CS4414 Recitation 10

Multithreading and Synchronization II

03/31/2023

Alicia Yang

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

--- 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
   }
}
fn_object_class   fn_instance;
std::thread thread_object(fn_instance,  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(){
     Hello hello;
     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();
}
```

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

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

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

x86-64 gcc 11.2

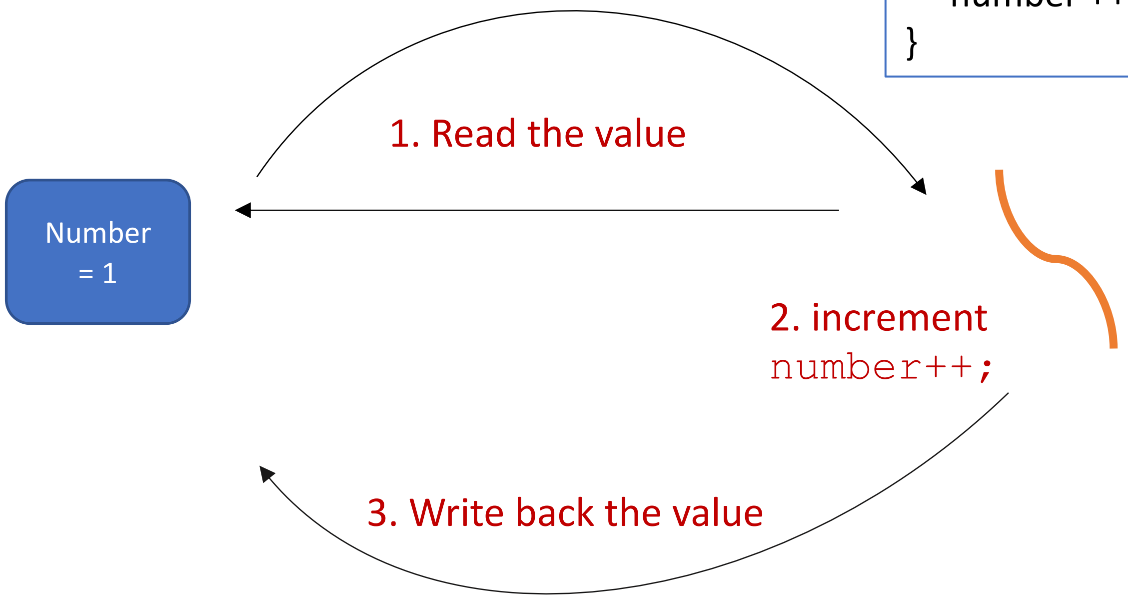
```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 0
        mov     eax, DWORD PTR [rbp-4]
        add     eax, 1
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

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 ++;
}
```

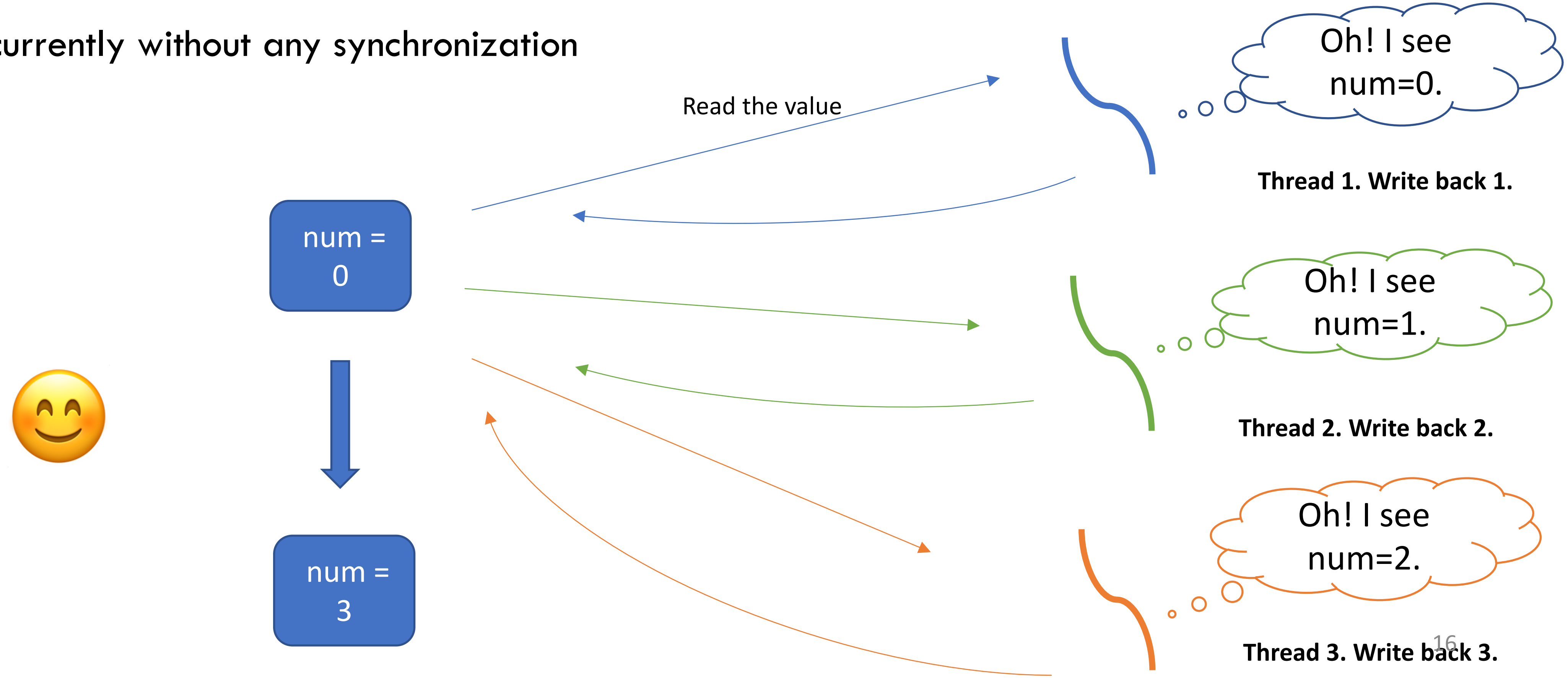


Number of threads	Final value
1	1000000
2	1059696
3	1155035
4	1369165

Sharing data among threads

---race condition

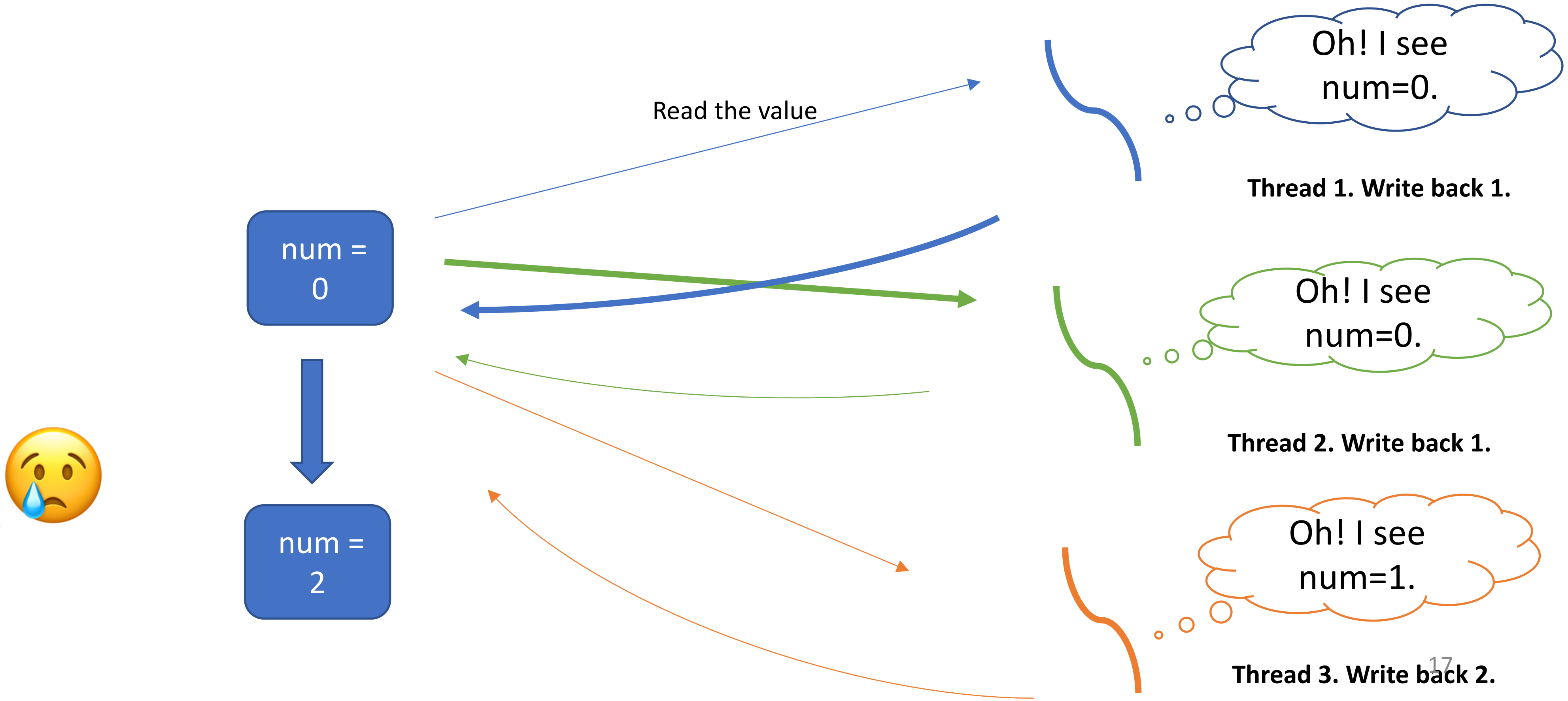
- <u>Example:</u> Concurrent increments of a shared integer variable.
  - Each thread shares an integer called count initialized to 0, increments it 1 million times concurrently without any synchronization



Sharing data among threads

---race condition

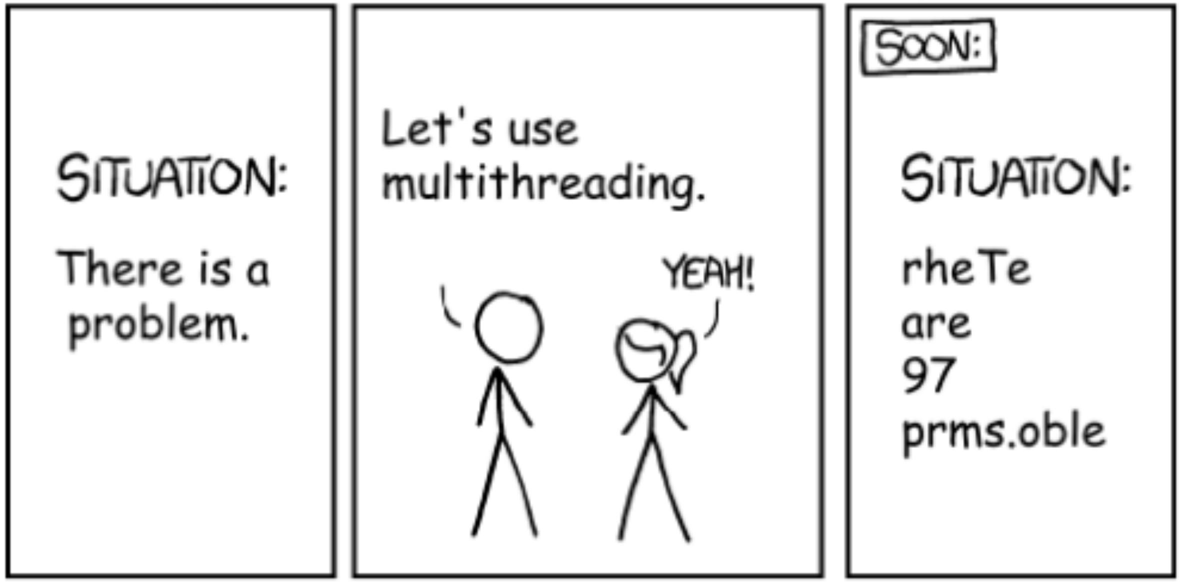
- Example: Concurrent increments of a shared integer variable.
  - The concurrent read, before the previous thread write back, caused the out of order inconsistent results.



Sharing data among threads

---race condition

- 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.



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:

Sharing data among threads

---race condition

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?

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 to alter the std::map, while accessing it from a different thread
  - Not thread-safe to vary size of vector(resize()), while adding element
  - …

Sharing data among threads

---race condition

demo

- 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 to alter the std::map, while accessing it from a different thread
  - Not thread-safe to vary size of vector(resize()), while adding element
- Avoid race condition
  - **Atomic variable**
  - Mutex lock

Atomic

- std::atomic<T> is a template, each instantiation and full specification of it defines an atomic type
- 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>
  - Constructor — std::atomic<bool> x(true); std::atomic<uint32_t> y(0);
  - store() — x.store(false); y.store(1, std::memory_order_relaxed);
  - load() — bool z = x.load();
  - exchange() — uint32_t m = y.exchange(100); // m = 0;
  - operator=
  - operator+=, operator -=
  - operator++, operator--

What happens when you call x+y?

C++ operator

```
class Coordinate{
public:
	int x;
	int y;
```

```
	Coordinate& operator=(const Coordinate&
other){
		x = other.x;
		y = other.y;
		return *this;
	}
```

```
	Coordinate operator+(const Coordinate&
other){
		return Coordinate( x + other.x, y +
other.y);
	}
```

```
int main(){
	Coordinate x(0,2);
	Coordinate y(3,5);
	y = x;
	Coordinate z = x + y;
}
```

demo

Atomic

- std::atomic<T> is a template, each instantiation and full specification of it defines an atomic type
- 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>
  - Constructor — std::atomic<bool> x(true); std::atomic<uint32_t> y(0);
  - store() — x.store(false); y.store(1, std::memory_order_relaxed);
  - load() — bool z = x.load();
  - exchange() — uint32_t m = y.exchange(100); // m = 0;
  - operator=
  - operator+=, operator -=
  - operator++, operator--
  - Note : operator + is not implemented by std::atomic library, same for copy and assignment operators

Atomic

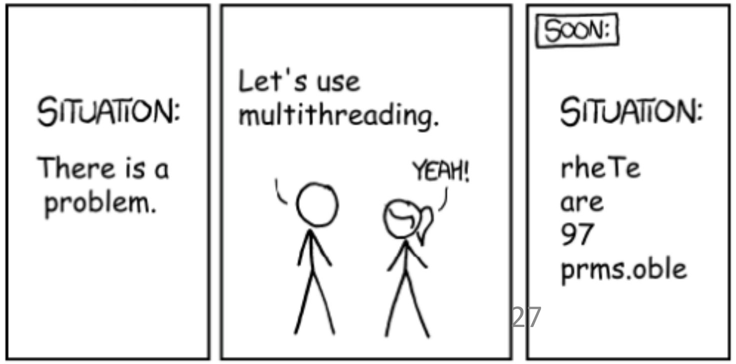
- An atomic operation is an indivisible operation. You can’t observe such an operation half-done from any thread in the system; it’s either done or not done.
- Atomic type: std::atomic<type>
  - An atomic type can be used to safely read and write to a memory location shared between two threads.
  - Accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by std::memory_order
    - memory_order::relaxed // no synchronization or ordering constraints imposed on other reads or writes
    - memory_order::consume // no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load
    - memory_order::acquire // no reads or writes in the current thread can be reordered before this load.
    - ….

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

- How can we avoid race condition?



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

---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 t1(incre, 10);
            std::thread t2(incre, 10);
            t1.join();
            t2.join();
}
```

Locking

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

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

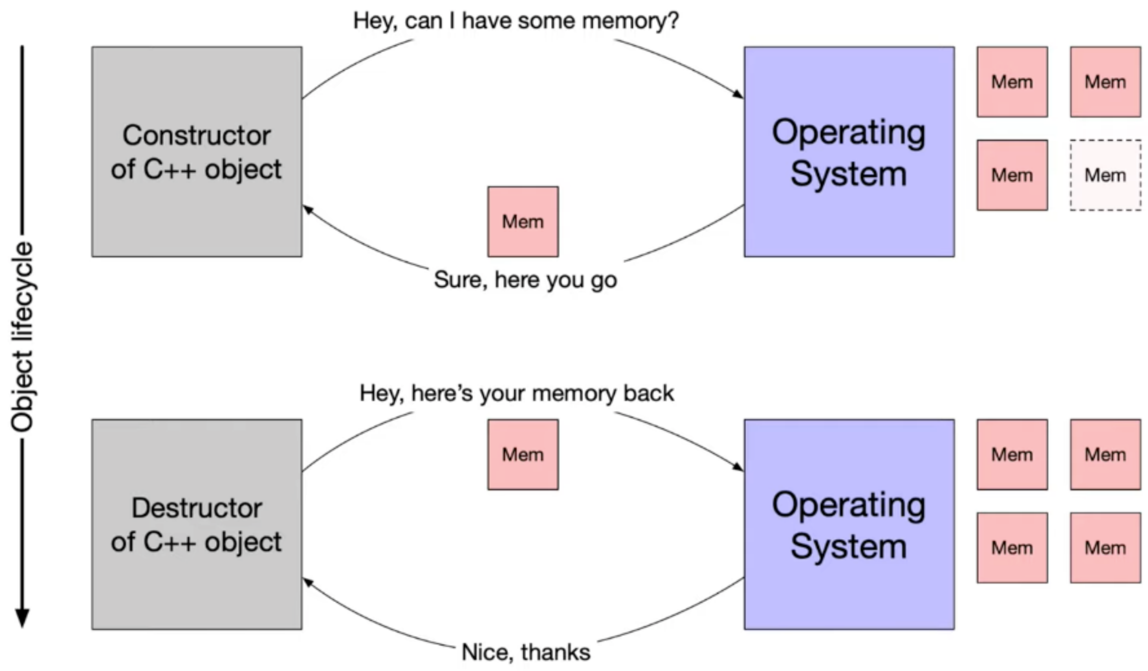
```
// problem #1
{
   int *arr = new int[10];
}   // arr goes out of scope but we didn’t delete it, we now have a memory leak 😢
```

```
// 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 : Object lifetime and resource management
  - guarantees that the resource is available to any function that may access the object
  - guarantees that all resources are released when the lifetime of their controlling object ends
- RAII summarization:
  - encapsulate each resource into a class
    - The constructor acquires the resource and establishes all class invariants
    - The destructor releases the resource and never throws exceptions
  - Use the resource via an instance
    - Automatic storage of resources with the duration/lifetime of the instance
    - Lifetime bounded to the instance

RAII (Resource Acquisition is initialization)

- RAII : Object lifetime and resource management
  - guarantees that the resource is available to any function that may access the object
  - guarantees that all resources are released when the lifetime of their controlling object ends
- 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

---scoped_lock

- 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;
            …
}
```

Locking

---scoped_lock

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

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.
  - 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)

Locking

---unique_lock

Unique_lock feature: Deferred locking

```
std::mutex mtx1;
std::mutex mtx2;
int  global_val;
void print_val () {
   std::unique_lock<std::mutex> lck (mtx1);
   std::cout << global_val << std::endl;
 }
int main () {
    std::thread th1 (print_val);
     std::thread th2 (print_val);
     th1.join();
     th2.join();
```

```
void print_val (int n, char c) {
    std::unique_lock<std::mutex> lock1{mtx1, std::defer_lock};
   std::unique_lock<std::mutex> lock2{mtx2, std::defer_lock};
    std::lock(lock1, lock2);
    std::cout << global_val << std::endl;
 }
```

Locking

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

Locking

---shared_lock

- 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();
```

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
- Multithreading and mutex:
  - https://en.cppreference.com/w/cpp/atomic/memory_order
  - 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]
  - https://stackoverflow.com/questions/58443465/stdscoped-lock-or-stdunique-lock-or-stdlock-guard
- Notes:
  - https://thispointer.com/c11-multithreading-part-3-carefully-pass-arguments-to-threads/