# CS4414 Recitation 11

## Multithreading and Synchronization

11/05/2021

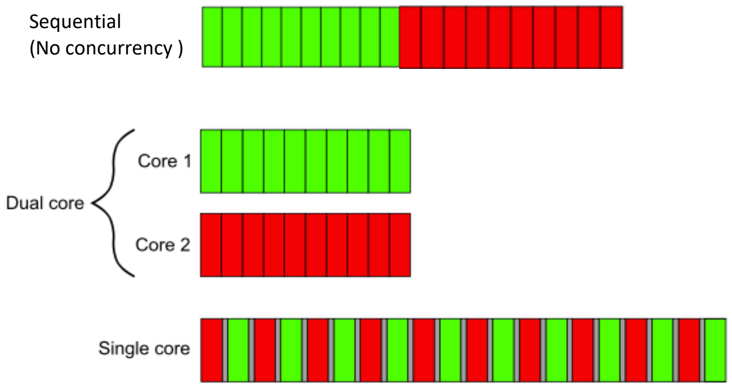
Alicia Yang

# Multithreading

- What is concurrency
- Multithreading
- Threads Management

# Concurrency

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




- Why use concurrency?
  - Separation of concerns
  - Performance

# 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.
- Example:

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

void hello() {
    std::cout<<"Hello Concurrent World\n";
}

int main() {
    std::thread t(hello);
    t.join();
}
```

Compile with –lpthread flag

# Multithreading

--- managing thread

- Launching a thread (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.
- A callable types:
  - A function pointer
  - A function object
  - A lambda expression

# Multithreading

--- managing thread

- Launching a thread (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.
- A callable types:
  - **A function pointer**
  - A function object
  - A lambda expression

# Multithreading

--- Launching thread with function pointer

- Launching a thread using function pointers and function parameters

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

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

- 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, "sagar");
     t1.join();
     t2.join();
}
```

# Multithreading

--- Launching thread with function pointer

- Launching a thread using function pointers and function parameters

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

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

- Example2: function takes multiple arguments (passing by values, and passing by reference)
  - 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));
      threadObj.join();
      std::cout << "After thread x=" << x << std::endl;
}
```

# Calling function of class on an object in a new thread

- First: 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.
- So, when I call print on different objects, why is the behavior different?

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

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

# Calling function of class on an object in a new thread

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();
}
```

# Multithreading

## --- Launching thread with function pointers

- Launching a thread using member function

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

- Example3: launching thread with member function

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

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

# Multithreading

--- managing thread

- Launching a thread (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.

- A callable types:
  - A function pointer
  - **A function object**
  - 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 <string>

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

- Launching a thread (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.
- 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);
```

- Example1:

basic lambda function

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

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

# Lambda function

- Lambda expression

```
[ 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

```
int main()
{
    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

--- managing threads

- Joining threads with std::thread
  - Wait for a thread to complete
  - Ensure that the thread was finished before the function was exited and thus before the local variables were destroyed.
  - Clean up any storage associated with the thread, so the std::thread object is no longer associated with the now- finished thread
  - join()  can be called only once for a given thread

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

demo

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

# Multithreading

--- managing threads

- Detach threads with std::thread
  - Run thread in the background, with no direct means of communicating with it. Ownership and control are passed over to the C++ Runtime Library
  - Detached threads are also called daemon / Background threads.
  - Such threads are typically long-running; they may well run for almost the entire lifetime of the application, performing a background task
  - If neither join or detach is called with a std::thread object that has associated executing thread then during that object's destruct, it will terminate the program.

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


demo

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

# Recap Multithreading

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

# Data Sharing between Threads

- Race condition
- Atomic
- Mutex

# 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.
- 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

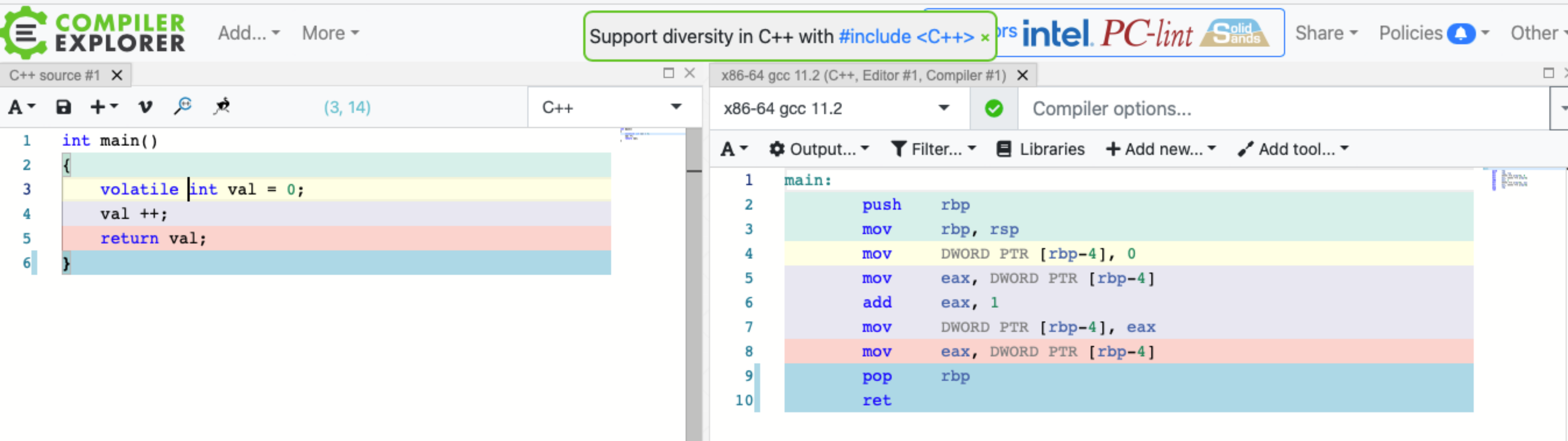
| Number of threads | Final value |
|---|---|
| 1 | 1000000 |
| 2 | 1059696 |
| 3 | 1155035 |
| 4 | 1369165 |

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

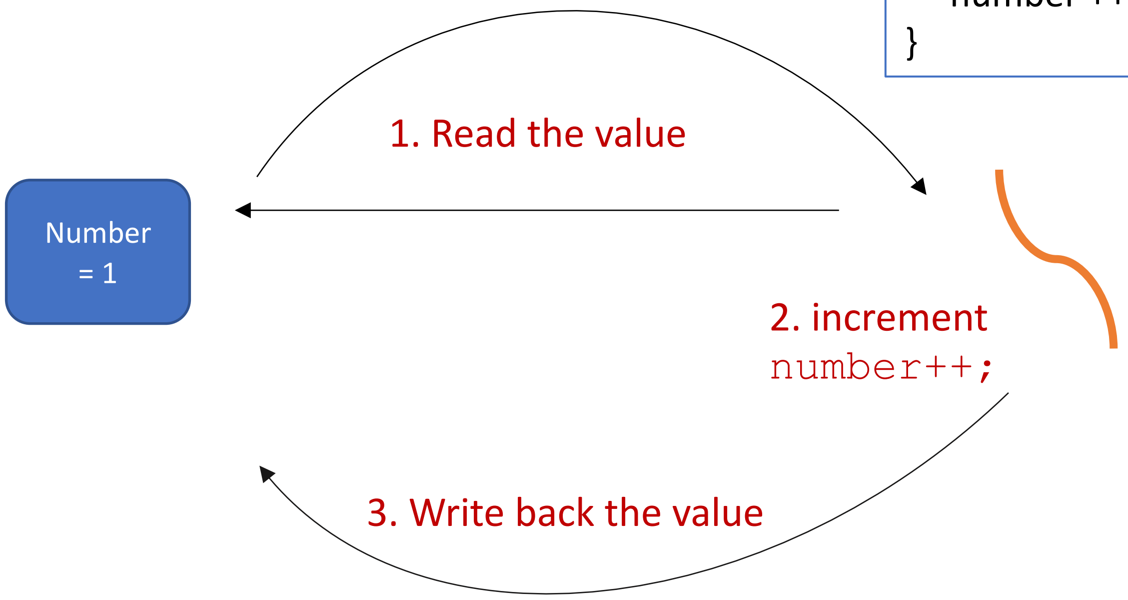


# Sharing data among threads

---race condition

- <u>Example:</u> 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

```
void Increment(){
    number ++;
}
```

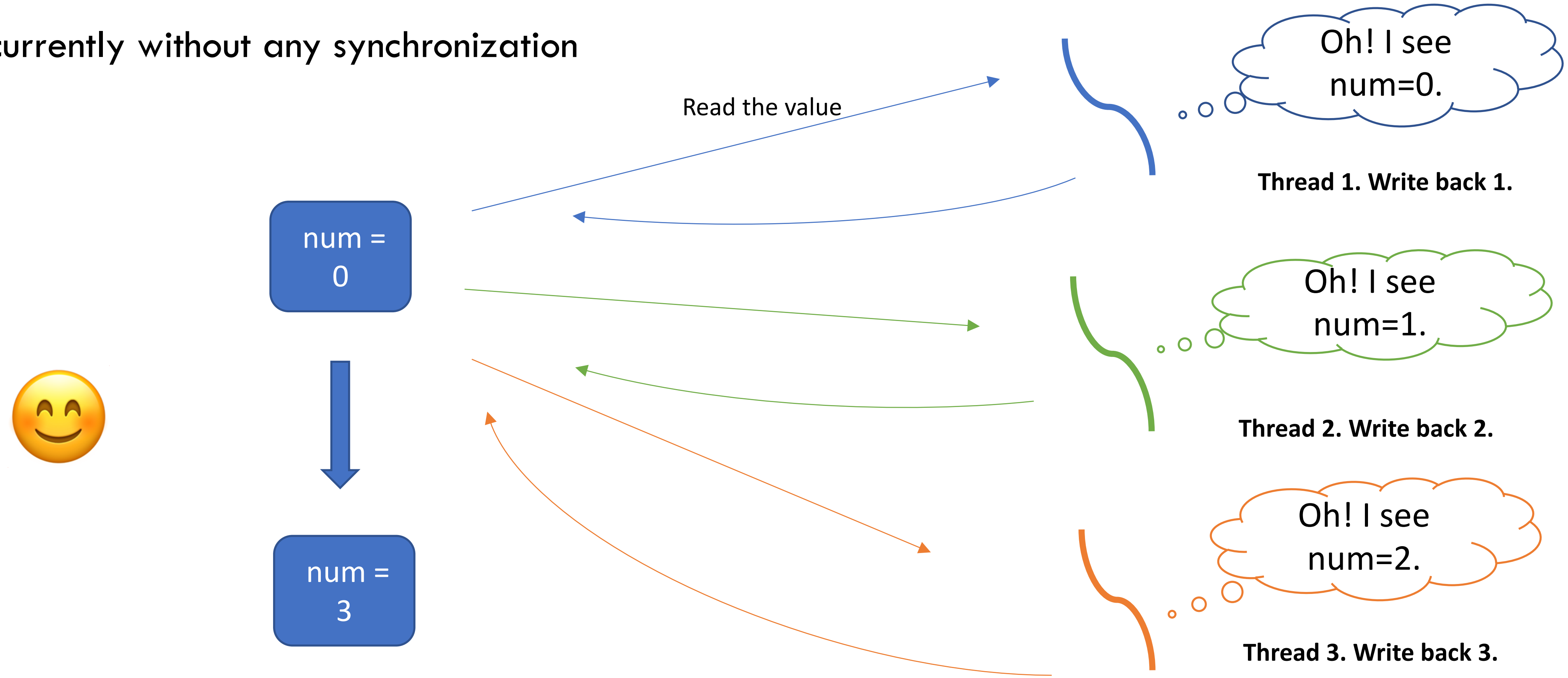



| Number of threads | Final value |
|---|---|
| 1 | 1000000 |
| 2 | 1059696 |
| 3 | 1155035 |
| 4 | 1369165 |

# Sharing data among threads

---race condition

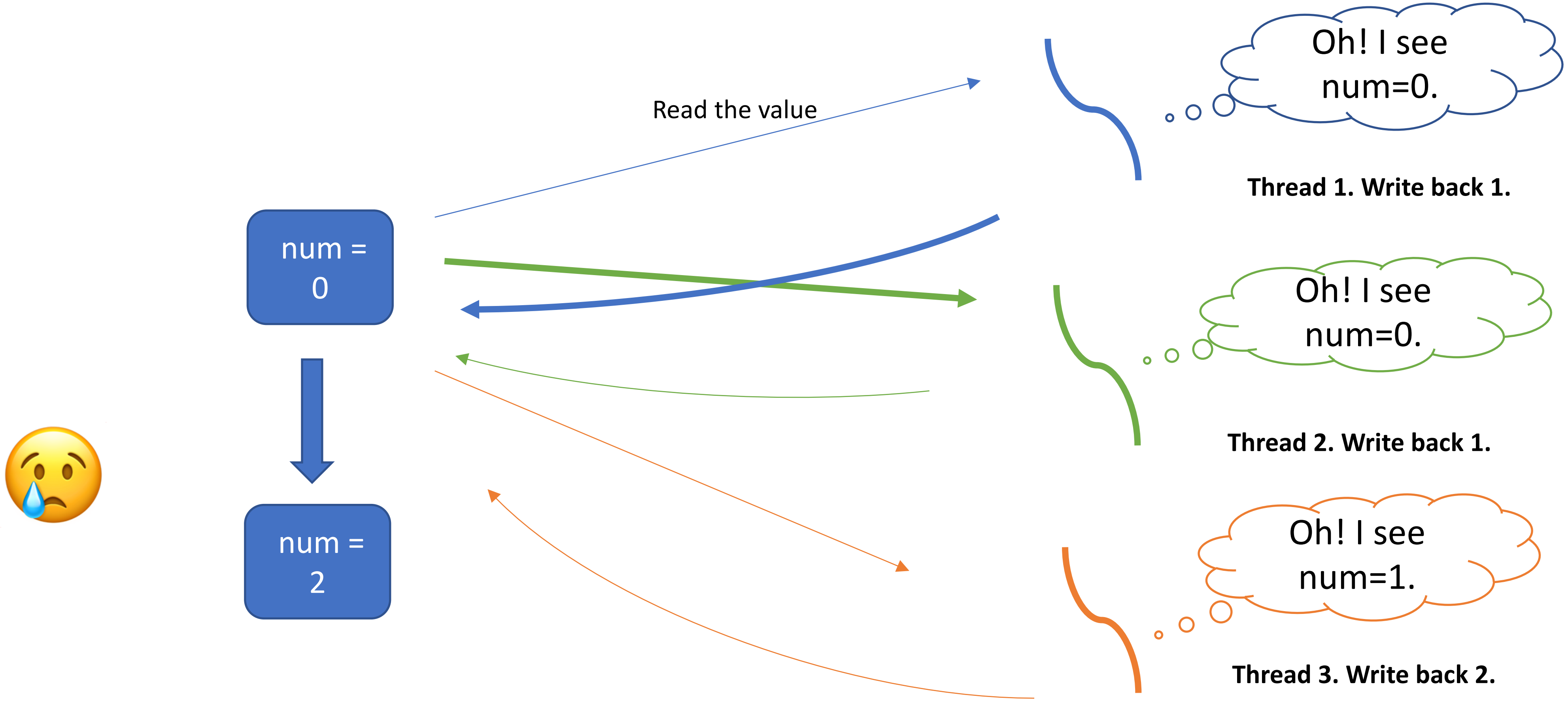
- Example: Concurrent increments of a shared integer variable.
    - Each thread shares an integer called count initialized to 0, increments it 1 million times concurrently without any synchronization



# Sharing data among threads

---race condition

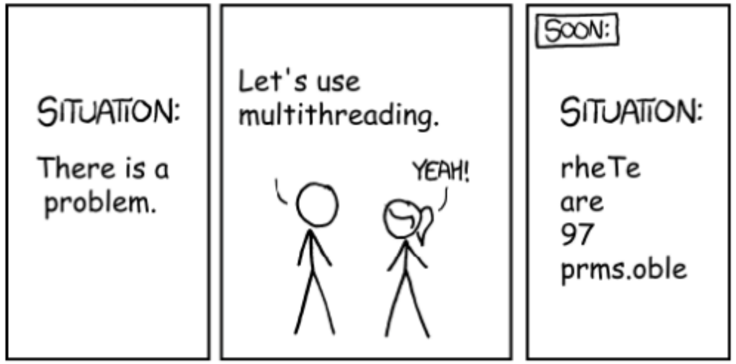
- Example: Concurrent increments of a shared integer variable.
  - The concurrent read, before the previous thread write back, caused the out of order inconsistent results.



# Sharing data among threads

---race condition

- Example of a race condition:
  - Not thread safe to add or remove values to/from std::map
  - Cannot vary size of std::vector, resizing when adding elements will cause segmentation fault
- Safe to read-only containers
- How can we avoid race condition?



# Sharing data among threads

---race condition

- 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.
- More example of a race condition:
  - Not thread safe add values to map, cannot delete
  - Cannot vary size of vector , resizing when adding element will cause segmentation fault
- Avoid race condition
  - **Atomic variable**
  - Mutex lock

# Atomic

- An atomic operation is an indivisible operation. You can't observe such an operation half-done from any thread in the system; it's either done or not done.
- Atomic type: std::atomic<type>
  - Requires hardware/PL support. Atomic can be applied to a specific set of types, such as int, long, bool
  - An atomic type can be used to safely read and write to a memory location shared between two threads.
  - Operations on atomic type that have the required memory-ordering semantics
    - Store operations
    - Load operations
    - Read-modify-write operations (simultaneous read and write)

demo

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

# 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.
  - 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.

# Locking

---protecting data with mutex

- Using mutex
  - It isn't recommended practice to call the member functions directly, because this means that you have to remember to call unlock() on every code path out of a function, including those due to exceptions.

# RAII (Resource Acquisition is initialization)

- The motivations of RAII

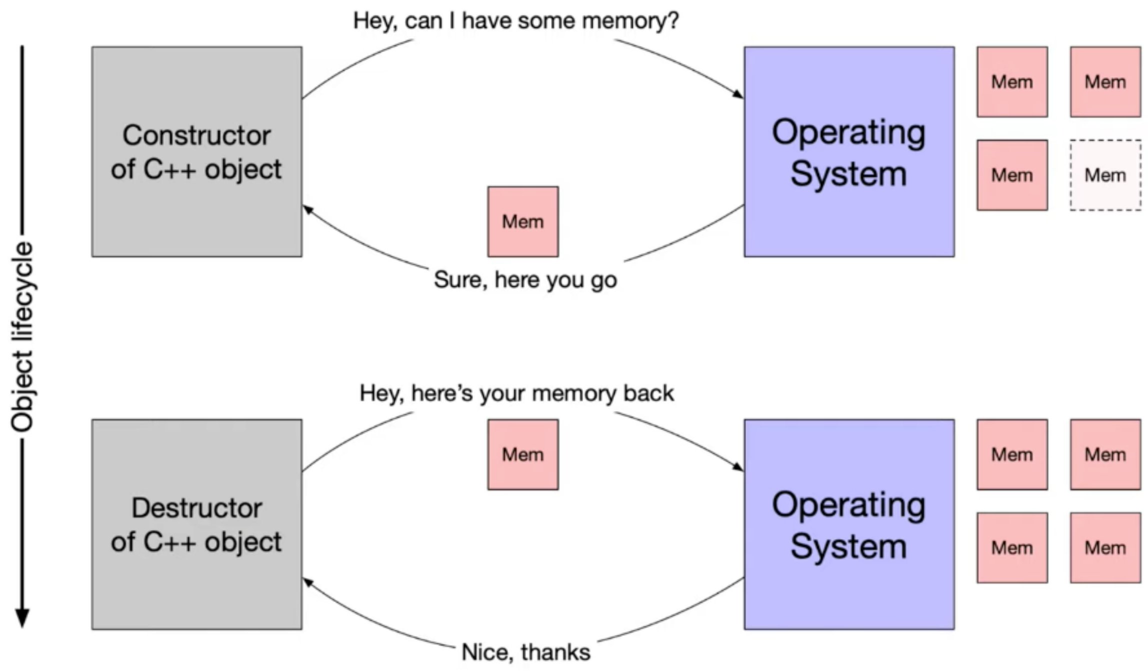
```
// problem #1
{
   int *arr = new int[10];
}   // arr goes out of scope but we didn’t delete it, we now have a memory leak 😢
```

```
// problem #2
Std::mutex globalMutex;
Void func() {
    globalMutex.lock();
}   // we never unlocked the mutex(or exception occurred before unlock), so this will cause a deadlock if other thread tries to acquire the lock 😢
```

```
// problem #3
{
    std::thread t1( [] () {
            // do some operations
    });
}   // thread goes out of scope and is joinable, std::terminate is called 😢
```

# 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



# RAII (Resource Acquisition is initialization)

- RAII :
  - Better way: automatically release the resources when objects go out of scope
- RAII Classes:
  - std::vector
  - std::string
  - std::unique_ptr
  - std::shared_ptr
  - std::unique_lock
  - std::scoped_lock

# Locking

- scoped_lock()
- unique_lock()
- shared_lock()

# Locking

---protecting data with mutex

- <u>Example:</u> Protecting vector with mutex and scoped_lock example

```
std::vector<int> my_vec;
std::mutex my_mutex;
void add_to_list(int new_value) {
    std::scoped_lock<std::mutex> lck(my_mutex);
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find) {
   std::scoped_lock<std::mutex> lck(my_mutex);
   return std::find(some_list.begin(),some_list.end(),value_to_find) !=
some_list.end();
}
```

if not using mutex, Abort error

```
a.out(41290,0x10d492600) malloc: *** error for object 0x600000393ffc: pointer being freed was not allocated
a.out(41290,0x10d492600) malloc: *** set a breakpoint in malloc_error_break to debug
Abort
```

# Locking

- scoped_lock()
- unique_lock()
- shared_lock()

# Locking

- 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. When the unique_lock is constructed it will lock the mutex, and when it gets destructed it will unlock the mutex.  If an exceptions is thrown, the std::unique_lock destructer will be called and so the mutex will be unlocked.

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;
void print_block (int n, char c) {
   std::unique_lock<std::mutex> lck (mtx)
   for (int i=0; i<n; ++i) {
       std::cout << c; } std::cout << '\n’;
 }
int main () {
    std::thread th1 (print_block,50,'*’);
    std::thread th2 (print_block,50,'$’);
    th1.join();
    th2.join();
    return 0;
}
```

# Locking

- scoped_lock()
- unique_lock()
- shared_lock()

# Locking

- shared_lock():
  - A shared_lock can be used in conjunction with a unique lock to allow multiple readers and exclusive writers.

# Where to find the resources?

- Concurrency programing:
  - [Book: C++Concurrency in Action Practice Multithreading]
- Multithreading and mutex:
  - https://www.geeksforgeeks.org/multithreading-in-cpp/
  - https://thispointer.com/c11-multithreading-part-2-joining-and-detaching-threads/
  - https://www.youtube.com/watch?v=q6dVKMgeEkk [helpful tutorial to understand RAII]
- Notes:
  - https://thispointer.com/c11-multithreading-part-3-carefully-pass-arguments-to-threads/