CS4414 Recitation 6

Multithreading and Synchronization III

10/03/2025

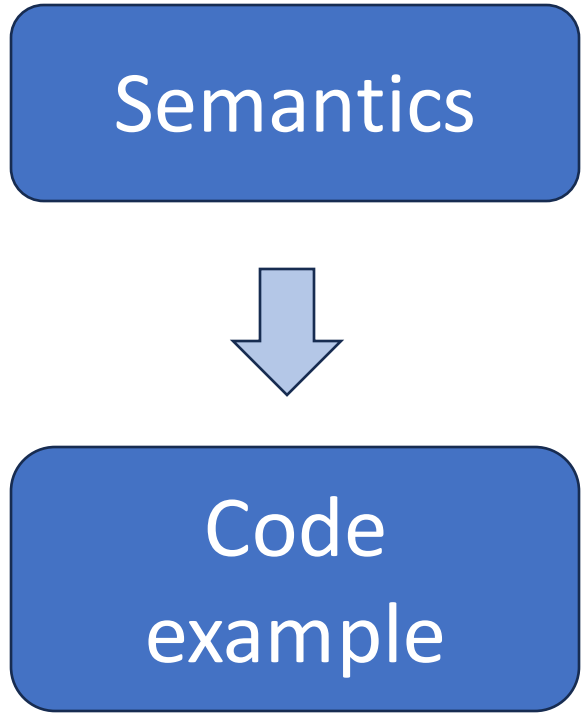
Alicia Yang

Overview

- Multithreading
  - Locking recap
  - Condition variable
- HW2 introduction

Multithreading

- Threads management
  - Launching threads
  - Threads completion
- Synchronization
  - Race condition
  - Atomic
  - **Mutex**
  - **Locks**
  - **Condition variable**



Recap

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

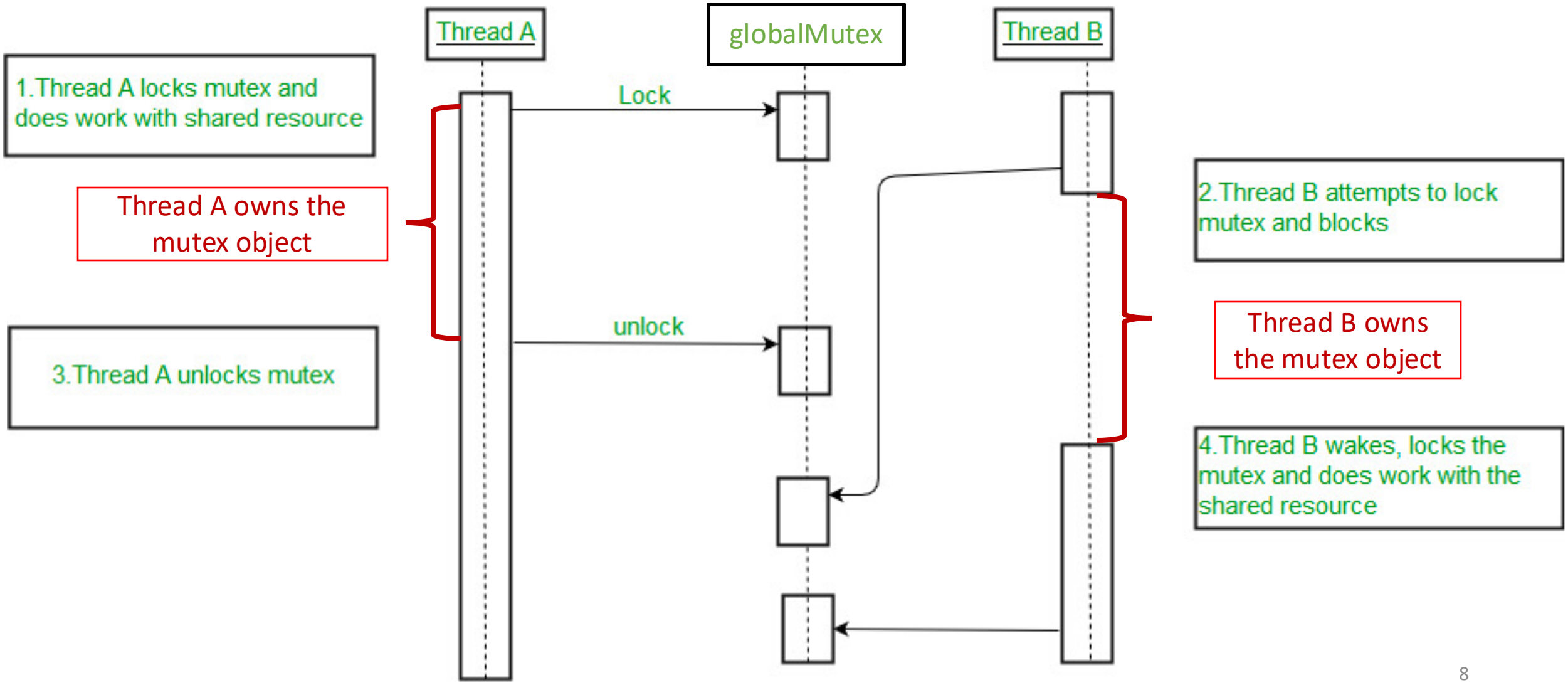
```
1   int       global_num = 0;
2   std::mutex      globalMutex;

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++



Locking

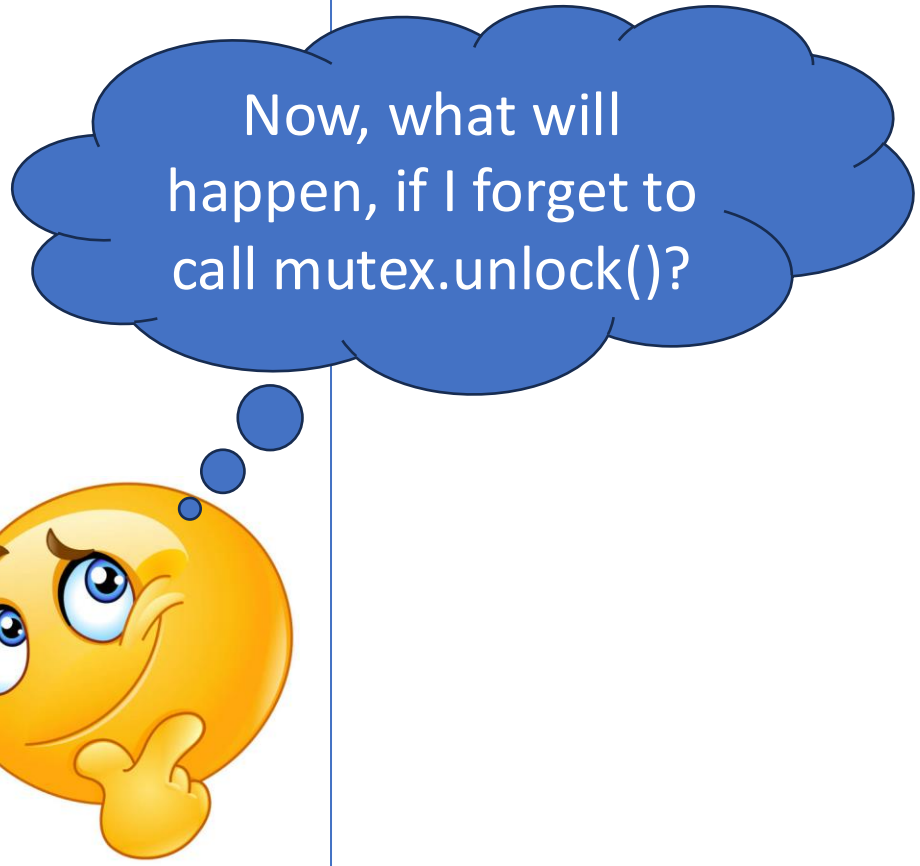
---std::mutex::lock(), unlock()

demo

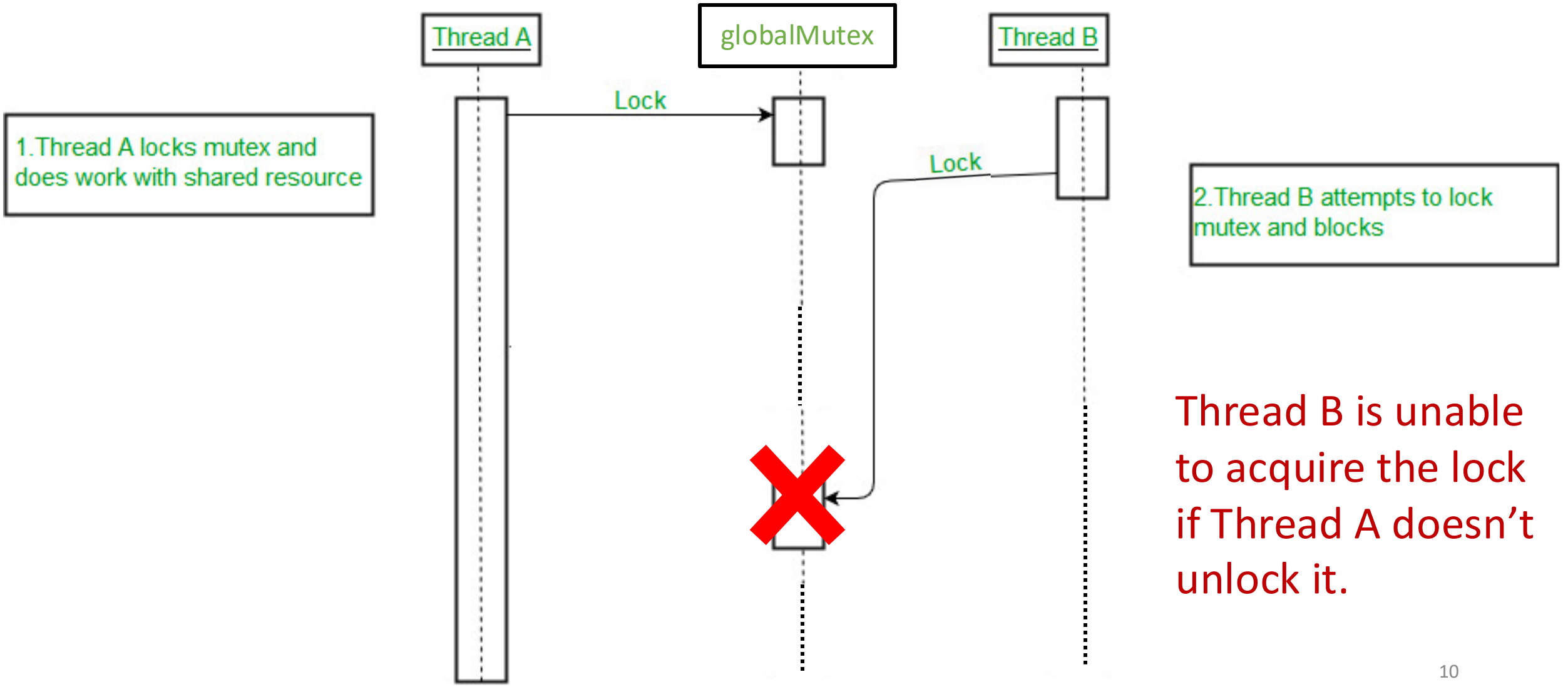
```
int         global_num = 0;
std::mutex          globalMutex;

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)

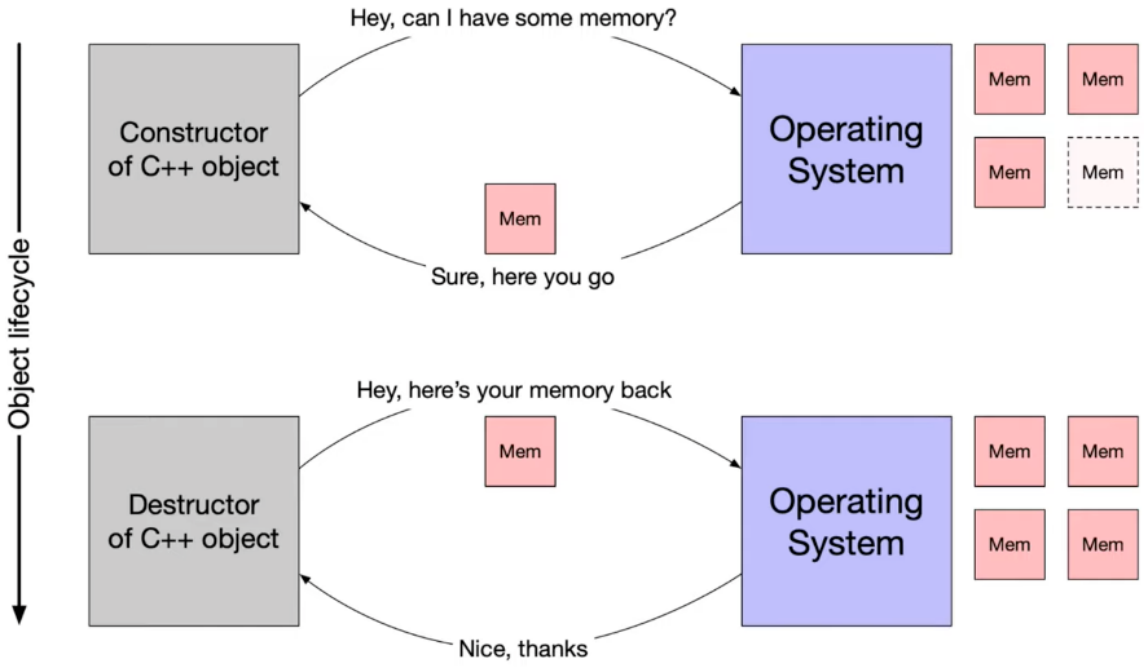
```
// problem #1's fix
{
   int *arr = new int[10];
   delete[] arr;
}
```

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

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

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

```
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

Locking

---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;
}
```

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

```
1   int         global_num = 0;
2   std::mutex          globalMutex;
3
4   void incre(int num){
5               {
6                       std::scoped_lock  s_lock(globalMutex);
7                       global_num = global_num + 1;
8               }
9               global_num = global_num + 1;
10              …
11  }
12
```

Locking

---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();
```

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

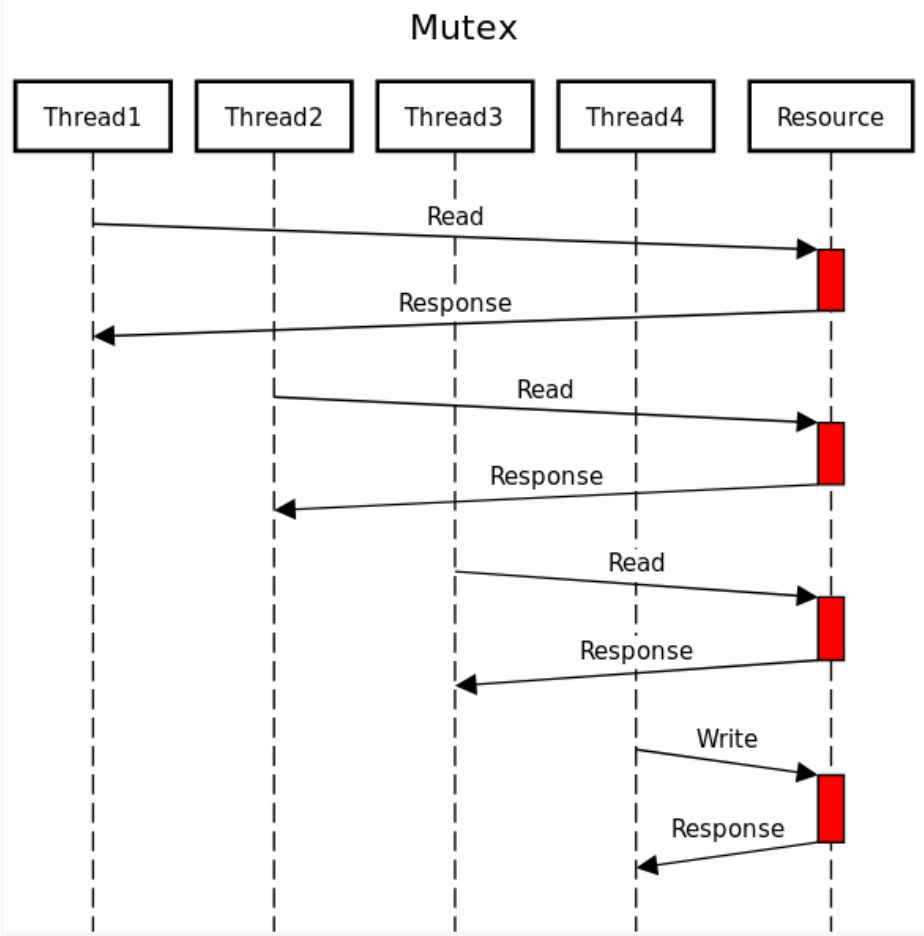
```
// problem #1's fix
{
    std::unique_ptr<int[]> arr(new int[10]);
.....
}
```

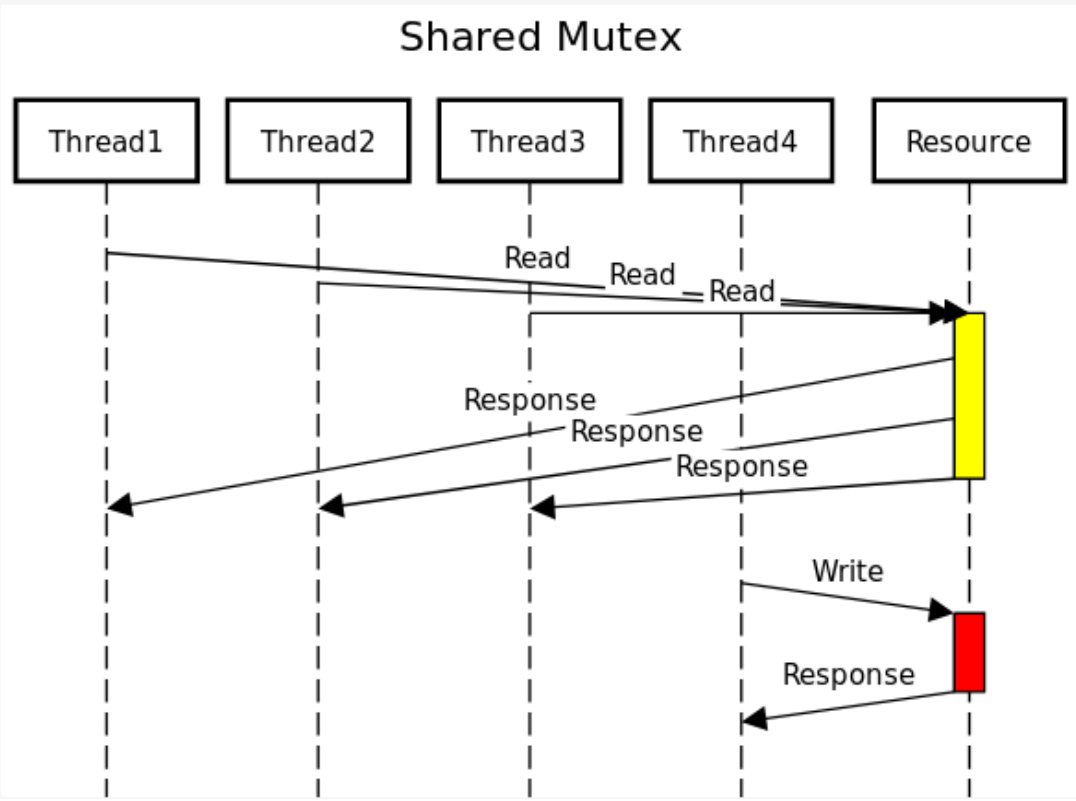
```
// problem #3's fix
Std::mutex globalMutex;
Void func() {
    std::unique_lock<std::mutex> lock(globalMutex);
....
}
```

Exercise --- RW lock

- Reader-writer lock
  - Single writer or multiple reader ownership

Why RW lock?


Mutex
Thread1
Thread2
Thread3
Thread4
Resource
Read
Response
Read
Response
Read
Response
Write
Response

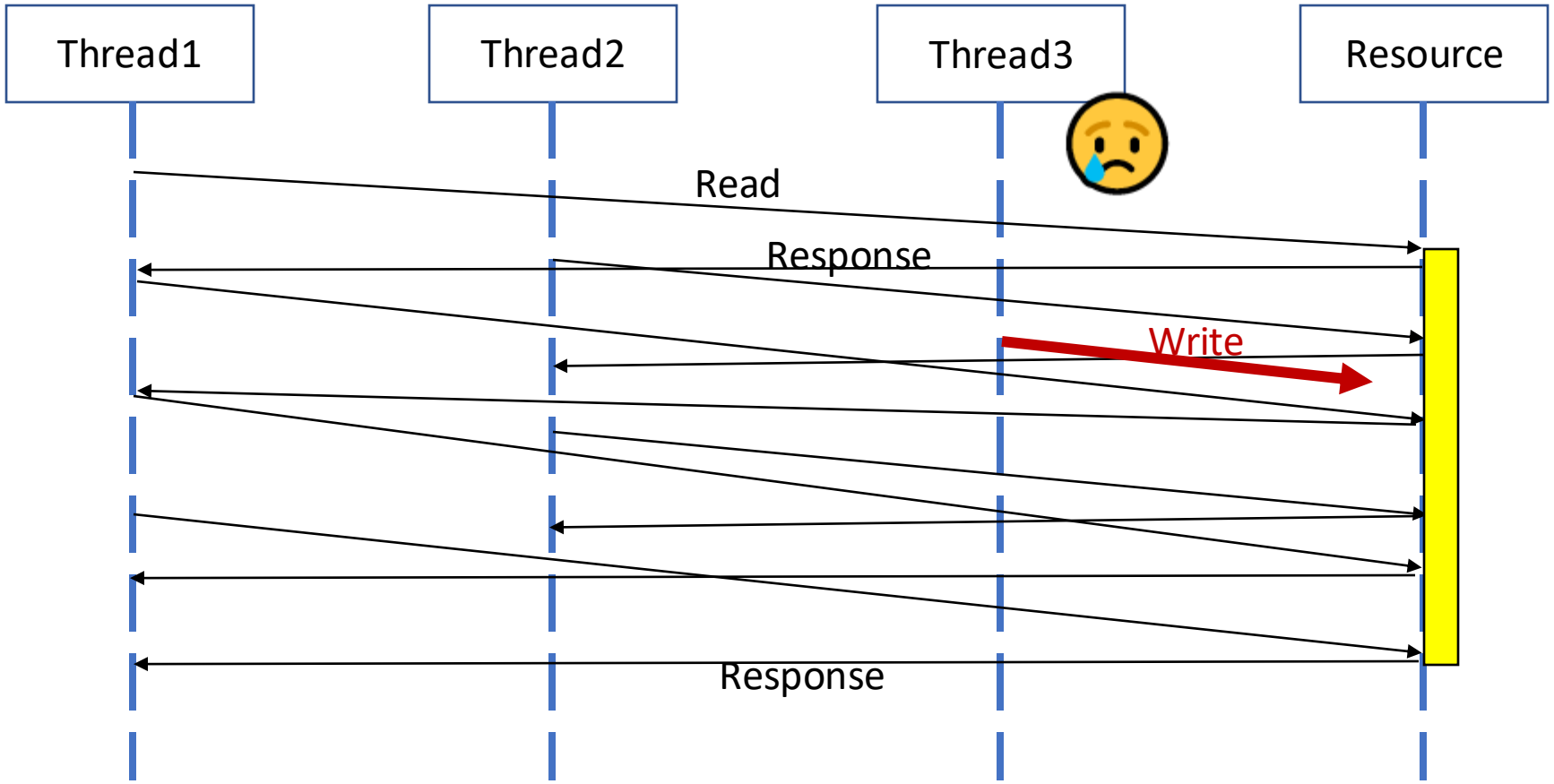

Shared Mutex
Thread1
Thread2
Thread3
Thread4
Resource
Read
Read
Read
Response
Response
Response
Write
Response

Exercise --- RW lock

demo

- Reader-writer lock
  - Single writer or multiple reader ownership
  - Expect higher concurrency when primarily reading
  - std::shared_mutex

What should I do if I want to prioritize the write?



Multithreading

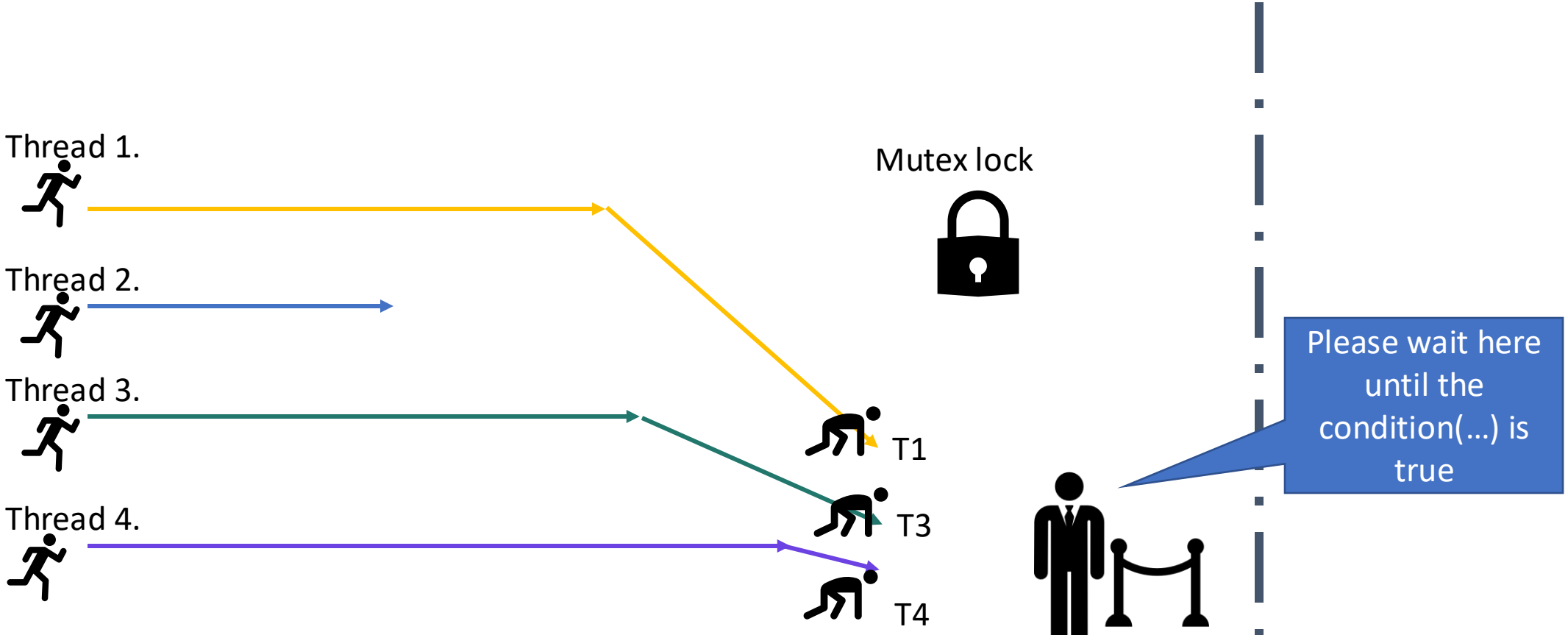
- Threads management
  - Launching threads
  - Threads completion
- Synchronization
  - Race condition
  - Atomic
  - Mutex
  - Locks
  - Condition variables

Condition Variable

Suppose a thread needs to wait for some other threads to do something for it, how would you encode this into the program?

Condition Variable

- Two main purpose of condition variable
  - Notify other threads
  - Waiting for some conditions that other thread can change



Condition Variable

1. Need mutex to use condition variable

   Two roles

   - Waiting threads: first acquire the lock, then wait() if condition not satisfied
   - Notifying threads: thread make the changes that can allow other thread's wait condition to true and move on.

Condition Variable

--- std::condition_variable

`class condition_variable;`	(since C++11)

```
std::condition_variable   cv;
```

Declare a condition_variable object

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

Condition Variable

--- std::condition_variable::wait

1. Need mutex to use condition variable
2. Condition Variable allows running threads to **wait** on some conditions and once the threads wake up
   - Atomically acquire the lock and check the condition
     - If the condition is satisfied, then it will continue the program
     - If not satisfied, it waits by releasing the lock, and goes back to waiting

Two types of wait functions for condition variable

`void wait( std::unique_lock<std::mutex>& lock );`	(1)	(since C++11)
`template< class Predicate >` `void wait( std::unique_lock<std::mutex>& lock, Predicate pred );`	(2)	(since C++11)

Automatically calls lck.unlock() and block *this

Unconditional wait(lock)

```
std::mutex mtx;
std::condition_variable cv;
int main(){
    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck);
    ……
}
```

predicate wait(lock, pred)

Equivalent to while (!pred()) wait(lock);

```
std::mutex mtx;
std::condition_variable cv;
int current_balance = 0;
int main() {
    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck, [] { return current_balance != 0; });
    ……
}
```


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

Two types of wait functions for condition variable

To avoid the affect of spurious wake ups, always use predicate wait() !

Unconditional wait(lock)

```
std::mutex mtx;
std::condition_variable cv;
{
    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck);
    ……
}
```

The thread will be unblocked when notify_all() or notify_one() is executed.

predicate wait(lock, pred)

```
std::mutex mtx;
std::condition_variable cv;
int current_balance = 0;
int main() {
    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck, [] { return current_balance != 0; });
    ……
}
```

Condition Variable --- wait

- When a thread calls the member function wait() on a condition variable
  - The execution of the current thread (which currently has the locked's mutex) is blocked until notified.
  - When the thread is blocked, the function automatically calls unlock(), allowing other threads to acquire the lock and continue.

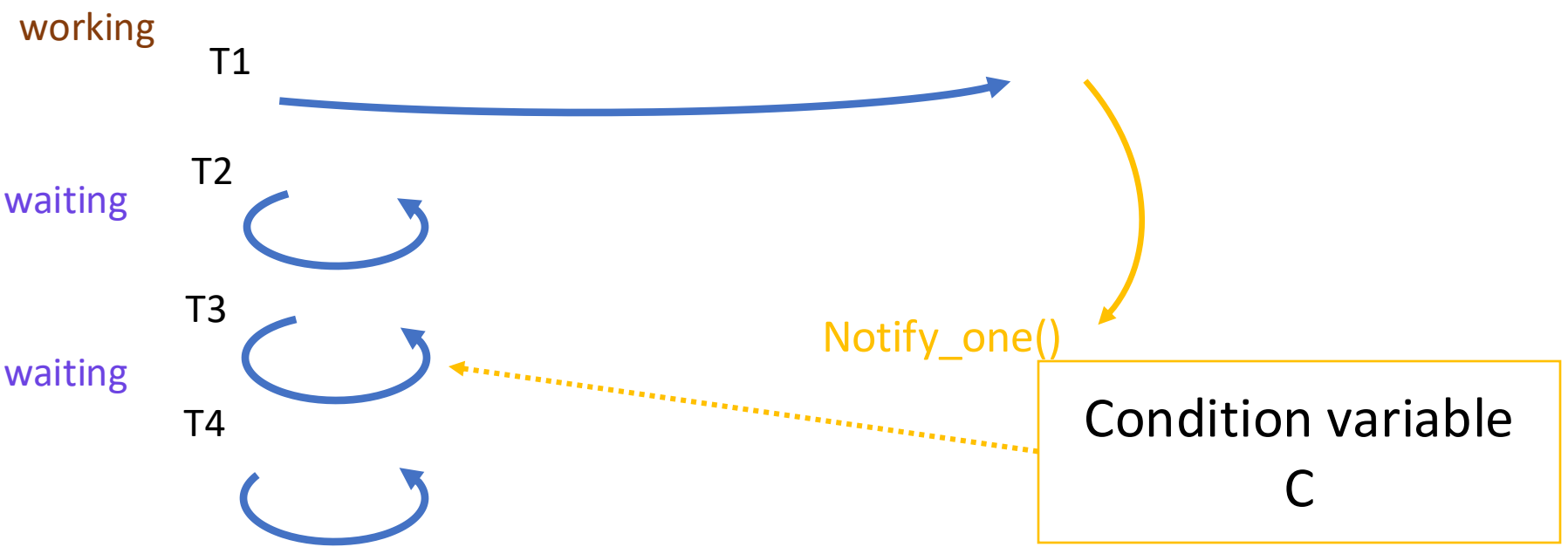
- The wait function performs three atomic operations:
  - The initial unlocking of mutex and simultaneous entry into the waiting state.
  - The unblocking of the waiting state.
  - The locking of mutex before returning.

Condition Variable --- notify

1. Need mutex to use condition variable
2. Condition Variable allows running threads to wait on some conditions
3. The waiting thread(s) is notified by working thread using:
   - notify_one();
   - notify_all();

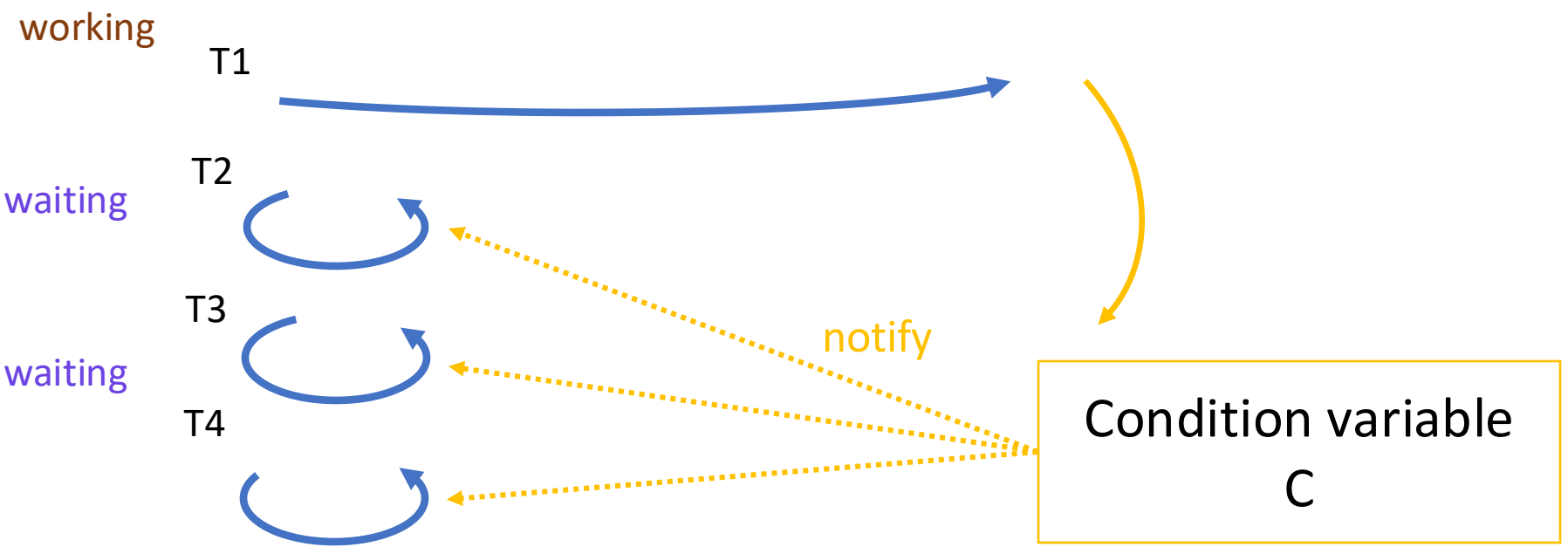
Condition Variable --- notify

- The waiting thread is notified by working thread using:
  - notify_one():
    - Unblocks one of the threads currently waiting for this condition.
    - If no threads are waiting, the function does nothing.
    - If more than one, it is unspecified which of the threads is selected.



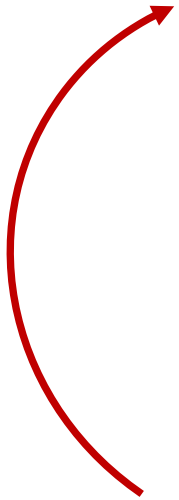
Condition Variable --- notify

- The waiting thread is notified by working thread using:
  - notify_all():
    - Unblocks all threads currently waiting for this condition.



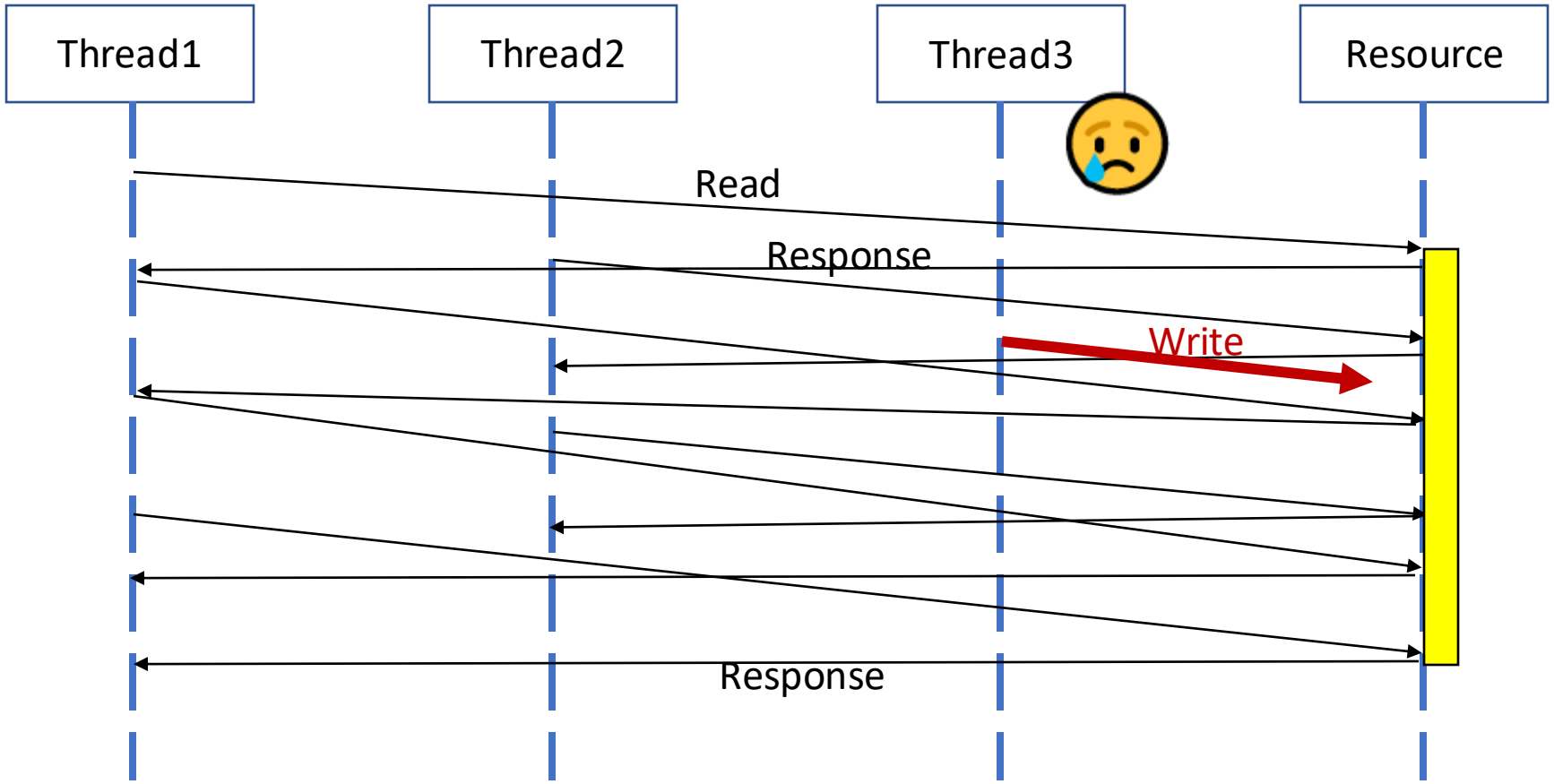
Condition Variable

1. Each thread first acquire the mutex lock
2. Then check the condition in wait()
3. Waiting thread(s) is notified by working thread
4. When thread(s) waiting at the condition variable gets notified,
   - it first try to acquire the lock of mutex
   - Check the condition, the thread will not go further until the condition is true:
     - if it is true, then go further;
     - if it is not, it will again wait for the condition variable



Demo

What should I do if I want to prioritize the write?



Exercise --- RW lock

demo

- Reader-writer lock
  - Single writer or multiple reader ownership
  - Expect higher concurrency when primarily reading
  - std::shared_mutex
  - Read/write preference

Where to find the resources?

- RW Lock: https://www.youtube.com/watch?v=KJS3ikoiLso
- Condition Variable:
  - https://www.cplusplus.com/reference/condition_variable/condition_variable/wait/
- Future and promise:
  - https://www.cplusplus.com/reference/future/async/
  - https://en.cppreference.com/w/cpp/thread/future/wait_for