Multithreading: Shared Data

This lesson gives an overall guide for best practices used to manage shared data in multithreaded applications in C++.

WE'LL COVER THE FOLLOWING

- Data Sharing
  - Pass data per default by copy
  - Minimize the time holding a lock
  - Put a mutex into a lock
  - Use std::lock or std::scoped_lock for locking more mutexes atomically
  - Never call unknown code while holding a lock

Data Sharing

With data sharing, the challenges in multithreading programming start.

Pass data per default by copy

```
#include <iostream>
#include <thread>
#include <string>

int main(){

std::string s{"C++11"};

std::thread t1([s]{std::cout << s << std::endl; }); // do something with s
t1.join();

std::thread t2([&s]{ std::cout << s << std::endl; }); // do something with s
t2.join();

// do something with s

s.replace(s.begin(), s.end(), 'C', 'Z');
}
```

If you pass data such as the `std::string s` to a thread `t1` by copy, the creator thread and the created thread `t1` will use independent data; this is in contrast to the thread `t2`. It gets its `std::string s` by reference. This means you have to synchronize the access to `s` in the creator thread and the created thread `t2` preventively. This is error-prone and expensive.

Minimize the time holding a lock

If you hold a lock, only one thread can enter the ***critical section*** and make progress.

```
#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

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

bool dataReady{false};

void setDataReadyBad(){
    std::lock_guard<std::mutex> lck(mutex_);
    //Work on Shared Variable
    dataReady = true;
    std::cout << "Data prepared" << std::endl;
    condVar.notify_one();
}                                 // unlock the mutex

void setDataReadyGood(){
    //Work on Shared Variable
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
    }                            // unlock the mutex
    std::cout << "Data prepared" << std::endl;
    condVar.notify_one();
}

int main(){

  std::cout << std::endl;

  std::thread t1(setDataReadyBad);
  std::thread t2(setDataReadyGood);

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

  std::cout << std::endl;
```

```
}
```

The functions `setDataReadyBad` and `setDataReadyGood` are the notification components of a condition variable. The variable `dataReady` is necessary to protect against spurious wakeups and lost wakeups. Because `dataReady` is a non-atomic variable, it has to be synchronized using the lock `lck`. To make the lifetime of the lock as short as possible, use an artificial scope (`{ ... }`) such as in the function `setDataReadyGood`.

Put a mutex into a lock

You should not use a mutex without a lock.

```
std::mutex m;
m.lock();
// critical section
m.unlock();
```

Something unexpected may happen in the critical section or you could simply forget to unlock the mutex; anyway, the result is the same. If you don't unlock a mutex, another thread requiring the mutex will be blocked and you will end with a deadlock.

Thanks to locks that automatically take care of the underlying mutex, your risks of getting a deadlock are considerably reduced. According to the RAII idiom, a lock automatically binds its mutex in the constructor and releases it in the destructor.

```
{
  std::mutex m,
  std::lock_guard<std::mutex> lockGuard(m);
  // critical section
}          // unlock the mutex
```

The artificial scope (`{ ... }`) ensures that the lifetime of the lock automatically ends; therefore, the underlying mutex will be unlocked.

Use std::lock or std::scoped_lock for locking more mutexes atomically

If a thread needs more than one mutex, you have to be extremely careful that you lock the mutex always in the same sequence. If not, you get a data race and a bad interleaving of threads may cause a deadlock.

```
void deadLock(CriticalData& a, CriticalData& b){
  std::lock_guard<std::mutex> guard1(a.mut);
  // some time passes
  std::lock_guard<std::mutex> guard2(b.mut);
  // do something with a and b
}

...

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

...
```

Thread `t1` and `t2` need two `CriticalData` resources to perform their job, as `CriticalData` has its own mutex `mut` to synchronize the access. Unfortunately, both invoke the function `deadlock` with the arguments `c1` and `c2` in a different sequence. Now we have a data race. If thread `t1` can lock the first mutex (`a.mut`) but not the second one (`b.mut`) because thread `t2` locks the second one in the meantime, then we will get a deadlock.

Thanks to `std::unique_lock`, you can defer the locking of its mutex; it's done by the function `std::lock`, which can lock an arbitrary number of mutexes in an atomic way.

```
void deadLock(CriticalData& a, CriticalData& b){
  unique_lock<mutex> guard1(a.mut,defer_lock);
  // some time passes
  unique_lock<mutex> guard2(b.mut,defer_lock);
  std::lock(guard1,guard2);
  // do something with a and b
}

...

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

...
```

C++17 has a new lock `std::scoped_lock`, which can get an arbitrary number of mutexes and locks them atomically. Now, the workflow becomes even simpler.

```
void deadLock(CriticalData& a, CriticalData& b){
   std::scoped_lock(a.mut, b.mut);
   // do something with a and b
}

...

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

...
```

Never call unknown code while holding a lock

Calling an `unknownFunction` while holding a mutex is a recipe for undefined behavior.

```
{
  std::mutex m,
  std::lock_guard<std::mutex> lockGuard(m);
  sharedVariable= unknownFunction();
}
```

I can only speculate about the `unknownFunction`. If `unknownFunction`

- tries to lock the mutex `m`, that will be undefined behavior. Most of the times, you will get a deadlock.
- starts a new thread that tries to lock the mutex `m`, you will get a deadlock.
- will not directly or indirectly try to lock the mutex `m`, all seems to be fine. “Seems” because your coworker can modify the function or the function is dynamically linked and you get a different version. All bets are open what may happen.