CS4414 Recitation 5

multi-threading II

09/26/2025

Alicia Yang, Shouxu Lin

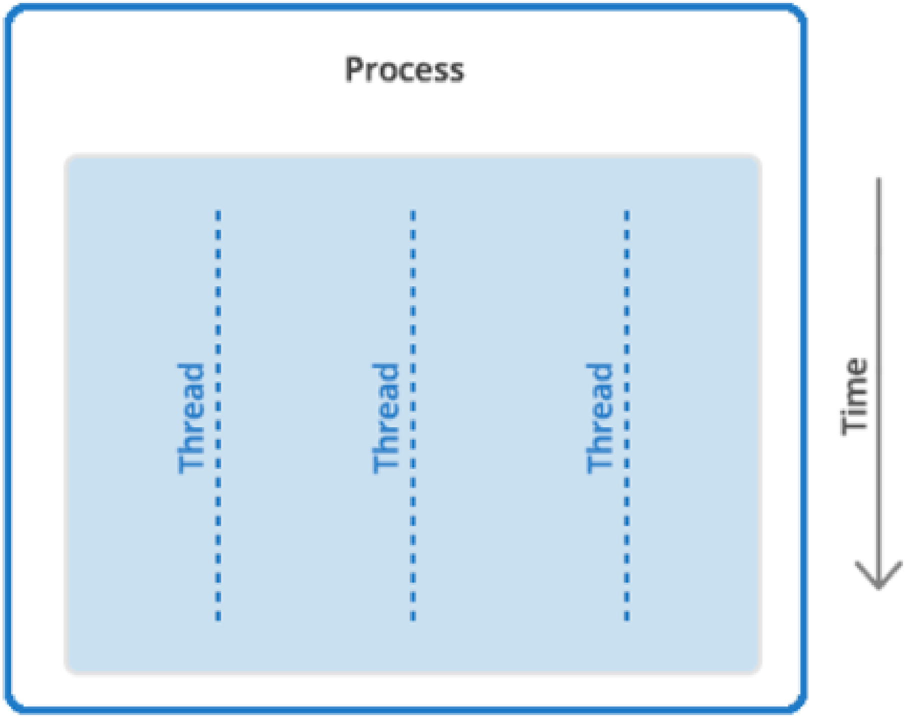
Overview

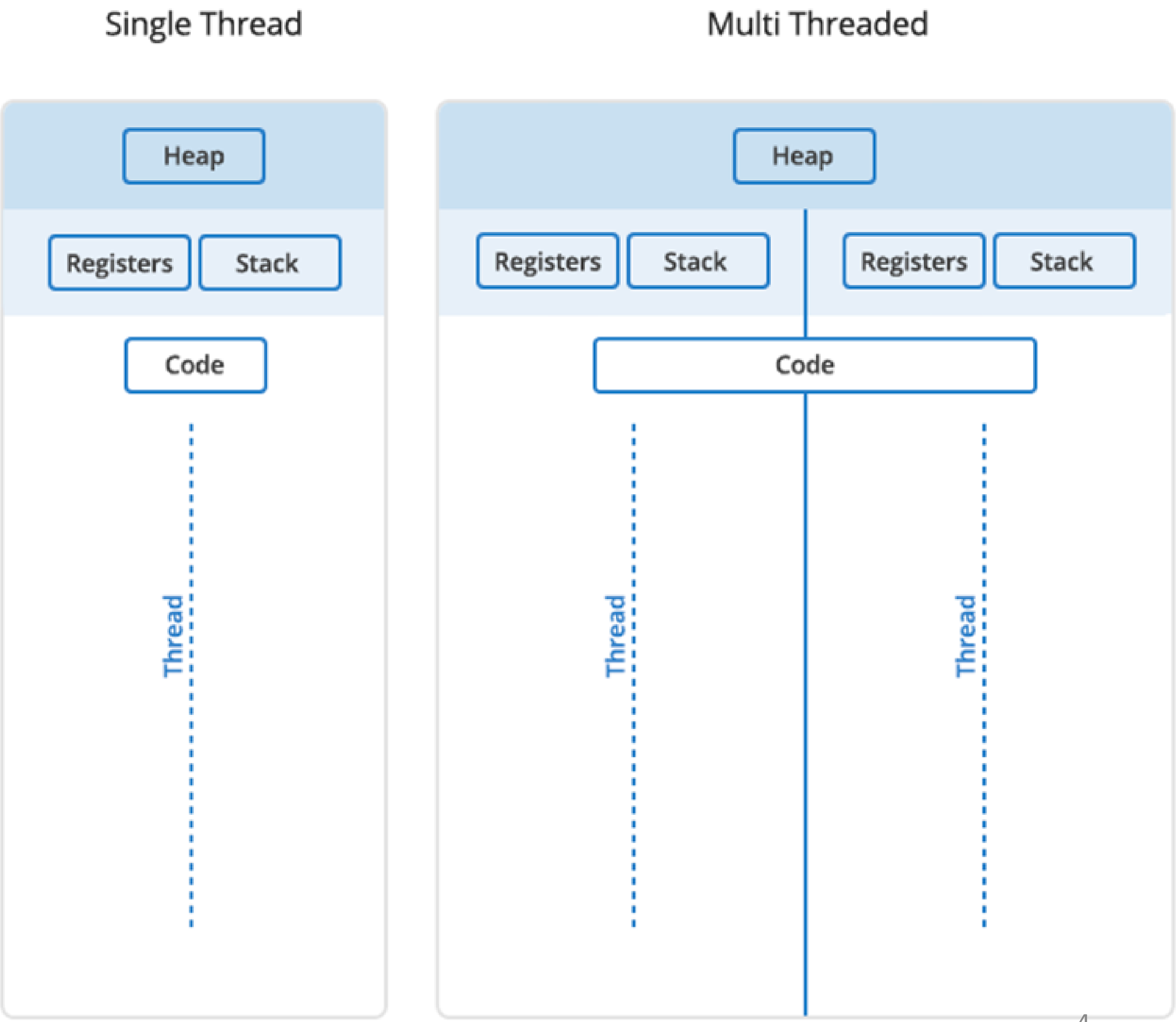
- Multithreading
  - Thread finishing
  - Race condition
  - Thread safety
    - std::atomic
    - Mutex locks
    - RAII locks

Recap

- What is concurrency
- Threads launching
- Thread finishing
  - join()

Concurrency





https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/

Launching thread (via std::thread)

- Create a new thread object.
- Pass the executing code to be called (i.e, a callable object) into the constructor of the thread object.
- Once the object is created a new thread is launched, it will execute the code specified in callable

```
#include <thread>   // part of the C++ Standard Library
```

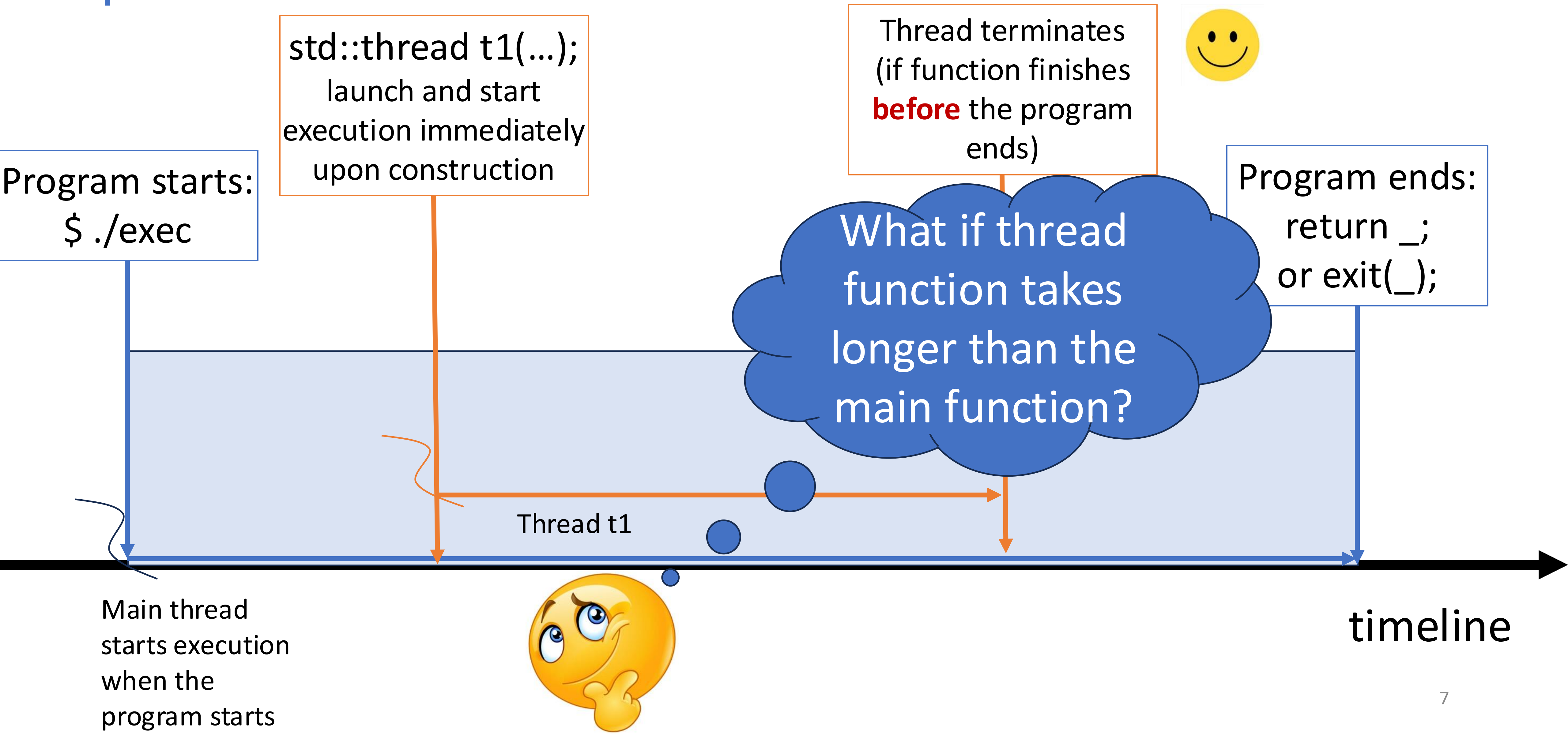
Launching thread

- Launching a thread using function pointers and function parameters

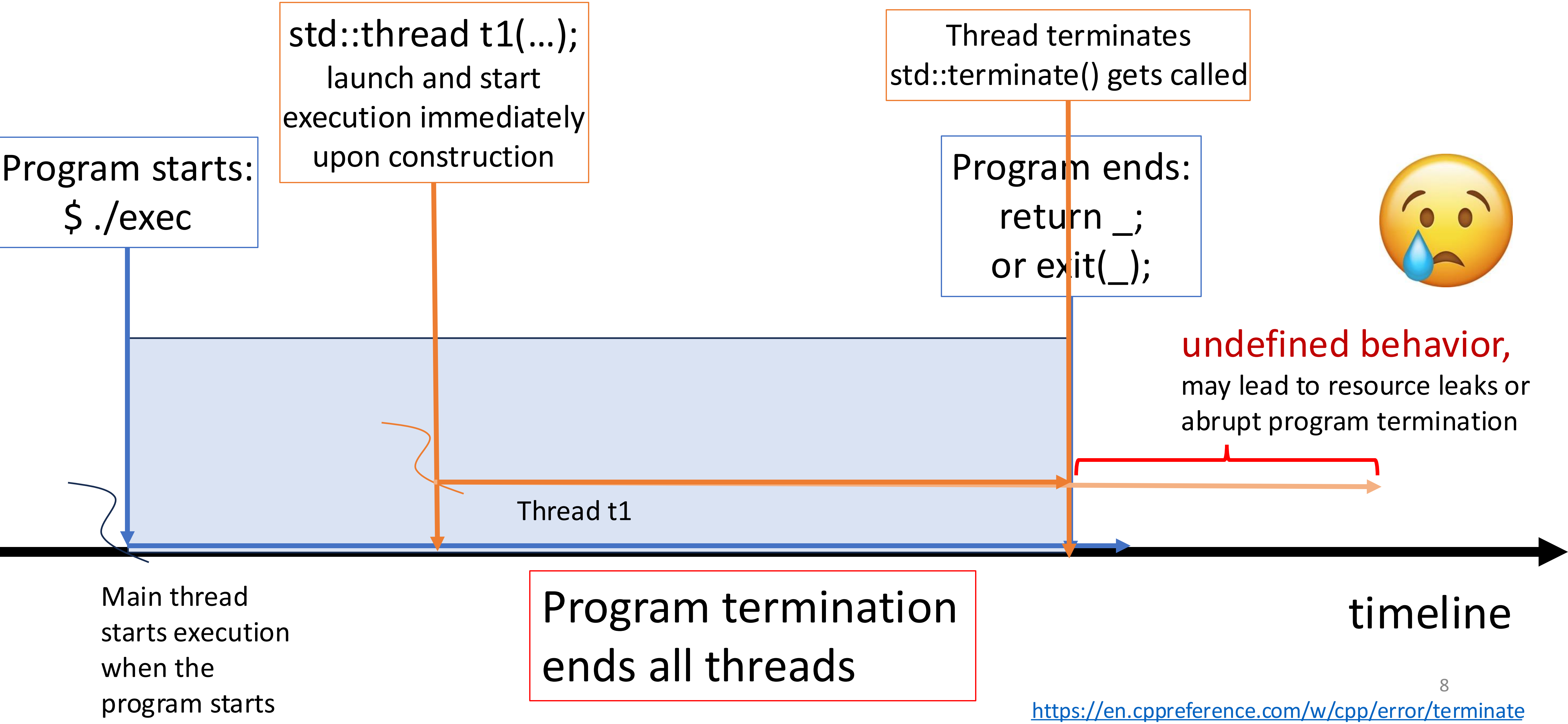
```
void func(params)
{
   // Do something
}

std::thread thread_obj(func, args);
```

demo

Thread lifecycle and program termination



Thread lifecycle and program termination



https://en.cppreference.com/w/cpp/error/terminate

Multithreading

- Launching a thread:
  - Function pointer
  - Function object
  - Lambda function
- Managing threads
  - Join()

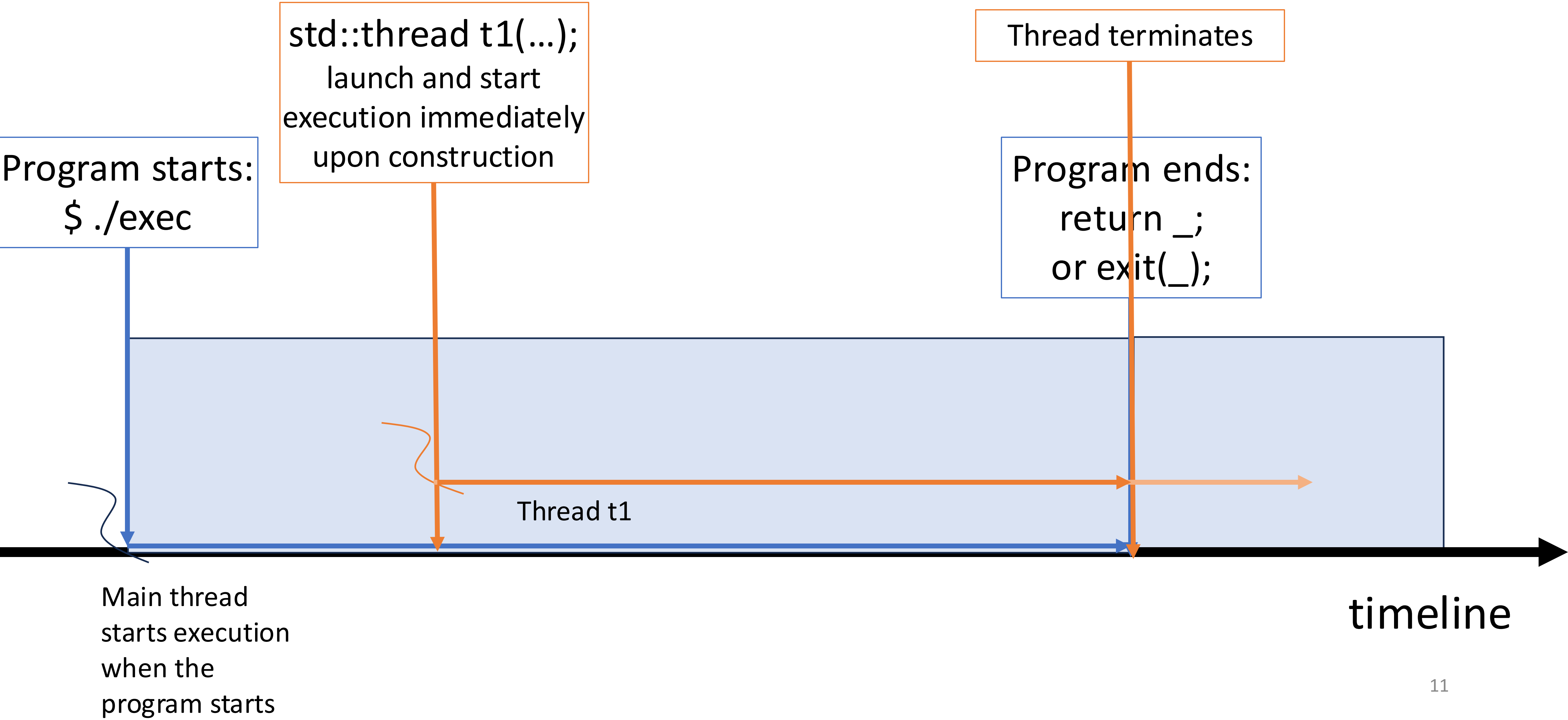
Joining threads with std::thread

demo

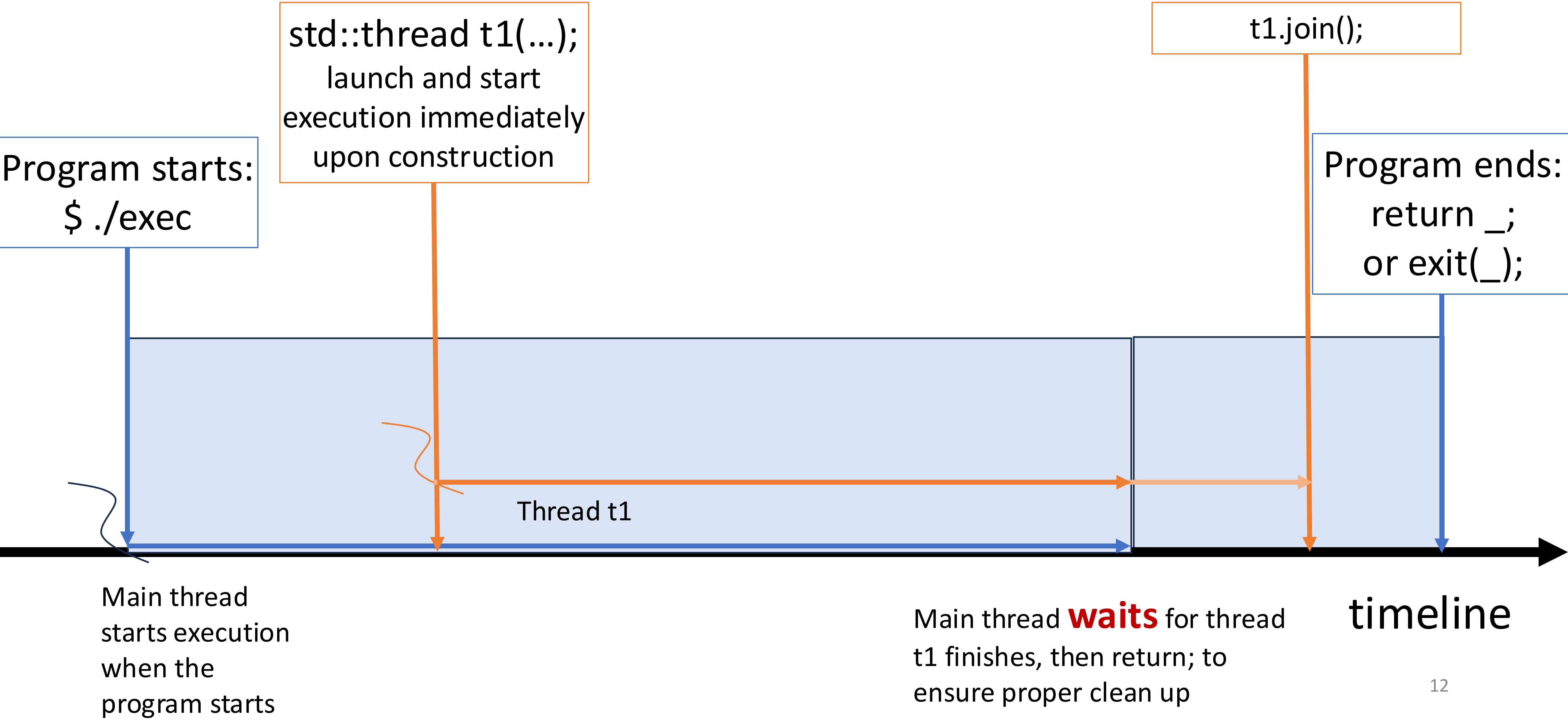
```
std::thread thread_obj(func, params);
Thread_obj.join();
```

- **Wait** for a thread to complete
- Ensure that the thread was **finished before** the function was **exited**
- **Clean up** any storage associated with the thread
- join() can be called only **once for a given thread**

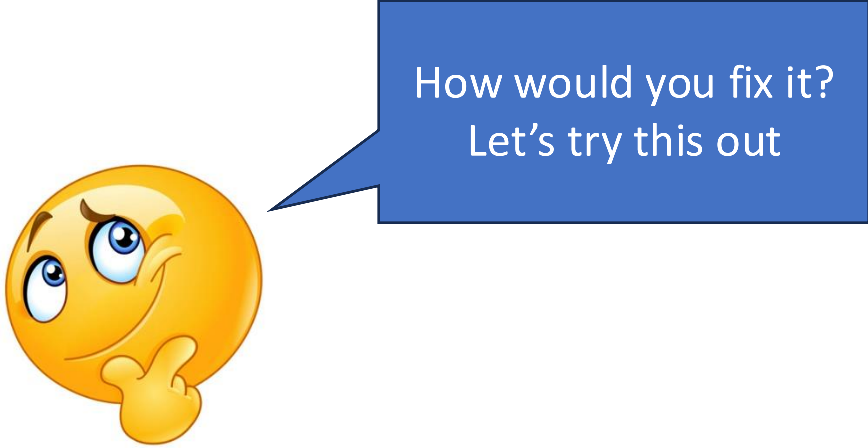
Thread lifecycle and program termination



Thread lifecycle and program termination



Exercise: would this code work?




```
std::thread foo(){
    std::vector<int> a = {1,2,3,5};
    std::thread threadObj([&](){
        for (int i : a){
            std::cout << i << std::endl;
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
     });
     return threadObj;
}

int main(){
    std::thread obj= foo();

    std::cout << "Back to main function" << std::endl;

    return 0;
}
```

Thread safety

Sharing data among threads

---race condition

- Race condition:
  - The situation where the **outcome depends** on the **relative ordering** of execution of operations on two or more threads; the threads **race** to perform their respective operations.

Code source: https://github.com/aliciayuting/CS4414Demo.git

Sharing data among threads

---race condition

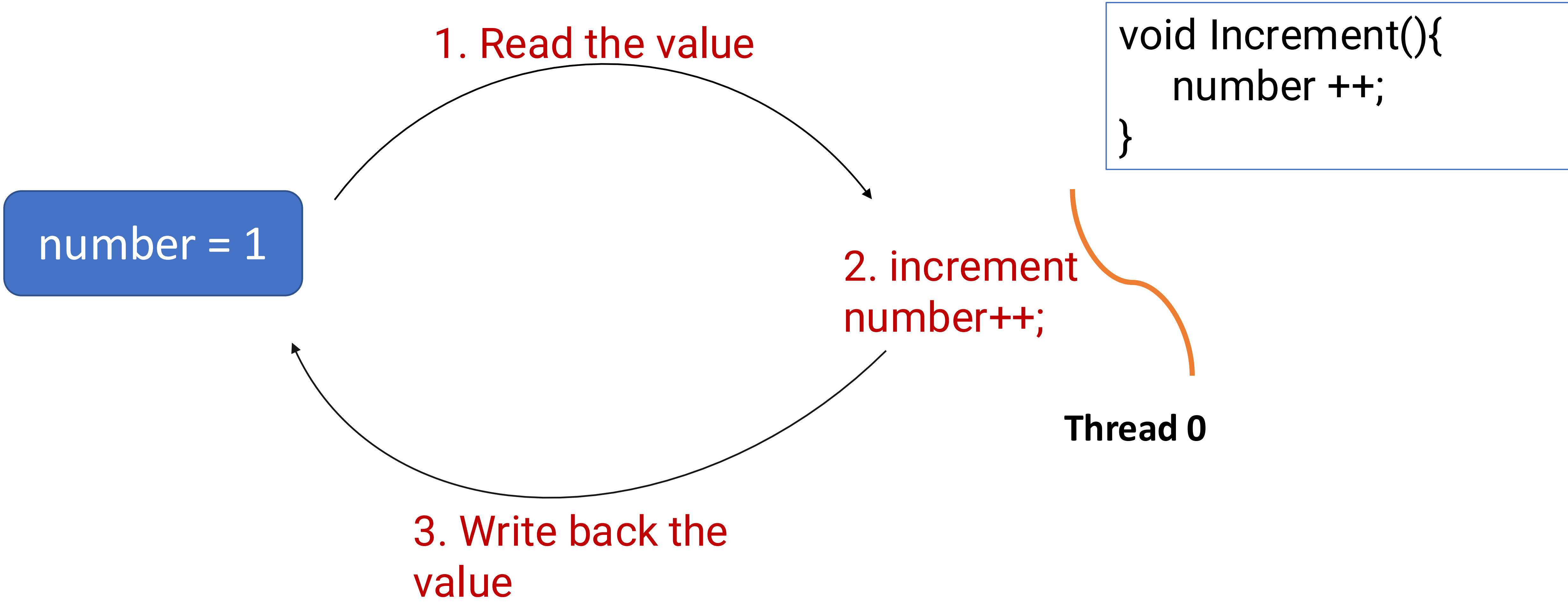
demo

- Example: Concurrent increments of a shared integer variable.
  - Each thread shares an integer called count initialized to 0, increments it 1 million times concurrently without any synchronization

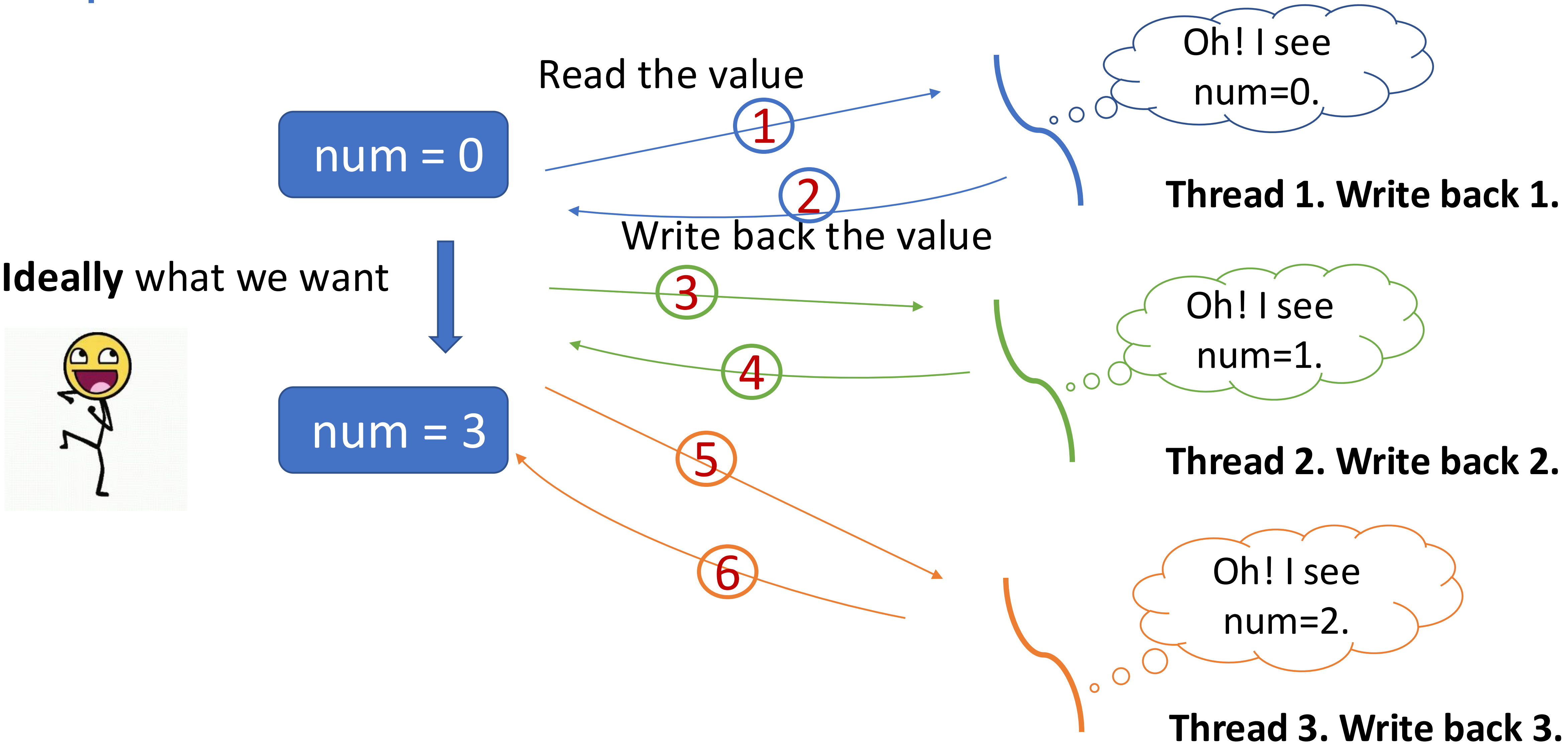
Number of threads	Final value
1	1000000
2	1059696
3	1155035
4	1369165

Code source: https://github.com/aliciayuting/CS4414Demo.git

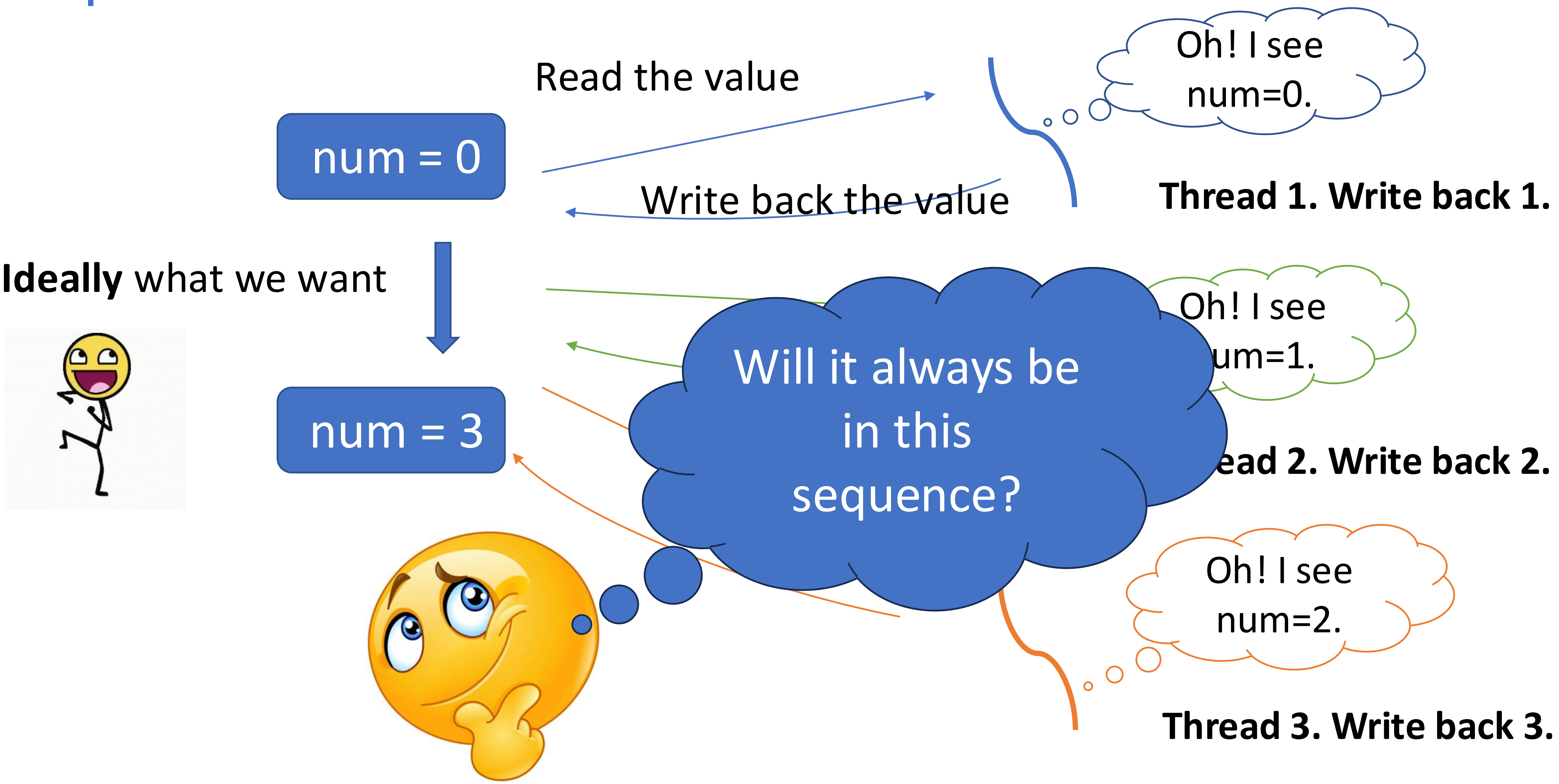
Example: Concurrent increments of a shared integer variable



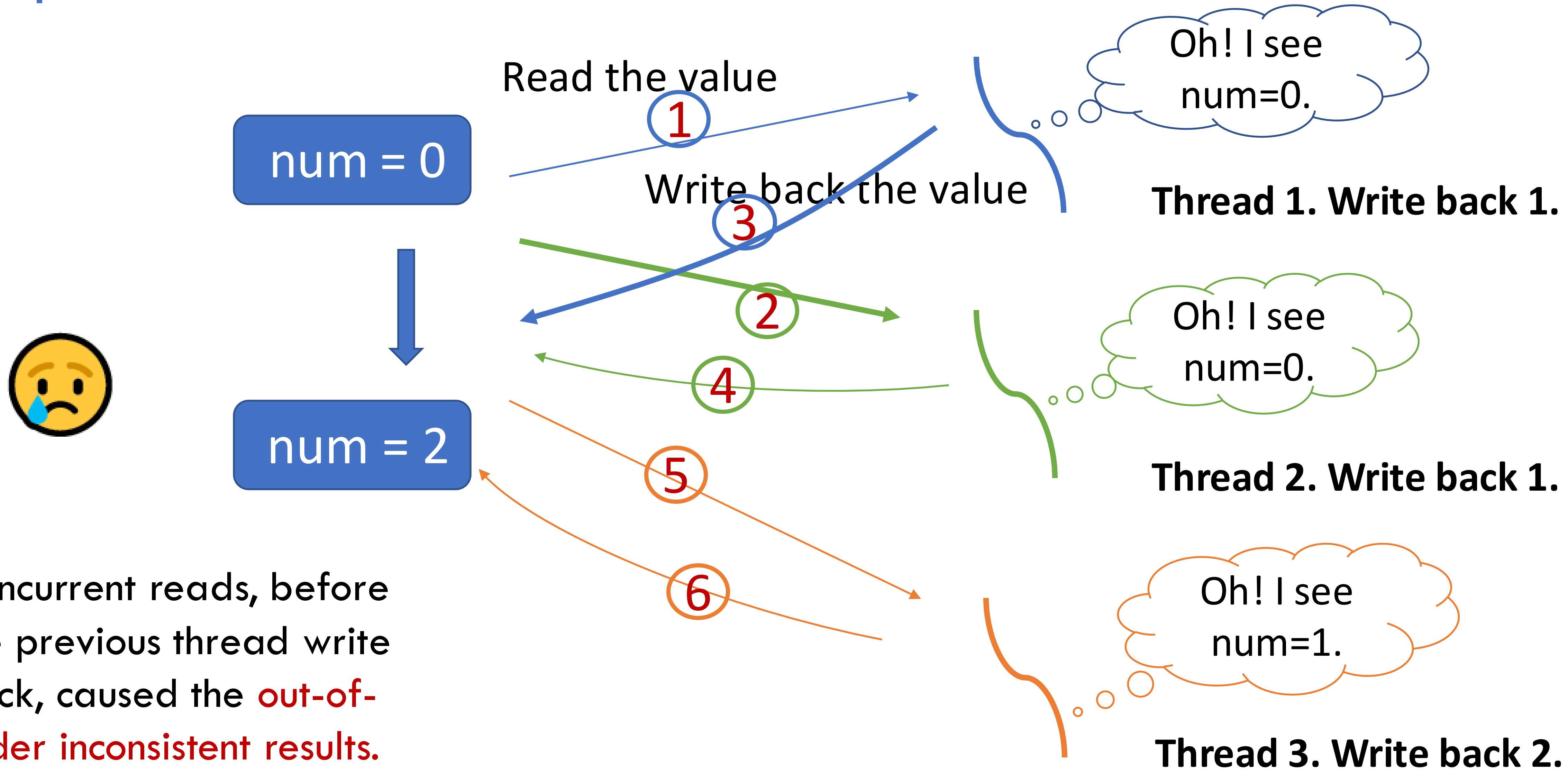
Example: Concurrent increments of a shared integer variable



Example: Concurrent increments of a shared integer variable



Example: Concurrent increments of a shared integer variable



Concurrent reads, before the previous thread write back, caused the out-of-order inconsistent results.

Thread Safety

- A function, a piece of code, or an object is **thread-safe** when it can be **invoked** or **accessed** **concurrently** by **multiple threads** **without** causing unexpected behavior, race conditions, or data corruption.

Thread safe

- Entities in C++ standard library and their thread-safety guarantees

Thread safe?

- Is integer type inherently thread-safe?
  - No, as we showed just now

std::atomic

- A template that defines an **atomic** type.

`template< class T > struct atomic;`	(1)	(since C++11)
`template< class U > struct atomic<U*>;`	(2)	(since C++11)
`template< class U > struct atomic<std::shared_ptr<U>>;`	(3)	(since C++20)
`template< class U > struct atomic<std::weak_ptr<U>>;`	(4)	(since C++20)

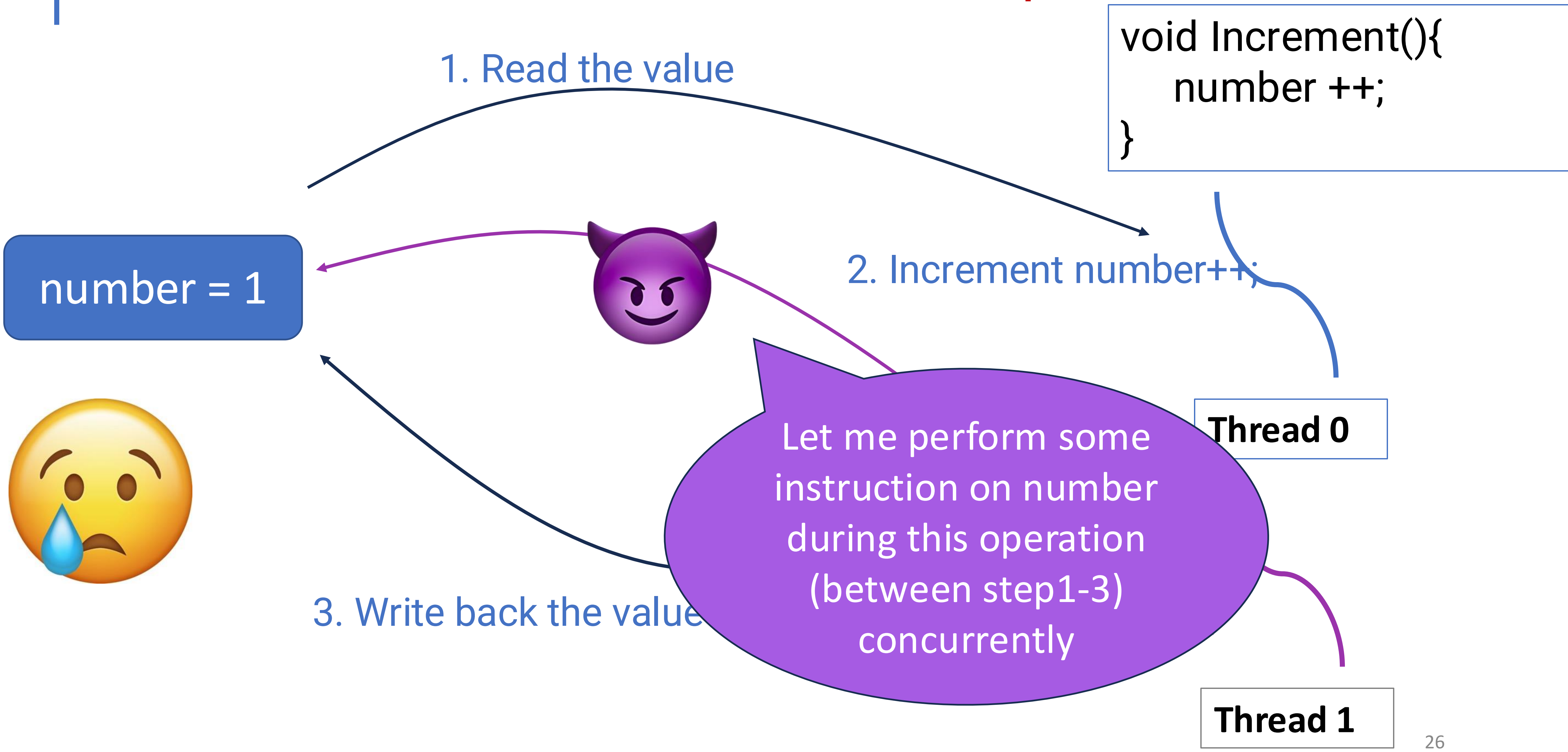
* (more at the end of recitation if have time)

https://en.cppreference.com/w/cpp/atomic/atomic

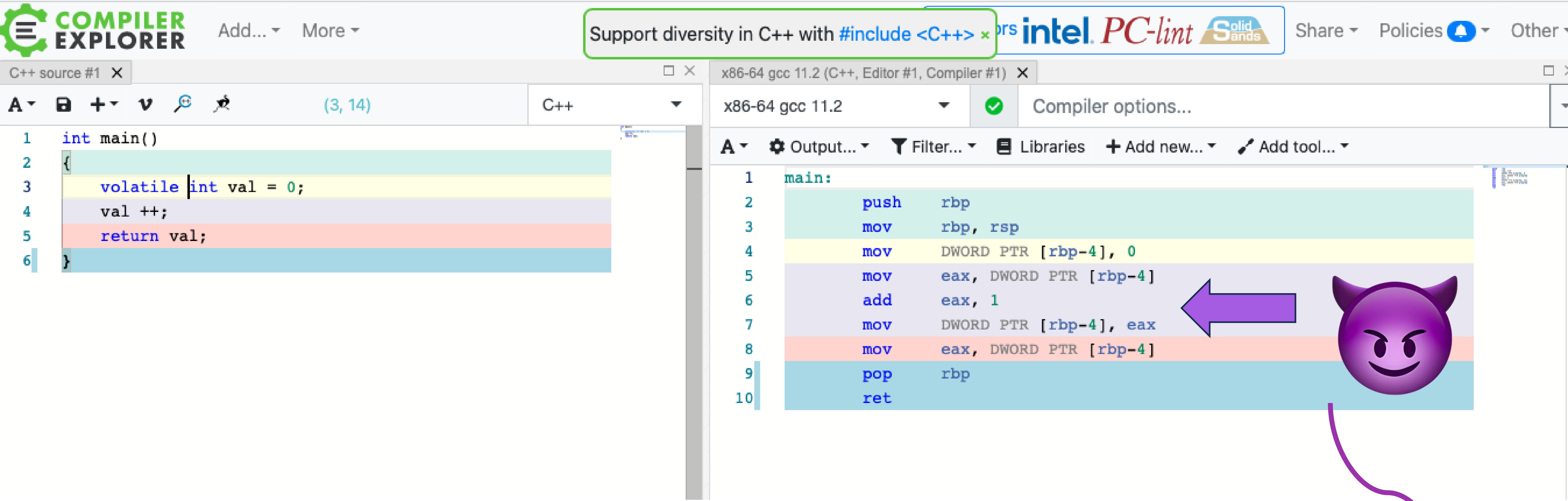
Atomic

- An atomic operation is an **indivisible operation.**
- The operation is **either done or not done**. Such an operation would **never be half-done** from any thread in the system.

Data race condition: non-atomic access pattern



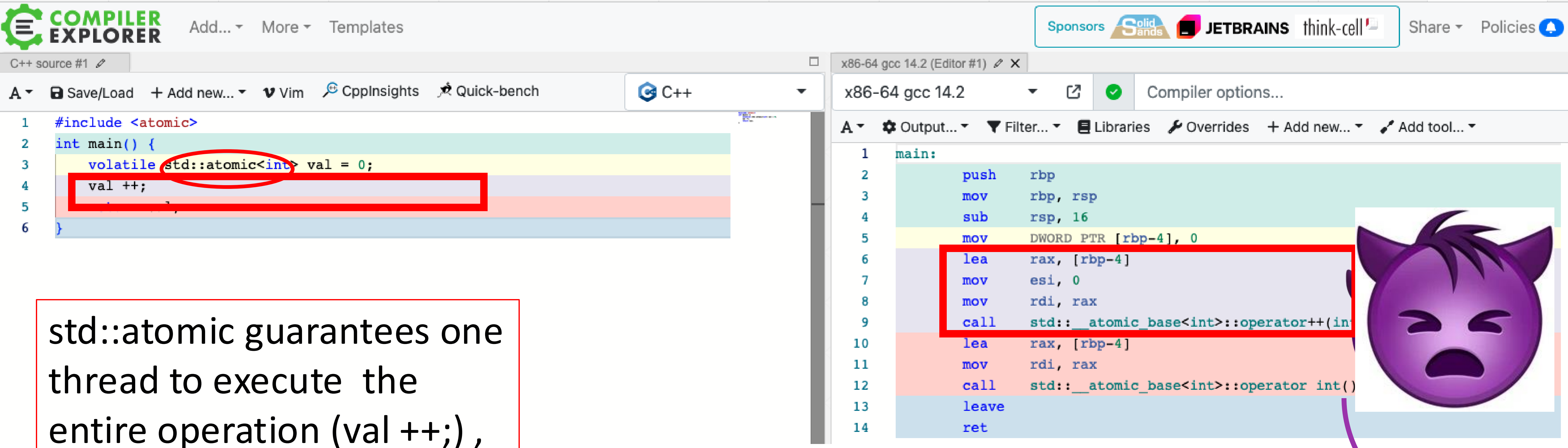
Data race condition: non-atomic access pattern




```
std::thread t1([&val]() {
        val++;
});
```

Another concurrent thread t1

Atomic access



std::atomic guarantees one thread to execute the entire operation (val ++;) , during which no other thread interfering or interrupting

```
std::thread t1([&val]() {
        val++;
});
```

Another concurrent thread t1

Atomic

- An atomic operation is an **indivisible operation.**
- std::atomic are **implemented** using hardware supports provided by modern CPU:
  - Examples of **atomic instructions**:
    - Compare-and-Swap (CAS)
    - Load-Linked/Store-conditional (LL/SC)
    - fetch_and_add (FAA)
  - **Different CPUs** provide **different sets** of **atomic instructions.** The implementation of std::atomic varies from architecture to architecture

Atomic member functions

- Atomic type: std::atomic<type>
- Constructor std::atomic<bool> x(true); std::atomic<uint32_t> y(0);
- store() x.store(false); y.store(1, std::memory_order_relaxed);

https://en.cppreference.com/w/cpp/atomic/atomic

More atomic member functions

- load() — `bool z = x.load();`
- exchange() — `uint32_t  m = y.exchange(100);`
- operator= — `y = 2;`
- operator+=, operator -= — `y += 1;`   `y.fetch_add(1);`   (since C++20)
- operator++, operator-- — `y ++;`

What about y = y + 1?

https://en.cppreference.com/w/cpp/atomic/atomic

More atomic member functions

- load() bool z = x.load();
- exchange() uint32_t m = y.exchange(100);
- operator= y = 2;
- operator+=, operator -= y += 1; y.fetch_add(1);
- operator++, operator-- y ++;

What about y = y + 1?

When multithreading, leads to **race condition**, because it involves multiple operations (read x, +1 and then assignment operation)

https://en.cppreference.com/w/cpp/atomic/atomic

Thread safe

- std::atomic
- std::shared_ptr

std::vector

- Does std::vector guarantee thread-safety?

Multithreads' data sharing with std::vector

- When is std::vector thread-safe?
  - Each thread has its own instance of std::vector (no concurrency)
  - Read-only access
- When is std::vector not thread-safe?
  - Simultaneous Read and Write
  - Concurrent modification
  - Reallocation access on reallocation or modification

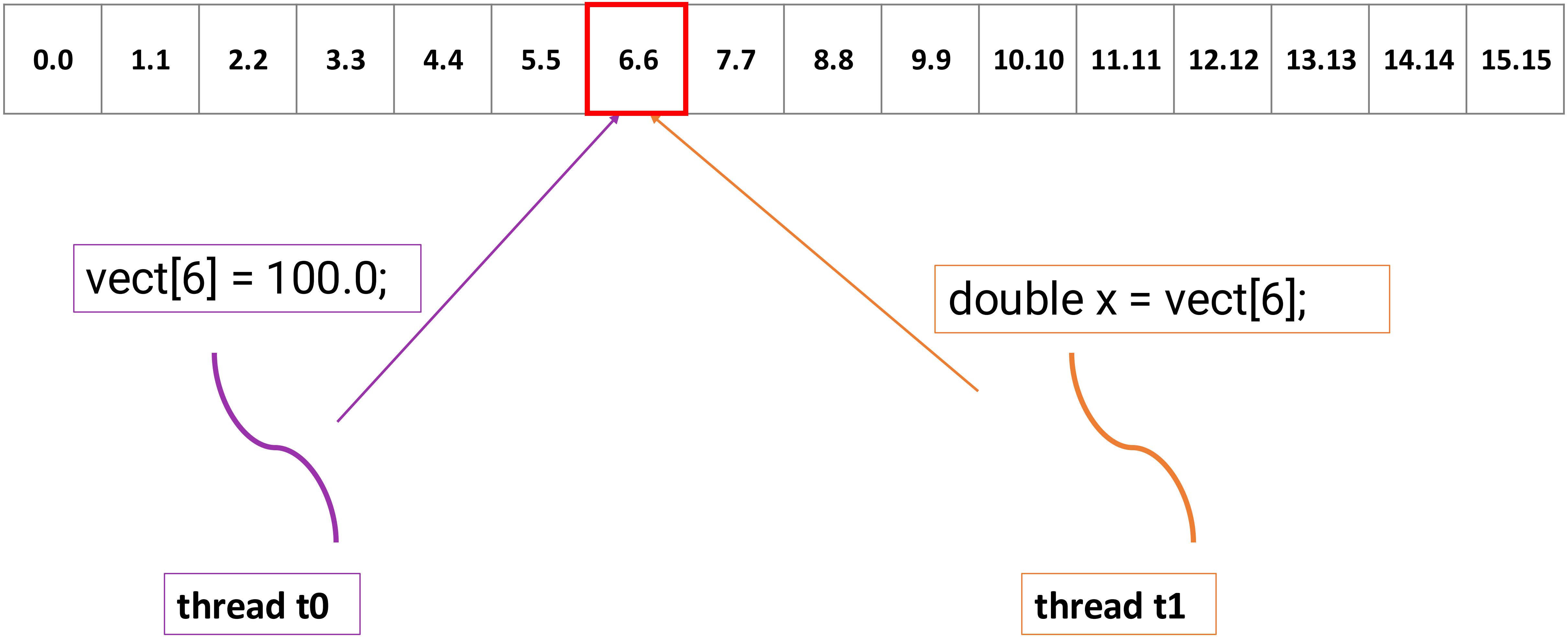
Read-only-access of std::vector

```
void read_vector(const std::vector<double>& vec, int thread_id, double& sum) {
    for (const auto& value : vec) {
        sum += value;
}}                                          // Each thread reads the vector and accumulates the sum
```

Thread safe, because only concurrent **reads**

```
int main() {
  std::vector<double> vec(100, 1.00);
  double t1_sum;
  double t2_sum;
  std::thread t1(read_vector,std::ref(vec), 1, std::ref(t1_sum));
  std::thread t2(read_vector,std::ref(vec), 2, std::ref(t2_sum));
  t1.join();
  t2.join();
  std::cout << "t1_sum="<< t1_sum << ",t2_sum=" << t2_sum;
...}
```

Simultaneous read and write



Simultaneous read and write

Concurrent Read+write to the **SAME** element is **NOT** thread-safe

0.0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.10	11.11	12.12	13.13	14.14	15.15

What if the threads are operating on different elements?

vect[6] = 100.0;

ect[6];

x could be 6.6 or 100.0 after this.

thread t1

Locking

Locking

---protecting data with mutex

- How does mutex work?
  - Before accessing a shared data structure, you lock the mutex associated with that data
  - When finished accessing the data structure, you unlock the mutex.

std::mutex

exclusive, non-recursive ownership

- A thread owns the mutex from the time when it call lock() until it calls unlock()
- The Thread Library then ensures that **once one thread** has locked a specific mutex, **all other threads** that try to lock the same mutex **have to wait** until the thread that successfully locked the mutex unlocks it.

https://en.cppreference.com/w/cpp/thread/mutex

Locking

---std::mutex::lock(), unlock()

demo

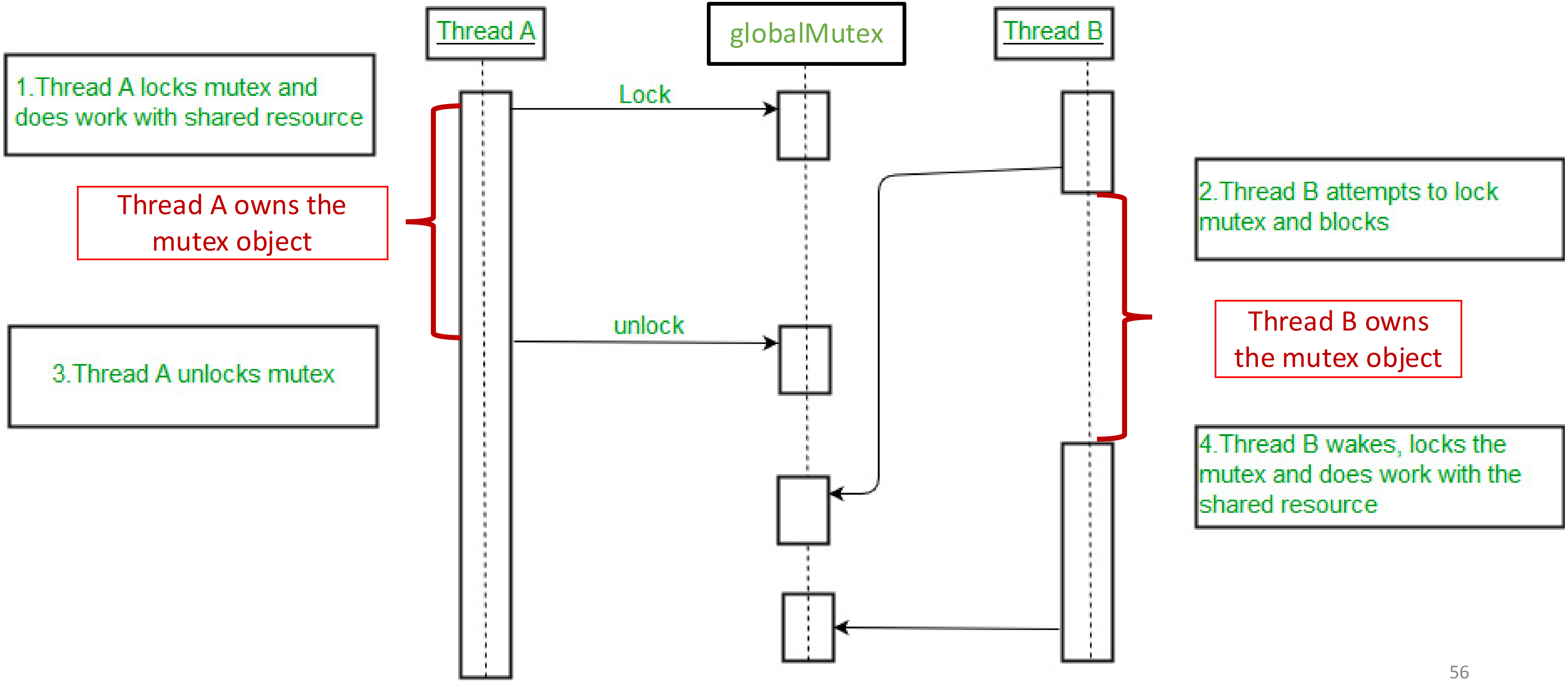
```
1   int       global_num = 0;
2   std::mutex      globalMutex;

3   void incre(int num){
4             globalMutex.lock();
5             global_num = global_num + 1;
6             globalMutex.unlock();
7   }

8   int main(){
9             std::thread threadA(incre, 10);
10            std::thread threadB(incre, 10);
11            threadA.join();
12            threadB.join();
    ...}
```

Only one thread could enter line 5 at a time

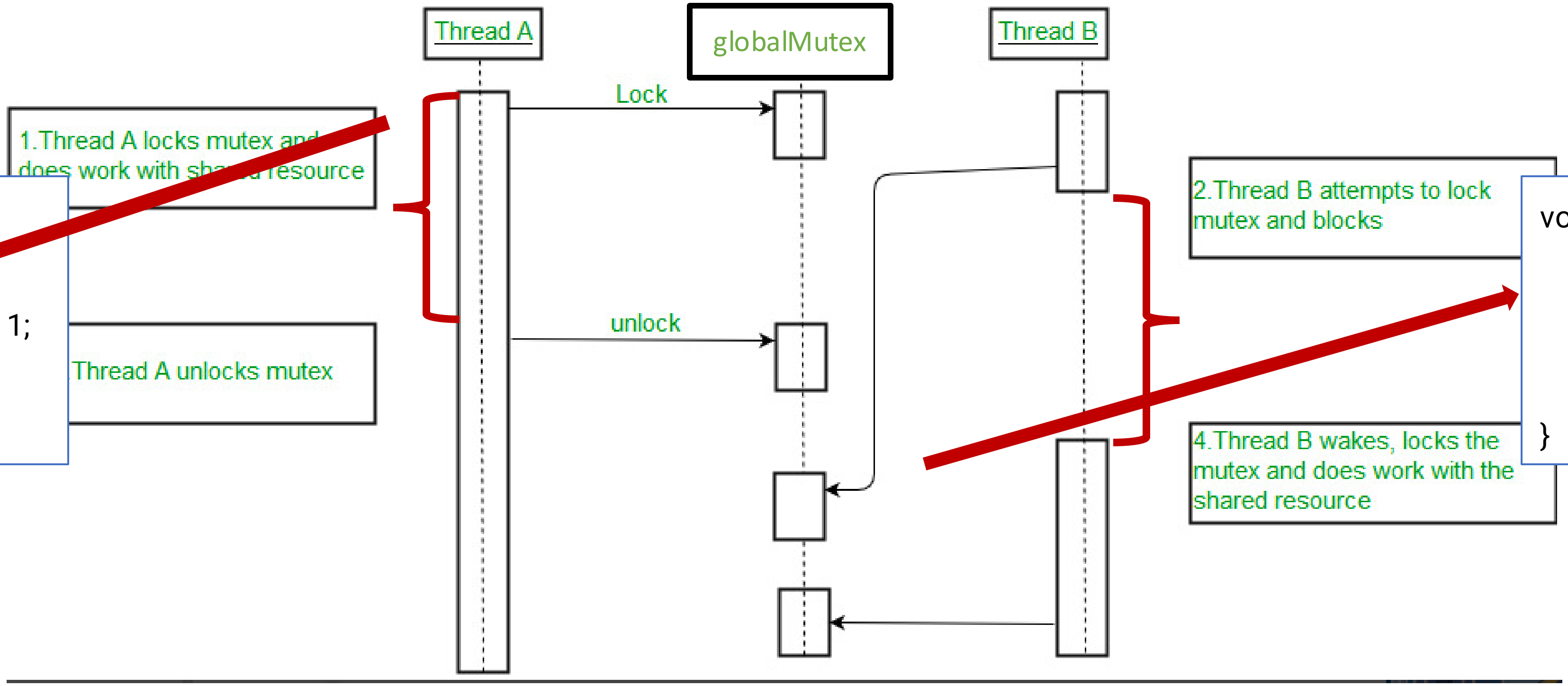
Mutex and Lock in C++

- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set by one thread, then no other thread can access the locked region of code.
- Mutex lock will only be released by the thread who locked it.

Mutex and Lock in C++



Mutex and Lock in C++




```
void incre(int num){
    globalMutex.lock();
    global_num = global_num + 1;
    globalMutex.unlock();
}
```

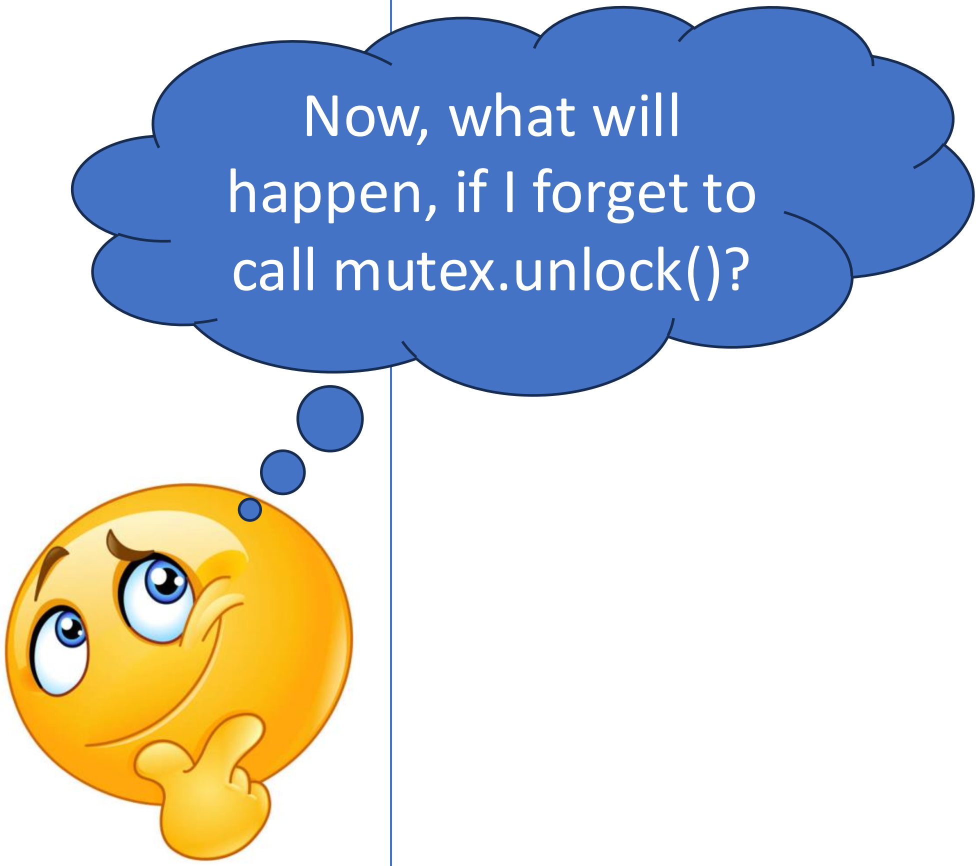
```
void incre(int num){
    globalMutex.lock();
    global_num = global_num + 1;
    globalMutex.unlock();
}
```

Locking

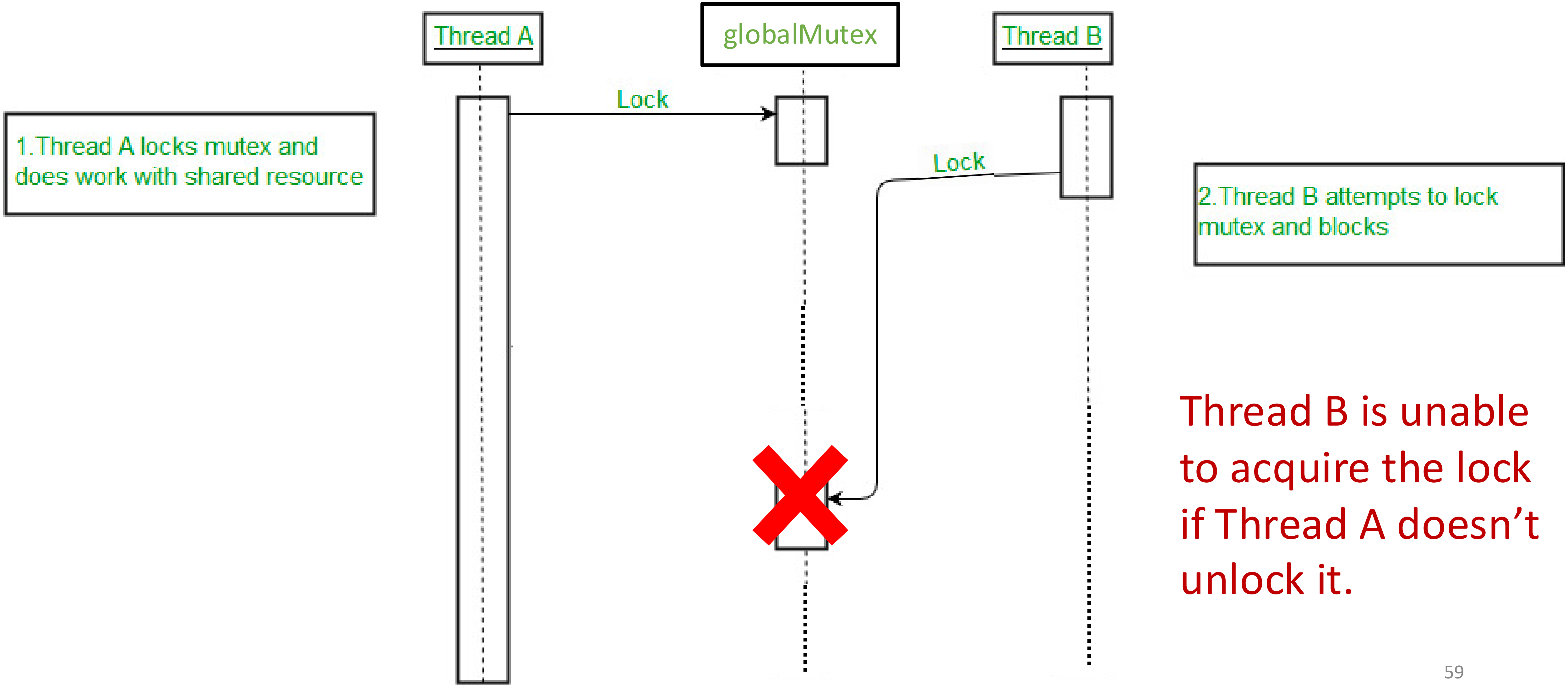
---std::mutex::lock(), unlock()

demo

```
int          global_num = 0;
std::mutex            globalMutex;

void incre(int num){
              globalMutex.lock();
              global_num = global_num + 1;
              globalMutex.unlock();
}

int main(){
              std::thread threadA(incre, 10);
              std::thread threadB(incre, 10);
              threadA.join();
              threadB.join();
}
```



Mutex and Lock in C++



Thread B is unable to acquire the lock if Thread A doesn't unlock it.

Mutex and Lock in C++

- A Mutex is a **lock** that we set **before** using **a shared resource** and **release after using it.**
- When the lock is set by one thread, then **no other thread** can access the locked region of code.
- Mutex lock could **only be released** by the **thread who locked it.**

Locking

---std::mutex::lock(), unlock()

- std::mutex::lock(), unlock()
  - It is **not recommended** practice to call lock(), unlock() directly, because this means that you have to remember to call **unlock()** on **every code path out of a function that called lock()**, including those due to exceptions.

RAII (Resource Acquisition is initialization) re-visit

- Resource acquisition must succeed for initialization to succeed:
  - In RAII, holding a resource is a class invariant is tied to object lifetime: resource allocation is done during object creation, by the constructor; while resource deallocation is done during object destruction, by the destructor.
- If there are no object leaks, there are no resource leaks.
  - The resource is guaranteed to be held between when initialization finishes and finalization starts, and to be held only when the object is alive.

RAII (Resource Acquisition is initialization)

```
// problem #1
{
   int *arr = new int[10];
}   // arr goes out of scope but we didn't delete it, we now have a memory leak 😢
```

```
// problem #2
{
    std::thread t1( [] () {
            // do some operations
    });
}             // thread t1 is created but not joined, if it goes out of scope, std::terminate is
              called,  this implementation doesn't properly handle the thread's life cycle 😢
```

```
// problem #3
Std::mutex globalMutex;
Void func() {
    globalMutex.lock();
}       // if we never unlocked the mutex(or exception occurred before unlock),
    it will cause a deadlock when other thread tries to acquire this lock 😢
```

RAII (Resource Acquisition is initialization)

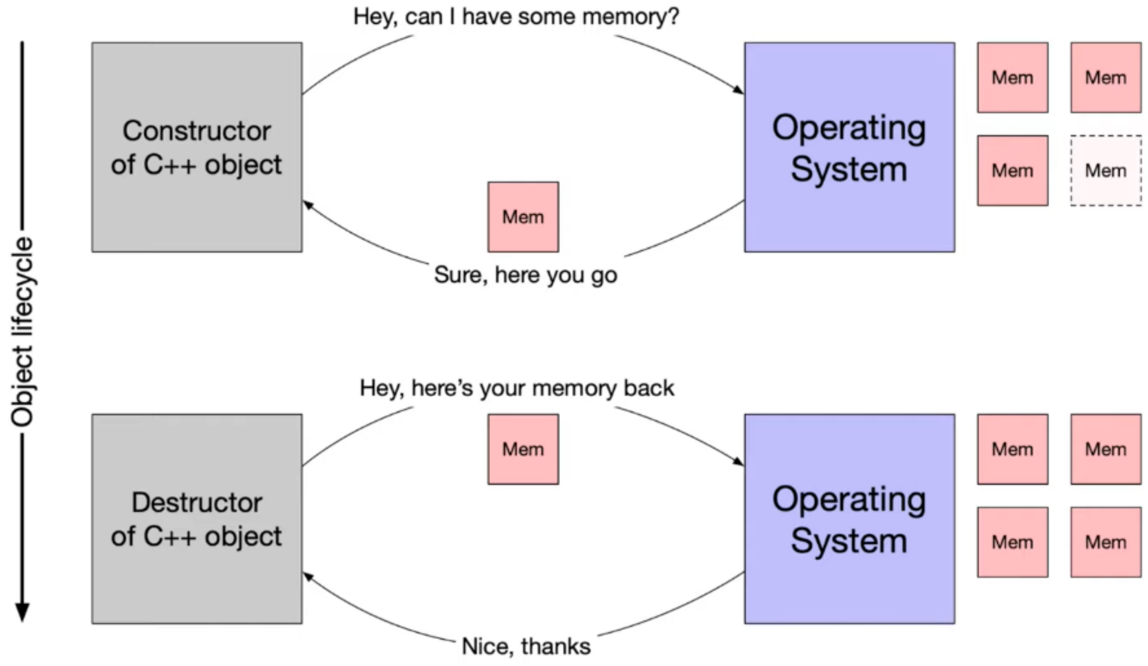
```
// problem #1's fix
{
   int *arr = new int[10];
   delete[] arr;
}
```

```
// problem #2's fix
{
    std::thread t1( [] () {
            // do some operations
    });
   t1.join();
}
```

```
// problem #3's fix
Std::mutex globalMutex;
Void func() {
    globalMutex.lock(); ….
    globalMutex.unlock();
}
```

RAII (Resource Acquisition is initialization)

- RAII
  - When acquire resources in a constructor, also need to release them in the corresponding destructor
  - Resources:
    - Heap memory,
    - files,
    - sockets,
    - mutexes



Locking

---std::mutex::lock(), unlock()

demo

```
int         global_num = 0;
std::mutex          globalMutex;

void incre(int num){
            globalMutex.lock();
            global_num = global_num + 1;
            globalMutex.unlock();
}

int main(){
            std::thread threadA(incre, 10);
            std::thread threadB(incre, 10);
            threadA.join();
            threadB.join();
}
```

Is there a better ways to manage the mutex that can automatically unlock it when not used?

Mutex and RAII locks

- std::unique_lock
- std::scoped_lock
- std::shared_lock

```
std::mutex my_mutex;
{
 std::unique_lock<std::mutex> lck(my_mutex);
    … …
}
```

```
{
 std::scoped_lock<std::mutex> lck(my_mutex);
    … …
}
```

```
std::shared_mutex shared_mutex;
{
    std::shared_lock<std::mutex> lck(shared_mutex);
    … …
}
```

std::unique_lock

- A unique lock is an **object** that **manages a mutex object** with unique ownership in both states: locked and unlocked.
- RAII: When creating a local variable of type std::unique_lock passing the mutex as parameter.
  - On construction, the object acquires a mutex object, for whose locking and unlocking operations becomes responsible.
  - This class guarantees an unlocked status on destruction (even if not called explicitly).
- Features:
  - Deferred locking, Timeout locks, adoption of mutexes, movable(transfer of ownership)

std::unique_lock

```
1  int         global_num = 0;
2  std::mutex      globalMutex;

3  void incre(int num){
4          std::unique_lock<std::mutex>  u_lock(globalMutex);
5          global_num = global_num + 1;
6          …
7  }

8  int main(){
9          std::thread t1(incre, 1);
10         std::thread t2(incre, 3);
11         t1.join();
12         t2.join();
…}
```

Only one thread could enter line 5-7 at a time

std::unique_lock

Unique_lock feature: Deferred locking

```
std::mutex mtx;
void conditional_locking(bool should_lock) {
    // Create lock but do not acquire it
    std::unique_lock<std::mutex> lock(mtx, std::defer_lock);
if (should_lock) {
    lock.lock();      // Conditionally acquire the lock
    std::cout << "Lock acquired." << std::endl;
  } else {
    std::cout << "Lock not acquired." << std::endl;
  }
}
```

```
int main() {
  std::thread t1(conditional_locking, true);
  std::thread t2(conditional_locking, false);
  t1.join();
  t2.join();
  return 0;
}
```

std::scoped_lock

a mutex wrapper which obtains access to (locks) the provided mutex, and ensures it is unlocked when the scoped lock goes out of scope

When does s_lock get released?

```
int                     global_num = 0;
std::mutex               globalMutex;

void incre(int num){
        {
                std::scoped_lock  s_lock(globalMutex);
                global_num = global_num + 1;
        }
        global_num = global_num + 1;
        ...
}
```

std::shared_lock

std::shared_lock allows for shared ownership of mutexes.

```
std::shared_mutex mtx;
int  global_val;
void print_val (int n, char c) {
   std::shared_lock<std::shared_mutex > lck (mtx);
   std::cout << global_val << std::endl;
 }
int main () {
    std::thread th1 (print_val);
    std::thread th2 (print_val);
    th1.join();
    th2.join();
```

std::shared_lock

Shared_lock allows for shared ownership of mutex. More than one thread could hold the mutex at the same time.

```
std::shared_mutex mtx;
int  global_val;
void print_val (int n, char c) {
   std::shared_lock<std::shared_mutex > lck (mtx);
   std::cout << global_val << std::endl;
 }
int main () {
    std::thread th1 (print_val);
    std::thread th2 (print_val);
    th1.join();
    th2.join();
… }
```

RAII (Resource Acquisition is initialization)

```
// problem #1
{
    int *arr = new int[10];
}       // arr goes out of scope but we didn't delete it, we now have a memory leak 😢
```

```
// problem #3
Std::mutex globalMutex;
Void func() {
    globalMutex.lock();
}               // if we never unlocked the mutex(or exception occurred before unlock),
            it will cause a deadlock when other thread tries to acquire this lock 😢
```

RAII

Better fixes

```
// problem #1's fix
{
    std::unique_ptr<int[]> arr(new int[10]);
.....
}
```

```
// problem #3's fix
Std::mutex globalMutex;
Void func() {
    std::unique_lock<std::mutex> lock(globalMutex);
....
}
```

Exercise

- How can I use the RAII class locks to implement R/W lock?
  - R/W locks allow multiple readers at the same time
  - But if there is writer, then there should be no readers, and only one writers.

Where to find the resources?

- Concurrency programing:
  - Book: C++Concurrency in Action Practice Multithreading
  - https://learn.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe
- Notes:
  - Atomic built-in: https://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Atomic-Builtins.html
  - Memory order: https://cplusplus.com/reference/atomic/memory_order/#google_vignette