## **Blocking Issues**

This lesson explains the challenges of blocking issues while using a condition variable in C++.

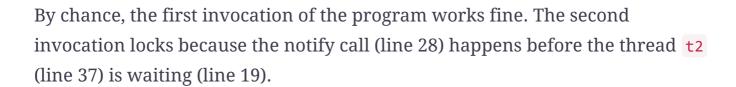
To make my point clear, you have to use a condition variable in combination with a predicate. If you don't, your program may become a victim of a spurious wakeup or lost wakeup.

If you use a condition variable without a predicate, it may happen that the notifying thread sends its notification before the waiting thread is in the waiting state; Therefore, the waiting thread waits forever. This phenomenon is called a lost wakeup. Here is the program.

```
// conditionVariableBlock.cpp
                                                                                            6
#include <iostream>
#include <condition variable>
#include <mutex>
#include <thread>
std::mutex mutex_;
std::condition_variable condVar;
bool dataReady;
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;</pre>
    std::unique lock<std::mutex> lck(mutex );
    condVar.wait(lck);
    // do the work
    std::cout << "Work done." << std::endl;</pre>
}
void setDataReady(){
    std::cout << "Sender: Data is ready." << std::endl;</pre>
    condVar.notify_one();
}
int main(){
```

```
std::cout << std::endl;
std::thread t1(setDataReady);
std::thread t2(waitingForWork);

t1.join();
t2.join();
std::cout << std::endl;
}</pre>
```



Of course, deadlocks and livelocks are side effects of race conditions. A deadlock depends in general on the interleaving of the threads and may sometimes occur or not. A livelock is similar to a deadlock; while a deadlock blocks, a livelock seems to make progress, with the emphasis on "seems." Think about a transaction in a transactional memory use case. Each time the transaction should be committed, a conflict happens and, therefore, a rollback takes place.