join()**  
If the above program did not produce any output, the two threads were not forced to work together. In the following program, an output is produced because the two threads have been forced to work together:

*#include <iostream>*  
*#include <thread>*  
using namespace std;  
  
void thrdFn() {  
        cout << "seen" << '\n';  
    }    
  
int main()  
{  
    thread thr(&thrdFn);  
  
    return 0;  
}

Now, there is an output, “seen” without any run-time error message. As soon as a thread object is created, with the encapsulation of the function, the thread starts running; i.e., the function starts executing. The join() statement of the new thread object in the main() thread tells the main thread (main() function) to wait until the new thread (function) has completed its execution (running). The main thread will halt and will not execute its statements below the join() statement until the second thread has finished running. The result of the second thread is correct after the second thread has completed its execution.

If a thread is not joined, it continues to run independently and may even end after the main() thread has ended. In that case, the thread is not really of any use.

The following program illustrates the coding of a thread whose function receives arguments:

*#include <iostream>*  
*#include <thread>*  
using namespace std;  
  
void thrdFn(char str1[], char str2[]) {  
        cout << str1 << str2 << '\n';  
    }    
  
int main()  
{  
    char st1[] = "I have ";  
    char st2[] = "seen it.";  
  
    thread thr(&thrdFn, st1, st2);  
    thr.join();  
  
    return 0;  
}

The output is:

“I have ” seen it.”

Without the double-quotes. The function arguments have just been added (in order), after the reference to the function, in the parentheses of the thread object constructor.

## Returning from a Thread

The effective thread is a function that runs concurrently with the main() function. The return value of the thread (encapsulated function) is not done ordinarily. “How to return value from a thread in C++” is explained below.

Note: It is not only the main() function that can call another thread. A second thread can also call the third thread.

**void detach()**  
After a thread has been joined, it can be detached. Detaching means separating the thread from the thread (main) it was attached to. When a thread is detached from its calling thread, the calling thread no longer waits for it to complete its execution. The thread continues to run on its own and may even end after the calling thread (main) has ended. In that case, the thread is not really of any use. A calling thread should join a called thread for both of them to be of use. Note that joining halts the calling thread from executing until the called thread has completed its own execution. The following program shows how to detach a thread:

*#include <iostream>*  
*#include <thread>*  
using namespace std;  
  
void thrdFn(char str1[], char str2[]) {  
        cout << str1 << str2 << '\n';  
    }    
  
int main()  
{  
    char st1[] = "I have ";  
    char st2[] = "seen it.";  
  
    thread thr(&thrdFn, st1, st2);  
    thr.join();  
    thr.detach();  
  
    return 0;  
}

Note the statement, “thr.detach();”. This program, as it is, will compile very well. However, when running the program, an error message may be issued. When the thread is detached, it is on its own and may complete its execution after the calling thread has completed its execution

**id get\_id()**  
id is a class in the thread class. The member function, get\_id(), returns an object, which is the ID object of the executing thread. The text for the ID can still be gotten from the id object – see later. The following code shows how to obtain the id object of the executing thread:

*#include <iostream>*  
*#include <thread>*  
using namespace std;  
  
void thrdFn() {  
        cout << "seen" << '\n';  
    }    
  
int main()  
{  
    thread thr(&thrdFn);  
    thread::id iD = thr.get\_id();    
    thr.join();  
  
    return 0;  
}

## Thread Returning a Value

The effective thread is a function. A function can return a value. So a thread should be able to return a value. However, as a rule, the thread in C++ does not return a value. This can be worked around using the C++ class, Future in the standard library, and the C++ async() function in the Future library. A top-level function for the thread is still used but without the direct thread object. The following code illustrates this:

*#include <iostream>*  
*#include <thread>*  
*#include <future>*  
using namespace std;  
  
future output;  
  
char\* thrdFn(char\* str) {  
        return str;  
    }    
  
int main()  
{  
    char st[] = "I have seen it.";  
  
    output = async(thrdFn, st);  
    char\* ret = output.get();   *//waits for thrdFn() to provide result*  
    cout<<ret<<'\n';  
  
    return 0;  
}

The output is

“I have seen it.”

Note the inclusion of the future library for the future class. The program begins with the instantiation of the future class for the object, output, of specialization . The async() function is a C++ function in the std namespace in the future library. The first argument to the function is the name of the function that would have been a thread function. The rest of the arguments for the async() function are arguments for the supposed thread function.

The calling function (main thread) waits for the executing function in the above code until it provides the result. It does this with the statement:

char\* ret = output.get();

This statement uses the get() member function of the future object. The expression “output.get()” halts the execution of the calling function (main() thread) until the supposed thread function completes its execution. If this statement is absent, the main() function may return before async() finishes the execution of the supposed thread function. The get() member function of the future returns the returned value of the supposed thread function. In this way, a thread has indirectly returned a value. There is no join() statement in the program.

## Communication Between Threads

The simplest way for threads to communicate is to be accessing the same global variables, which are the different arguments to their different thread functions. The following program illustrates this. The main thread of the main() function is assumed to be thread-0. It is thread-1, and there is thread-2. Thread-0 calls thread-1 and joins it. Thread-1 calls thread-2 and joins it.

The simplest way for threads to communicate is to be accessing the same global variables, which are the different arguments to their different thread functions. The following program illustrates this. The main thread of the main() function is assumed to be thread-0. It is thread-1, and there is thread-2. Thread-0 calls thread-1 and joins it. Thread-1 calls thread-2 and joins it.

*#include <iostream>*  
*#include <thread>*  
*#include <string>*  
using namespace std;  
  
string global1 = string("I have ");  
string global2 = string("seen it.");  
  
void thrdFn2(string str2) {  
        string globl = global1 + str2;  
        cout << globl << endl;  
    }  
  
void thrdFn1(string str1) {  
        global1 = "Yes, " + str1;  
  
        thread thr2(&thrdFn2, global2);    
        thr2.join();  
    }    
  
int main()  
{  
    thread thr1(&thrdFn1, global1);    
    thr1.join();  
  
    return 0;  
}

The output is:

“Yes, I have seen it.”  
Note that the string class has been used this time, instead of the array-of-characters, for convenience. Note that thrdFn2() has been defined before thrdFn1() in the overall code; otherwise thrdFn2() would not be seen in thrdFn1(). Thread-1 modified global1 before Thread-2 used it. That is communication.

More communication can be got with the use of condition\_variable or Future – see below.

## The thread\_local Specifier

A global variable must not necessarily be passed to a thread as an argument of the thread. Any thread body can see a global variable. However, it is possible to make a global variable have different instances in different threads. In this way, each thread can modify the original value of the global variable to its own different value. This is done with the use of the thread\_local specifier as in the following program:

*#include <iostream>*  
*#include <thread>*  
using namespace std;  
  
thread\_local int inte = 0;  
  
void thrdFn2() {  
    inte = inte + 2;  
    cout << inte << " of 2nd thread\n";  
}  
  
void thrdFn1() {  
    thread thr2(&thrdFn2);  
    inte = inte + 1;  
    cout << inte << " of 1st thread\n";  
  
    thr2.join();  
}    
  
int main()  
{  
    thread thr1(&thrdFn1);    
    cout << inte << " of 0th thread\n";  
    thr1.join();  
  
    return 0;  
}

The output is:

0, of 0th thread  
1, of 1st thread  
2, of 2nd thread

## Sequences, Synchronous, Asynchronous, Parallel, Concurrent, Order

## Atomic Operations

Atomic operations are like unit operations. Three important atomic operations are store(), load() and the read-modify-write operation. The store() operation can store an integer value, for example, into the microprocessor accumulator (a kind of memory location in the microprocessor). The load() operation can read an integer value, for example, from the accumulator, into the program.

## Sequences

An atomic operation consists of one or more actions. These actions are sequences. A bigger operation can be made up of more than one atomic operation (more sequences). The verb “sequence ” can mean whether an operation is placed before another operation.

## Synchronous

Operations operating one after the other, consistently in one thread, are said to operate synchronously. Suppose two or more threads are operating concurrently without interfering with one another, and no thread has an asynchronous callback function scheme. In that case, the threads are said to be operating synchronously.

If one operation operates on an object and ends as expected, then another operation operates on that same object; the two operations will be said to have operated synchronously, as neither interfered with the other on the use of the object.

## Asynchronous

Assume that there are three operations, called operation1, operation2, and operation3, in one thread. Assume that the expected order of working is: operation1, operation2, and operation3. If working takes place as expected, that is a synchronous operation. However, if, for some special reason, the operation goes as operation1, operation3, and operation2, then it would now be asynchronous. Asynchronous behavior is when the order is not the normal flow.

Also, if two threads are operating, and along the way, one has to wait for the other to complete before it continues to its own completion, then that is asynchronous behavior.

## Parallel

Assume that there are two threads. Assume that if they are to run one after the other, they will take two minutes, one minute per thread. With parallel execution, the two threads will run simultaneously, and the total execution time would be one minute. This needs a dual-core microprocessor. With three threads, a three-core microprocessor would be needed, and so on.

If asynchronous code segments operate in parallel with synchronous code segments, there would be an increase in speed for the whole program. Note: the asynchronous segments can still be coded as different threads.

## Concurrent

With concurrent execution, the above two threads will still run separately. However, this time they will take two minutes (for the same processor speed, everything equal). There is a single-core microprocessor here. There will be interleaved between the threads. A segment of the first thread will run, then a segment of the second thread runs, then a segment of the first thread runs, then a segment of the second, and so on.

In practice, in many situations, parallel execution does some interleaving for the threads to communicate.

## Order

For the actions of an atomic operation to be successful, there must be an order for the actions to achieve synchronous operation. For a set of operations to work successfully, there must be an order for the operations for synchronous execution.

## Blocking a Thread

By employing the join() function, the calling thread waits for the called thread to complete its execution before it continues its own execution. That wait is blocking.

## Locking

A code segment (critical section) of a thread of execution can be locked just before it starts and unlocked after it ends. When that segment is locked, only that segment can use the computer resources it needs; no other running thread can use those resources. An example of such a resource is the memory location of a global variable. Different threads can access a global variable. Locking allows only one thread, a segment of it, that has been locked to access the variable when that segment is running.

## Mutex

Mutex stands for Mutual Exclusion. A mutex is an instantiated object that enables the programmer to lock and unlock a critical code section of a thread. There is a mutex library in the C++ standard library. It has the classes: mutex and timed\_mutex – see details below.

A mutex owns its lock.

## Timeout in C++

An action can be made to occur after a duration or at a particular point in time. To achieve this, “Chrono” has to be included, with the directive, “#include <chrono>”.

**duration**  
duration is the class-name for duration, in the namespace chrono, which is in namespace std. Duration objects can be created as follows:

chrono::hours hrs(2);  
chrono::minutes mins(2);  
chrono::seconds secs(2);  
chrono::milliseconds msecs(2);  
chrono::microseconds micsecs(2);

Here, there are 2 hours with the name, hrs; 2 minutes with the name, mins; 2 seconds with the name, secs; 2 milliseconds with the name, msecs; and 2 microseconds with the name, micsecs.

1 millisecond = 1/1000 seconds. 1 microsecond = 1/1000000 seconds.

## Lockable Requirements

Let m be the instantiated object of the class, mutex.

## BasicLockable Requirements

**m.lock()**  
This expression blocks the thread (current thread) when it is typed until a lock is acquired. Until the next code segment is the only segment in control of the computer resources that it needs (for data access). If a lock cannot be acquired, an exception (error message) would be thrown.

**m.unlock()**  
This expression unlocks the lock from the previous segment, and the resources can now be used by any thread or by more than one thread (which unfortunately may conflict with each other). The following program illustrates the use of m.lock() and m.unlock(), where m is the mutex object.

*#include <iostream>*  
*#include <thread>*  
*#include <mutex>*  
using namespace std;  
  
int globl = 5;  
mutex m;  
  
void thrdFn() {  
    *//some statements*  
    m.lock();  
        globl = globl + 2;  
        cout << globl << endl;  
    m.unlock();  
}  
  
int main()  
{  
    thread thr(&thrdFn);  
    thr.join();  
  
    return 0;  
}

The output is 7. There are two threads here: the main() thread and the thread for thrdFn(). Note that the mutex library has been included. The expression to instantiate the mutex is “mutex m;”. Because of the use of lock() and unlock(), the code segment,

globl = globl + 2;  
cout << globl << endl;

Which must not necessarily be indented, is the only code that has access to the memory location (resource), identified by globl, and the computer screen (resource) represented by cout, at the time of execution.

**m.try\_lock()**  
This is the same as m.lock() but does not block the current execution agent. It goes straight ahead and attempts a lock. If it cannot lock, probably because another thread has already locked the resources, it throws an exception.

It returns a bool: true if the lock was acquired and false if the lock was not acquired.

“m.try\_lock()” must be unlocked with “m.unlock()”, after the appropriate code segment.

## Mutex Types

Mutex types are: mutex, recursive\_mutex, shared\_mutex, timed\_mutex, recursive\_timed\_-mutex, and shared\_timed\_mutex. The recursive mutexes shall not be addressed in this article.

Note: a thread owns a mutex from the time the call to lock is made until unlock.

**mutex**  
Important member functions for the ordinary mutex type (class) are: mutex() for mutex object construction, “void lock()”, “bool try\_lock()” and “void unlock()”. These functions have been explained above.

**shared\_mutex**  
With shared mutex, more than one thread can share access to the computer resources. So, by the time the threads with shared mutexes have completed their execution, while they were at lock-down, they were all manipulating the same set of resources (all accessing the value of a global variable, for example).

Important member functions for the shared\_mutex type are: shared\_mutex() for construction, “void lock\_shared()”, “bool try\_lock\_shared()” and “void unlock\_shared()”.

lock\_shared() blocks the calling thread (thread it is typed in) till the lock for the resources is acquired. The calling thread may be the first thread to acquire the lock, or it may join other threads that have already acquired the lock. If the lock cannot be acquired, because for example, too many threads are already sharing the resources, then an exception would be thrown.

try\_lock\_shared() is the same as lock\_shared(), but does not block.

unlock\_shared() is not really the same as unlock(). unlock\_shared() unlocks shared mutex. After one thread share-unlocks itself, other threads may still hold a shared lock on the mutex from the shared mutex.

**timed\_mutex**  
Important member functions for the timed\_mutex type are: “timed\_mutex()” for construction, “void lock()”, “bool try\_lock()”, “bool try\_lock\_for(rel\_time)”, “bool try\_lock\_until(abs\_time)”, and “void unlock()”. These functions have been explained above, though try\_lock\_for() and try\_lock\_until() still need more explanation – see later.

**shared\_timed\_mutex**  
With shared\_timed\_mutex, more than one thread can share access to the computer resources, depending on time (duration or time\_point). So, by the time the threads with shared timed mutexes have completed their execution, while they were at lock-down, they were all manipulating the resources (all accessing the value of a global variable, for example).

Important member functions for the shared\_timed\_mutex type are: shared\_timed\_mutex() for construction, “bool try\_lock\_shared\_for(rel\_time);”, “bool try\_lock\_shared\_until(abs\_time)” and “void unlock\_shared()”.

“bool try\_lock\_shared\_for()” takes the argument, rel\_time (for relative time). “bool try\_lock\_shared\_until()” takes the argument, abs\_time (for absolute time). If the lock cannot be acquired, because for example, too many threads are already sharing the resources, then an exception would be thrown.

unlock\_shared() is not really the same as unlock(). unlock\_shared() unlocks shared\_mutex or shared\_timed\_mutex. After one thread share-unlocks itself from the shared\_timed\_mutex, other threads may still hold a shared lock on the mutex.

## Data Race

Data Race is a situation where more than one thread access the same memory location simultaneously, and at least one writes. This is clearly a conflict.

A data race is minimized (solved) by blocking or locking, as illustrated above. It can also be handled using, Call Once – see below. These three features are in the mutex library. These are the fundamental ways of a handling data race. There are other more advanced ways, which bring in more convenience – see below.

## Locks

A lock is an object (instantiated). It is like a wrapper over a mutex. With locks, there is automatic (coded) unlocking when the lock goes out of scope. That is, with a lock, there is no need to unlock it. The unlocking is done as the lock goes out of scope. A lock needs a mutex to operate. It is more convenient to use a lock than to use a mutex. C++ locks are: lock\_guard, scoped\_lock, unique\_lock, shared\_lock.

**lock\_guard**  
The following code shows how a lock\_guard is used:

*#include <iostream>*  
*#include <thread>*  
*#include <mutex>*  
using namespace std;  
  
int globl = 5;  
mutex m;  
  
void thrdFn() {  
    *//some statements*  
    lock\_guard<mutex> lck(m);  
        globl = globl + 2;  
        cout << globl << endl;  
    *//statements*  
}  
  
int main()  
{  
    thread thr(&thrdFn);  
    thr.join();  
  
    return 0;  
}

The output is 7. The type (class) is lock\_guard in the mutex library. In constructing its lock object, it takes the template argument, mutex. In the code, the name of the lock\_guard instantiated object is lck. It needs an actual mutex object for its construction (m). Notice that there is no statement to unlock the lock in the program. This lock died (unlocked) as it went out of the scope of the thrdFn() function.

**unique\_lock**  
Only its current thread can be active when any lock is on, in the interval, while the lock is on. The main difference between unique\_lock and lock\_guard is that ownership of the mutex by a unique\_lock, can be transferred to another unique\_lock. unique\_lock has more member functions than lock\_guard.

Important functions of unique\_lock are: “void lock()”, “bool try\_lock()”, “template <class Rep, class Period>bool try\_lock\_for(const chrono::duration <Rep, Period>& rel\_time)”, and “template <class Clock, class Duration>bool try\_lock\_until(const chrono::time\_point <Clock, Duration>& abs\_time)” .

Note that the return type for try\_lock\_for() and try\_lock\_until() is not bool here – see later. The basic forms of these functions have been explained above.

Ownership of a mutex can be transferred from unique\_lock1 to unique\_lock2 by first releasing it off unique\_lock1, and then allowing unique\_lock2 to be constructed with it. unique\_lock has an unlock() function for this releasing. In the following program, ownership is transferred in this way:

*#include <iostream>*  
*#include <thread>*  
*#include <mutex>*  
using namespace std;  
  
mutex m;  
  
int globl = 5;  
  
void thrdFn2() {  
    unique\_lock<mutex> lck2(m);  
        globl = globl + 2;  
        cout << globl << endl;  
    }  
  
void thrdFn1() {  
    unique\_lock<mutex> lck1(m);  
        globl = globl + 2;  
        cout << globl << endl;  
  
        lck1.unlock();  
        thread thr2(&thrdFn2);  
        thr2.join();    
    }    
  
int main()  
{  
    thread thr1(&thrdFn1);    
    thr1.join();  
  
    return 0;  
}

The output is:

7  
9

The mutex of unique\_lock, lck1 was transferred to unique\_lock, lck2. The unlock() member function for unique\_lock does not destroy the mutex.

**shared\_lock**  
More than one shared\_lock object (instantiated) can share the same mutex. This mutex shared has to be shared\_mutex. The shared mutex can be transferred to another shared\_lock, in the same way, that the mutex of a unique\_lock can be transferred to another unique\_lock, with the help of the unlock() or release() member function.

Important functions of shared\_lock are: "void lock()", "bool try\_lock()", "template<class Rep, class Period>bool try\_lock\_for(const chrono::duration<Rep, Period>& rel\_time)", "template<class Clock, class Duration>bool try\_lock\_until(const chrono::time\_point<Clock, Duration>& abs\_time)", and "void unlock()". These functions are the same as those for unique\_lock.

## Call Once

A thread is an encapsulated function. So, the same thread can be for different thread objects (for some reason). Should this same function, but in different threads, not be called once, independent of the concurrency nature of threading? – It should. Imagine that there is a function that has to increment a global variable of 10 by 5. If this function is called once, the result would be 15 – fine. If it is called twice, the result would be 20 – not fine. If it is called three times, the result would be 25 – still not fine. The following program illustrates the use of the “call once” feature:

*#include <iostream>*  
*#include <thread>*  
*#include <mutex>*  
using namespace std;  
  
auto globl = 10;  
  
once\_flag flag1;  
  
void thrdFn(int no) {  
    call\_once(flag1, [no]() {  
        globl =  globl + no;});  
    }  
  
int main()  
{  
    thread thr1(&thrdFn, 5);    
    thread thr2(&thrdFn, 6);    
    thread thr3(&thrdFn, 7);    
    thr1.join();  
    thr2.join();  
    thr3.join();  
  
    cout << globl << endl;  
  
    return 0;  
}

The output is 15, confirming that the function, thrdFn(), was called once. That is, the first thread was executed, and the following two threads in main()were not executed. “void call\_once()” is a predefined function in the mutex library. It is called the function of interest (thrdFn), which would be the function of the different threads. Its first argument is a flag – see later. In this program, its second argument is a void lambda function. In effect, the lambda function has been called once, not really the thrdFn() function. It is the lambda function in this program that really increments the global variable

## Condition Variable

When a thread is running, and it halts, that is blocking. When the critical section of the thread “holds” the computer resources, such that no other thread would use the resources, except itself, that is locking.

Blocking and its accompanied locking is the main way to solve the data race between threads. However, that is not good enough. What if critical sections of different threads, where no thread calls any other thread, want the resources simultaneously? That would introduce a data race! Blocking with its accompanied locking as described above is good when one thread calls another thread, and the thread called, calls another thread, called thread calls another, and so on. This provides synchronization between the threads in that the critical section of one thread uses the resources to its satisfaction. The critical section of the called thread uses the resources to its own satisfaction, then the next to its satisfaction, and so on. If the threads were to run in parallel (or concurrently), there would be a data race between the critical sections.

Call Once handles this problem by executing only one of the threads, assuming that the threads are similar in content. In many situations, the threads are not similar in content, and so some other strategy is needed. Some other strategy is needed for synchronization. Condition Variable can be used, but it is primitive. However, it has the advantage that the programmer has more flexibility, similar to how the programmer has more flexibility in coding with mutexes over locks.

A condition variable is a class with member functions. It is its instantiated object that is used. A condition variable allows the programmer to program a thread (function). It would block itself until a condition is met before it locks onto the resources and uses them alone. This avoids data race between locks.

Condition variable has two important member functions, which are wait() and notify\_one(). wait() takes arguments. Imagine two threads: wait() is in the thread that intentionally blocks itself by waiting until a condition is met. notify\_one() is in the other thread, which must signal the waiting thread, through the condition variable, that the condition has been met.

The waiting thread must have unique\_lock. The notifying thread can have lock\_guard. The wait() function statement should be coded just after the locking statement in the waiting thread. All locks in this thread synchronization scheme use the same mutex.

The following program illustrates the use of the condition variable, with two threads:

*#include <iostream>*  
*#include <thread>*  
*#include <condition\_variable>*  
using namespace std;  
  
mutex m;  
condition\_variable cv;  
  
bool dataReady = false;  
  
void waitingForWork(){  
    cout << "Waiting" << '\n';  
    unique\_lock<std::mutex> lck1(m);  
    cv.wait(lck1, []{ return dataReady; });    
    cout << "Running" << '\n';  
}  
  
void setDataReady(){  
  
        lock\_guard<mutex> lck2(m);  
        dataReady = true;  
  
    cout << "Data prepared" << '\n';  
    cv.notify\_one();                          
}  
  
int main(){  
    cout << '\n';  
  
    thread thr1(waitingForWork);  
    thread thr2(setDataReady);  
  
    thr1.join();  
    thr2.join();  
   
    cout << '\n';  
  
  return 0;  
   
}

The output is:

Waiting  
Data prepared  
Running

## Conclusion

A thread (thread of execution) is a single flow of control in a program. More than one thread can be in a program, to run concurrently or in parallel. In C++, a thread object has to be instantiated from the thread class to have a thread.

Data Race is a situation where more than one thread is trying to access the same memory location simultaneously, and at least one is writing. This is clearly a conflict. The fundamental way to resolve the data race for threads is to block the calling thread while waiting for the resources. When it could get the resources, it locks them so that it alone and no other thread would use the resources while it needs them. It must release the lock after using the resources so that some other thread can lock onto the resources.

Literature:

https://linuxhint.com/multi-thread-and-data-race-basics-in-cpp/#1