Multithreading: Condition Variables

This lesson elucidates best practices involving condition variables such as notify_one and notify_all in multithreaded applications in C++.

WE'LL COVER THE FOLLOWING

^

- Condition Variables
 - Don't use condition variables without a predicate
 - Use Promises and Futures instead of Condition Variables
 - Promises and Futures
 - If possible, go for std::async

Condition Variables

Synchronizing threads via notifications is a simple concept, but condition variables make this task really challenging - mostly because the condition variables have no state.

- If a condition variable gets a notification, it may be the wrong one spurious wakeup.
- If a condition variable gets its notification before it was ready, the notification will be lost lost wakeup.

Don't use condition variables without a predicate

Using a condition variable without a predicate is a race condition.

```
// conditionVariableLostWakeup.cpp

#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex :
```

```
std::condition_variable condVar;

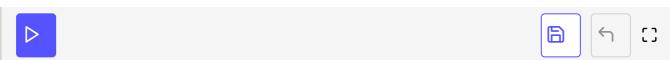
void waitingForWork(){
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    // do the work
}

void setDataReady(){
    condVar.notify_one();
}

int main(){

std::thread t1(setDataReady);
    std::thread t2(waitingForWork);

t1.join();
    t2.join();
}
```



If the thread t1 runs before the thread t2, you will get a deadlock. t1 will send its notification before t2 can accept it, and the notification is lost. This will happen very often because thread t1 starts before thread t2, and thread t1 has less work to perform.

Adding a bool variable dataReady to the workflow will solve this issue.

dataReady will also protect against a spurious wakeup, as the waiting thread first checks if the notification was from the right thread.

```
// conditioVarialbleLostWakeupSolved.cpp

#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;
std::condition_variable condVar;

bool dataReady{false};

void waitingForWork(){
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    // do the work
}

void setDataReady(){
    {
        std::lock_guard<std::mutex> lck(mutex_);
    }
}
```

```
dataReady = true;
}
condVar.notify_one();
}
int main(){

std::thread t1(waitingForWork);
std::thread t2(setDataReady);

t1.join();
t2.join();
}
```

Use Promises and Futures instead of Condition Variables

For one-time notifications, promises and futures are the better choice. The workflow of the previous program conditioVarialbleLostWakeupSolved.cpp can directly be implemented with a promise and a future.

```
// notificationWithPromiseAndFuture.cpp
                                                                                         #include <future>
#include <utility>
void waitingForWork(std::future<void>&& fut){
   fut.wait();
    // do the work
}
void setDataReady(std::promise<void>&& prom){
    prom.set_value();
}
int main(){
  std::promise<void> sendReady;
  auto fut = sendReady.get_future();
  std::thread t1(waitingForWork, std::move(fut));
  std::thread t2(setDataReady, std::move(sendReady));
 t1.join();
 t2.join();
}
```







[]

The workflow is reduced to its bare minimum. The promise <code>prom.set_value()</code> sends the notification the future <code>fut.wait()</code> is waiting for. The program needs no mutexes and locks because there is no critical section. Because no lost wakeup or spurious wakeup can happen, a predicate is also not necessary.

If your workflow requires that you use a condition variable many times, then a promise and a future pair is no alternative.

Promises and Futures

std::async can often be used as an easy-to-use replacement for threads or condition variables.

```
If possible, go for std::async
```

If possible, you should go for std::async to execute an asynchronous task.

```
auto fut = std::async([]{ return 2000 + 11; });
// some time passes
std::cout << "fut.get(): " << fut.get() << std::endl;</pre>
```

Let's see this in action:

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

int main(){

    std::cout << std::endl;

    auto fut= std::async([]{ return 2000 + 11; });
    std::this_thread::sleep_for (std::chrono::seconds(2)); //Work for 2 seconds
    std::cout << "fut.get(): " << fut.get() << std::endl;

    std::cout << std::endl;
}</pre>
```

By invoking auto fut = std::async([]{ return 2000 + 11; }), you say to the C++ runtime: "Run my job". I don't care if it will be executed immediately, if it

GPU; You are only interested in picking up the result in the future: fut.get().

From a conceptional view, a thread is just an implementation detail for running your job. You only specify *what* should be done and not *how* it should be done.