CONCURRENCY IN C++

Yuqing Xia
CSCI 5828
Prof. Kenneth M. Anderson
University of Colorado at Boulder

OUTLINE

- Introduction
- Importance of Concurrency
- Threads launch
- Protection on shared data
  - Atomic
  - Mutex
- Communication
  - Condition variables
- Memory model
  - Definition
  - Operaton orders

INTRODUCTION-WHY CONCURRENCY IS NECESSARY

- Processor speed has slowed and we can use transistors from Moore's law for parallelism to increase speed.
- Concurrent programming is necessary to utilize parallel hardware.
- Sometimes it is natural to describe a problem with multi-threads just like divide a problem into several steps.

CONCURRENCY INTRODUCED TO C++11

- Original C++ Stan-dard was published in 1998 only support single thread programming
- The new C++ Standard (referred to as C++11 or C++0x) was published in 2011. It will acknowledge the existence of multithreaded programs. Memory models for concurrency is also introduced.

WHY WRITE CONCURRENT PROGRAMS

- Dividing a problem into multiple executing threads is an important programming technique.
- Multiple executing threads may be the best way to describe a problem
- With multiple executing threads, we can write highly efficient program by taking advantage of any parallelism available in computer system.

CONCURRENCY REQUIREMENT ON PROGRAMMING LANGUAGE

- Thread creation – be able to create another thread of control.
- Thread synchronization – be able to establish timing relationships among threads.
  - One thread waits until another thread has reached a certain point in its code.
  - One threads is ready to transmit information while the other is ready to receive the message, simultaneously.
- Thread communication – be able to correctly transmit data among threads.

THREAD CREATION

- C++11 introduced a new thread library including utilities for starting and managing threads.
- Creating an instance of std::thread will automatically start a new thread.

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

void threadFunction(){
    std::cout << "Hello from thread! \n";
}

int main(){
    std::thread th(threadFunction);
    std::cout<<"Hello from main! \n";
    th.join();
}
```

Two thread will be created. The main thread will launch a new thread when it encounter the code std::thread th( ) to executive the function threadFucntion();

- The join function here is to force the current thread to wait for the thread to th finish. Otherwise the main function may exit without the thread th finished

CRITICAL SECTION

- Data are usually shared between threads. There is a problem when multiple threads attempting to operate on the same object simultaneously.
  - If the operation is atomic(not divisible) which means no other thread can modify any partial results during the operation on the object, then it is safe. Otherwise, we are in a race condition.
- a **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution
- Preventing simultaneous execution of critical section by multiple thread is called mutual exclusion.

EXAMPLE

- Shared objects between threads will lead synchronization issues. For example

```cpp
struct Counter{
    int value;
    Counter():value(0){}
    void increment(){
        ++value;
    }
};
int main(){
    Counter counter;
    std::vector<std::thread> threads;
    for(int i=0;i<5;++i){
        threads.push_back(std::thread([&counter](){
            for(int i=0;i<5000;++i){
                counter.increment();
            }
        }));
    }
    for(auto & thread : threads){
        thread.join();
    }
    std::cout<< counter.value << std::endl;
    return 0;
}
```

5 threads created try to increase the counter 5000 times. This program has a synchronization problem. Here are some result obtained on my computer:

24138
20326
23345
25000
17715

It is not the same every time.

PROTECT SHARED DATA

- The problem is that the increment is not an atomic operation
  - Atomic operation: during which a processor can simultaneously read a location and write it in the same bus operation. Atomic implies indivisibility and irreducibility, so an atomic operation must be performed entirely or not performed at all.
- The increment in the example is made of three operations:
  - Read the current value of vaue
  - Add one to the current value
  - Write that new value to value
- So when you launch more than one thread, they might interleave with each other and make the result impossible to predict.

PROTECT SHARED DATA

- Solutions
  - Semaphores — Mutex is a binary semaphore.
  - Atomic references
  - Monitors — guarantee on thread can be active within monitor at a time. C++ does not support monitor but Java does.
  - Condition variables.
  - Compare-and-swap — It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value
  - Etc.
- Here we will only introduce the most common solutions mutexes and atomic reference in C++.

PROTECT SHARED DATA WITH MUTEXES

- Mutexes(name after mutual exclusion) enable us to mark the code that access the data structure as mutually exclusive so that if any thread was running one of them, any other tread that tried to access the data had to wait until the first thread was finished
- In C++, you create a mutex by constructing an instance of **std::mutex**, lock it with a call to the member function **lock()** and unlock it with a call to the function unlock().

```
struct Counter {
    std::mutex mutex;
    int value;

    Counter() : value(0) {}

    void increment(){
        mutex.lock();
        ++value;
        mutex.unlock();
    }
};
```

- Lock(): enable a thread to obtain the lock to block other thread.
- Unlock(): release the lock to unblock waiting threads.

RAII IDIOM

It is not wise to call the member functions directly because you have to remember to Unlock() on every code path out of a function including those due to exceptions.

The template std::lock_guard implements that Resource Acquisition Is Initialization (RAII) idiom for a mutex

```
struct ConcurrentCounter {
    std::mutex mutex;
    Counter counter;

    void increment(){
        mutex.lock();
        counter.increment();
        mutex.unlock();
    }

    void decrement(){
        mutex.lock();
        counter.decrement();
        mutex.unlock();
    }
};
```

mutex.lock() is called when the instance of std::lock_guard is constructed and mutex.unlock() is called when the instance guard is descontructed.

Because of mutexes, only one thread can do counter.increment() each time ensuring the correctness of our result.

ADANCED LOCKING WITH MUTEXES

- Recursive Lokcing .
  - **std::recursive_mutex**
  - Recursive locking enable the same thread to lock the same mutex twice and won't deadlock.
- Timed Locking
  - **std::timed_mutex,   std::recursive_timed_mutex**
  - Timed locking enable a thread to do something else when waiting for a thread to finish.
- Call once
  - **Std::call_once(std::once_flag falg, function**);
  - It is possible we only want a function to be called only one time no matter how many thread is launched. Each std::call_once is matched to a std:::once_flag variable.

USING ATOMIC TYPES

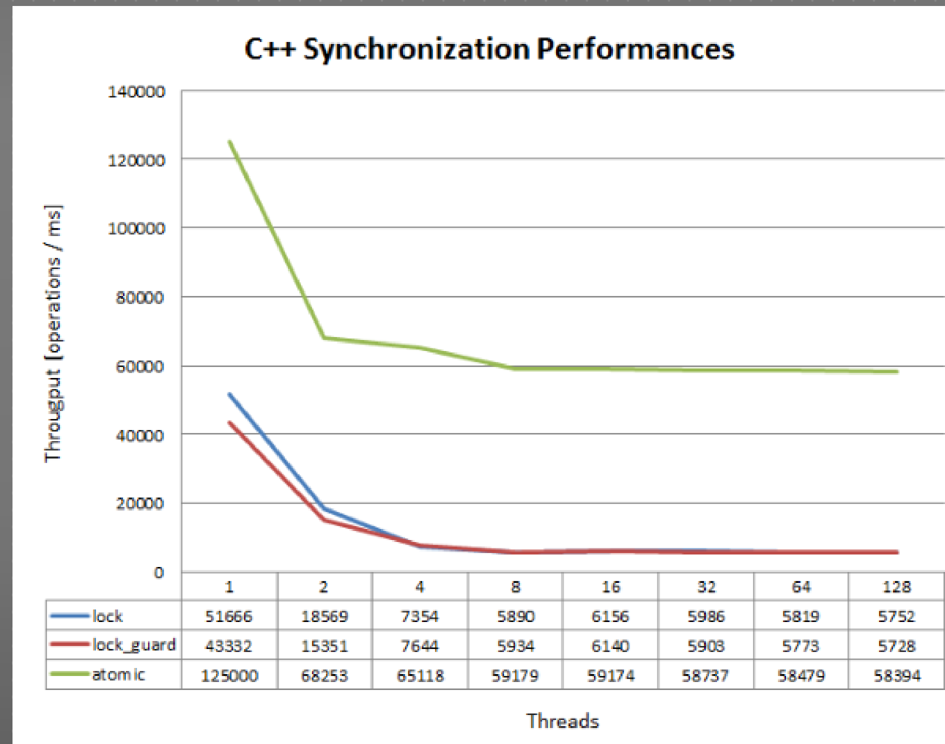
- C++11 concurrency library introduces atomic types as a template class: std::atomic. You can use any type you want with the template and the operation on that variable will be atomic and so thread-safe.
  - std::atomic<Type> object.
- Different locking technique is applied according to the data type and size.
  - lock-free technique: integral types like int, long, float. It is much faster than mutexes technique.
  - Mutexes technique: for big type(such as 2MB storage). There is no performance advantage for atomic type over mutexes.

EXAMPLE OF USING ATOMIC TYPES

The same example with atomic template

```
#include <atomic>

struct AtomicCounter {
    std::atomic<int> value;

    void increment(){
        ++value;
    }

    void decrement(){
        --value;
    }

    int get(){
        return value.load();
    }
};
```

Speed comparison between atomic type and mutexes



	1	2	4	8	16	32	64	128
lock	51666	18569	7354	5890	6156	5986	5819	5752
lock_guard	43332	15351	7644	5934	6140	5903	5773	5728
atomic	125000	68253	65118	59179	59174	58737	58479	58394

SYNCHRONIZATION BETWEEN THREADS

- Except for protecting shared data, we also need to synchronization action on separate threads.
- In C++ Standard Library, **conditional variables** and **futures** are provided to handle synchronization problems.
  - The condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until:
    - a notification is received from another thread
    - a timeout expires
  - Any thread that intends to wait on std::condition_variable has to acquire a **std::unique_lock** first. The wait operations atomically release the mutex and suspend the execution of the thread. When the condition variable is notified, the thread is awakened, and the mutex is reacquired.

EXAMPLE

```
std::mutex mut;
std::queue<data_chunk> data_queue;                      #1
std::condition_variable data_cond;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);                          #2
        data_cond.notify_one();                         #3
    }
}

void data_processing_thread()
{
    while(true)
```

- queue is used to pass data between two threads
- When data is ready, the thread locks the mutex and push the data into the queue(#2) and then call notify_one() member function in std::condition_variable instance to notify the waiting thread(#3)

EXAMPLE

- On the other hand, the processing thread first lock the mutex with std::unique_lock. The thread calls wait() in the condition varaible and checking the condition in the lambda function.

```
        data_cond.notify_one();                              #3
    }
}

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);                #4
        data_cond.wait(lk,[]{return !data_queue.empty();});  #5
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();                                         #6
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

- When the condition variable is notified by a call to notify_one() from the data preparation thread, the thread wakes and check the condition and lock the mutex if the condition is true and then process the next command.

MORE ABOUT UNIQUE_LOCK

- The condition variables require std::unique_lock rather than the std::lock_quard — the waiting thread must unlock the mutex while it is waiting , the lock it again afterwards and the std::lock_guard does not provide such flexibility.
- The flexibility to unlock a std::unique_lock is not just used for the call to wait() , it is also used once we've got the data to process, but before processing it (#6): processing data can potentially be a time-consuming operation, and as we saw in chapter 3, it is a bad idea to hold a lock on a mutex for longer than necessary.

ONE-OFF EVENT WITH FUTURES

- If a thread needs to wait for a specific one-off event, then it obtains a **future** representing this event. The thread can poll the future to see if the event has occurred while performing some other task.
- Two sorts of futures templates in C++ Standard Library.
  - std::unique_furture<> — the instance is the only one that refers to its associated event.
  - std::shared_future<> — multiple instances of it may refer to the same event. All the instance become ready at the same time, and they may all access any data associated with the event.

ONE-OFF EVENT WITH FUTURES

```
void wait_for_flight1(flight_number flight)
{
    std::shared_future<boarding_information>   #1
        boarding_info=get_boarding_info(flight);
    board_flight(boarding_info.get());
}

void wait_for_flight2(flight_number flight)
{
    std::shared_future<boarding_information>   #2
        boarding_info=get_boarding_info(flight);
    while(!boarding_info.is_ready())            #4
    {
        eat_in_cafe();                          #3
        buy_duty_free_goods();
    }
    board_flight(boarding_info.get());
}
```

- The first thread, running wait_for_flight1() obtains a std::shared_future<bording_information> with the bording information(#1), and call get(), which waits for the future to become ready.
- The second thread, running wait_for_flight2() , after obtaining the future(#2), does something else while periodically checking to see if the flight is ready to board by calling is_ready() on the future(#4).

A GLANCE OF MEMORY MODEL

- Why a C++ Memory Model
  - Problem: Hard for programmers to reason about correctness
  - Without precise semantics, hard to reason if compiler will violate semantics
  - Compiler transformations could introduce data races without violating language specification.

    ——resulting execution could yield unexpected behaviors.

MEMORY MODEL

- Two aspects to the memory model:
  I. the basic structural aspects — how things are laid out in memory
     1. Every variable is an object, including those that are members of other objects.
     2. Every object occupies at least one memory location.
     3. Variables of fundamental type such as int or char are exactly one memory location, whatever their size, even if they're adjacent or part of an array.
     4. Adjacent bit fields are part of the same memory location.

  II. The concurrency aspects
     1. If there is no enforced ordering between two accesses to a single memory location from separate threads, these accesses is not atomic,
     2. if one or both accesses is a write, this is a data race, and causes undefined behaviour.

MEMORY MODEL

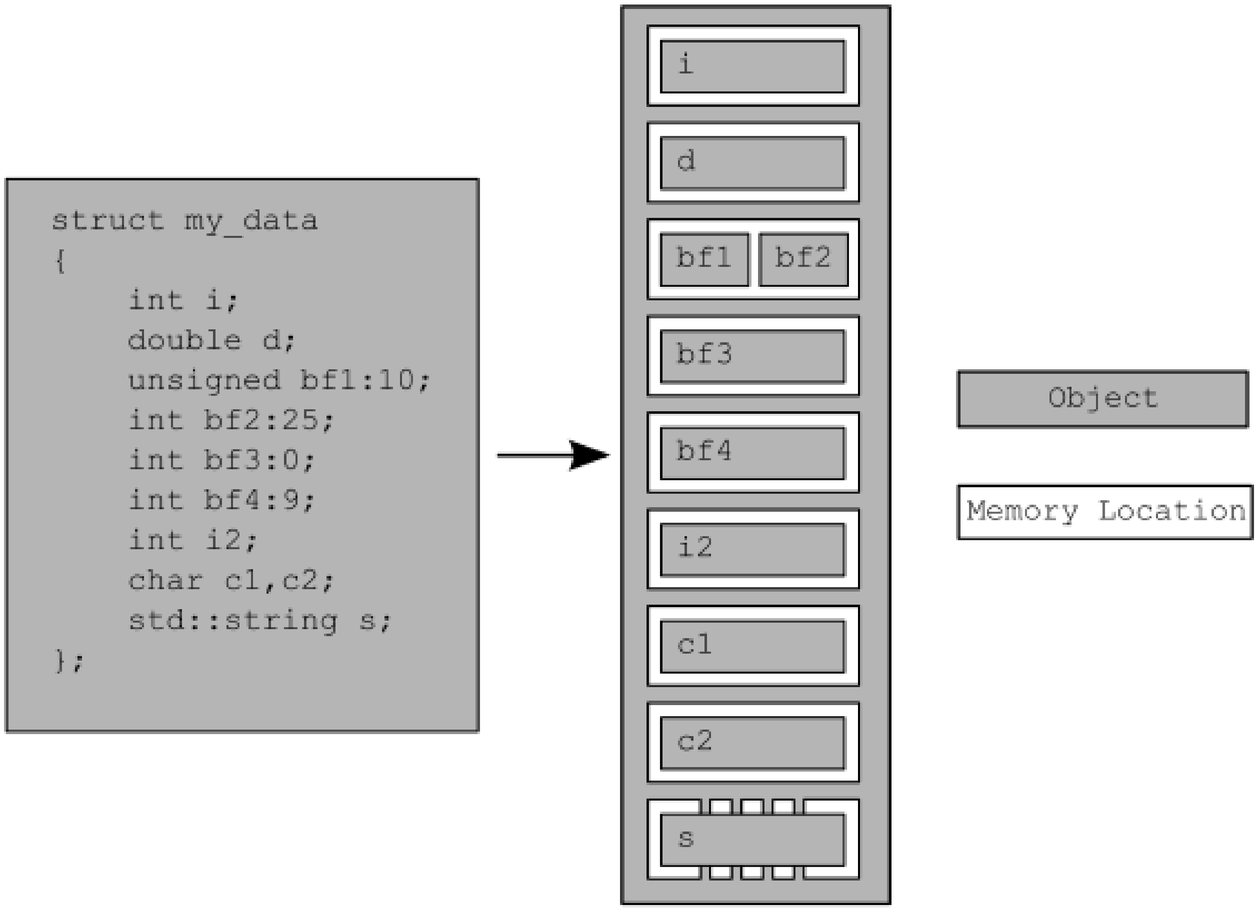


Figure 5.1 The division of a `struct` into objects and memory locations

OPERATIONS ORDERS

- Each of the operations on atomic types has an optional memory-ordering argument that be used to specify the required memory-ordering semantics. These operations can be divided into three categories:

1. Store operation, which can have memory_order_relaxed, memorey_order_release or memory_order_seq_cst ordering
2. Load operations, which can have memory_order_relaxed, memory_order_consume, memory_order_acquire, or memory_order_seq_cst ordering
3. Read-modify-write operations, which can have memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel, or memory_order_seq_cst ordering

EXAMPLE

Let see how the threads ordering will affect the result.

```
        Global
        int x, y;

Thread 1                Thread 2
x = 17;                 cout << y << " ";
y = 37;                 cout << x << endl;
```

- The behavior here is undefined.

```
        Global
        atomic<int> x, y;

Thread 1                Thread 2
x.store(17);            cout << y.load() << " ";
y.store(37);            cout << x.load() << endl;
```

- The default mode for atomic loads/stores in c++11 is to enforce sequential consistency which means all loads and stores must be "as if" they happened in the order you wrote them within each thread. The possible output are:
  0 0 (thread2 runs before thread 1)
  37 17 (thread 2 runs after thread 1)
  0 17 (threads 2 runs after thread 1 assigns to x but before it assigns to y)

EXAMPLE

```
              Global
              atomic<int> x, y;

Thread 1                              Thread 2
x.store(17,memory_order_relaxed);    cout << y.load(memory_order_relaxed) << " ";
y.store(37,memory_order_relaxed);    cout << x.load(memory_order_relaxed) << endl;
```

Relaxed ordering: there are no constraints on reordering of memory accesses around the atomic variable. So you might get the output

37 0

```
              Global
              atomic<int> x, y;

Thread 1                              Thread 2
x.store(17,memory_order_release);    cout << y.load(memory_order_acquire) << " ";
y.store(37,memory_order_release);    cout << x.load(memory_order_acquire) << endl;
```

The result is the same as sequential consistency.

MEMORY MODEL

- Memory model provides low-level atomic operations
- Expert programmers can maximize performance
- Atomic variables can be explicitly parameterized with respect to memory ordering constrain allowing instruction to be reordered with other memory operations.
- For read-modify-write operations, programmer can specify whether the operations acts as an acquire, a release operation, neither(relaxed), or both.

CONCLUSION

- C++ committee introduced the concurrency into C++0X and C++11 make it support multi-threads which make C++ a adaptive to the current programming style.
- To make concurrency possible, language should support threads launch, threads synchronization and threads communication.
- The basic problem in concurrency is how to protected shared data and synchronize threads. Mutexes and atomic template are the solution to race conditions.
- Lower level control based on memory models allow us to clear the semantic meaning in concurrency operation

Questions?

Thanks you!