CS4414 Recitation 9

multi-threading I

10/25/2024

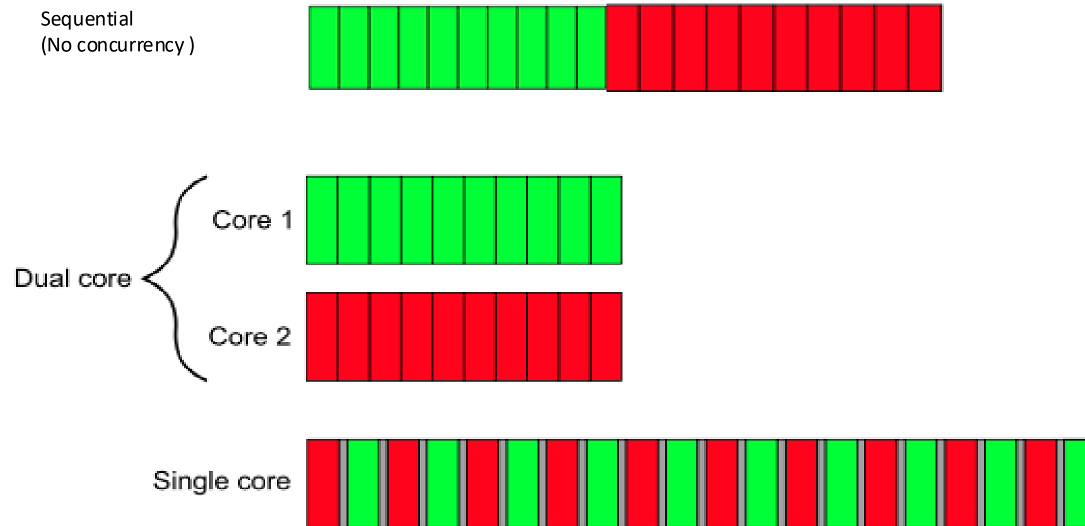
Alicia Yang

Multithreading

- What is concurrency
- Threads launching
- Thread finishing
- Threads safety

Concurrency

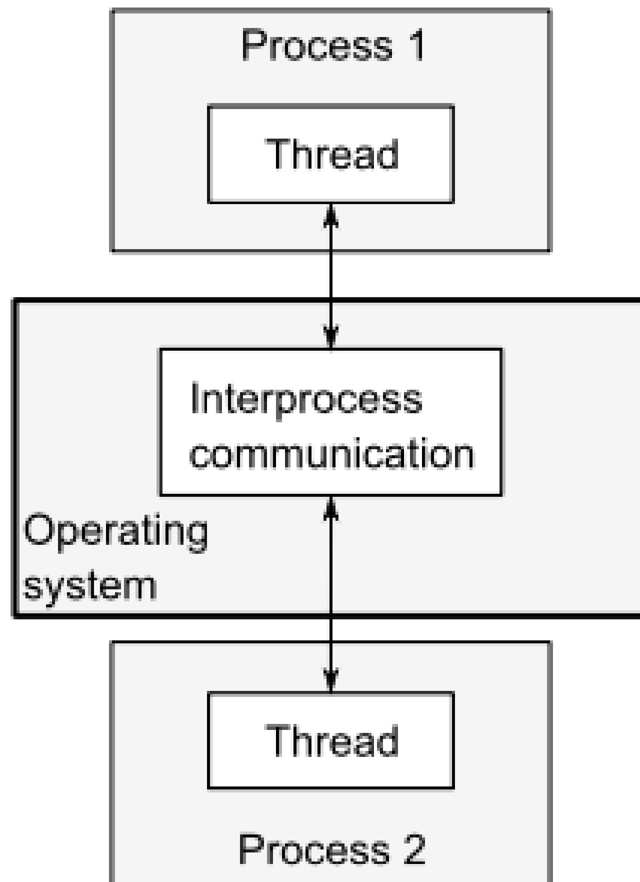
- What is concurrency?
  - a single system performs multiple independent activities in parallel



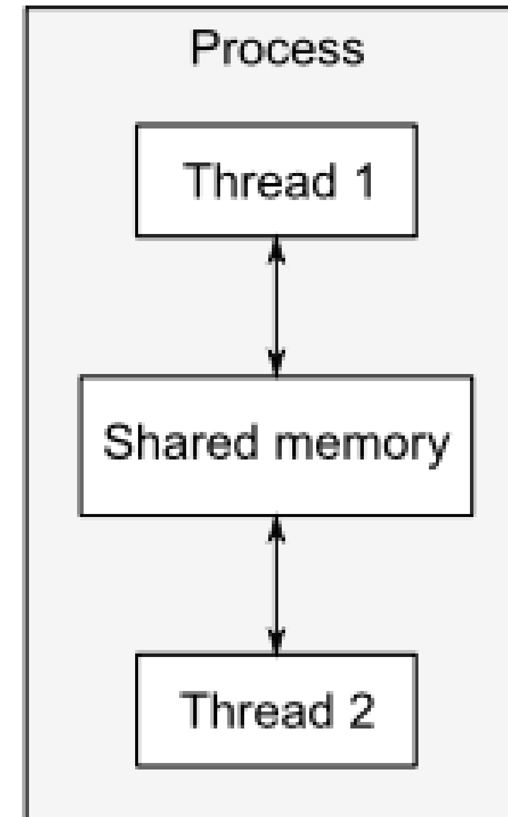
- Why use concurrency?
  - Separation of concerns
  - Performance

Types of concurrency

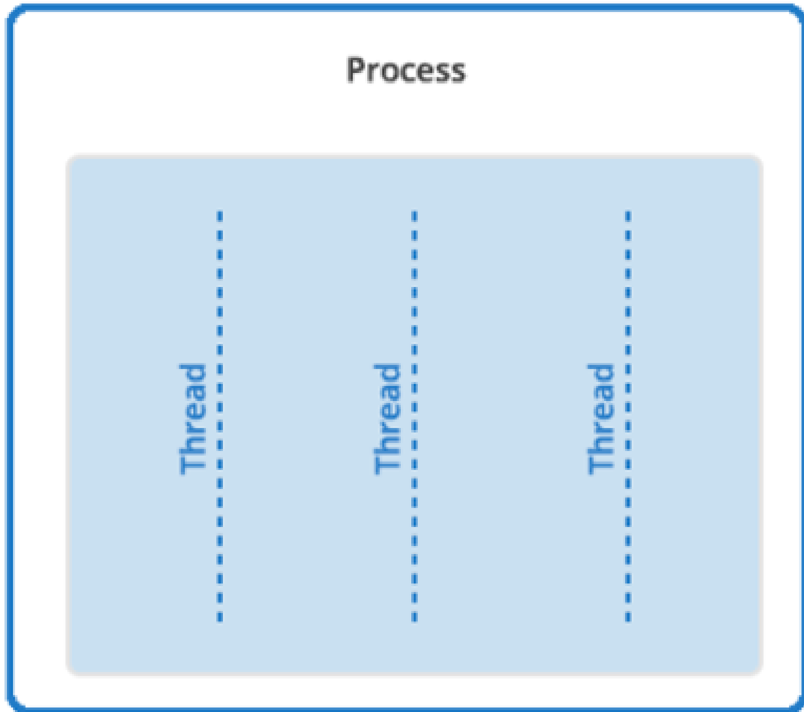
Concurrent Processes



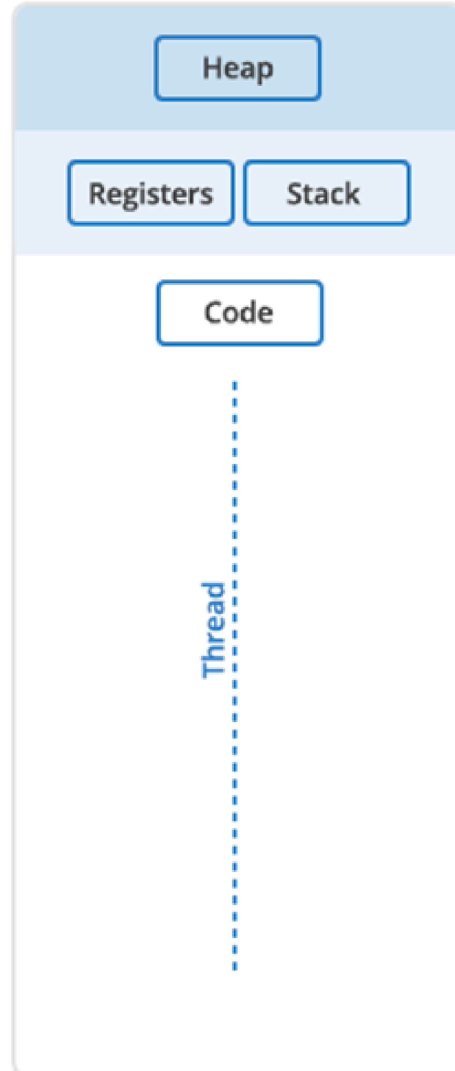
Concurrent Threads



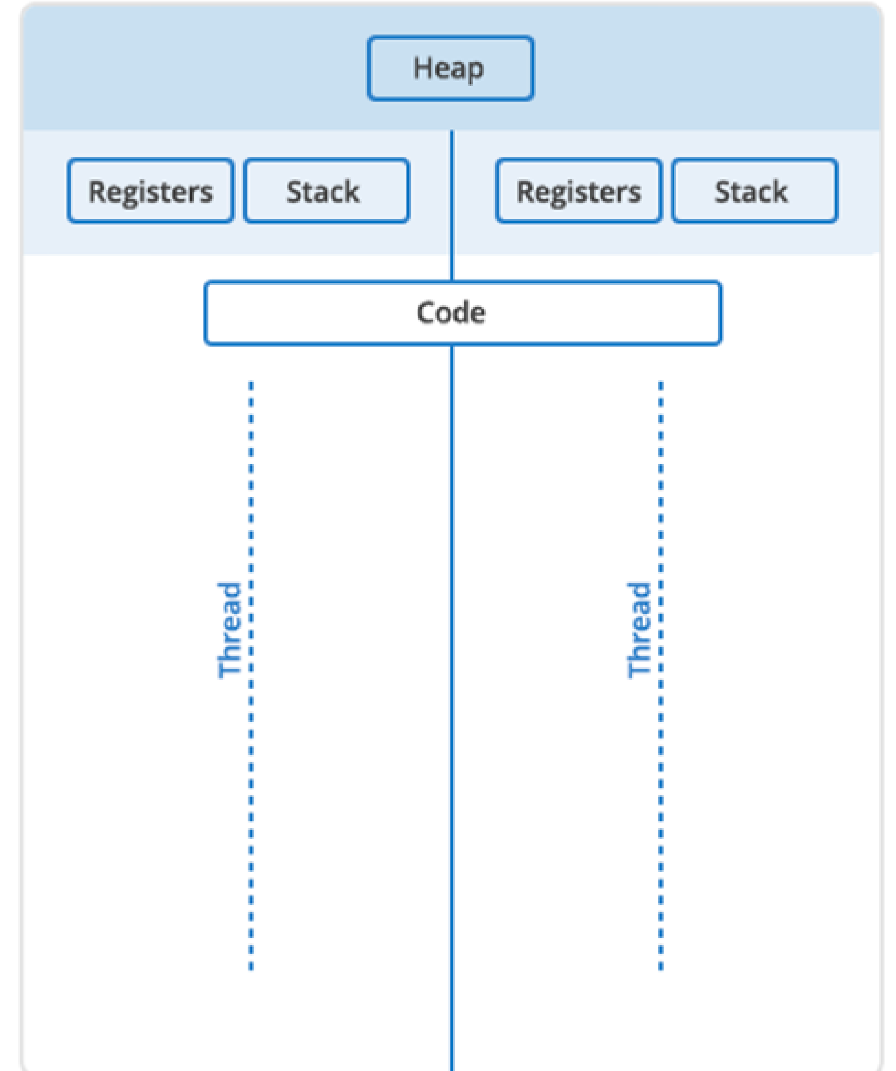
Concurrency



Single Thread



Multi Threaded



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

Multithreading

- Threads:
  - Threads are lightweight **executions**: each thread runs independently of the others and may run a different sequence of instructions.
  - All threads in a process **share the same address space**, and most of the data can be accessed directly from all threads—global variables remain global, and pointers or references to objects or data can be passed around among threads.

Multithreading

- What is concurrency
- Threads launching
  - std::thread
  - (Thread pool)
  - (openmp)
- Thread finishing
- Threads safety

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 (via std::thread)

- A callable types:
  - **A function pointer**
    - **Free function (non-member function)**
    - **Member function**
  - A function object (functor)
  - A lambda expression

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

Launching thread --- function pointer

- Launching a thread using function pointers and function parameters

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

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

Launching thread --- function pointer

Example1: function takes one argument

```
#include <thread>
void hello(std::string to)
{
    std::cout << "Hello Concurrent World to " << to << "\n";
}
int main()
{
    std::thread t1( &hello, "alicia");
    std::thread t2( hello, ”jonathan");
    t1.join();
    t2.join();
}
```

&(address-of) is **optional**
the function name decays to function pointer **automatically**, due to function-to-function-pointer decay

Launching thread --- function pointer

Example2: function takes multiple arguments (passing by values, references)

- std::ref for reference arguments

```
#include <thread>
void hello_count(std::string to, int &x){
    x++;
    std::cout << "Hello to " << to << x << std::endl;
}
int main(){
    int x = 0;
    std::thread threadObj(hello_count, "alicia", std::ref(x));
    ... // join
}
```

Launching thread (via std::thread)

- A callable types:
  - **A function pointer**
    - **Free function (non-member function)**
    - **Member function**
  - A function object (functor)
  - A lambda expression

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

How does calling a function on a class object work in C++?

- Suppose I have a class with an attribute x, a function print() that prints x.
- All objects of the class have their own copy of the non-static data members, but they share the class functions.
- When I call print() on different objects, why are their behavior different?

```
class myClass{
public:
    int x;
    void print(){
        std::cout << x << std::endl;
    }
};
```

```
int main(){
    myClass obj;
    obj.print();
}
```

Solution to the puzzle:

- All class functions automatically receive a pointer to the class object as their first argument
- For example, myClass::print() behaves as if it's written as myClass::print(myClass* obj_ptr)
- All references to x in the function resolve as obj_ptr->x

```
class myClass{
public:
    int x;
    void print(){
        std::cout << x << std::endl;
    }
};
```

```
int main(){
    myClass obj;
    obj.print();
}
```

Launching thread --- member function pointer

- Launching a thread using (non-static) member function

```
class FunClass {
    void func(params) {
        // Do Something
    }
};
FunClass x;
std::thread thread_object(&FunClass::func, &x, params);
```

Launching thread --- member function pointer

- Example3: launching thread with (non-static) member function

```
class Hello
{
public:
    void greeting(std::string const &message) const{
        std::cout << message << std::endl;
    }
};

int main(){
    Hello x;
    std::string msg("hello");
    std::thread t(&Hello::greeting, &x, msg);
… // join}
```

Multithreading

--- managing thread

- A callable types:
  - A function pointer
  - **A function object (functor)**
  - A lambda expression

Multithreading

--- Launching thread with function object

- Launching a thread using function object and taking function parameters

```
class fn_object_class {
    // Overload () operator
   void operator()(params) {
      // Do Something
   }
}
std::thread thread_object(fn_object_class(), params)
```

- Example: launching thread with function object
  - Create a callable object using the constructor
  - The thread calls the function call operator on the object

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

class Hello{
public:
        void operator()(std::string name)
        {
                std::cout << "Hello to " << name << std::endl;
        }
};

int main(){
        std::thread t(Hello(), "alicia");
        t.join();
}
```

Multithreading

--- managing thread

- A callable types:
  - A function pointer
  - A function object
  - **A lambda expression**

Multithreading

--- Launching thread with lambda function

- Launching a thread using lambda function

```
std::thread thread_object([](params) {
    // Do Something
}, params);
```

- Example:

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

int main()
{
    std::thread t([](std::string name){
        std::cout << "Hello World ! " << name <<" \n";
    }, “Alicia”);
    t.join();
}
```

Lambda function

- Lambda expression

```
[ capture clause ] (parameters) -> return-type
{
   definition of method
}
```

Lambda function

```
[ capture clause ] (parameters) -> return-type
{
   definition of method
}
```

- Capture variables:
  - [&] : capture all external variables by reference
  - [=] : capture all external variables by value
  - [a, &b] : capture a by value and b by reference

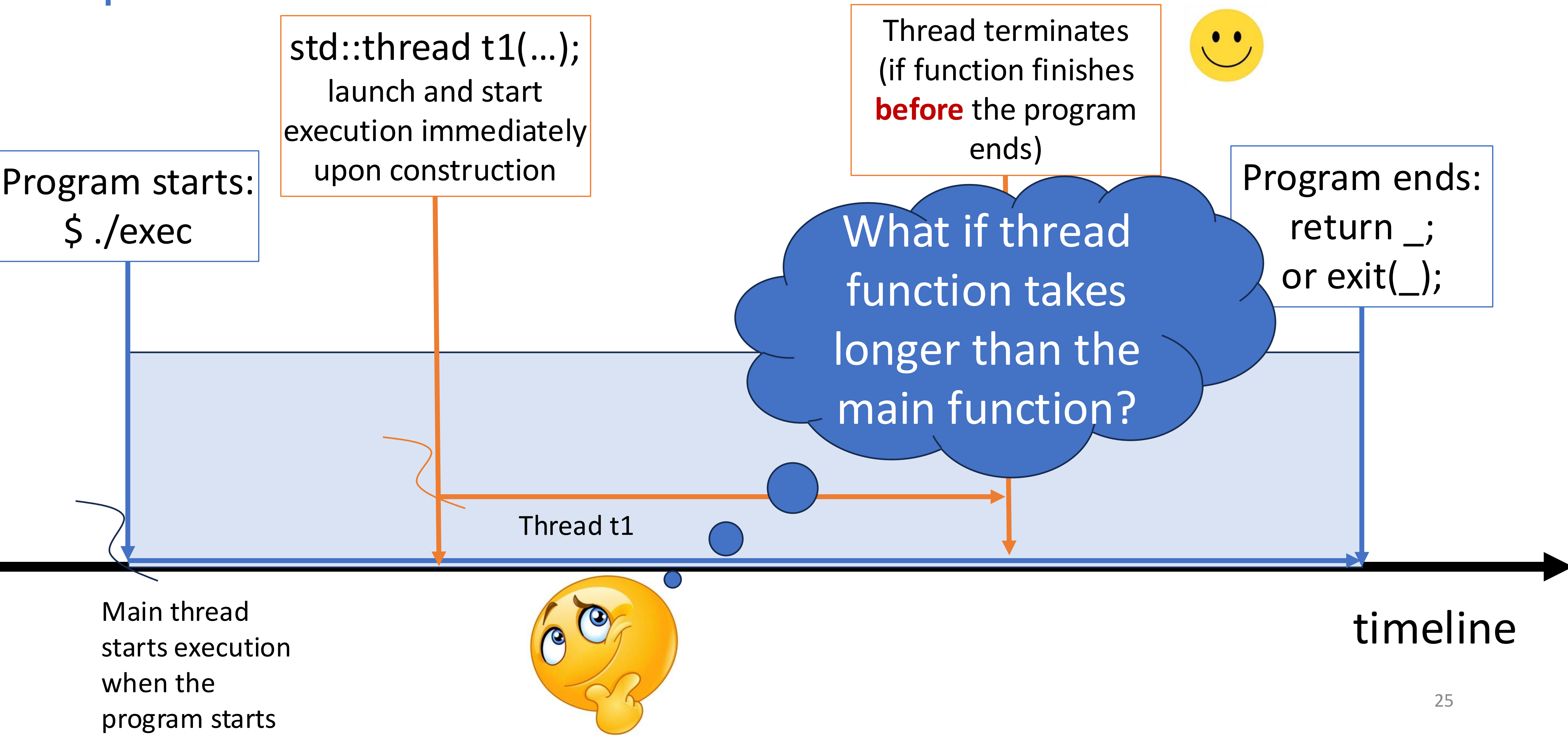
```
std::vector<int> v1 = {3, 1, 7, 9};
std::vector<int> v2 = {10, 2, 7, 16, 9};
//  access v1 and v2 by reference
auto pushinto = [&] (int m){
    v1.push_back(m);
    v2.push_back(m);
};
pushinto(100);
...
```

& can access all the variables that are in scope.

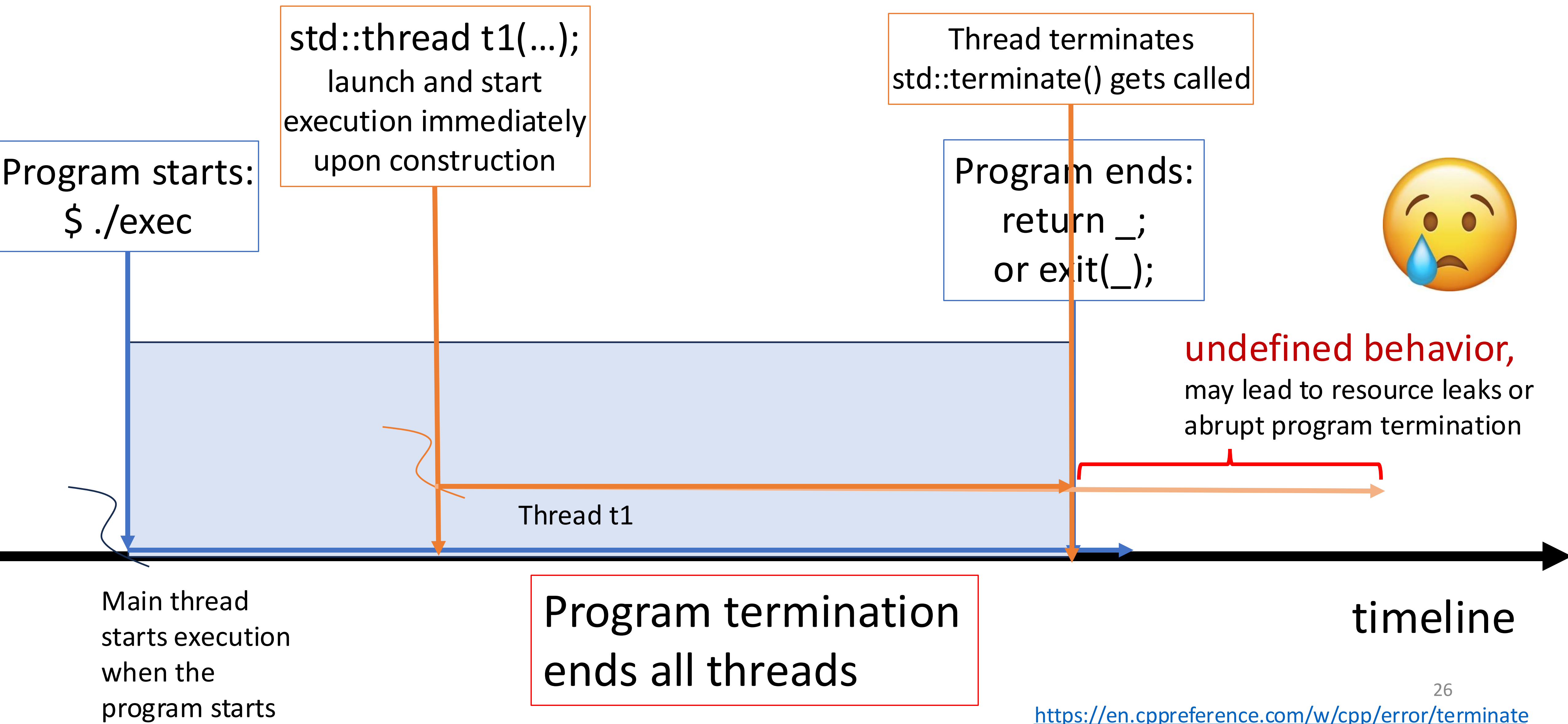
Multithreading

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

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()
  - Detach()

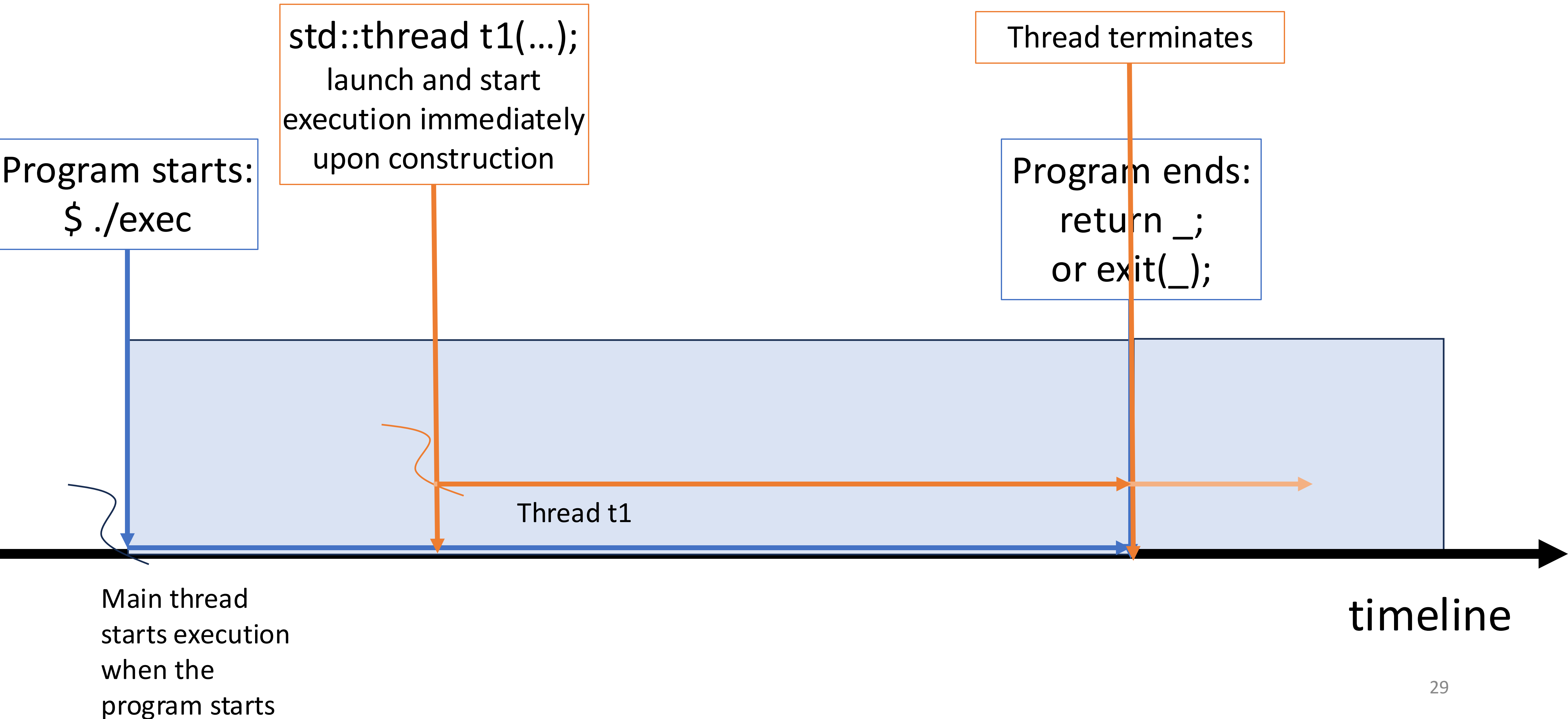
demo

Joining threads with std::thread

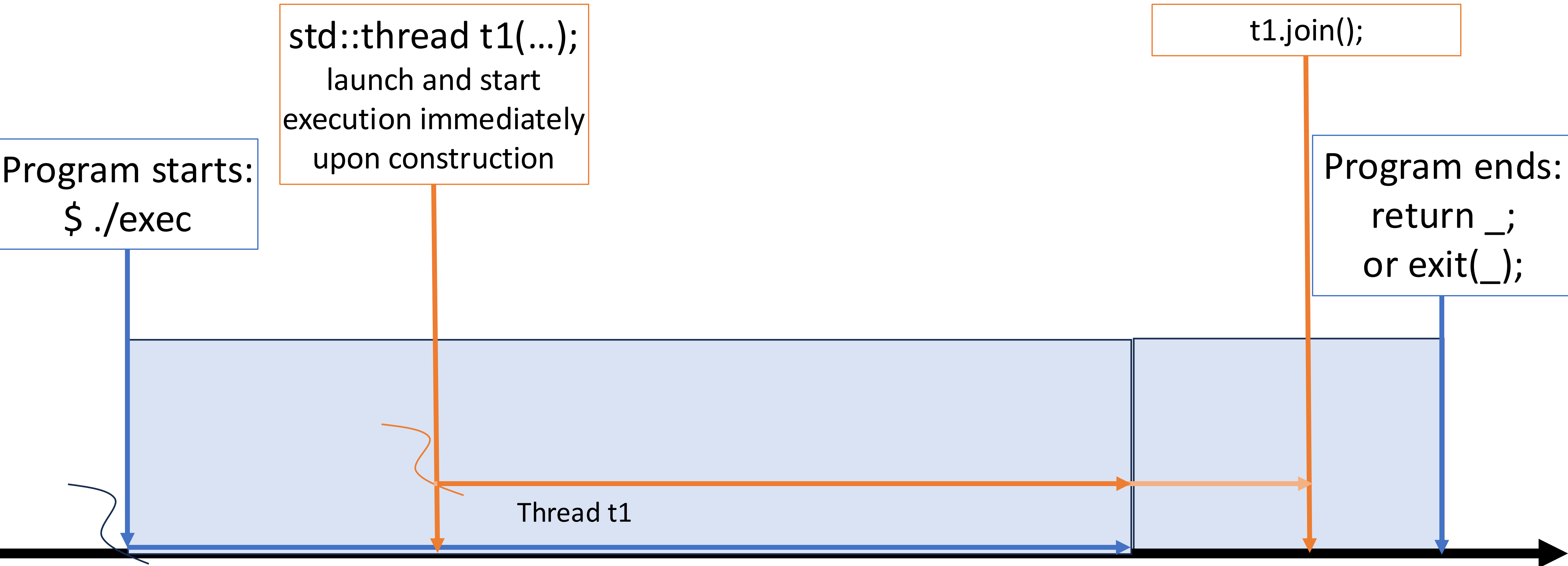
```
std::thread thread_obj(func, params);
Thread_obj.join();
```

- **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



Detaching threads with std::thread

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

- Run thread **independently**, with no direct means of communicating with it.

  Ownership and control are passed over to the C++ Runtime Library
- Detached threads **terminate** when the program ends
- For **long-running** tasks; they may run for the entire lifetime of applications, such as background logging or monitoring tasks, async notification or alert

demo

Multithreading

- What is concurrency
- Threads launching
- Threads safety
  - Race condition
  - Examples of data types that are/not thread-safe

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.

What could go wrong with concurrent access?

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

- 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

Sharing data among threads

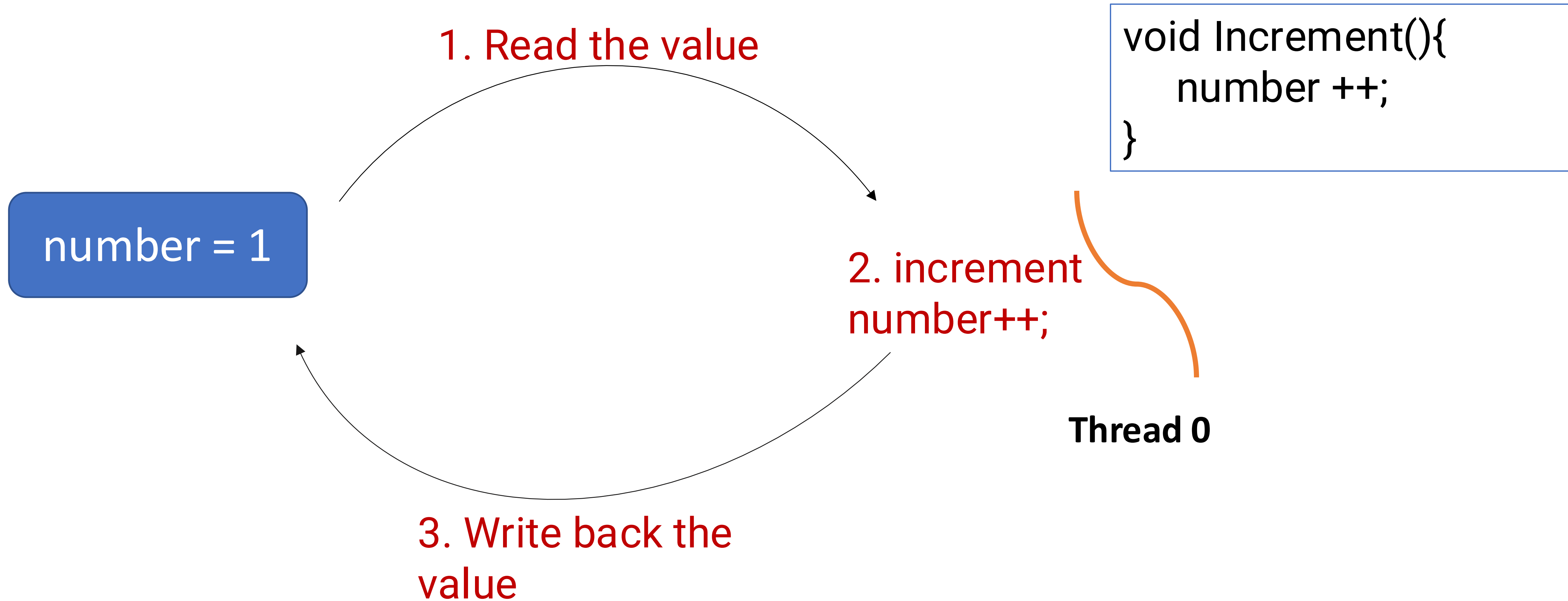
---race condition

- Example: Concurrent increments of a shared integer variable.
  - Increment in assembly


```cpp
int main()
{
    volatile int val = 0;
    val ++;
    return val;
}
```

```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 0
        mov     eax, DWORD PTR [rbp-4]
        add     eax, 1
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

Example: Concurrent increments of a shared integer variable



Example: Concurrent increments of a shared integer variable

num = 0

Read the value

1

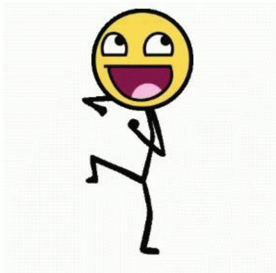
Oh! I see num=0.

2

Write back the value

Thread 1. Write back 1.

Ideally what we want



3

Oh! I see num=1.

4

num = 3

5

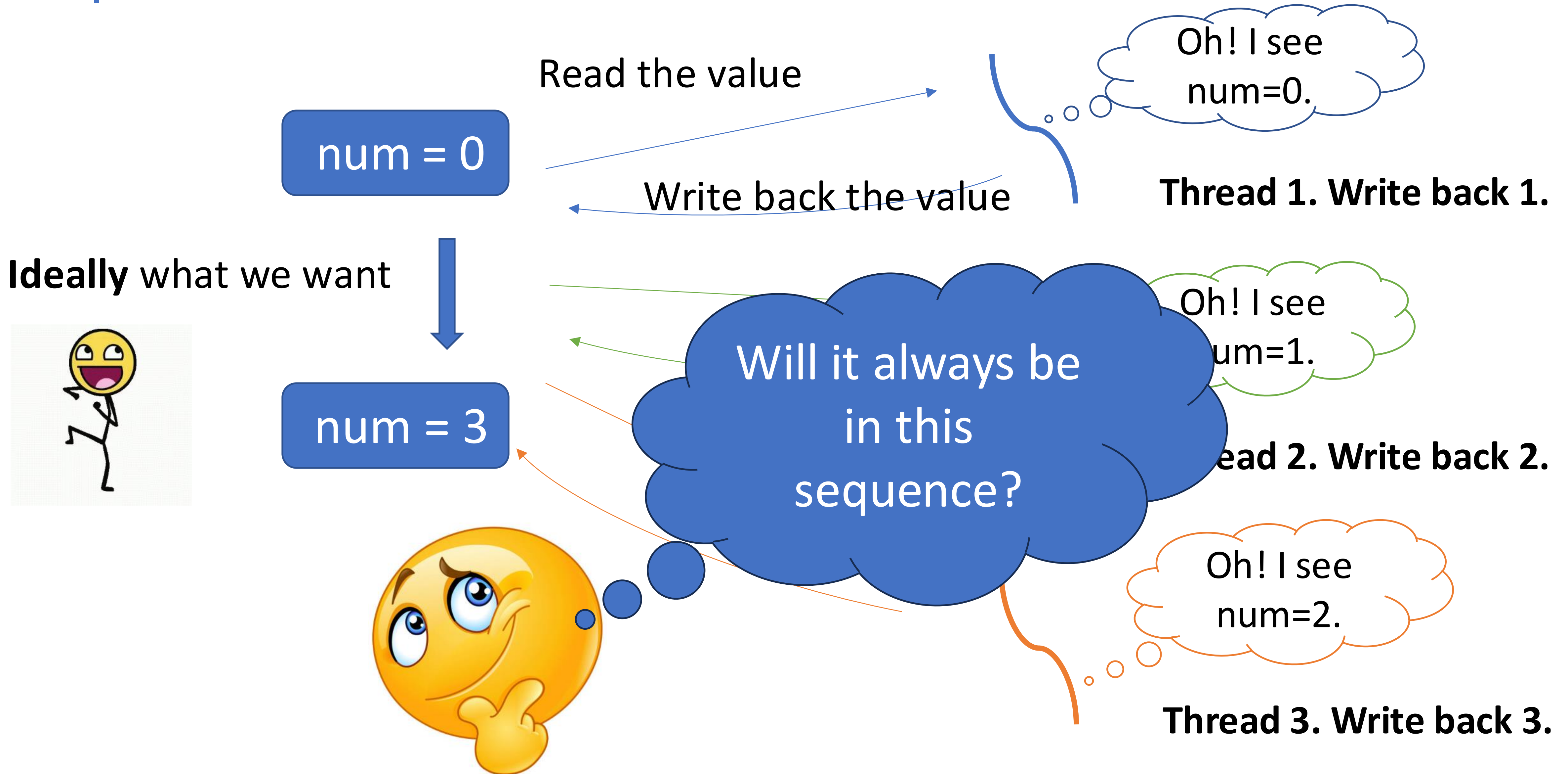
Thread 2. Write back 2.

6

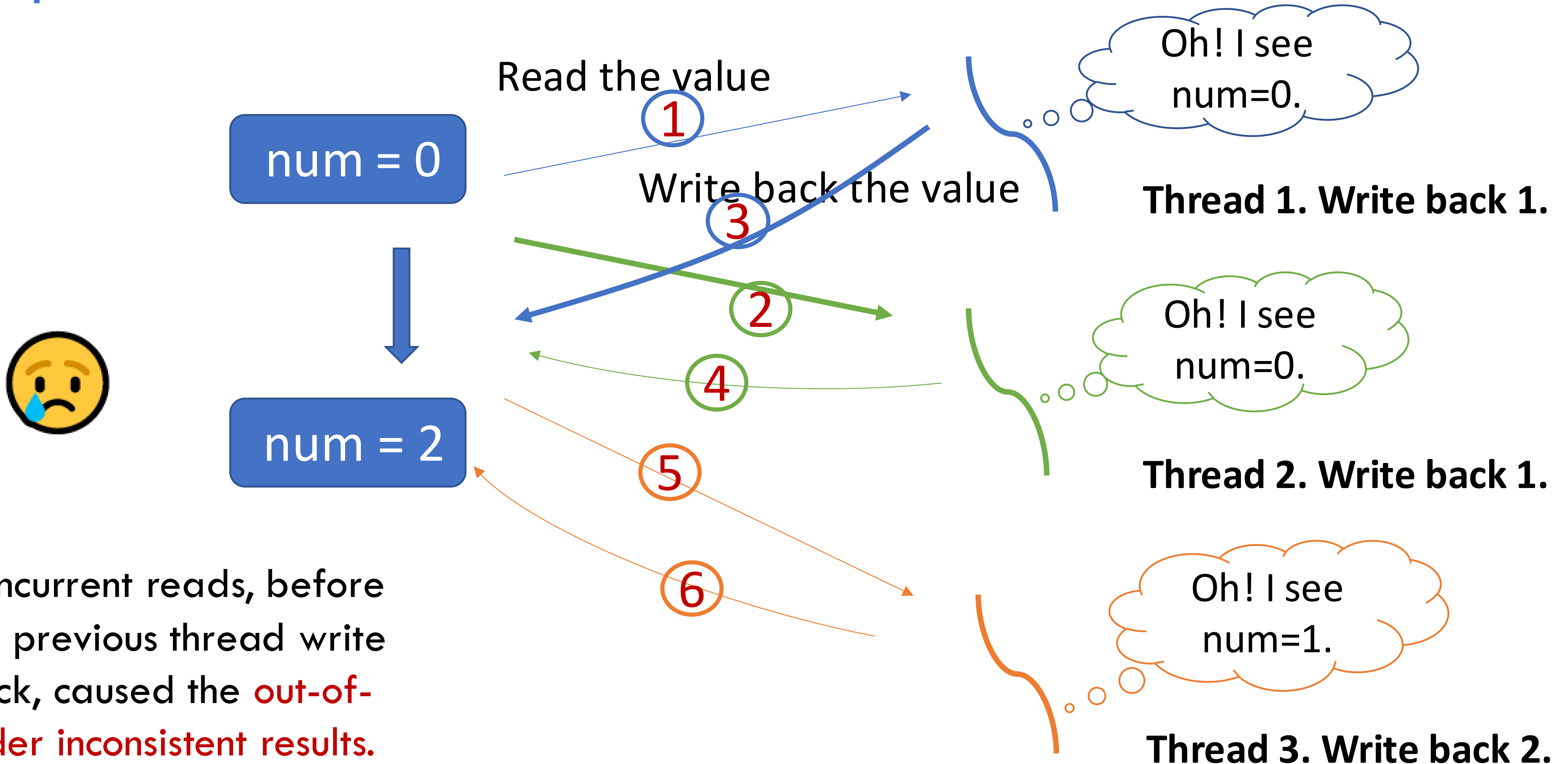
Oh! I see num=2.

Thread 3. Write back 3.

Example: Concurrent increments of a shared integer variable



Example: Concurrent increments of a shared integer variable

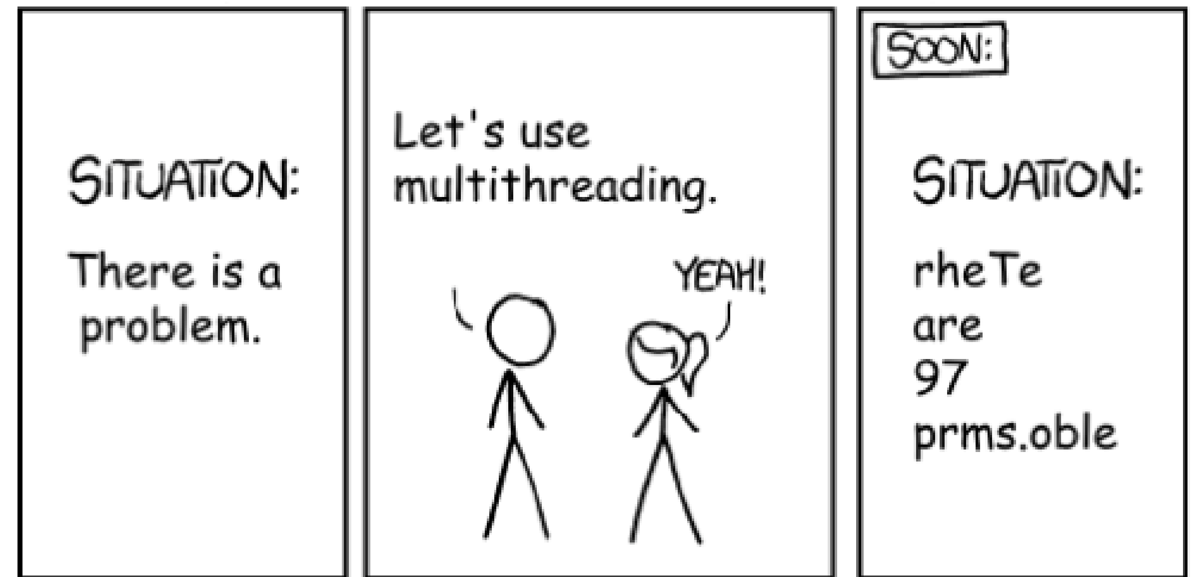


Concurrent reads, before the previous thread write back, caused the out-of-order inconsistent results.

Race condition

a race condition is 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.



Thread safe?

- Is integer inherently thread safe?
  - No, as we showed just now
- Next recitation :
  - What about other standard libraries classes and types thread-safety
  - How can multi-thread programing share data while guaranteeing thread safety?

std::map

demo

```
std::map<int, int> global_map;

int main(){
	for (int i = 0; i < 1000000; ++i){
		global_map[i] = i;
	}
	std::thread r_thread(read_map);
	std::thread e_thread(erase_map);

	read_map_thread.join();
	erase_map_thread.join();
}
```

```
void read_map(){
	for (int i=0;i<1000000;++i){
		if(global_map.find(i) == global_map.end())
			continue;
		int val = global_map.at(i);
		if(val != i){
			std::cout << i << "," << val << std::endl;
		}
	}
}
```

```
void erase_map(){
	for (int i = 20000; i < 80000; ++i){
		global_map.erase(i);
	}
}
```

What could go wrong?

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
  - cppcon thread-safe: https://youtu.be/s5PCh_FaMfM?si=-3h7nszcy_jesQAH
- Notes:
  - https://thispointer.com/c11-multithreading-part-3-carefully-pass-arguments-to-threads/