**Condition Variables**

Sometimes, tasks performed by different threads have to wait for each other. Thus, you sometimes have to synchronize concurrent operations for other reasons than to access the same data.

Condition variables, which can be used to synchronize logical dependencies in data flow between threads multiple times.

Futures is already present for such a mechanism. Futures allow to block until data by another thread is provided or another thread is done.

**Limitation of futures**

A future can pass data from one thread to another only once.

In fact, a future’s major purpose is to deal with return values or exceptions of threads.

# Purpose of Condition Variables

**Limitation with polling**

To let one thread wait for another we introduced a naive approach by using something like a ready flag. This usually means that the waiting thread polls to notice that its required data or precondition has arrived. However, such a polling for a fulfilled condition is usually not a good solution.

**Williams: C++ Conc points out:**

The waiting thread consumes valuable processing time repeatedly checking the flag and when it locks the mutex the thread setting the ready flag is blocked. ... In addition, it’s hard to get the sleep period right: too short a sleep in between checks and the thread still wastes processing time checking, too long a sleep and the thread will carry on sleeping even when the task it is waiting for is complete, introducing a delay.

## Condition Variable

C++ standard library provides condition variable. A condition variable is a variable by which a thread can wake up one or multiple other waiting threads.

**A condition variable works as follows:**

1. You have to include both <mutex> and <condition\_variable> to declare a mutex and a condition variable:

#include <mutex>

#include <condition\_variable>

std::mutex readyMutex;

std::condition\_variable readyCondVar;

1. The thread (or one of multiple threads) that signals the fulfillment of a condition has to call

readyCondVar.notify\_one(); // notify one of the waiting threads

or

readyCondVar.notify\_all(); // notify all the waiting threads

1. Any thread that waits for the condition has to call

std::unique\_lock<std::mutex> l(readyMutex);

readyCondVar.wait(l);

## Note

To wait for the condition variable, **you need a mutex and a unique\_lock**. A lock\_guard is not enough, because the waiting function might lock and unlock the mutex.

## Spurious wakeups

In addition, condition variables in general might have so-called spurious wakeups. That is, a wait on a condition variable may return even if the condition variable has not been notified.

Thus, a wakeup does not necessarily mean that the required condition now holds. Rather, after a wakeup you still need some code to verify that the condition in fact has arrived. Therefore, for example, we have to check whether provided data is really available, or we still need something like a ready flag. To set and query this provided data or this ready flag, we can use the same mutex.

# Example for Condition Variables

The following code is a complete example that demonstrates how to use condition variables:

#include <condition\_variable>

#include <mutex>

#include <future>

#include <iostream>

bool readyFlag;

std::mutex readyMutex;

std::condition\_variable readyCondVar;

void thread1() {

// do something thread2 needs as preparation

std::cout << "<return>" << std::endl;

std::cin.get();

// signal that thread1 has prepared a condition

{

std::lock\_guard<std::mutex> lg(readyMutex);

readyFlag = true;

} // release lock

readyCondVar.notify\_one();

// notification itself does not have to be inside the protected area

}

void thread2() {

// wait until thread1 is ready (readyFlag is true)

{

std::unique\_lock<std::mutex> ul(readyMutex);

readyCondVar.wait(ul, []{ return readyFlag; });

} // release lock

// do whatever shall happen after thread1 has prepared things

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

}

int main() {

auto f1 = std::async(std::launch::async, thread1);

auto f2 = std::async(std::launch::async, thread2);

return 0;

}

Output:

<return>

Hello

done

The code in the thread 2 has same effect as the following code, where the loop necessary for dealing with spurious wakeups is explicitly visible:

{

std::unique\_lock<std::mutex> ul(readyMutex);

while (!readyFlag) {

readyCondVar.wait(ul);

}

} // release lock

Again note that you have to use a unique\_lock and can’t use a lock\_guard here, because internally, wait() explicitly unlocks and locks the mutex.

# Example A Queue for Multiple Threads

#include <condition\_variable>

#include <mutex>

#include <future>

#include <thread>

#include <iostream>

#include <queue>

std::queue<int> queue;

std::mutex queueMutex;

std::condition\_variable queueCondVar;

void provider (int val) {

// push different values (val til val+5 with timeouts of val ms into the queue

for (int i=0; i<6; ++i) {

{

std::lock\_guard<std::mutex> lg(queueMutex);

queue.push(val+i);

} // release lock

queueCondVar.notify\_one();

std::this\_thread::sleep\_for(std::chrono::milliseconds(val));

}

}

void consumer (int num) {

// pop values if available (num identifies the consumer)

while (true) {

int val;

{

std::unique\_lock<std::mutex> ul(queueMutex);

queueCondVar.wait(ul,[]{ return !queue.empty(); });

val = queue.front();

queue.pop();

} // release lock

std::cout << "consumer " << num << ": " << val << std::endl;

}

}

int main() {

// start three providers for values 100+, 200+, and 300+

auto p1 = std::async(std::launch::async,provider,100);

auto p2 = std::async(std::launch::async,provider,200);

auto p3 = std::async(std::launch::async,provider,300);

// start two consumers printing the values

auto c1 = std::async(std::launch::async,consumer,1);

auto c2 = std::async(std::launch::async,consumer,2);

return 0;

}

Output:

consumer 1: 300

consumer 2: 200

consumer 1: 100

consumer 1: 101

consumer 2: 201

consumer 1: 102

consumer 2: 301

consumer 1: 103

consumer 2: 202

consumer 1: 104

consumer 2: 105

consumer 1: 302

consumer 2: 203

consumer 1: 204

consumer 2: 303

consumer 1: 205

consumer 2: 304

consumer 1: 305

# Condition Variables in Detail

The header file <condition\_variable> provides two classes for condition variables

1. condition\_variable
2. condition\_variable\_any

## Class condition\_variable

Below is interface of condition\_variable in detail. Class condition\_variable\_any provides the same interface except native\_handle() and notify\_all\_at\_thread\_exit().

If it can’t create a condition variable, the constructor might throw a std::system\_error exception with the error code resource\_unavailable\_try\_again, which is equivalent to the POSIX errno EAGAIN.

Copies and assignments are not allowed.

|  |  |
| --- | --- |
| Operation | Effect |
| condvar cv | Default constructor; creates a condition variable |
| cv.~condvar() | Destroys the condition variable |
| cv.notify\_one() | Wakes up one of the waiting threads, if any |
| cv.notify\_all() | Wakes up all waiting threads |
| cv.wait(ul) | Waits for notification, using the unique lock ul |
| cv.wait(ul, pred) | Waits for notification, using the unique lock ul, until pred yields true after a wakeup |
| cv.wait\_for(ul, duration) | Waits for a notification, using the unique lock ul, for duration |
| cv.wait\_for(ul, duration, pred) | Waits for a notification, using the unique lock ul, for duration or until pred yields true after a wakeup |
| cv.wait\_until(ul, timepoint) | Waits for a notification, using the unique lock ul, until timepoint |
| cv.wait\_until(ul, timepoint, pred) | Waits for a notification, using the unique lock ul, until timepoint or until pred yields true after a wakeup |
| cv.native\_handle() | Returns a platform-specific type native\_handle\_type for nonportable extensions |
| notify\_all\_at\_thread\_exit(cv, ul) | Wakes up all waiting threads of cv, using the unique lock ul, at the end of the calling thread |

Notifications are automatically synchronized so that concurrent calls of notify\_one() and notify\_all() cause no trouble.

All threads waiting for a condition variable have to use the same mutex, which has to be locked by a unique\_lock when one of the wait() members is called. Otherwise, undefined behavior occurs.

Note that consumers of a condition variable always operate on mutexes that are usually locked. Only the waiting functions temporarily unlock the mutex performing the following three atomic steps:

1. Unlocking the mutex and entering the waiting state
2. Unblocking the wait
3. Locking the mutex again

This implies that predicates passed to waiting functions are always called under the lock, so they may safely access the object(s) protected by the mutex. The calls to lock and unlock the mutex might throw the corresponding exceptions.

Called without the predicate, both wait\_for() and wait\_until() return the following enumeration class values:

* std::cv\_status::timeout if the absolute timeout happened
* std::cv\_status::no\_timeout if a notification happened

Called with a predicate as third argument, wait\_for() and wait\_until() return the result of the predicate (whether the condition holds).

The global function notify\_all\_at\_thread\_exit(cv,l) is provided to call notify\_all() when the calling thread exits. For this, it temporarily locks the corresponding lock l, which must use the same mutex all waiting threads use. To avoid deadlocks, the thread should be exited directly after calling notify\_all\_at\_thread\_exit(). Thus, this call is only to cleanup before notifying waiting threads, and this cleanup should never block.

## Class condition\_variable\_any

std::condition\_variable\_any, which does not require using an object of class std::unique\_lock as lock.

As the C++ standard library notes: “If a lock type other than one of the standard mutex types or a unique\_lock wrapper for a standard mutex type is used with condition\_variable\_any, the user must ensure that any necessary synchronization is in place with respect to the predicate associated with the condition\_variable\_any instance.”

In fact, the object has to fulfill the so-called BasicLockable requirements, which require providing synchronized lock() and unlock() member functions.

# END