Issues of Mutexes: Deadlocks

This lesson gives an overview of deadlocks caused by improper mutex locking in C++.

WE'LL COVER THE FOLLOWING

The issues with mutexes boil down to one main concern: deadlocks.

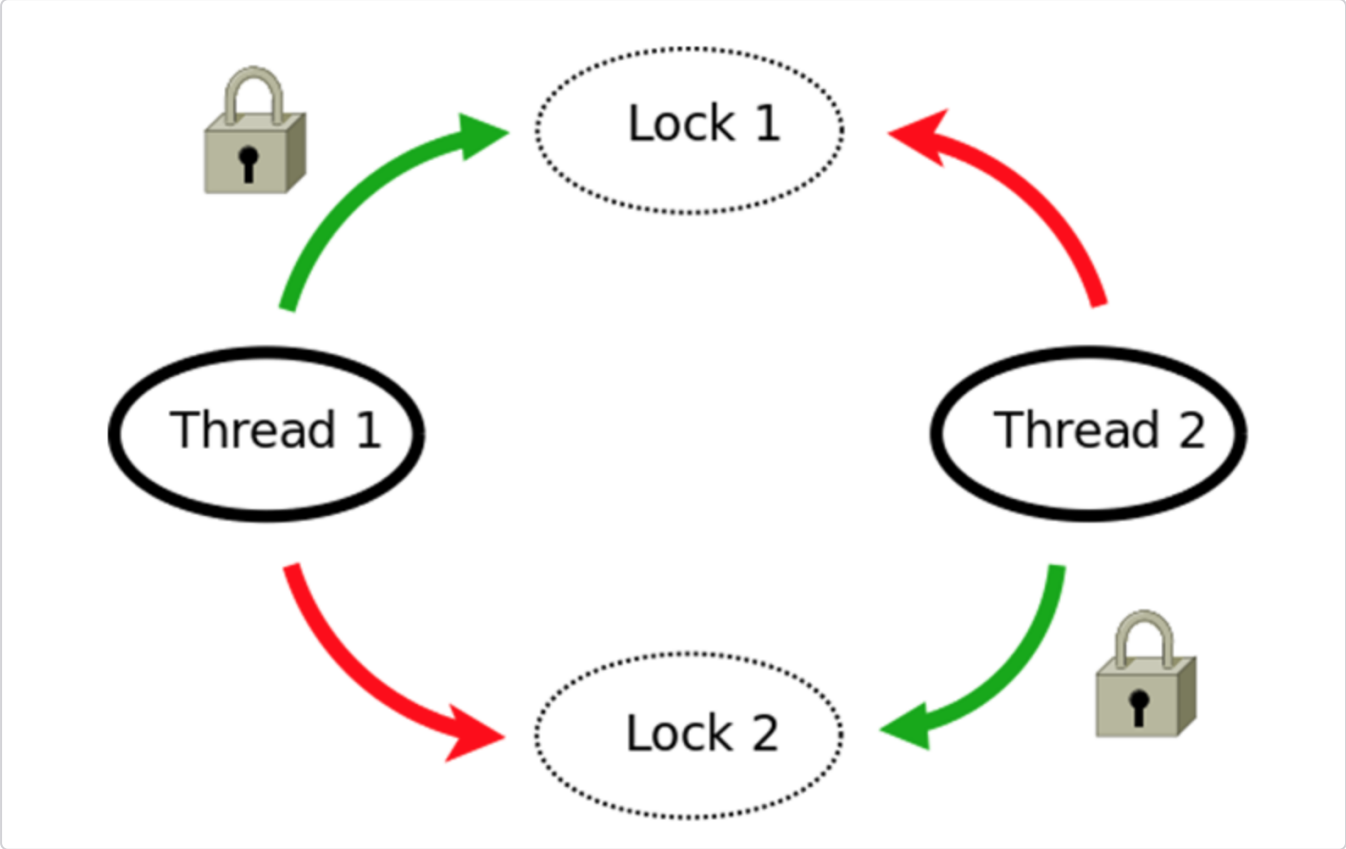
Deadlock

A deadlock is a state where two or more threads are blocked because each thread waits for the release of a resource before it releases its own resource.

The result of a deadlock is a total standstill. The thread that tries to acquire the resource - and usually the whole program - is blocked forever. It is easy to produce a deadlock. Curious? Let's see how a deadlock occurs.

Deadlocks caused by mutex locking order

Here is a typical scenario of a deadlock resulting from locking in a certain order.



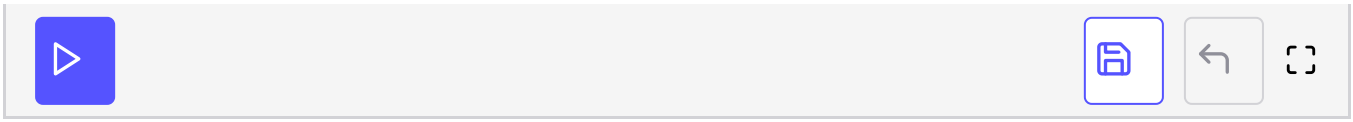
Thread 1 and thread 2 need access to two resources to finish their work. The problem arises when the requested resources are protected by two separate mutexes and are locked in different orders (Thread 1: Lock 1, Lock 2; Thread 2: Lock 2, Lock 1). In this case, the thread executions will interleave in such a way that thread 1 gets mutex 1, then thread 2 gets mutex 2, and then we reach a standstill. Each thread wants to get the other's mutex but, to get the other's mutex, the first thread has to release its mutex first. The expression "deadly embrace" describes this kind of deadlock very well.

Deadlock example

Translating this picture into code is easy.

```
// deadlock.cpp

#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>

struct CriticalData{
  std::mutex mut;
};
```

```
void deadLock(CriticalData& a, CriticalData& b){

  a.mut.lock();
  std::cout << "get the first mutex" << std::endl;
  std::this_thread::sleep_for(std::chrono::milliseconds(1));
  b.mut.lock();
  std::cout << "get the second mutex" << std::endl;
  // do something with a and b
  a.mut.unlock();
  b.mut.unlock();

}

int main(){

  CriticalData c1;
  CriticalData c2;

  std::thread t1([&]{deadLock(c1,c2);});
  std::thread t2([&]{deadLock(c2,c1);});

  t1.join();
  t2.join();

}
```



Explanation:

Threads `t1` and `t2` call `deadlock` (lines 12 - 23). The function `deadlock` needs variables `CriticalData c1` and `c2` (lines 27 and 28). Because objects `c1` and `c2` have to be protected from shared access, they internally hold a mutex; (To keep this example short and simple, `CriticalData` doesn't have any other methods or members apart from a mutex).

A short sleep of about 1 millisecond in line 16 is sufficient to produce the deadlock.

Locks will not solve all the issues with mutexes, but they'll come to the rescue in many cases. We will cover the issues of mutexes in detail in the next lesson.