CS4414 Recitation 13

Prelim2 Review

11/22/2024

Alicia Yang

Disclaimer: due to time constraints, the review recitation can not cover everything lectures/recitations that could be appeared on the exam. The selections of topics are based on questions that raised most commonly on Ed. Please also review other lectures/recitations that are not covered here to have a better preparation.

Logistics

Additional review session

- TA: **Abhijeet Saha**
- Time: **12/3 (Tuesday) 7-8PM**
- Location: **Gates G01**

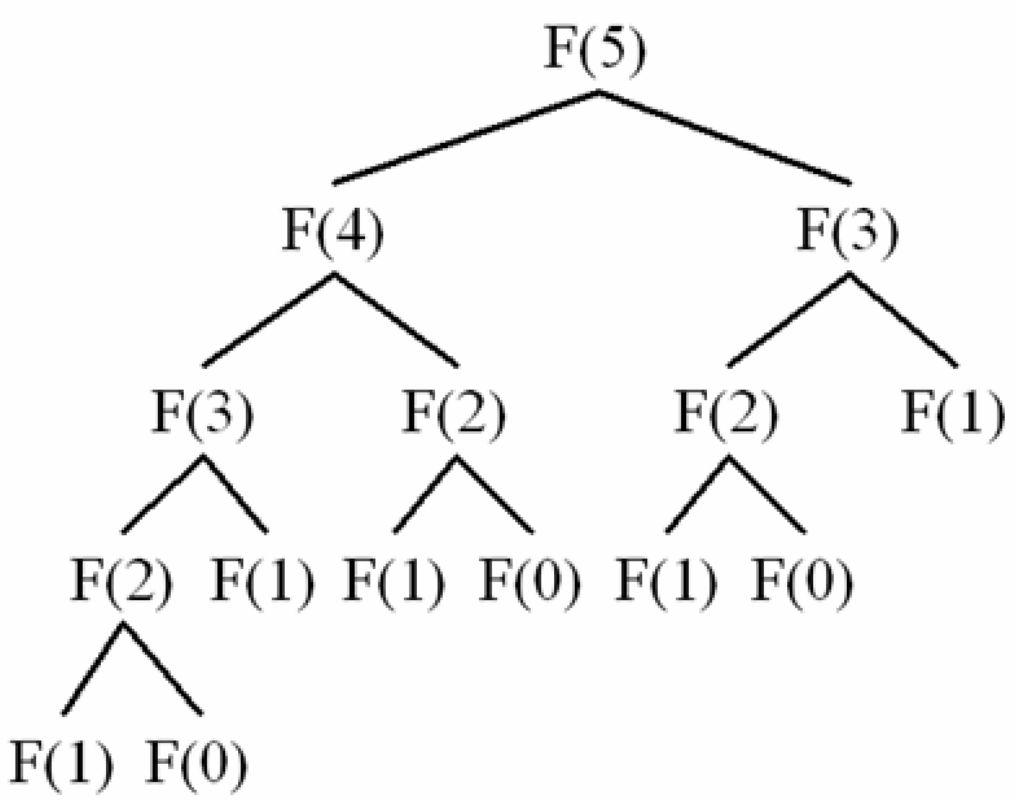
C++ Compile-time and runtime concepts

- Constexpr (lecture 9)
- Templates (lecture 10, recitation 6)
- Linking (lecture 13, recitation 8)

fibonacci

```
int fibonacci(int n)
{
        if(n <= 1)
                return n;
        return fibonacci(n-2)+fibonacci(n-1
}
```

```
fibonacci(5) = fibonacci(3)+Fibonacci(4)
fibonacci(4) = fibonacci(2)+Fibonacci(3)
fibonacci(3) = fibonacci(1)+Fibonacci(2)
fibonacci(2) = fibonacci(0)+Fibonacci(1)
fibonacci(1) = 1
Fibonacci(0) = 1
```



Due to repeatitive pattern, requires 15 calls to Fibonacci!

C++ "CONST" ANNOTATION

- Expresses the **promise** that something will not be changed.
- The compiler can then use that knowledge to produce better code, in situations where an opportunity arises.
- Can only be used if you genuinely won't change the value!

Example of declaring a compile-time constant

```
const int MAXD = 10000;  // length of myvec
Char myvec[MAXD];
```

Example of marking an argument to a method with const

```
void point_x (const int& x){
   std::cout << x << std::endl;
}
```

constexpr

- This keyword says that “this expression should be entirely constant”. *The expression can even include function calls.*

- C++ will complain if for some reason it can’t compute the result at **compile time**: a constant expression turns into a “result” during the compilation stage.

- If successful, it treats the result as a const.

constexpr

- This keyword says that “this expression should be entirely constant”. *The expression can even include function calls.*

```
constexpr float x = 42.0;
constexpr float z = exp(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

- If successful, it treats the result as a const.

Practice prelim question

3. true/false

A constexpr expression cannot include variables that hold values the program reads from a user or from some other kind of input.

If a constexpr performs a zero divide, then when you run C++ to compile the code, the compiler will exit with a zero-divide exception.

gprof won't count the time C++ spends evaluating a constexpr when it prints a formatted profile report for the program.

Practice prelim question

3. true/false

Constexpr needs to be evaluated at compile-time

T A constexpr expression cannot include variables that hold values the program reads from a user or from some other kind of input.

Runtime input

F If a constexpr performs a zero divide, then when you run C++ to compile the code, the compiler will exit with a zero-divide exception.

Compile-time error

T gprof won't count the time C++ spends evaluating a constexpr when it prints a formatted profile report for the program.

Gprof measures the program's runtime profiling

C++ Compile-time and runtime concepts

- Constexpr (lecture 9)
- Templates (lecture 10, recitation 6)
- Linking (lecture 13, recitation 8)

The goal for template

- **Compile time** type checking and type-based specialization.
- A way to create classes that are specialized for different types
- **Conditional compilation**, with dead code automatically removed
- Code polymorphism and varargs without runtime polymorphism

The basic idea is extremely simple

- Suppose we have an array of objects of type int:

```
int      myArray[10];
```

- With a template, the user supplies a type by coding something like Things<long>, like:

```
template<Typename T>
T        myArray[10];
```

- More example: std::vector

```
template<
   class T,
   class Allocator = std::allocator<T>
> class vector;
```

Template class

```
template<typename T>
class Things {
    T       myArray[10];
    T       getElement(int);        // People often index by a constant, hence not int&

    void  setElement(int,T&);
}
```

Template functions

- Templates can also be associated with individual functions. The entire class can have a type parameter, but a function can have its own (perhaps additional) type par

This really should require that T be a type supporting "comparable". We'll see how to specify that restriction in a moment.

```
Template<typename T>
T max(T a, T b)  // Again, not T& to allow caller to provide a constant
{
        return a>b? a : b;                    // T must support a > b
}
```

Template: compile-time check

```
template<class T, T::type n = 0>
class X;
struct S {
        using type = int;
};
using T1 = X<S, int, int>; // error: too many arguments
using T2 = X<>; // error: no default argument for first template parameter
using T3 = X<1>; // error: value 1 does not match type-parameter
using T4 = X<int>; // error: substitution failure for second template parameter
using T5 = X<S>; // OK
```

- Templates are expanded at **compiler time**
- it does **type-checking** before template expansion.

https://en.cppreference.com/w/cpp/language/templates

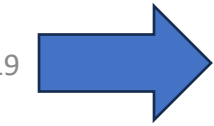
Practice prelim question

3. true/false

T Template code is expanded (as much as possible) at compile time. As a result, gdb and the profiler won't necessarily be able to associate bugs that cause a crash to the proper line within the template, or give proper runtime cost-accounting for templated methods

In gdb or gprof, a variable with a templated type will often have more

T type-signature content than you used to define that variable, because of expansion of default template type parameters and argument.



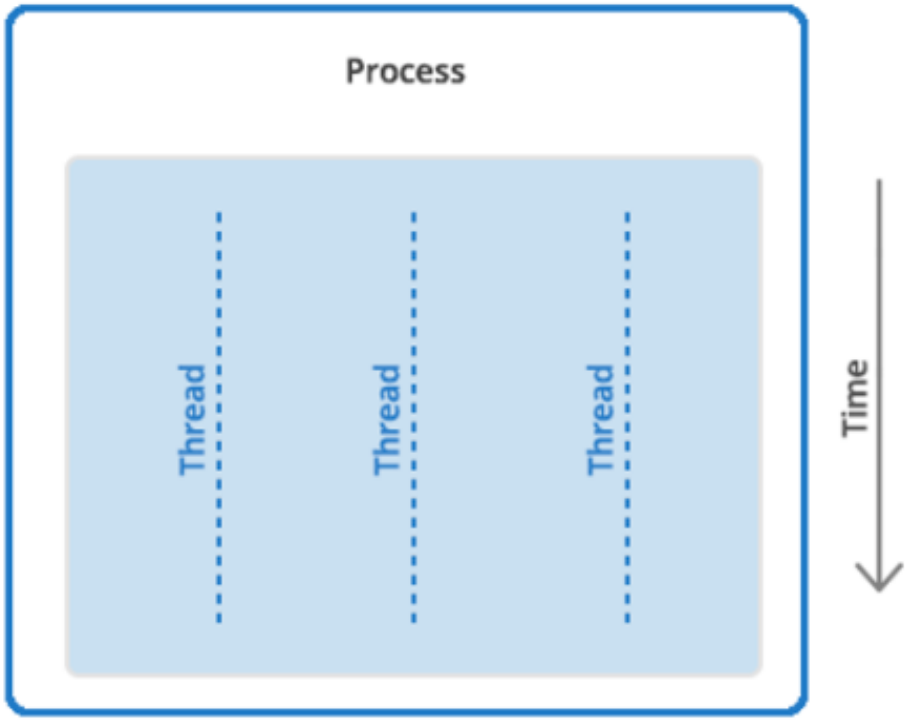
Multithreading

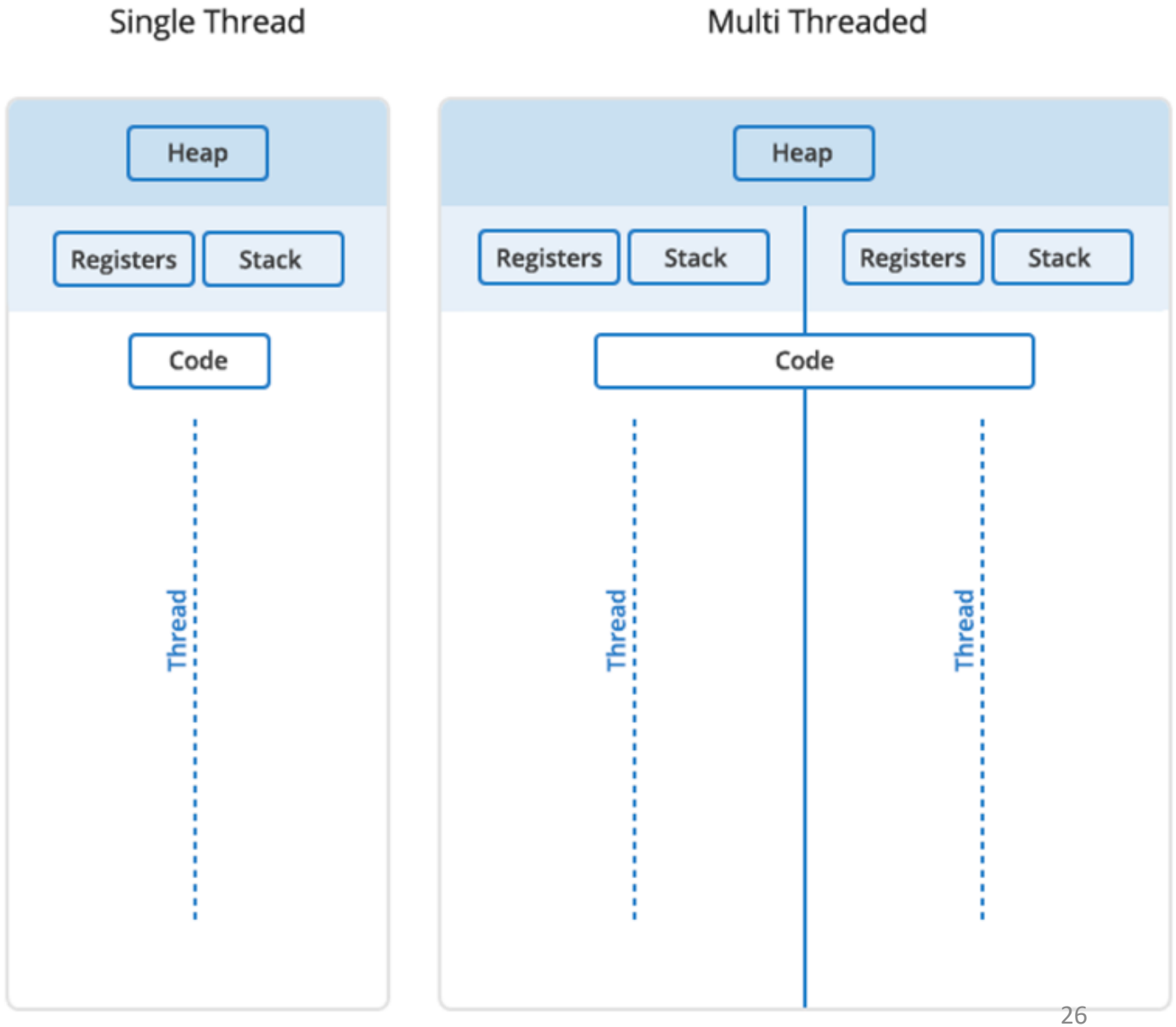
- Lecture 14-17
- Recitation 9-11

Multithreading

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

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 --- function pointer

- Launching a thread using function pointers and function parameters

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

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

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**

Multithreading

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

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

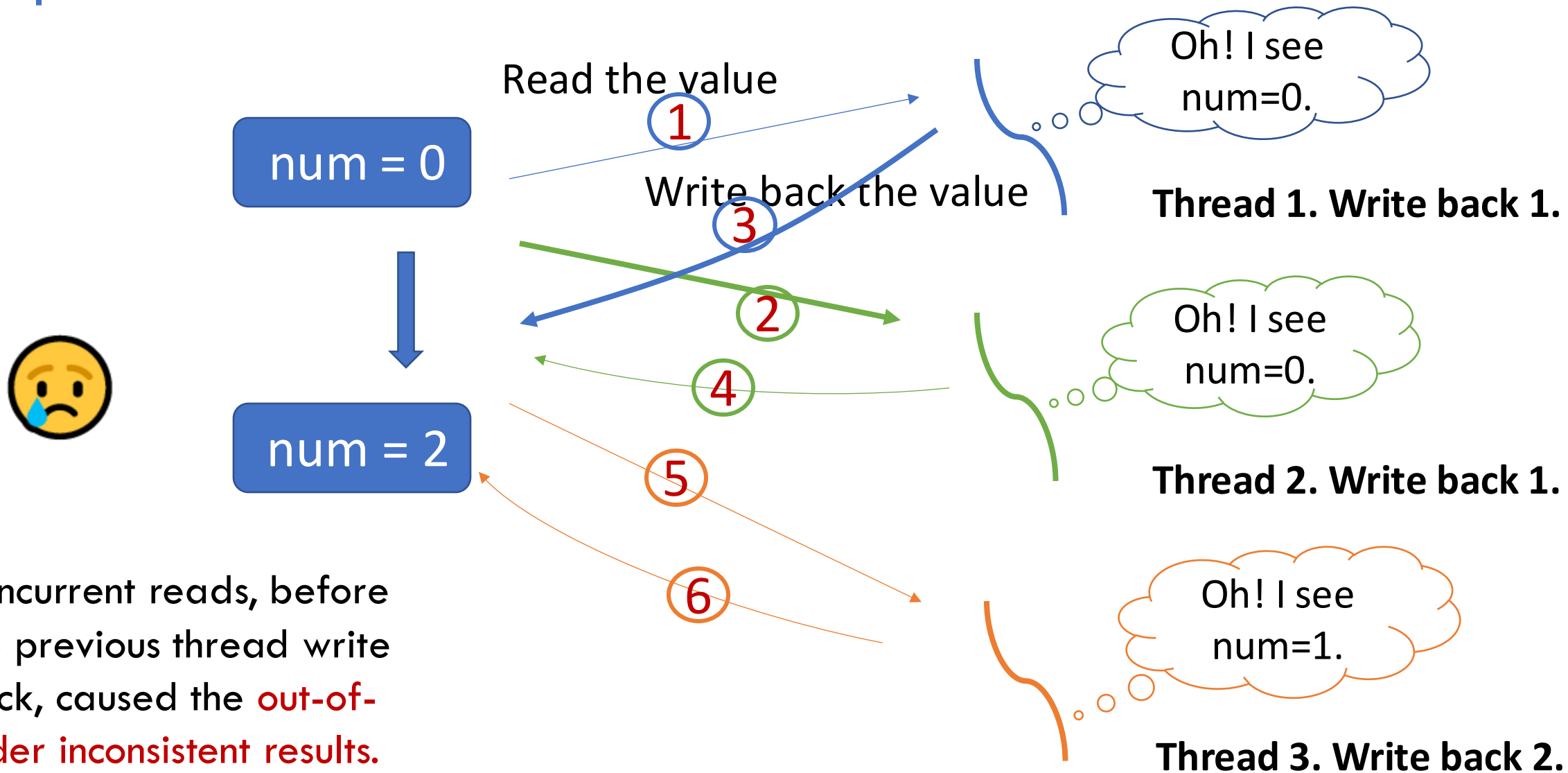
---race condition

- 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

Example: Concurrent increments of a shared integer variable



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

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)

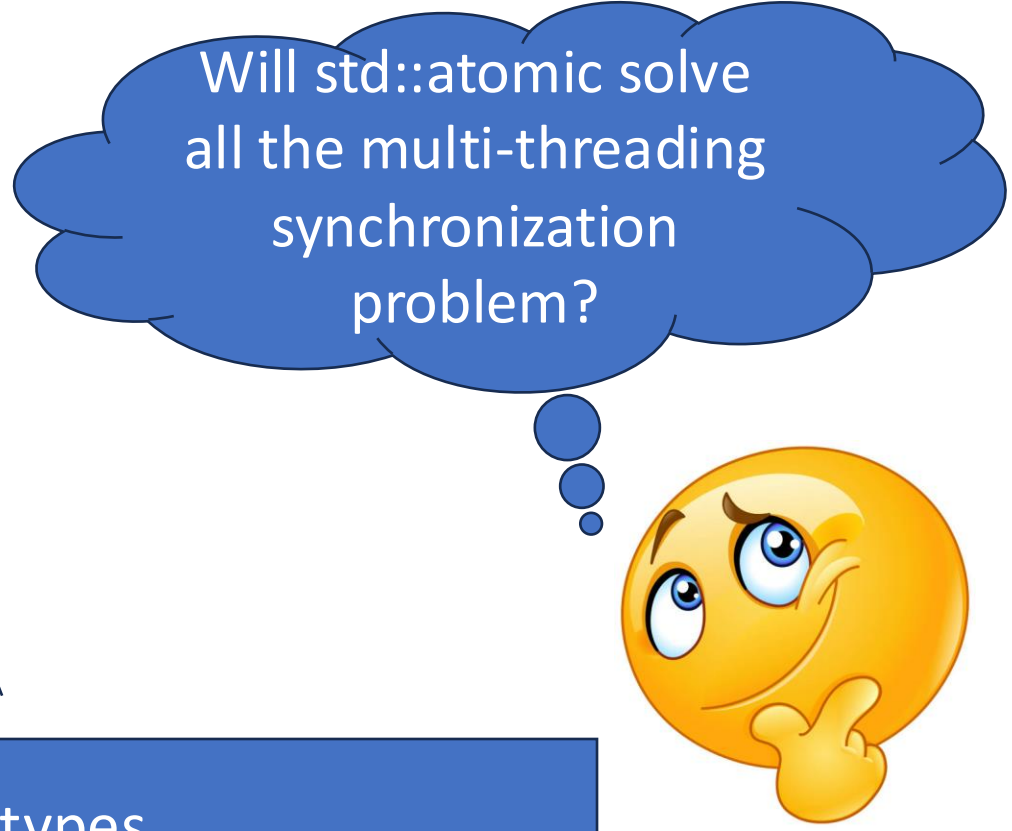
https://en.cppreference.com/w/cpp/atomic/atomic

Atomic 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);`
- load() `bool z = x.load();`
- operator= `y = 2;`
- operator+=, operator -= `y += 1;` `y.fetch_add(1);` (since C++20)
- operator++, operator-- `y ++;`

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

Atomic member functions



- Atomic type: std::atomic<type>
- Constructor
- store()
- load()
- operator=
- operator+=, operator -=
- operator++, operator--

Only for specific types.
- Full specializations:

Character types, Standard signed integer types, Standard unsigned integer types, Integral types …
- Partial specializations:

All pointer types

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

std::vector

- Does std::vector<T> guarantee thread-safety?

Not necessarily

- What about std::atomic<std::vector<T>>? Is this thread safe?

Not necessarily

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

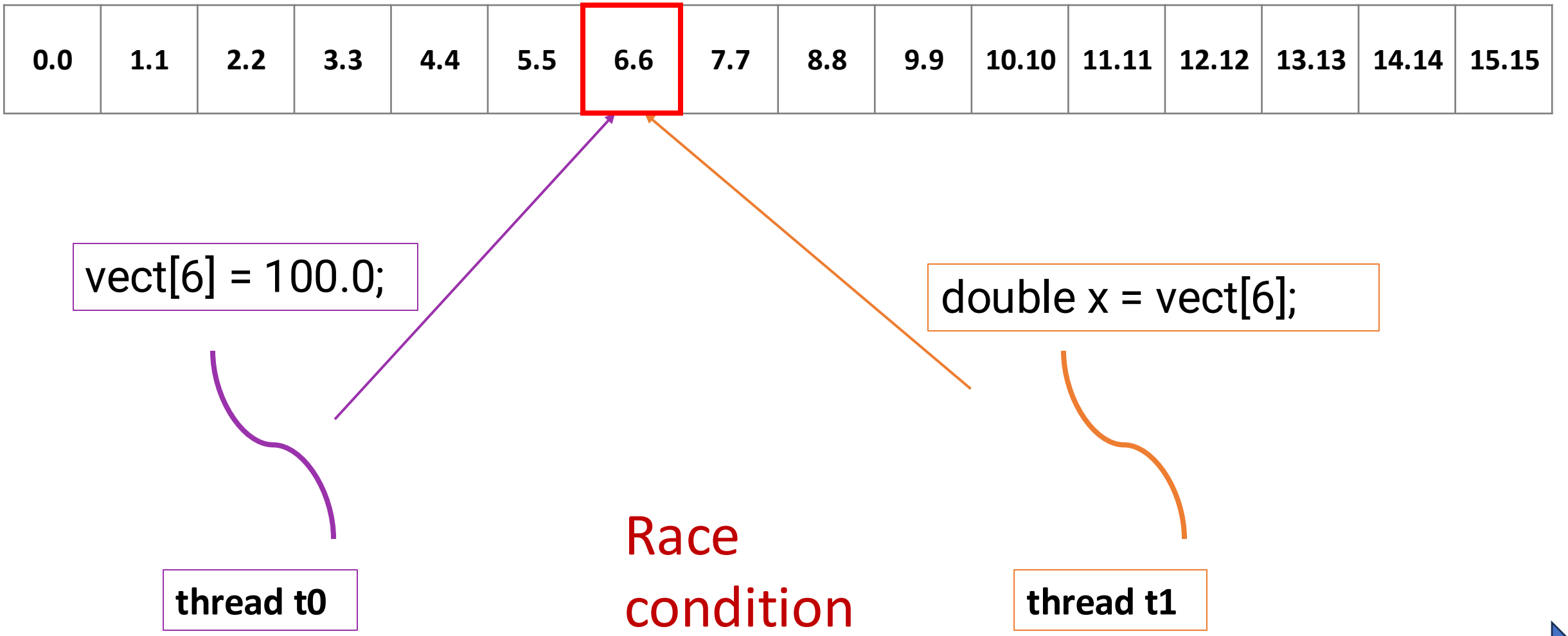
Read-only-access of std::vector

```
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



Multithreading

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

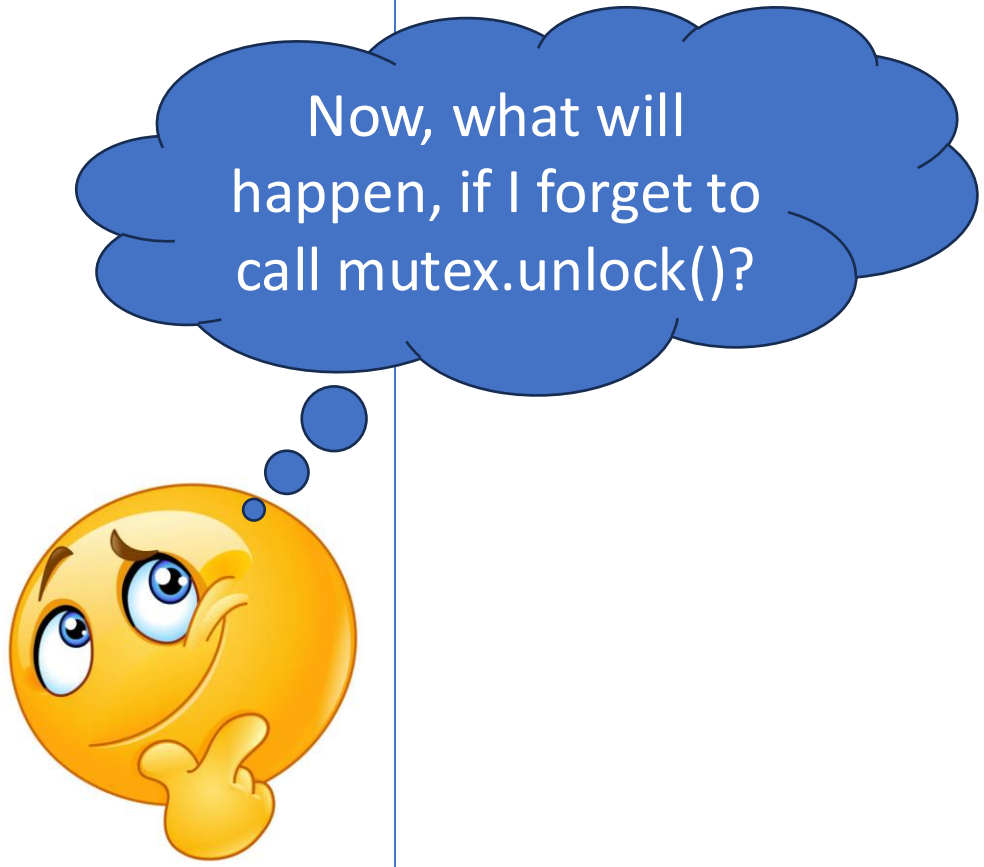
Recap Mutex and Lock in C++

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

```
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();
}
```



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 😢
```

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

---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

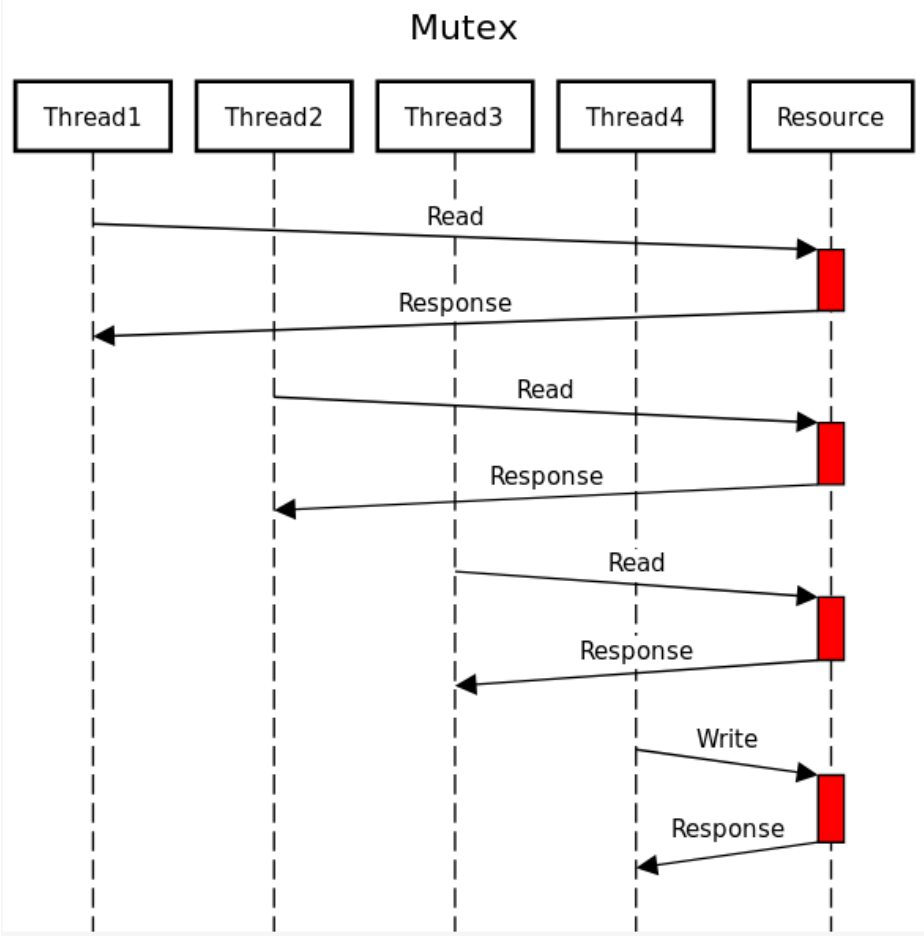
fixes

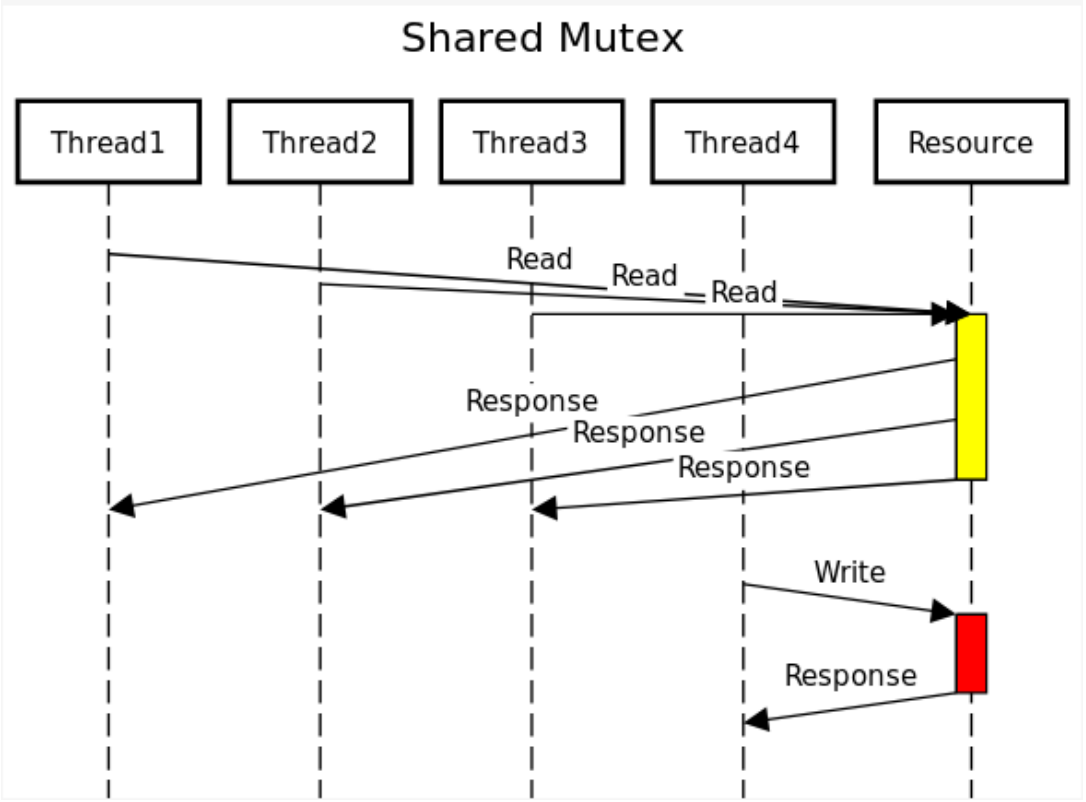
```
// problem #1's fix
{
   std::unique_ptr<int[]> arr(new int[10]);
.....}
```

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

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

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

RW lock simple implementation

std::shared_mutex two level of access:

- **exclusive**: If one thread has acquired the exclusive lock, no other threads can acquire the lock (including the shared).

- **shared**: If one thread has acquired the shared lock, no other thread can acquire the exclusive lock, but can acquire the shared lock.

```
std::shared_mutex mutex_;
int value_ = 0;                                  // Global variables

unsigned int get() const
  {
    std::shared_lock lock(mutex_);               // Multiple threads/readers can read the counter's value at the same time.
    return value_;
  }

 void increment()
  {
    std::unique_lock lock(mutex_);               // Only one thread/writer can increment/write the counter's value.
    ++value_;
  }
```

RW lock simple implementation

```
std::shared_mutex mutex_;
int value_ = 0;

unsigned int get() const
   {
      std::shared_lock lock(mutex_);
      return value_;
   }

 void increment()
   {
      std::unique_lock lock(mutex_);
      ++value_;
   }
```

```
int main() {
  std::thread reader_thread([]() {
              unsigned int val = get();
              std::cout << val << '\n';
              });
   std::thread writer_thread([]() {
              increment();
              });
   reader_thread.join();
   writer_thread.join();
…}
```

Practice prelim

1. true/false

When using the std::scoped_lock type to create a lock, you do need to specify a mutex object but do not need to give the std::scoped_lock a variable name, because you would never perform any operations on the object.

It is not necessary to use a std::mutex to protect shared objects marked as “const”.

The readers and writers pattern can be used to safely protect an STL object like a std::list or std::map that will be read by some threads and written by other threads.

Practice prelim

1. true/false

F When using the std::scoped_lock type to create a lock, you do need to specify a mutex object but do not need to give the std::scoped_lock a variable name, because you would never perform any operations on the object.

T It is not necessary to use a std::mutex to protect shared objects marked as "const".

const promises no changes to the variable

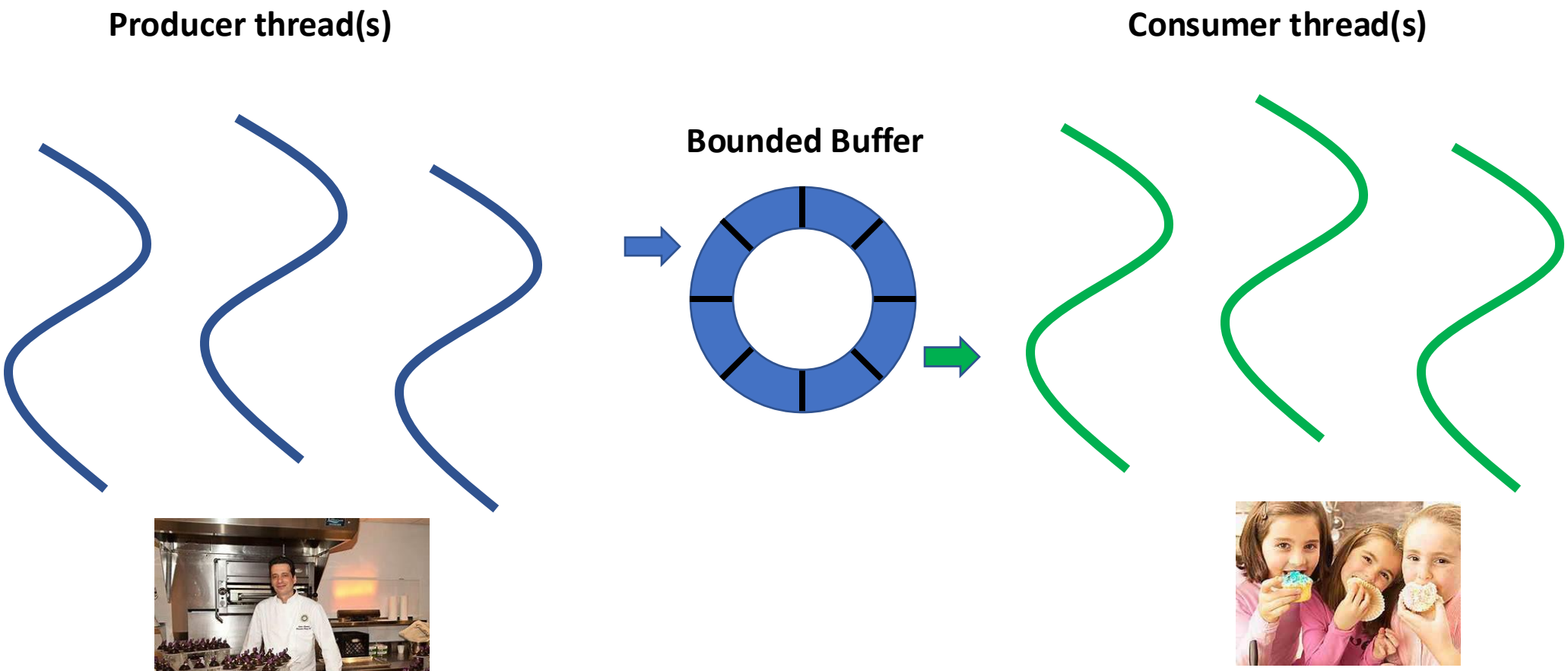
T The readers and writers pattern can be used to safely protect an STL object like a std::list or std::map that will be read by some threads and written by other threads.

Multiple threads could read at the same time, but writer requires exclusive access

Coordination

- Lecture 18, 19

Producer – consumer Pattern



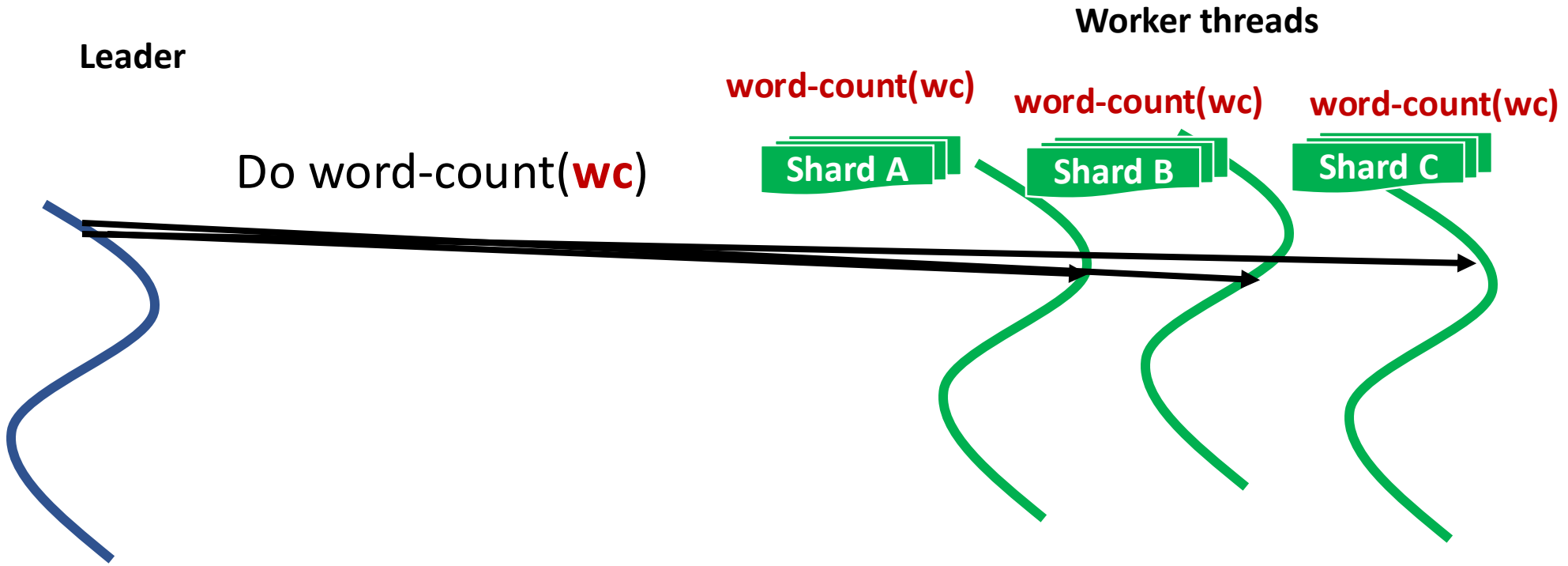
CCL Pattern (All-Reduce and Map-Reduce)

It assumes that there is a large (key,value) data set divided so that worker k has the k'th shard of the data set.

- For example, with integer keys, perhaps (key % n) == k
- With arbitrary objects, you can use the built-in C++ "hash" method.

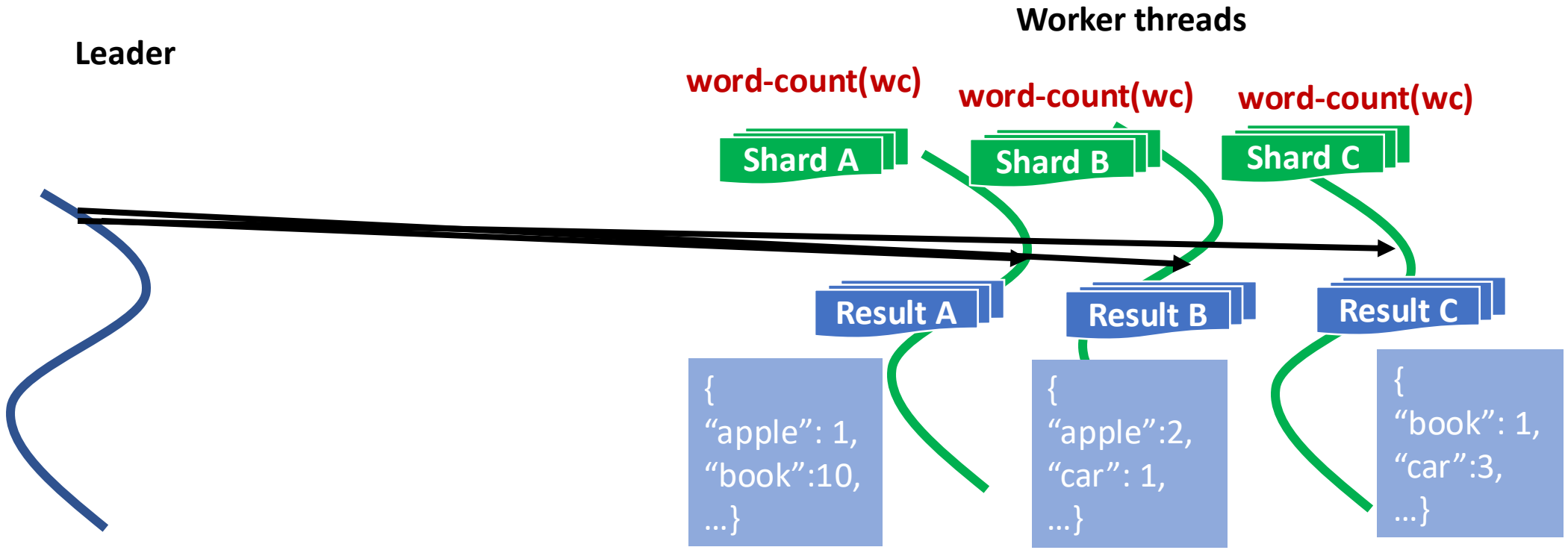
All-Reduce pattern: Map (first step)

example



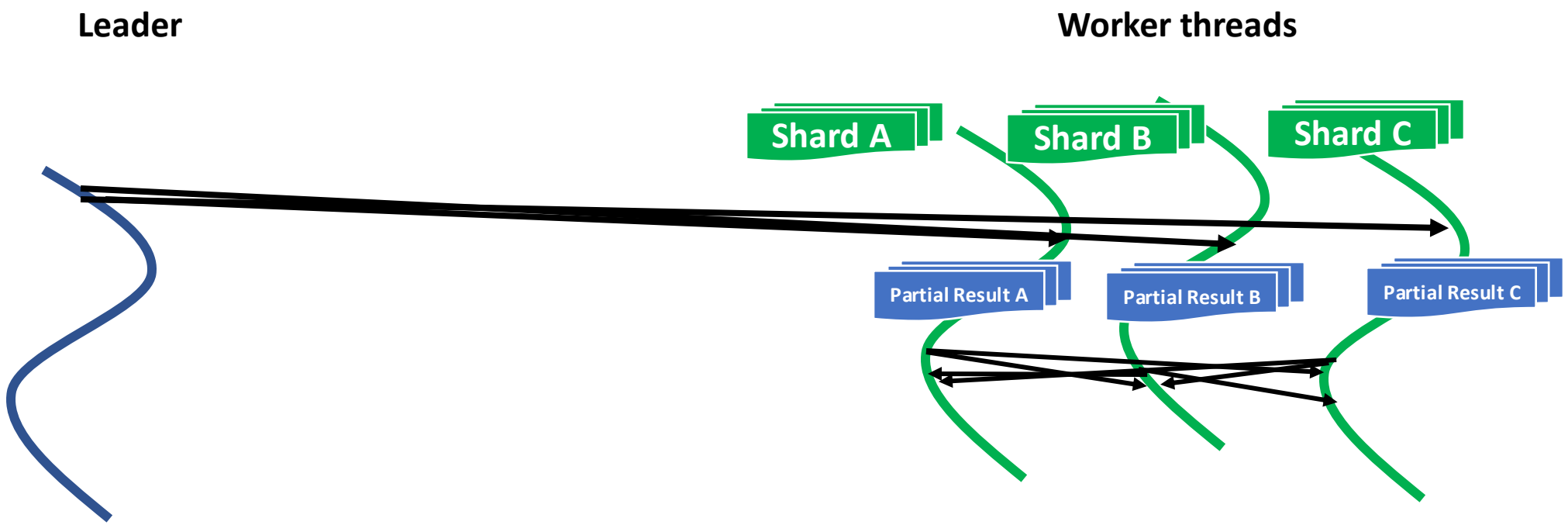
- The leader **maps** some task over the n workers.
- Each worker applies the requested function to the data in its shard.

All-Reduce pattern: Map (first step)

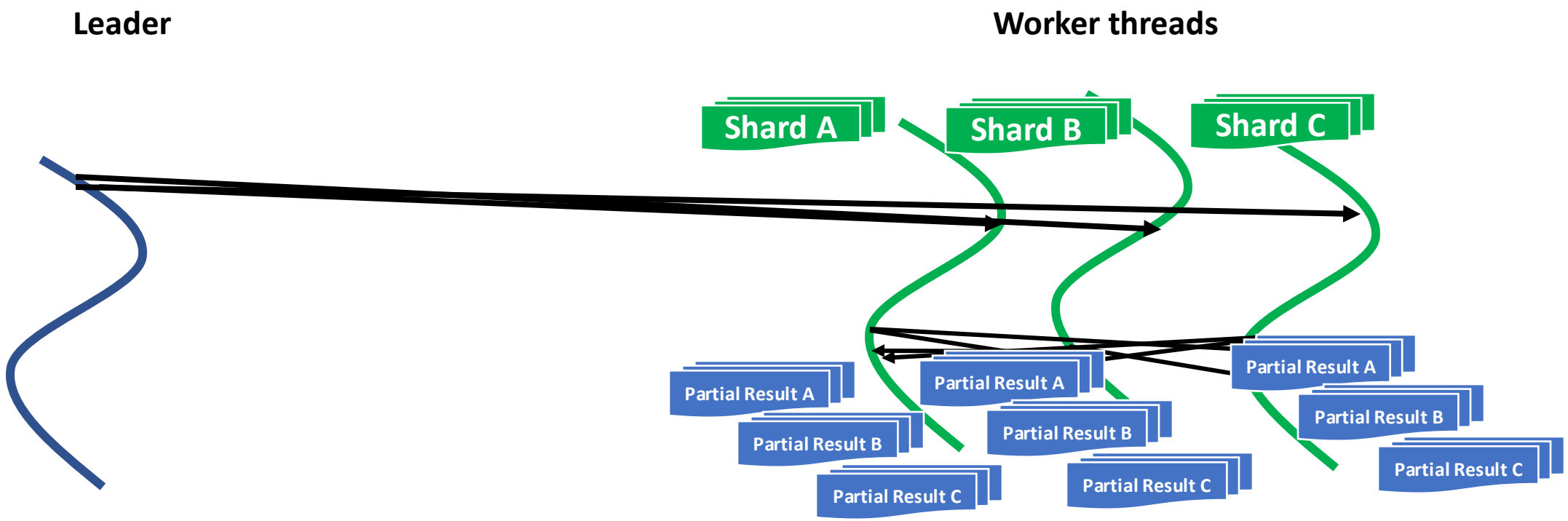


- When finished, each worker will have a list of new (key,value) pairs as its share of the result.

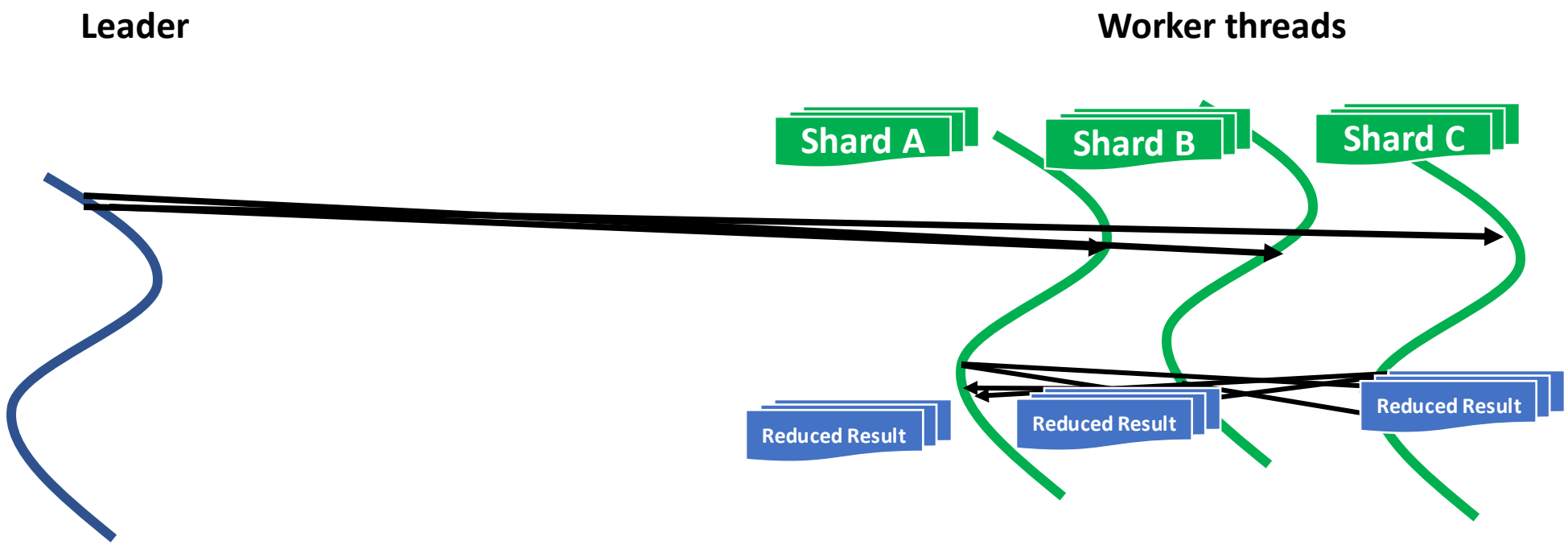
All-Reduce pattern: Shuffle



All-Reduce pattern: Shuffle



All-Reduce pattern: Shuffle



With AllReduce, at the end of the pattern all participants have identical "replicas" of the reduced result. The map step is usually the slow one, and reducing is usually fast

Map-reduce pattern

- With Map-Reduce, each worker ends up with a ***distinct share*** of the results. Data is "spread out" at the start and at the end. Useful if the final result would be too big to hold on a single computer.

- Instead of a set of all-to-all broadcasts, MapReduce uses point-to-point messages: worker1 sends data intended for worker2 only to worker2, etc.

Practice prelim

2. true/false

T MapReduce is valuable if a data set is so large that it can't fit on any one computer and must be split into pieces ("sharded") and spread over many computers.

F Unlike parallel computing, a MapReduce computation is sequential. Every shard will be processed but the computations occur one by one, with each worker running in turn in an order decided by the leader.

What have we learnt in this class?

System programming

- Modern systems: various types of resources (memory, CPU cycles, files…), OS, file systems, …
- How to realize the best performance of the systems, via the hardware, the compiler, the linker, etc; performance analysis, and profiling.
- Coordination in a systems: multiple threads, multiple processes, perhaps multiple computers, perhaps even attached hardware accelerators that are themselves programmable.

How to write **good** system program in C++

1. clean and correct code

2. Develop efficient system

What is C++?

A federation of related languages, with four primary sublanguages

- **C:** C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** "C with Classes", classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classed.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

Write clean and correct code

- The basics: C++ types, variable … 
- Classes and functions 
- Memory management in C++, RAII principle 
- Smart pointers in C++ 
- C++ templates 
- Standard containers – std::vector<T>, std::map<K,V> 

Develop efficient system

- Cmake for large system compilation management, gprof for program profiling 
- Make efficient use of hardware
  - Hardware parallelism 
  - Multithreading and synchronization