Condition variables allow us to synchronize threads via notifications. So, you can implement workflows like sender/receiver or producer/consumer. In such a workflow, the receiver is waiting for the the sender's notification. If the receiver gets the notification, it continues its work.

std::condition\_variable

The condition variable can fulfil the roles of a sender or a receiver. As a sender, it can notify one or more receivers.

That's all you have to know to use condition variables.

// conditionVariable.cpp

#include <iostream>

#include <condition\_variable>

#include <mutex>

#include <thread>

std::mutex mutex\_;

std::condition\_variable condVar;

void doTheWork(){

std::cout << "Processing shared data." << std::endl;

}

void waitingForWork(){

std::cout << "Worker: Waiting for work." << std::endl;

std::unique\_lock<std::mutex> lck(mutex\_);

condVar.wait(lck);

doTheWork();

std::cout << "Work done." << std::endl;

}

void setDataReady(){

std::cout << "Sender: Data is ready." << std::endl;

condVar.notify\_one();

}

int main(){

std::cout << std::endl;

std::thread t1(waitingForWork);

std::thread t2(setDataReady);

t1.join();

t2.join();

std::cout << std::endl;

}

The program has two child threads: t1 and t2. They get their callable payload (functions or functors) waitingForWork and setDataReady in lines 33 and 34. The function setDataReady notifies  - using the condition variable condVar - that it is done with the preparation of the work: condVar.notify\_one(). While holding the lock, thread t2 is waiting for its notification: condVar.wait(lck). The waiting thread always performs same steps. It wakes up, tries to get the lock, checks if he's holding the lock, if the notifications arrived and, in case of failure,  puts himself  back to sleep. In case of success, the thread leaves the endless loop and continues with its work.

The output of the program is not so thrilling. That was my first impression. But wait.

## Spurious wakeup

The devil is in the details. In fact, it can happen, that the receiver finished its task before the sender has sent its notification. How is that possible? The receiver is susceptible for spurious wakeups. So the receiver wakes up, although no notification happens. To protect it from this,  I had to add a predicate to the wait method. That's exactly what I had done in the next example

// conditionVariableFixed.cpp

#include <iostream>

#include <condition\_variable>

#include <mutex>

#include <thread>

std::mutex mutex\_;

std::condition\_variable condVar;

bool dataReady;

void doTheWork(){

std::cout << "Processing shared data." << std::endl;

}

void waitingForWork(){

std::cout << "Worker: Waiting for work." << std::endl;

std::unique\_lock<std::mutex> lck(mutex\_);

condVar.wait(lck,[]{return dataReady;});

doTheWork();

std::cout << "Work done." << std::endl;

}

void setDataReady(){

std::lock\_guard<std::mutex> lck(mutex\_);

dataReady=true;

std::cout << "Sender: Data is ready." << std::endl;

condVar.notify\_one();

}

int main(){

std::cout << std::endl;

std::thread t1(waitingForWork);

std::thread t2(setDataReady);

t1.join();

t2.join();

std::cout << std::endl;

}

The key difference from the first example conditionVariable.cpp is  the boolean dataReady  used in line 11 as an additional condition. dataReady is set to true in line 28. It is checked in the function waitingForWork: condVar.waint(lck,[]return dataReady;}). That's why wait() method has an additional overload which accepts a predicate. A predicate is a callable returning true or false. In this example the callable is a lambda function. So, the condition variable checks two conditions: if the predicate is true or if the notification happened.

A short remark -  dataReady. dataReady is a shared variable, which will be changed. So I had to protect it with a lock. Because thread t1 sets and releases the lock just once, std::lock\_guard is fine for that job. That will not hold for thread t2. The wait method will continuously lock and unlock the mutex. So I need the more powerful lock: std::unique\_lock.

But that's not all. Condition variables have a lot of challenges. They must be protected by locks and are susceptible for spurious wakeups. Most use cases are easier to solve with tasks. More about tasks in the next post.

## Lost wakeup

The problem is, if the sender sends its notification before the receiver gets to a wait state, notification gets lost. The C++ standard describes condition variables as synchronization mechanism at the same time: "The condition\_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, ...". So the notfication gets lost and the receiver is waiting and waiting and ... .

How can this issue be solved? The predicate which got rid of spurious wakeups will also help with lost wakeups. In case the predicate is true, the receiver is able to continue its work independently of the notification of the sender. The variable dataReady is like a memory. Because as far as the variable data in line 28 is set to true, the receiver assumes in line 21, that the notification was delivered.