18.6. 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.

Now, you can argue that we have introduced such a mechanism already: Futures (see Section 18.1, page 946) allow you to block until data by another thread is provided or another thread is done. However, 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.

Here we introduce and discuss condition variables, which can be used to synchronize logical dependencies in data flow between threads multiple times.

18.6.1. Purpose of Condition Variables

Section 18.5.1, page 997, introduced a naive approach to let one thread wait for another by using something like a *ready flag*, signaling when one thread has prepared or provided something for another thread. This usually means that the waiting thread *polls* to notice that its required data or precondition has arrived:

Click here to view code image

```
bool readyFlag;
std::mutex readyFlagMutex;

// wait until readyFlag is true:
{
      std::unique_lock<std::mutex> ul(readyFlagMutex);
      while (!readyFlag) {
          ul.unlock();
          std::this_thread::yield();      // hint to reschedule to the next thread
          std::this_thread::sleep_for(std::chrono::milliseconds(100));
          ul.lock();
      }
}   // release lock
```

However, such a *polling* for a fulfilled condition is usually not a good solution. Or as [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.

A better approach is to use *condition variables*, which the C++ standard library provides in `<condition_variable>`. A condition variable is a variable by which a thread can wake up one or multiple other waiting threads.

In principle, a condition variable works as follows:

- 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;
```

- 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
```

- Any thread that waits for the condition has to call

```
std::unique_lock<std::mutex> l(readyMutex);
readyCondVar.wait(l);
```

Thus, the thread providing or preparing something simply calls `notify_one()` or `notify_all()` for the condition variable, which for one or all the waiting threads is the moment to wake up.

So far, so good. Sounds simple. But there's more. First, note that to wait for the condition variable, you need a mutex and a `unique_lock`, introduced in Section 18.5.1, page 996. A `lock_guard` is not enough, because the waiting function might lock and unlock the mutex.

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. To quote Anthony Williams from [Williams:CondVar]: “Spurious wakes cannot be predicted: they are essentially random from the user’s point of view. However, they commonly occur when the thread library cannot reliably ensure that a waiting thread will not miss a notification. Since a missed notification would render the condition variable useless, the thread library wakes the thread from its wait rather than take the risk.”

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.

18.6.2. A First Complete Example for Condition Variables

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

Click here to view code image

```
// concurrency/condvar1.cpp

#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();
}

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);
}
```

After including the necessary header files, we need three things to communicate between threads:

1. An object for the data provided to process or a flag signaling that the condition is indeed satisfied (here: `readyFlag)`)
2. A mutex (here: `readyMutex`)
3. A condition variable (here `readyCondVar`)

The providing thread `thread1()` locks the mutex `readyMutex` , updates the condition (the object for the data or for the ready flag), unlocks the mutex, and notifies the condition variable:

```
{
    std::lock_guard<std::mutex> lg(readyMutex);
    readyFlag = true;
} // release lock
readyCondVar.notify_one();
```

Note that the notification itself does not have to be inside the protected area of the lock.

A waiting (consuming/processing) thread locks the mutex with a `unique_lock` (Section 18.5.1, page 996), waits for the notification while checking the condition, and releases the lock:

```
{
    std::unique_lock<std::mutex> ul(readyMutex);
    readyCondVar.wait(ul, []{ return readyFlag; });
} // release lock
```

Here, a `wait()` member for condition variables is used as follows: You pass the lock `ul` for the mutex `readyMutex` as first argument and a lambda as *callable object* (see Section 4.4, page 54) double checking the condition as second argument. The effect is that `wait()` internally calls a loop until the passed callable returns `true`. Thus, the code has the 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.

You can argue that this is a bad example for using condition variables, because futures can be used for blocking until some data arrives. So let's present a second example.

18.6.3. Using Condition Variables to Implement a Queue for Multiple Threads

In this example, three threads push values into a queue that two other threads read and process:

Click here to view code image

```
// concurrency/condvar2.cpp

#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 milliseconds 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+, 300+, and 500+
    auto p1 = std::async(std::launch::async,provider,100);
    auto p2 = std::async(std::launch::async,provider,300);
```

```
        auto p3 = std::async(std::launch::async,provider,500);

        // start two consumers printing the values
        auto c1 = std::async(std::launch::async,consumer,1);
        auto c2 = std::async(std::launch::async,consumer,2);
    }
```

Here, we have a global queue (see Section 12.2, page 638) concurrently used and protected by a mutex and a condition variable:

```
    std::queue<int> queue;
    std::mutex queueMutex;
    std::condition_variable queueCondVar;
```

The mutex ensures that reads and writes are atomic, and the condition variable is to signal and wake up processing threads when new values are available.

Now, three threads provide data by pushing it into the queue:

```
    {
        std::lock_guard<std::mutex> lg(queueMutex);
        queue.push(val+i);
    } // release lock
    queueCondVar.notify_one();
```

With `notify_one()` , they wake up one of the waiting threads to process the next value. Note again that this call does not have to be part of the protected section, so we close the block where the lock guard is declared before.

The threads waiting for new values to process operate as follows:

Click here to view code image

```
    int val;
    {
        std::unique_lock<std::mutex> ul(queueMutex);
        queueCondVar.wait(ul,[]{ return !queue.empty(); });
        val = queue.front();
        queue.pop();
    } // release lock
    ...
```

Here, according to the interface of a queue (see Section 12.2, page 638), we need three calls to get the next value out of the queue: `empty()` checks whether a value is available. Calling `empty()` is the double-check to deal with spurious wakeups in `wait()` . `front()` queries the next value, and `pop()` removes it. All three are inside the protected region of the unique lock `ul` . However, the processing of the value returned by `front()` happens afterward to minimize the lock duration.

A possible output of this program is:

```
    consumer 1: 300
    consumer 1: 100
    consumer 2: 500
    consumer 1: 101
    consumer 2: 102
    consumer 1: 301
    consumer 2: 103
    consumer 1: 104
    consumer consumer 1: 105
    2: 501
    consumer 1: 302
    consumer 2: 303
    consumer 1: 502
    consumer 2: 304
    consumer 1: 503
    consumer 2: 305
    consumer 1: 504
    consumer 2: 505
```

Note that the output of the two consumers is not synchronized, so characters might interleave. Note also that the order in which concurrent waiting threads are notified is not defined.

In the same way, you can call `notify_all()` if multiple consumers will have to process the same data. A typical example would be an event-driven system, where an event has to get published to all registered consumers.

Also note that for condition variables, you have the interface of waiting for a maximum amount of time: `wait_for()` waits for a duration of time, whereas `wait_until()` waits until a timepoint has arrived.

18.6.4. Condition Variables in Detail

The header file `<condition_variable>` provides two classes for condition variables, class `condition_variable` and class `condition_variable_any`.

Class `condition_variable`

As introduced in Section 18.6, page 1003, class `std::condition_variable` is provided by the C++ standard library to be able to wake up one or multiple threads waiting for a specific condition (something necessary prepared or performed or some necessary data provided). Multiple threads can wait for the same condition variable. When a condition is fulfilled, a thread can notify one or all of the waiting threads.

Due to *spurious wakeups*, notifying a thread is not enough when a condition is fulfilled. Waiting threads have and need to use the ability to double-check that the condition holds after a wakeup.

Table 18.10 lists the interface the C++ standard library provides for class `condition_variable` in detail. Class `condition_variable_any` provides the same interface except `native_handle()` and `notify_all_at_thread_exit()`.

Table 18.10. Operations of Class *`condition_variable`*

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

If it can't create a condition variable, the constructor might throw a `std::system_error` exception (see Section 4.3.1, page 43) with the error code `resource_unavailable_try_again`, which is equivalent to the POSIX `errno EAGAIN` (see Section 4.3.2, page 45). Copies and assignments are not allowed.

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:[23]

[23] The problem with a naive approach like "*lock*, *check state*, *unlock*, *wait*" is that notifications arising between *unlock* and *wait* would get lost.

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.[24] The calls to lock and unlock the mutex might throw the corresponding exceptions (see Section 18.5.2, page 1000).

[24] Thanks to Bartosz Milewski for pointing this out.

Called without the predicate, both `wait_for()` and `wait_until()` return the following *enumeration class* (see Section

3.1.13, page 32) 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.[25]

[25] A typical example would be to signal the end of a detached thread (see Section 18.2.1, page 967). By using `notify_all_at_thread_exit()` , you can ensure that thread local objects are destroyed before the main program (or master thread) processes the fact that the detached thread terminated.

Class `condition_variable_any`

Besides class `std::condition_variable` , the C++ standard library also provides a class `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.