MULTITHREADING

DATA SHARING

CODERS SCHOOL

BARTOSZ SZURGOT

ŁUKASZ ZIOBROŃ

AGENDA

1. Data sharing for reading
2. Data sharing for writing
3. Data races
4. Thread sanitizer
5. `mutex`
6. Types of mutexes
7. `shared_mutex`
8. locks (RAII wrappers)
9. Memory model
10. `std::atomic`
11. Good practices
12. Deadlock

https://github.com/coders-school/multithreading/tree/master/module2

LET'S GET TO KNOW EACH OTHER

- Have you been to thread training?
- One of the most interesting things from Cpp Core Guidelines pre-work?

ŁUKASZ ZIOBROŃ

EXPERIENCE NOT ONLY IN PROGRAMMING

- Trainer and DevOps @ Coders School
- C ++ and Python developer @ Nokia and Credit Suisse
- Team leader and trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C ++ courses @ Coders School
- Practical Aspects of Software Engineering @ PWr & UWr
- Nokia Academy
- Internal corporate training

EXPERIENCE IN PUBLIC SPEAKING

- code::dive conference
- code::dive community
- Polish Academic Championship in Team Programming

INTERESTS

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy
- Marketing 🆕

CONTRACT

- 🎰 The Vegas rule
- 🗣️ Discussion, not lecture
- ☕ Additional breaks on demand
- ⌚ Punctuality

DATA SHARING

DATA SHARING FOR READING

- Imagine you are watching TV with your friends. Many friends can watch TV with you. Neither of you influences the material you are watching in any way (nobody has a remote :)).
- It's exactly the same with sharing read-only data
  - Threads = you and friends
  - Data = movie
  - Reading data by one thread does not interfere with reading data by another thread
- No modification = no problems

DATA SHARING WITH THEIR MODIFICATION

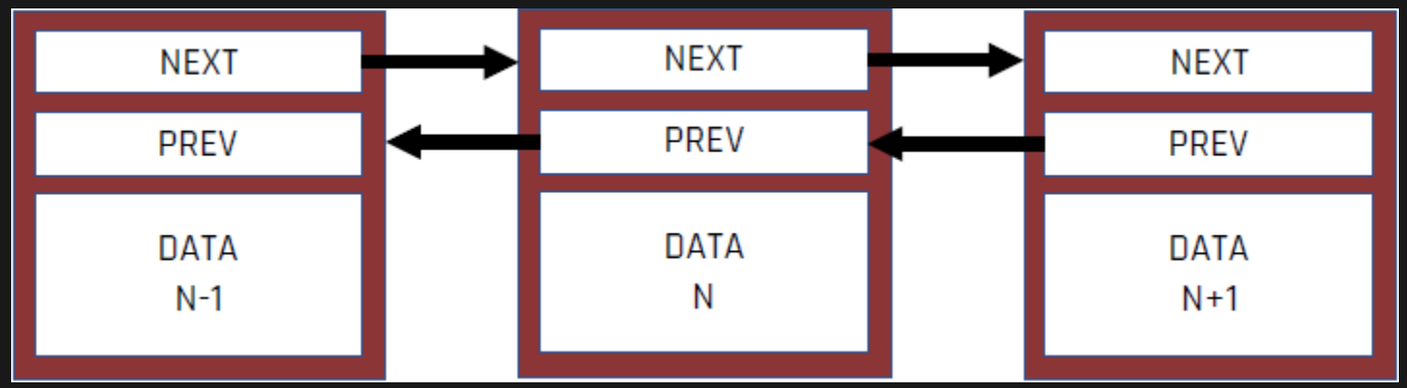
- Imagine you are sharing an apartment with your flatmates. Each of you has your own room, but the toilet is shared. When someone is in it, they have to lock the door. If you want to use the toilet at the same time, you can't do it. Waiting for a long time for another person to stop occupying the toilet is frustrating. After use, the toilet is in a different state than before (less paper, used water, more interesting smells ...)
- Exactly the same is with data sharing :)
  - Threads = you and flatmates
  - Resource (Data) = Toilet (Paper, Water)
  - Mutex = door lock
- Data modification (toilet use) = problem - synchronization required
- What if there was no door with a lock to the toilet? Someone could flush the toilet during its use or pick up the last sheet of toilet paper.

EXAMPLE: REMOVING A NODE FROM A BIDIRECTIONAL LIST

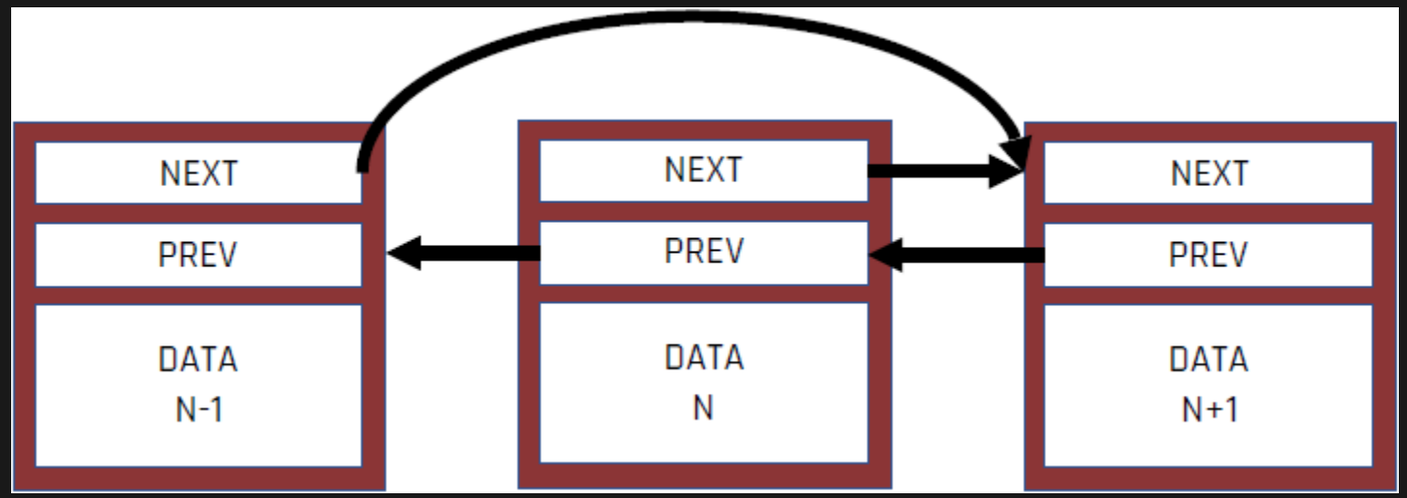
1. Find the node to be deleted (node N)
2. Set the pointer `NEXT` at node N-1 to N+1
3. Set the pointer `PREV` at node N+1 to N-1
4. Delete node N

The pointers are not consistent between steps 2 and 3.

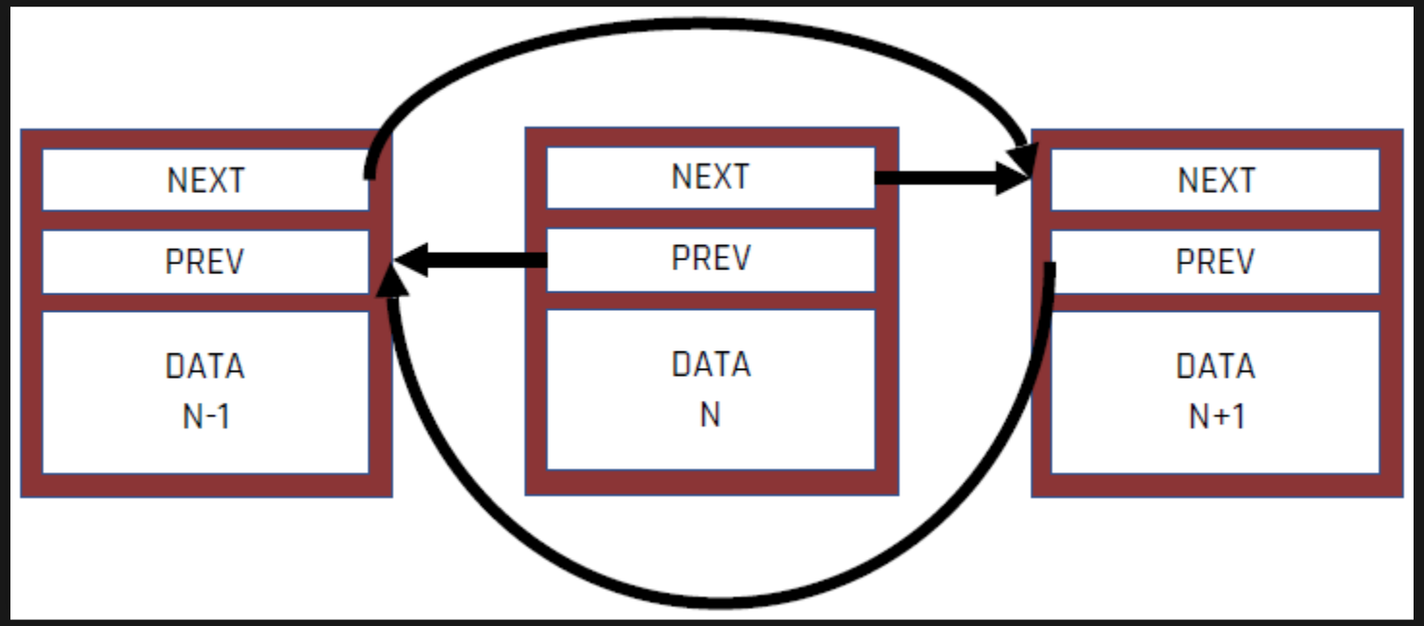
FIND THE NODE TO BE DELETED



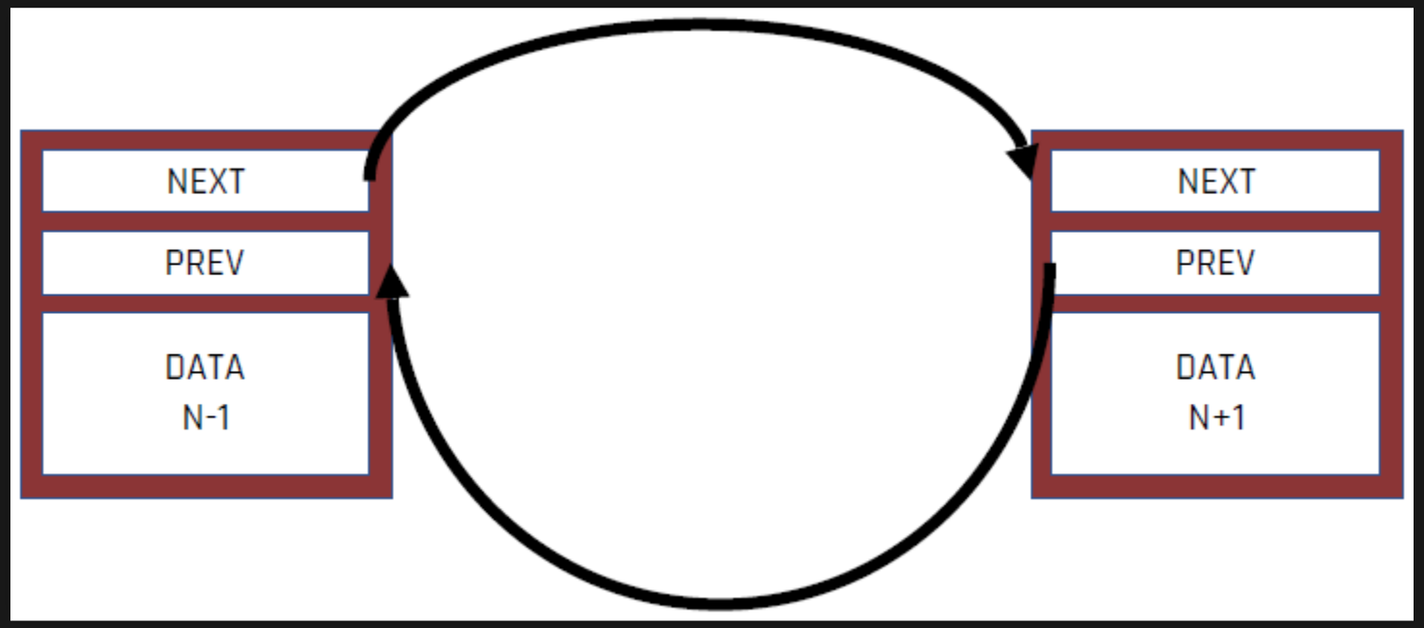
SET THE POINTER **NEXT** AT NODE N-1 TO N+1



SET THE POINTER **PREV** AT NODE N+1 TO N-1



DELETE NODE N



EXAMPLE: REMOVING A NODE FROM A BIDIRECTIONAL LIST

- Removing a node from the list consists of several steps
- What if another thread iterates through the list while removing the N node?
- What if another thread starts removing node N + 1 while node N is being removed?
- This phenomenon is called race conditions
- Race conditions typically occur when two or more separate pieces of data need to be modified, such as the pointers in the list example

Data races are a slightly different phenomenon - they occur when two or more threads modify the same (one) memory area without synchronization

RACE CONDITIONS

CODERS SCHOOL

ARE THERE RACES HERE?

```cpp
#include <thread>
#include <iostream>
#include <functional>

void abc(int &a) { a = 2; }
void def(int &a) { a = 3; }

int main() {
    int x = 1;
    std::thread t1(abc, std::ref(x));
    std::thread t2(def, std::ref(x));

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

    std::cout << x << std::endl;
}
```

What will appear on the screen?

```
$> g++ 01_threads_write.cpp -lpthread
$> ./a.out
3
$> ./a.out
3
$> ./a.out
3
$> ./a.out
3
$> ./a.out
3
$> ./a.out
3
$> ./a.out
3
$> ./a.out
2
```

RACE CONDITIONS

- Races generate nondeterministic (random) program behavior.
- Race = Undefined behavior.
- Unwanted side effects usually do not occur during most runs of the application.
- Problems may occur less than 1 in 1000 runs.
- Races are incredibly hard to spot. Usually, when the CPU is not under load, everything works as it should. The more loaded the CPU, the more the risk of a different order of data access increases. Such problems are difficult to reproduce.
- The data race problem is time-critical and can be completely impossible to catch under the debugger. The debugger affects the execution times of individual instructions.
- It is much easier to prevent races. The treatment is time-consuming and expensive.
- Thread Sanitizer (TSan) - data race detector.

THREAD SANITIZER

```
$> g++ examples/01_threads_write.cpp -lpthread -fsanitize=thread -O2 -g
$> ./a.out
```

```
==================
WARNING: ThreadSanitizer: data race (pid=3180)
  Write of size 4 at 0x7ffc806f24ac by thread T2:
    #0 def(int&) /home/ziobron/coders.school/data_sharing/threads_write.cpp:6 (a.out+0x12c5)
    #1 void std::__invoke_impl<void, void (*)(int&), std::reference_wrapper<int> >(std::__invoke_other, void (*&&)(int&), std::reference_wrapper
    #2 std::__invoke_result<void (*)(int&), std::reference_wrapper<int> >::type std::__invoke<void (*)(int&), std::reference_wrapper<int> >(void
a.out+0x130e)
    #3 decltype (__invoke((_S_declval<0ul>)(), (_S_declval<1ul>)())) std::thread::_Invoker<std::tuple<void (*)(int&), std::reference_wrapper<int
234 (a.out+0x130e)
    #4 std::thread::_Invoker<std::tuple<void (*)(int&), std::reference_wrapper<int> > >::operator()() /usr/include/c++/7/thread:243 (a.out+0x130
    #5 std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)(int&), std::reference_wrapper<int> > > >::_M_run() /usr/include/c++/7/
    #6 <null> <null> (libstdc++.so.6+0xbd57e)

  Previous write of size 4 at 0x7ffc806f24ac by thread T1:
    #0 abc(int&) /home/ziobron/coders.school/data_sharing/threads_write.cpp:5 (a.out+0x1295)
    #1 void std::__invoke_impl<void, void (*)(int&), std::reference_wrapper<int> >(std::__invoke_other, void (*&&)(int&), std::reference_wrapper
    #2 std::__invoke_result<void (*)(int&), std::reference_wrapper<int> >::type std::__invoke<void (*)(int&), std::reference_wrapper<int> >(void
a.out+0x130e)
    #3 decltype (__invoke((_S_declval<0ul>)(), (_S_declval<1ul>)())) std::thread::_Invoker<std::tuple<void (*)(int&), std::reference_wrapper<int
234 (a.out+0x130e)
    #4 std::thread::_Invoker<std::tuple<void (*)(int&), std::reference_wrapper<int> > >::operator()() /usr/include/c++/7/thread:243 (a.out+0x130
    #5                                                                                                                          /c++/7/
    #6

  Loca

  Location is global '<null>' at 0x000000000000 ([stack]+0x00000001f4ac)

  Thread T2 (tid=3183, running) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x2bcfe)
    #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State> >, void (*)()) <null> (libstdc
    #2 main /home/ziobron/coders.school/data_sharing/threads_write.cpp:12 (a.out+0x1065)

  Thread T1 (tid=3182, finished) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x2bcfe)
    #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, std::default_delete<std::thread::_State> >, void (*)()) <null> (libstdc
    #2 main /home/ziobron/coders.school/data_sharing/threads_write.cpp:11 (a.out+0x1041)

SUMMARY: ThreadSanitizer: data race /home/ziobron/coders.school/data_sharing/threads_write.cpp:6 in def(int&)
==================
3
ThreadSanitizer: reported 1 warnings
```

TASK: REPRODUCTION OF RACES UNDER A DEBUGGER

- Compile threads_write.cpp
  `g++ 01_threads_write.cpp -lpthread -g`
- Run the program under the debugger (gdb or other)
  `gdb --tui ./a.out`
- Make `x` have the value 2 at the end of the program
- Make `x` have the value 3 at the end of the program
- Useful commands:
  - `b 5` - sets the breakpoint on the fifth line
  - `watch x` - observe changes of the variable x (the debugger will stop when it is modified)
  - `c` - continue debugging
  - `info threads` - thread information
  - `thread 3` - switch to thread 3
  - `n` - next instruction
  - `fin` - execute everything until the end of the current function
  - `up` - go up to the upper stack frame
  - `down` - go down to the lower stack frame
  - `del br` - remove all breakpoints
  - `CTRL + L` - refresh view

One of the possible sequences:

- `b 5`
- `b 17`
- `r`
- `c`
- `p x`
  - `$ = 2`
- `del br`
- `b 6`
- `b 17`
- `r`
- `c`
- `p x`
  - `$ = 3`

MUTEX

CODERS SCHOOL

MUTEX

- Mutex (Mutual Exclusion) - mutual exclusion
- Lock implementation
- `#include <mutex>`
- `std::mutex`
- The most important operations:
  - `void lock()` - mutex locking. Blocking operation. If the mutex is locked by another thread, we wait to be unlocked.
  - `void unlock()` - mutex unlocking.
  - `bool try_lock()` - mutex locking. Non-blocking operation. If the mutex is already locked by another thread, it continues with further thread execution. Returns true if it succeeds in locking the mutex

TASK: SYNCHRONIZE STREAM WRITE OPERATIONS

```cpp
#include <vector>
#include <thread>
#include <chrono>
#include <iostream>

using namespace std;

void do_work(int id) {
    this_thread::sleep_for(100ms);
    cout << "Thread [" << id << "]: "
         << "Job done!" << endl;
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- Secure your code so that each thread can safely enter its full text into the stream.
- The following cases should not be possible

```
$> g++ 02_threads_in_collection.cpp -lpthread
$> ./a.out
...
Thread [10]: Job done!
Thread [9]: Thread [Job done!11]: Job done!
Thread [6]: Job done!
...
$> ./a.out
...
Thread [Thread [5]: Job done!
Thread [13]: Job done!
17]: Job done!
...
```

TASK - A SOLUTION WITH A MUTEX

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

using namespace std;

mutex m;

void do_work(ind id) {
    this_thread::sleep_for(100ms);
    m.lock();
    cout << "Thread [" << id << "]: "
         << "Job done!" << endl;
    m.unlock();
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i));
    }
    for (auto && t : threads) {
        t.join();
```

- Works
- The code locked with mutex is called critical section.
- The duration of the lock must be as short as possible
  - Locking the function `sleep_for()` is an error.
- Global mutex
  - Usually mutexes are placed in classes whose resources access must be synchronized.
- What if an exception occurs during the lockout period?
  - `unlock()` will not be called
  - Other threads will never finish
  - No RAII
- Can it be better?

TASK - A SOLUTION WITH `lock_guard` + `mutex`

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

using namespace std;

void do_work(ind id, mutex & m) {
    this_thread::sleep_for(100ms);
    lock_guard<mutex> lock(m);
    cout << "Thread [" << id << "]: "
         << "Job done!" << endl;
}

int main() {
    mutex m;
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i, ref(m)));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- It works.
- Safe in case of exceptions - RAII.
- Mutex passed by reference.
- Formatting and composing a text under a lock - streams are not famous for their performance.
- Can it be faster?

TASK - A SOLUTION WITH `stringstream`

```
#include <vector>
#include <thread>
#include <chrono>
#include <iostream>
#include <sstream>

using namespace std;

void do_work(ind id) {
    this_thread::sleep_for(100ms);
    stringstream ss;
    ss << "Thread [" << id << "]: "
       << "Job done!" << endl;
    cout << ss.rdbuf();
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- It works.
- Text formatting outside the lock.
- Mutex is unnecessary
- A single write to a stream is atomic.

CRITICAL SECTION

```
class SafeList {
    mutex m;
public:
    void remove_safely(Node* node) {
        lock_guard<mutex> lock(m);
        node->prev->next = node->next;
        node->next->prev = node->prev;
        delete node;
    } // automatic unlocking
    // other methods
};
```

- A critical section is a piece of a code that can only be executed by one thread at a time
- It is usually implemented using a mutex
- Always use a lock manager (e.g. `lock_guard`) to provide the RAII mechanism
- Usually, the entire critical section is a separate function

TYPES OF MUTEXES

- `mutex`
  - `void lock()` - the current thread is suspended until a lock is acquired
  - `void unlock()` - if the current thread is the holder of the lock, it is released. Otherwise we have UB
  - `bool try_lock()` - attempt to obtain a lock without blocking the current thread. Returns true if the lock has been acquired, false otherwise
- `timed_mutex`
  - has methods to define the maximum waiting time for a thread to acquire a lock
    - `bool try_lock_until(timeout_time)`
    - `bool try_lock_for(timeout_duration)`
- `recursive_mutex`
  - the same thread can obtain a mutex multiple times by calling `lock()` or `try_lock()`
  - to release the mutex `unlock()` must be called a number of times it was locked
- `recursive_timed_mutex`
  - owns property `timed_mutex`
  - owns property `recursive_mutex`
- `shared_mutex`
  - the possibility of obtaining shared locks using the methods:
    - `void lock_shared()`
    - `bool try_lock_shared()`
    - `bool try_lock_shared_for(rel_time)`
    - `bool try_lock_shared_until(abs_time)`
    - `void unlock_shared()`

SHARED LOCK - `shared_mutex`

```
#include <deque>
#include <shared_mutex>
std::deque<int> ids;
std::shared_mutex mtxIds;
int getIdsIndex() { /* ... */ }
void process(int) { /* ... */ }
int newValue() { /* ... */ }
void reader() {
    int index = getIdsIndex();
    std::shared_lock<std::shared_mutex> lock(mtxIds);
    int value = ids[index];
    lock.unlock();
    process(value);
}
void writer() {
    int index = getIdsIndex();
    std::lock_guard<std::shared_mutex> lock(mtxIds);
    ids[index] = newValue();
}
```

- Shared locks are used when reading data
  - `shared_lock`
- You must obtain an exclusive lock to write data
  - `lock_guard`
  - `unique_lock`

LOCK MANAGERS

CODERS SCHOOL

LOCK MANAGERS

- `lock_guard<Mutex>`
  - the simplest one, primary choice
  - constructor calls `lock()` on the mutex
  - the destructor calls `unlock()`
  - non-copyable
- `unique_lock<Mutex>`
  - delayed blocking
  - time-limited locking attempts
  - recursive locking
  - making non-blocking attempts to obtain a lock - `try_lock()`
  - cooperation with `timed_mutex`
  - cooperation with condition variables
  - non-copyable
  - movable
- `shared_lock<SharedMutex>`
  - shared lock manager for reading variables
  - several threads can share a lock on `SharedMutex`
  - another thread may acquire an exclusive lock on `SharedMutex` through the `unique_lock`
  - the same properties as `unique_lock`
- `scoped_lock<Mutexes...>`
  - locks multiple mutexes
  - prevents deadlock
  - the constructor locks all mutexes in a safe order
  - the destructor unlocks them in reversed order
  - non-copyable

TASK 3: SECURING THE PROGRAM WITH APPROPRIATE LOCKS

```
#include <vector>
#include <iostream>
#include <thread>
using namespace std;
vector<int> numbers = {};

int getNextValue() {
    static int i = 0;
    return i+=10;
}

void read(int index) {
    for(int i = 0; i < 10; i++) {
        int value = numbers[index];
        cout << value << " ";
    }
}

void write() {
    for(int i = 0; i < 3; i++) {
        int newValue = getNextValue();
        numbers.emplace_back(newValue);
    }
}
```

- Use shared and / or regular locks
- Use the appropriate lock managers
- Compile with TSan

```
$> g++ 03_shared_mutex.cpp -lpthread
   -std=c++17 -fsanitize=thread
```

TIPS

- Shared locks are used when reading data
  - `shared_lock`
- You must obtain an exclusive lock to write data
  - `lock_guard`
  - `unique_lock`

TASK 3: PROPOSED SOLUTION

- What problems can this code encounter?
- How can this solution be improved?

```
// #includes ...
using namespace std;
vector<int> numbers = {};
shared_mutex numbersMtx;
mutex coutMtx;

int getNextValue() {
    static int i = 0;
    return i+=10;
}
void read(int index) {
    for(int i = 0; i < 10; i++) {
        shared_lock<shared_mutex> lock{numbersMtx};
        int value = numbers[index];
        lock.unlock();
        lock_guard<mutex> coutLock(coutMtx);
        cout << value << " ";
    }
}
void write() {
    for(int i = 0; i < 3; i++) {
        lock_guard<shared_mutex> lock(numbersMtx);
        int newValue = getNextValue();
        numbers.emplace_back(newValue);
    }
}
```

```
int main() {
    vector<thread> writers;
    for(int i = 0; i < 10; i++)
        writers.emplace_back(write);

    cout << "Writers produced: ";
    for(const auto & n : numbers)
        cout << n << " ";
    cout << endl;

    cout << "Readers consumed: ";
    vector<thread> readers;
    for(int i = 0; i < 10; i++)
        readers.emplace_back(read, i);

    for(auto && writer : writers)
        writer.join();
    for(auto && reader : readers)
        reader.join();

    cout << endl;
    return 0;
}
```

TASK 3: PROBLEMS IN THE PROPOSED SOLUTION

- Is vector `numbers` always filled in before readers read it?
  - First, we create writers
  - `join()` is called on writers first
  - But that doesn't guarantee anything
  - Use of `at()` will throw an exception. How do we want to handle it?
  - You can reserve memory in the vector in advance, but what is the default behavior?
  - There is no unambiguous answer, you have to accept some convention or ...
  - You can use a synchronized queue (covered in another training)
- No need to lock `cout` if there is a single << operation
  - With the proper design, we do not have to use locks at all (so-called lock-free programming), but it is absolute expert knowledge
  - We gain in efficiency, but we lose in program flexibility
  - Any change of the lock-free code must be preceded by a reliable code-review of many experts

C++ MEMORY MODEL

CODERS SCHOOL

MEMORY MODEL

- The smallest memory unit is 1 byte
- Each byte has a unique memory address
- Synchronization is not needed if you are writing something concurrently to different areas of memory

```
vector<int> v(10);
thread t1([&]{ v[0] = 5; });
thread t2([&]{ v[1] = 15; });
```

- Synchronization is needed when you are writing something concurrently to the same memory areas
- Synchronization is needed if at least one thread is writing and others are reading the same area of memory
- No synchronization when required == race == undefined behavior
- `const` implies multi-threaded security

C ++ Memory model on cppreference.com

IS SYNCHRONIZATION NEEDED HERE?

```
struct S {
    char a;
    int b;
} obj;
thread t1([&]{ obj.a = 'a'; });
thread t2([&]{ obj.b = 4; });
```

- Race is possible in old POSIX threads
- From C++11, the code is valid and does not need to be synchronized
- Despite the same structure, the memory areas in which we save data are disjoint

IS SYNCHRONIZATION NEEDED HERE?

```
vector<int> v(10, 0);
for (int = 0; i < 10; i++)
   thread t([&]{ v[i] = i; });
```

- YES
- There is a race on variable `i`
- But the vector access is race free
- Despite the same structure, the memory areas in which we save data are disjoint
- Proper version that do not require synchronization:

```
vector<int> v(10, 0);
for (int = 0; i < 10; i++)
   thread t([&, i]{ v[i] = i; });
```

IS SYNCHRONIZATION NEEDED HERE?

```
vector<int> v;
for (int = 0; i < 10; i++)
    thread t([&]{ v.emplace_back(i); });
```

- YES
- We have to increment the `end()` iterator while we are adding new elements - possible race condition
- When adding a new object, the vector may be reallocated. Some threads may have iterators on the deprecated vector position.

WAYS OF SYNCHRONIZATION

std::mutex

```
int a = 0;
mutex m;
thread t1([&]{
    lock_guard<mutex> lg(m);
    a = 1;
});
thread t2([&]{
    lock_guard<mutex> lg(m);
    a = 2;
});
```

std::atomic<T>

```
atomic<int> a = 0;
thread t1([&]{ a = 1; });
thread t2([&]{ a = 2; });
```

TYPE `std::atomic<T>`

- `#include <atomic>`
- `std::atomic`
- Light synchronization
- Allows simple arithmetic and bitwise operations: ++, -, + =, - =, & =, | =, ^ =
- Typically: numbers, pointers
- The most important operations:
  - `operator=()` - saves a value in an atomic variable
  - `store()` - saves a value in an atomic variable, you can also specify `std::memory_order`
  - `operator T()` - reads a value from an atomic variable
  - `load()` - reads a value from an atomic variable, you can also specify `std::memory_order`

`std::atomic<T>` will be covered in details in a separate training course

TASK: SYNCHRONIZATION

`exercises/05_synchronization.cpp`

- Use proper synchronization mechanisms
  - `std::mutex` + locks
  - `std::atomic`
- Choose proper places for locking
- Avoid unnecessary locking

GOOD PRACTICES

CODERS SCHOOL

HOW TO LOCK?

Never use bare `lock()/unlock()` on mutex

Always use a mutex with the appropriate manager (RAII wrapper):

- `lock_guard` - the simplest, main choice
- `unique_lock` - delayed blocking, time-limited blocking attempts, ...
- `shared_lock` - shared lock for reading variables
- If threads need to block multiple mutexes, they should always be blocked in the same order on every thread that uses them. This prevents deadlocks.
- `scoped_lock` - blocks several mutexes to prevent deadlock
- Each RAII wrapper must be named! Unnamed guards are r-values that are destroyed automatically in the next line

WHEN TO LOCK?

- Whenever necessary, but...
- Avoid locking wherever possible
- The duration of the lock should be as short as possible
- If there is a lot of data to be modified and they are not arranged next to each other in memory, the duration of the mutex lock can be very long so that all data can be loaded to the cache memory.
- I/O operations such as reading and writing files, downloading data via the network or generally calling a system functions are long processes

WHERE TO LOCK?

- The mutex should exist along with the locking object it is used for
- If a mutex locks access to a class field, it should exist in that class
- M&M rule - mutable & mutex go together (in classes)
- Each resource should have its own mutex, as this simplifies the locking logic
- In really rare cases, one mutex can block several objects, but you need to think very carefully about the logic and all timing

WHAT IS THE PROBLEM WITH THE CODE BELOW?

```
#include <thread>
#include <mutex>
using namespace std;

class X {
    mutable mutex mtx_;
    int value_ = 0;
public:
    explicit X(int v) : value_(v) {}

    bool operator<(const X & other) const {
        lock_guard<mutex> ownGuard(mtx_);
        lock_guard<mutex> otherGuard(other.mtx_);
        return value_ < other.value_;
    }
};

int main() {
    X x1(5);
    X x2(6);
    thread t1([&](){ x1 < x2; });
    thread t2([&](){ x2 < x1; });
    t1.join();
    t2.join();
    return 0;
}
```

t1 thread	t2 thread
`x1 < x2`	`x2 < x1`
x1: mtx_.lock() by ownGuard	x2: mtx_.lock() by ownGuard
x2: mtx_.lock() by otherGuard - waiting	x1: mtx_.lock() by otherGuard - waiting

DEADLOCK

The deadlock occurs randomly on some executions

DEADLOCK

A situation where at least two different threads are waiting for each other, so neither can proceed

TASK: DEADLOCK IN COMPARISON

```cpp
#include <thread>
#include <mutex>
using namespace std;

class X {
    mutable mutex mtx_;
    int value_ = 0;
public:
    explicit X(int v) : value_(v) {}

    bool operator<(const X & other) const {
        lock_guard<mutex> ownGuard(mtx_);
        lock_guard<mutex> otherGuard(other.mtx_);
        return value_ < other.value_;
    }
};

int main() {
    X x1(5);
    X x2(6);
    thread t1([&](){ x1 < x2; });
    thread t2([&](){ x2 < x1; });
    t1.join();
    t2.join();
    return 0;
}
```

- The deadlock occurs randomly on some executions
- Use `std::scoped_lock` to fix the deadlock

```
$> g++ 04_deadlock.cpp -lpthread -fsanitize=threa
$> ./a.out
...
WARNING: ThreadSanitizer: lock-order-inversion
(potential deadlock) (pid=5509)
Cycle in lock order graph: M6
(0x7fffac4d4430) => M7 (0x7fffac4d4460) => M6
...
```

SOLUTIONS: DEADLOCK IN COMPARISON

ORIGINAL EXAMPLE WITH DEADLOCK

```
bool operator<(const X & other) const {
    lock_guard<mutex> ownGuard(mtx_);
    lock_guard<mutex> otherGuard(other.mtx_);
    return value_ < other.value_;
}
```

SCOPED LOCK - PREFERRED SOLUTION

```
bool operator<(const X & other) const {
    scoped_lock bothLock(mtx_, other.mtx_);
    return value_ < other.value_;
}
```

`std::scoped_lock` (C ++ 17)

- requires passing mutexes in the constructor, which it tries to lock immediately
- does not require creating additional lock objects

SOLUTIONS: DEADLOCK IN COMPARISON

DEFER UNIQUE_LOCK + LOCK

```
bool operator<(const X & other) const {
    unique_lock<mutex> l1(mtx_, defer_lock);
    unique_lock<mutex> l2(other.mtx_, defer_lock);
    lock(l1, l2);
    return value_ < other.value_;
}
```

LOCK + ADOPT LOCK_GUARD

```
bool operator<(const X & other) const {
    lock(mtx_, other.mtx_);
    lock_guard<mutex> l1(mtx_, adopt_lock);
    lock_guard<mutex> l2(other.mtx_, adopt_lock);
    return value_ < other.value_;
}
```

`std::lock()`

- guarantees that all mutexes will be blocked without deadlock, regardless of the order in which they are obtained
- requires that delayed locks are passed as parameters - `std::unique_lock` with `defer_lock`
- alternatively it requires passing mutexes and then creating locked locks `std::lock_guard` with `adopt_lock`
- l1 and l2 do not block mutexes in constructors, the `std::lock()` function does block

DEADLOCK

- Can occur when we have 2 or more mutexes blocked in different orders
- Block mutexes everywhere in the same order, but manual reordering does not always fix the problem (as in the example with the `operator<`)
- Use `std::scoped_lock` - a lock that takes multiple mutexes and always locks them in a certain (correct) order

RECAP

CODERS SCHOOL

POST-WORK

- Game - The Deadlock Empire
- Implementation - The dining philosophers problem (next slide)
- Post-test
- Training evaluation

HOMEWORK

Implement the dining philosophers problem using threads and mutexes.

- Each philosopher is controlled by a separate thread
- Each fork is protected by 1 mutex
- Try to visualize the problem
- Beware of deadlocks
- Play around with the number of philosophers and see if the behavior of the program changes
- Additional task: prevent starvations. This is a situation where at least 1 thread will never get all the required resources due to implementation or its lower priority.

WHAT WAS NEW FOR YOU?

WHAT WAS THE MOST DIFFICULT FOR YOU?

SUMMARY

- Sharing read-only data is secure and does not require locking. Word `const` guarantees data security.
- When a mutex is locked in a thread, any other thread wishing to lock it must wait for unlocking. Blocking too large critical sections is a mistake.
- The deadlock problem occurs when 2 mutexes (or more) may be blocked in a different order by different threads, or we have 1 mutex that will not be unlocked (eg. due to an exception) and the program will not call `unlock()` function.
- Mutexes slow down the program execution, but keep your data safe, if used correctly. The main cost is the waiting time on `lock()` function. Mutex lock/unlock operations are delegated to kernel-space, but usually, their cost may be neglected.
- In small programs, mutexes can be kept as global objects, but in larger programs, they should be kept in an appropriate class whose operations are to be blocked. If such a class has several resources that can be modified independently, then you should have several mutexes (1 for each resource). You have to watch out for deadlocks.
- Despite the blocked mutexes, if we use pointers or references to objects somewhere outside the critical section, all protection is for nothing. Therefore, data protection with mutexes requires careful interface modelling.
- Always use mutexes via lock managers (RAII wrappers)
- Use `std::atomic` (atomic variables). They provide a convenient way to lightly synchronize data without the use of mutexes, but more on that will be covered in the next training

USEFUL LINKS

- What every programmer should know about memory - Ulrich Drepper
- The C ++ memory model
- The dining philosophers problem

CODERS SCHOOL