## Deadlock

Deadlock is the biggest problem with having to lock two or more mutexes in order to perform an operation.

### Mutex

Before we dive into a deadlock case, let's start with a simple code that needs mutex protection. The following program prints out numbers: threads 1-5 for (0,49) and main thread for (-49,0).

```
// t11.cpp

#include <iostream>

#include <mutex>

#include <thread>


using namespace std;


mutex myMutex;


void shared_cout(int i)

{

   lock_guard<mutex> g(myMutex);

   cout << " " << i << " ";

}


void f(int n)

{

   for(int i = 10*(n-1); i < 10*n ; i++) {

      shared_cout(i);

   }

}
```

```
int main()
{
    thread t1(f, 1);  // 0-9
    thread t2(f, 2);  // 10-19
    thread t3(f, 3);  // 20-29
    thread t4(f, 4);  // 30-39
    thread t5(f, 5);  // 40-49


    for(int i = 0; i > -50; i--)
        shared_cout(i);  // (0, -49)

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();

    return 0;
```

If we do not use mutex, we get the following outputs:

Run #1

Output-

```
$ g++ t11.cpp -o t11 -std=c++11 -pthread
$ ./t11

  20  21  22  23  24  25  26  27  28  29 10  11  12  13  14  15  16  17  18  19  30  31  32  33  34  350 0 40  41
42  43   -1  -2  44-3    45-4    46-5    47-6    48-7    49-8   -9  -10  -11  -12  -13  -14  -15  -16  -17  -18  -19  -20
-21  -22  -23  -24  -25  -26  -27  -28  -29  -30  -31  -32  -33  -34  -35  -36  -37  -38  -39  -40  -41  -42  -43  -44
-45  -46  -47  -48  -49   1  2  3  4  5  6  7  8  9   36  37  38  39
```

Run #2

```
$ ./t11

   30  031    32-1    -2  -3  -433  -5  -6   -7 34  35  36    37-8   -9   -1038    -1139   -12  -13  -14  -15  -16  -17
-18  -19  -20  -21  -22  -23  -24  -25  -26  -27  -28  -29  -30  -31  -32  -33  -34  -35  -36  -37  -38  -39  -40  -41
-42  -43  -44  -45  -46  -47  -48  -49 20  21  22  23  24  25  26  27  28  29 0  1  2  3  4  5  6  7  8  9 10  11  12
13  14  15  16  17  18  19  40  41  42  43  44  45  46  47  48  49
```

We can see somewhat random output, and the cout << " " not worked as the code intended. However, if we use the mutex, we get:

```
$ ./t11

 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
31  32  33  34  35  36  37  38  39  0  -1  -2  -3  -4  -5  -6  -7  -8  -9  -10  -11  -12  -13  -14  -15  -16  -17  -18
-19  -20  -21  -22  -23  -24  -25  -26  -27  -28  -29  -30  -31  -32  -33  -34  -35  -36  -37  -38  -39  -40  -41
-42  -43  -44  -45  -46  -47  -48  -49  40  41  42  43  44  45  46  47  48  49
```

Now, we want to make a deadlock situation:

```
#include <iostream>

#include <mutex>

#include <thread>

#include <mutex>


using namespace std;

const int SIZE = 10;


mutex myMutex, myMutex1, myMutex2;
```

```
void shared_cout_thread_even(int i)

{

   lock_guard<mutex> g1(myMutex1);

   lock_guard<mutex> g2(myMutex2);

   cout << " " << i << " ";

}


void shared_cout_thread_odd(int i)

{

   lock_guard<mutex> g2(myMutex2);

   lock_guard<mutex> g1(myMutex1);

   cout << " " << i << " ";

}


void shared_cout_main(int i)

{

   lock_guard<mutex> g(myMutex);

   cout << " " << i << " ";

}


void f(int n)

{

   for(int i = SIZE*(n-1); i < SIZE*n ; i++) {

      if(n % 2 == 0)
```

```
            shared_cout_thread_even(i);

        else

            shared_cout_thread_odd(i);

    }

}


int main()

{

    thread t1(f, 1);  // 0-9

    thread t2(f, 2);  // 10-19

    thread t3(f, 3);  // 20-29

    thread t4(f, 4);  // 30-39

    thread t5(f, 5);  // 40-49


    for(int i = 0; i > -SIZE; i--)

        shared_cout_main(i);  // (0, -49)


    t1.join();

    t2.join();

    t3.join();

    t4.join();

    t5.join();


    return 0;

}
```

The code sample is rather a concocted one. The "std::cout" resource is protected with two mutex locks with different order: either mutex1->mutex2 or mutex2->mutex1. With this setup, sometimes it works fine but sometimes we have a deadlock. In our case, the deadlock happens when two threads are waiting for a mutex owned by the other.

One of the most common ways of avoiding a deadlock is to always lock the two mutexes in the same order. If we always lock mutex A before mutex B, then we'll never have a deadlock. Sometimes this is straightforward, because the mutexes are serving different purposes, but other times it's not so simple, such as when the mutexes are each protecting a separate instance of the same class.

As an example, let's think about an operation that exchanges data between two instances of the same class. In order to ensure that the data is exchanged correctly, without being affected by concurrent modifications, the mutexes on both instances must be locked. However, if a fixed order is chosen such as, the mutex for the instance supplied as the first parameter, then the mutex for the instance supplied as the second parameter. This can backfire, however, all it takes is for two threads to try to exchange data between the same two instances with the parameters swapped, and we have deadlock!

However, the C++ Standard Library has a cure for this in the form of std::lock which is a function that can lock two or more mutexes at once without risk of deadlock.

The example below shows how to use std::lock for a simple swap operation.

```
#include <mutex>


using namespace std;


class MyObjectClass {};


void swap(MyObjectClass& lhs,MyObjectClass& rhs);


class X

{

private:
        MyObjectClass myObj;

        std::mutex m;

public:

        X(MyObjectClass const& obj):myObj(obj){}

        friend void swap(X& lhs, X& rhs)

        {

                // the arguments are checked to ensure they are different instances,

                // because attempting to acquire a lock on a std::mutex

                // when we already hold it is undefined behavior.

                if(&lhs;==&rhs;) return;


                // the call to std::lock() locks the two mutexes

                std::lock(lhs.m,rhs.m);


                // two std::lock_guard instances are constructed one for each mutex.

                std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);

                std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);


                swap(lhs.myObj, rhs.myObj);

        }

};
```

The code checks the arguments to ensure they are different instances, because attempting to acquire a lock on a std::mutex when we already hold it is undefined behavior. A mutex that does permit multiple locks by the same thread is provided in the form of std::recursive_mutex. Then, the call to std::lock() locks the two mutexes, and two std::lock_guard instances are constructed one for each mutex. The std::adopt_lock parameter is supplied in addition to the mutex to indicate to the std::lock_guard objects that the mutexes are already locked, and they should just adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor.

This ensures that the mutexes are correctly unlocked on function exit in the general case where the protected operation might throw an exception; it also allows for a simple return. Also, it's worth noting that locking either lhs.m or rhs.m inside the call to std::lock can throw an exception. in that case, the exception is propagated out of std::lock. If std::lock has successfully acquired a lock on one mutex and an exception is thrown when it tries to acquire a lock on the other mutex, this first lock is released automatically: std::lock provides all-or-nothing semantics with regard to locking the supplied mutexes.

Although std::lock can help us to avoid deadlock in those cases where we need to acquire two or more locks together, it doesn't help if they're acquired separately. In that case we have to rely on our discipline as developers to ensure we don't get deadlock. This isn't easy: deadlocks are one of the nastiest problems to encounter in multithreaded code and are often unpredictable, with everything working fine the majority of the time. There are, however, some relatively simple rules that can help us to write deadlock-free code.

## How to avoid deadlock

Though locks are the most frequent cause of deadlock, it does not just occur with locks. We can create deadlock with two threads and no locks just by having each thread call join() on the std::thread object for the other. In this case, neither thread can make progress because it's waiting for the other to finish.

This simple cycle can occur anywhere that a thread can wait for another thread to perform some action if the other thread can simultaneously be waiting for the first thread, and it isn't limited to two threads: a cycle of three or more threads will still cause deadlock.

The guidelines for avoiding deadlock all boil down to one idea: don't wait for another thread if there's a chance it's waiting for you. The individual guidelines provide ways of identifying and eliminating the possibility that the other thread is waiting for you.