Module M59

Partha Pratim Das

Programming in Modern C++

Module M59: C++11 and beyond: Concurrency: Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced the notion of concurrent programming in C++11 using thread support
- Explored library support through std::thread and std::bind
- Exposed to the bugs in thread programming - race condition and data race
- Discussed examples of thread programs with bugs and their solution

Module Objectives

- To understand synchronization issues in multi-thread programming in C++
- To study various synchronization mechanisms through example
- To self-study the details of synchronization mechanisms:
  - *Mutex*
  - *Lock*
  - *Atomics*
  - *Condition Variable*
  - *Future and Promises*
  - *Async*

Module Outline

Sources:

- C++11 - the new ISO C++ standard: Threads, Stroustrup, 2016
- C++11 Standard Library Extensions — Concurrency: Threads, isocpp
- std::thread, cppreference
- Concurrency memory model, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- A tutorial on modern multithreading and concurrency in C++, 2020
- **C++11 Multi-threading Tutorials**: Parts 1-8, thisPointer
    - C++11 Multithreading – Part 1 : Three Different ways to Create Threads
- **C++20 Concurrency**: Parts 1-3, Gajendra Gulgulia, 2021
    - C++20 Concurrency: Part 1: synchronized output stream
    - C++20 Concurrency: Part 2: jthreads
    - C++20 Concurrency: Part 3: request_stop and stop_token for std::jthread

Threads

- A **thread** is a *light-weight process*
- We have learnt the basic thread operations using `std::thread`:
  - Create a thread
  - Pass parameter/s to a thread - directly or by `std::bind`
  - Return result/s from a thread - directly or by `std::bind`
  - Join threads
- We have also observed race condition and data race in simple multi-threaded program
- To alleviate such bugs we need to understand the synchronization of threads and various mechanisms for it

Race Condition and Data Race

- We often talk about bugs in multi-threading:
  - *Race Condition*
  - *Data Race*
- Are they same?
  - No, they are not
  - They are not a subset of one another
  - They are also neither the necessary, nor the sufficient condition for one another
- *Race Condition*: A race condition is a semantic error
  - A race condition is a situation, in which the result of an operation depends on the interleaving of certain individual operations
  - Many race conditions can be caused by data races, but this is not necessary
- *Data Race*: A data race occurs when 2 instructions from different threads access the same memory location without synchronization
  - A data race is a situation, in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable.
  - The discovery of data race can be automated
- We take examples to illustrate both

Example 1 (Module 58)

- Let us write a simple program to compute:

$$\sum_{i=1}^{20} i^2 = \frac{20 \times (20+1) \times (2 \times 20 + 1)}{6} = 2870$$

- Assuming that `x*x` is a heavy computation (fake it!) we developed a simple multi-threaded program for the above:
  - Spawn 20 threads
  - Each thread computes `square` for a distinct value
  - The accumulated result is available after the threads join
- We added random delay and repeated run support to setup scenarios for race conditions to be observed. We observed that bugs exist
- We have also discussed two fixes – by `mutex` and by `atomic` which we will recap here
- We also discuss other solutions by `lock`, `future`, and `async`

Example 1: Random Delay + Repeat (Module 58)

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int accum = 0; // init accumulator
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}
int main() { int trial_count = 0; // counting trials before failure
    do { ++trial_count; // increment trial counter
    if (0 == trial_count % 100) // message after every 100 trials - that the process is alive
        cout << "trials = " << trial_count << endl;
        accum = 0; // reset to start a trial
        vector<thread> ths; // vector of threads
        for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
        for (auto& th : ths) { th.join(); } // join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
    cout << "trials = " << trial_count << " accum = " << accum << endl; // print if there is bad result
}
```

Race Condition and Data Race: Example 1: Solution by Mutex

Example 1: Solution by Mutex (Module 58)

- A mutex (mutual exlusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time. Keeping the main function the same:

```
int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    accum_mutex.lock();   // gets the lock on accum_mutex
    accum += temp;
    accum_mutex.unlock(); // release the lock on accum_mutex
}
```

- We try running the program repeatedly again and the problem should now be fixed
- The first thread that calls `lock()` gets the `lock`
- During this time, all other threads that call `lock()`, will wait at that line for the mutex to be unlocked. Creates a *Critical Section*
- It is important to introduce the variable `temp`, since we want the `x * x` calculations to be outside the lock-unlock block, otherwise we would be hogging the lock while we are running our heavy calculations

Example 1: Solution by Mutex (Module 58)

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <mutex>   // mutex
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // init accumulator
mutex accum_mutex; // mutex variable
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum_mutex.lock();   // gets the lock on accum_mutex
    accum += p;           // accumulate product
    accum_mutex.unlock(); // release the lock on accum_mutex
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

Example 1: Solution by Lock

Race Condition and Data Race: Example 1: Solution by Lock

Example 1: Solution by Lock

- A lock is an object that can hold a reference to a mutex and may `unlock()` the mutex during the lock's destruction (such as when leaving block scope)

```
int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    std::unique_lock<std::mutex> // acquires and owns the lock on accum_mutex
        lck(accum_mutex);
    //accum_mutex.lock();    // gets the lock on accum_mutex
    accum += temp;
    //accum_mutex.unlock(); // release the lock on accum_mutex
} // release the lock and ownership on accum_mutex
```

- Use of `lock` makes the coding and understanding simpler than using bare `mutex`
- `std::unique_lock` has the similar resource ownership advantages as of `std::unique_ptr`
- Particularly useful when we have multiple resources to mutually exclusively manage

Example 1: Solution by Lock

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <mutex>   // mutex, unique_lock
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // init accumulator
mutex accum_mutex; // mutex variable
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    std::unique_lock<std::mutex> lck(accum_mutex); // acquires and owns the lock on accum_mutex
    accum += p;           // accumulate product
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

Race Condition and Data Race: Example 1: Solution by Atomic

- With Mutex / Lock the problem gets fixed. The program does not produce a wrong result even after 6000+ trials
- Interestingly, **C++11** offers even nicer abstractions to solve this problem. For instance, the `atomic` container:

```
#include <atomic>

atomic<int> accum(0); // makes accum and initializes to 0

void square(int x) {
    accum += x * x;
}
```

- We do not need to introduce temp here, since `x * x` will be evaluated before handed off to `accum`, so it will be outside the atomic event
- However, we will continue to show the solution using the temporary

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <atomic>  // atomic
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
atomic<int> accum(0); // makes accum and initializes to 0
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum += p;            // accumulate product
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

- Works fine. Does not produce a wrong result even after 5000+ trials

Example 1: Solution by Future

Race Condition and Data Race: Example 1: Solution by Future

Example 1: Solution by Future

- Future, represented by `std::future`, is a way to access the results of asynchronous operations
- Imagine if our main `thread A` wants to open a new `thread B` to perform some of our expected tasks and return me a result. At this time, `thread A` may be busy with other things and have no time to take into account the results of `thread B`. So we naturally hope to get the result of `thread B` at a certain time
- Before the introduction of `std::future` in **C++11**, the usual practice used to be:
    - Create a `thread A`
    - start `task B` in `thread A`
    - send an `event` when it is ready, and
    - save the result in a *global variable*
    - The main function thread A is doing other things. When the result is needed, a thread is called to `wait` for the function to get the result of the execution
- The `std::future` provided by **C++11** simplifies this process and can be used to get the results of asynchronous tasks. Naturally, we can easily imagine it as a simple means of thread synchronization, namely the *barrier*
- We engage Example 1 to illustrate the way Future works

Example 1: Solution by Future

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <future>  // future
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // define accumulator
void square(future<int>& fut) { // called in different threads - one each for 1 .. 20
    int x = fut.get(); // get parameter from future
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}
int main() {
    vector<promise<int>> vp; /*promises*/ vector<future<int>> vf; /*futures*/ vector<thread> vt;
    for (int i = 0; i < 20; i++) {
        vp.push_back(promise<int>()); // vector of promise objects
        vf.push_back(vp[i].get_future()); // vector of engaged future objects from promise objects
        vt.push_back(thread(&square, ref(vf[i]))); // vector of threads - pass future object
        vp[i].set_value(i+1); // fulfil promise with needed value
    }
    for (auto& t : vt) { t.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

Example 1: Solution by Async

Race Condition and Data Race: Example 1: Solution by Async

Example 1: Solution by Async

- An even higher level of abstraction that avoids the use of promise and future directly, talking in terms of *tasks* is `async` given in `std::future`. Consider the following example:

```
#include <iostream>
#include <future> // future
using namespace std;

int square(int x) { return x * x; }
int main() {
    auto a = async(&square, 10); // returns a future<int>
    int v = a.get();             // waits to fulfil the promise

    cout << "The thread returned " << v << endl;
}
```

- The `async` construct uses an object pair called a `promise` and a `future`
- The former has made a promise to eventually provide a value
- The `future` is linked to the `promise` and can at any time try to retrieve the value by `get()`
- If the `promise` has not been fulfilled yet, it will simply wait until the value is ready
- The `async` hides most of this for us, except that it returns in this case a `future<int>` object

Example 1: Solution by Async

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <future>  // future
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    return p;
}
int main() {
    int accum = 0; // define accumulator
    vector<future<int>> fts; // vector of future objects
    for (int i = 1; i <= 20; i++) { fts.push_back(async(&square, i)); } // 20 future objects
    for (auto& ft : fts) { accum += ft.get(); } // wait to get value from future and accumulate
    cout << " accum = " << accum << endl; // print final value
}
```

- Works fine. Does not produce a wrong result even after 30000+ trials

Synchronization

Synchronization Errors: Symptoms and Causes

• **Conflicting access to shared memory**	• one thread begins an operation on shared memory, is suspended, and leaves that memory region incompletely transformed • a second thread is activated and accesses the shared memory in the corrupted state, causing errors in its operation and potentially errors in the operation of the suspended thread when it resumes
• **Race Conditions**	• correct operation depends on the order of completion of two or more independent activities • the order of completion is not deterministic
• **Deadlock**	• two or more tasks each own resources needed by the other preventing either one from running so neither ever completes and never releases its resource
• **Starvation**	• a high priority thread dominates CPU resources, preventing lower priority threads from running often enough or at all
• **Priority inversion**	• a low priority task holds a resource needed by a higher priority task, blocking it from running

Synchronization

- A program may need multiple threads to share some data
- If access is not controlled to be sequential, then shared data may become corrupted
  - One thread accesses the data, begins to modify the data, and then is put to sleep because its time slice has expired. The problem arises when the data is in an incomplete state of modification.
  - Another thread awakes and accesses the data, that is only partially modified. The result is very likely to be corrupt data.
- The process of making access serial is called serialization or synchronization
- Synchronization may be achieved in various ways including:
  - *Mutex* (self-study)
  - *Lock* (self-study)
  - *Atomics* (self-study)
  - *Condition Variable* (self-study)
  - *Future and Promises* (self-study)
  - *Async* (self-study)

Synchronization: thread_local

Sources:

- Thread-local storage, isocpp.org
- Storage class specifiers, cppreference
- Thread-Local Data, modernescpp, 2016
- What does the thread_local mean in C++11?, stackoverflow

thread_local

- `thread_local` is a storage class specifier. Thread local data will be created for each thread as needed
- `thread_local` data exclusively belongs to the thread and behaves like `static` data
- Created at its first use and *lifetime* bound to the *lifetime of the thread* (*lifetime* in **Module 13**, **23**, & **35**)

Thread Local

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
thread_local int i = 0; //

void f(int newval) { i = newval; }
void g() { cout << i; }
void threadfunc(int id) {
    f(id); ++i; /*          */  g();
}
int main() { i = 9;
    vector<thread> th;
    for(int i = 1; i < 4; ++i)
        th.push_back(thread(threadfunc, i));
    for(auto& t: th) t.join();
    cout << i << endl;
}
// 2349, 3249, 4239, 4329, 2439 or 3429
```

Global

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
// int i = 0;

void f(int newval) { i = newval; }
void g() { cout << i; }
void threadfunc(int id) {
    f(id); ++i; /*          */  g();
}
int main() { i = 9;
    vector<thread> th;
    for(int i = 1; i < 4; ++i)
        th.push_back(thread(threadfunc, i));
    for(auto& t: th) t.join();
    cout << i << endl;
}
```

Self-Study

Synchronization: `mutex`

Sources:

- Mutual exclusion, isocpp.org
- std::mutex, cplusplus
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

mutex

- We have used mutex from <mutex> in the solution for Example 1
- A mutex is a primitive object used for controlling access in a multi-threaded system

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// manipulate shared data
sh+=1;
m.unlock();
```

- **Only one thread at a time can be in the region of code between the lock() and the unlock() (critical region)**
- If a second thread tries m.lock() while a first thread is executing in that region, that second thread is blocked until the first executes the m.unlock()
- There may give rise to serious problems like:
  - What if a thread "forgets" to unlock()?
  - What if a thread tries to lock() the same mutex twice?
  - What if a thread waits a very long time before doing an unlock()?
  - What if a thread needs to lock() two mutexes to do its job?
  - What if ...?

mutex

- In addition to `lock()`, a mutex has a `try_lock()` operation which can be used to try to get into the critical region without the risk of getting blocked:

```
std::mutex m;
int sh; // shared data
// ...
if (m.try_lock()) { // either get the lock or fail and continue - no blocking
    // manipulate shared data
    sh+=1;
    m.unlock();
}
else { /* maybe do something else */ }
```

- Use a `recursive_mutex` to acquire it more than once by a thread in a recursive or co-recursive function
- We can also set a duration (relative time) to try for a lock:

```
m.try_lock_for(std::chrono::seconds(10)) // get it in the next 10 seconds or fail
```

- Or we may want to wait until a fixed point in time, a `time_point`:

```
m.try_lock_until(midnight) // wait till midnight or fail
```

- A `recursive_timed_mutex` is a `recursive_mutex` that can be timed

Synchronization: `lock`

Sources:

- Locks, isocpp.org
- std::lock, cplusplus
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

lock

- We have used lock from `<mutex>` in the solution for Example 1
- A lock is an object that can hold a reference to a mutex and may `unlock()` the mutex during the lock's destruction (such as when leaving block scope)
- A thread may use a lock to aid in managing mutex ownership in an exception safe manner
- That is, a lock implements *Resource Acquisition Is Initialization (RAII)* for mutual exclusion (Recall RAII in smart pointers). For example:

```
std::mutex m;
int sh; // shared data
// ...
void f() {
    // ...
    std::unique_lock<std::mutex> lck(m); // m.lock()
    // manipulate shared data:
    // lock will be released even if this code throws an exception
    sh+=1;
} // m.unlock()
```

- A lock can be *moved* (the purpose of a lock is to represent local ownership of a non-local resource), but not *copied* (which copy would own the resource/mutex?)

lock

- This straightforward picture of a lock is clouded by `unique_lock` having facilities to do just about everything a mutex can, but *safer* and *simpler*
- For example, we can use a `unique_lock` to do `try_lock`:

```
std::mutex m;
int sh; // shared data
// ...
void f() {
    // ...
    std::unique_lock<std::mutex> lck(m, std::defer_lock); // make a lock, but do not
    // ...                                                 // acquire the mutex
    if (lck.try_lock()) {
        // manipulate shared data:
        sh+=1;
    }
    else { /* maybe do something else */ }
}
```

- Similarly, `unique_lock` supports `try_lock_for()` and `try_lock_until()`
- What you get from using a lock rather than the mutex directly is exception handling and protection against forgetting to `unlock()`

Synchronization: `lock`: Deadlock

lock: Deadlock

- What if we need two resources represented by two mutexes? The naive way is to acquire the mutexes in order:

```
std::mutex m1;
std::mutex m2;
int sh1;    // shared data
int sh2
// ...
void f() {
    // ...
    std::unique_lock<std::mutex> lck1(m1);
    std::unique_lock<std::mutex> lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```

- This has the potentially deadly flaw that some other thread could try to acquire `m1` and `m2` in the opposite order so that each had one of the locks needed to proceed and would wait forever for the second (*deadlock*)
- With many locks in a system, that is a real danger

lock: Deadlock

- The standard locks provide two functions for (safely) trying to acquire two or more locks:

```
void f() { // ...
    std::unique_lock<std::mutex> lck1(m1, std::defer_lock); // make locks but
    std::unique_lock<std::mutex> lck2(m2, std::defer_lock); // do not yet try to
    std::unique_lock<std::mutex> lck3(m3, std::defer_lock); // acquire the mutexes
    std::lock(lck1, lck2, lck3);
    // manipulate shared data:
}
```

- The implementation of `lock()` is carefully crafted to avoid *deadlock*
- In essence, it will do the equivalent to careful use of `try_lock()`s
- If `lock()` fails to acquire all locks it will throw an exception
- If you prefer to use `try_lock()`s yourself, there is an equivalent to `lock()` to help:

```
void f() { int x; // ...
    std::unique_lock<std::mutex> lck1(m1, std::defer_lock); // make locks but
    std::unique_lock<std::mutex> lck2(m2, std::defer_lock); // do not yet try to
    std::unique_lock<std::mutex> lck3(m3, std::defer_lock); // acquire the mutexes
    if ((x = try_lock(lck1, lck2, lck3))==-1) { // manipulate shared data:
    } else { // x holds the index of a mutex we could not acquire
             // for example, if lck2.try_lock() failed x==1
    }
}
```

Synchronization: atomic

Sources:

- Atomics, isocpp.org
- std::atomic, cplusplus
- atomic Weapons: The C++ Memory Model and Modern Hardware, herbsutter.com, 2013
- C++ and Beyond, Scott Meyers, Herb Sutter, and Andrei Alexandrescu, 2010-14
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

atomic

- We have used atomic from <atomic> in the solution for Example 1
- Each instantiation of the std::atomic template defines an atomic type. **If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined**
- std::atomic is neither copyable nor movable
- Type aliases are provided for bool (std::atomic_bool) and integral types like int, short, etc.

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::atomic_flag, ATOMIC_FLAG_INIT
#include <thread>         // std::thread, std::this_thread::yield
#include <vector>         // std::vector
std::atomic<bool> ready (false);
std::atomic_flag winner = ATOMIC_FLAG_INIT; // set false. atomic_flag has no load / store
void count1m (int id) {
    while (!ready) { std::this_thread::yield(); } // all threads wait for the ready signal to start
    for (volatile int i=0; i<1000000; ++i)        // go!, count to 1 million
    if (!winner.test_and_set()) // atomically sets the flag to true and obtains its previous value
        { std::cout << "thread #" << id << " won!\n"; }
};
int main () { std::vector<std::thread> threads;
    std::cout << "spawning 10 threads that count to 1 million...\n";
    for (int i=1; i<=10; ++i) threads.push_back(std::thread(count1m,i));
    ready = true; // signal ready to start
    for (auto& th : threads) th.join();
} // thread #8 won! // thread #4 won! // thread #1 won! // thread #9 won! ...
```

Synchronization: condition_variable

Sources:

- Condition variables, isocpp.org
- std::condition_variable, cplusplus
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

condition_variable

- A condition variable is an object able to block the calling thread until notified to resume
- It uses a `unique_lock` (over a `mutex`) to lock the thread when one of its *wait functions* is called
- The thread remains blocked until woken up by another thread that calls a *notification function* on the same `condition_variable` object
- Objects of type `condition_variable` always use `unique_lock<mutex>` to wait: for an alternative that works with any kind of lockable type, use `condition_variable_any`
- Condition variables provide good solutions to deadlock problem too

condition_variable


```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable
std::mutex mtx;
std::condition_variable cv;
bool ready = false;
void print_id (int id) { std::unique_lock<std::mutex> lck(mtx);
    while (!ready) cv.wait(lck);
    // ...
    std::cout << "thr. " << id << ' ';
}
void go() { std::unique_lock<std::mutex> lck(mtx);
    ready = true;
    cv.notify_all();
}
int main () {
    std::thread threads[10];
    // spawn 10 threads:
    for (int i=0; i<10; ++i) threads[i] = std::thread(print_id,i);
    std::cout << "10 threads ready to race...\n";
    go();                          // go!
    for (auto& th : threads) th.join();
}
```

Synchronization: `future` and `promise`

Sources:

- Futures and promises, isocpp.org
- std::future, cplusplus
- std::promise, cplusplus
- Copying and rethrowing exceptions, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

future and promise

- C++11 offers future and promise for returning a value from a task spawned on a separate thread, and packaged_task to help launch tasks
- The important point about future and promise is that they enable a transfer of a value between two tasks without explicit use of a lock; *the system* implements the transfer efficiently
- The basic idea is simple: When a task wants to return a value to the thread that launched it, it puts the value into a promise. Somehow, the implementation makes that value appear in the future attached to the promise
- The caller (typically the launcher of the task) can then read the value
- The standard provides three kinds of futures, future for most simple uses, and shared_future and atomic_future for some trickier cases
- Here, we will just present future because it is the simplest and does all we need. If we have a future<X> called f, we can get() a value of type X from it:

```
X v = f.get();  // if necessary wait for the value to get computed
```

- If the value is not there yet, our thread is blocked until it arrives
- If the value could not be computed and the task will throw an exception, calling get() will rethrow that exception to the code calling get()

future and promise

- We might not want to wait for a result, so we can ask the future if a result has arrived:

```
if (f.wait_for(0)) { // there is a value to get()
    // do something
}
else { /* do something else */ }
```

- However, the main purpose of future is to provide that simple get()
- The main purpose of promise is to provide a simple set() to match future's get()
- If you have a promise and need to send a result of type X (back) to a future, there are basically two things you can do:
  - pass a value
  - pass an exception

```
try { X res;
    // compute a value for res
    p.set_value(res);
}
catch (...) { /* could not compute res */ p.set_exception(std::current_exception()); }
if (f.wait_for(0)) { // there is a value to get()
    // do something
}
else { /* do something else */ }
```

Synchronization: async

Sources:

- async, isocpp.org
- std::async, cplusplus
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

async

- We have used async from <future> in the solution for Example 1
- async is a way for the programmer to rise above the messy threads-plus-lock level of programming:

```
#include <iostream>  // cout
#include <future>    // async
#include <vector>    // vector
#include <numeric>   // accumulate
using namespace std;
template<class T, class V> struct Accum { // simple accumulator function object
    T* b; T* e; V val; Accum(T* bb, T* ee, const V& v): b{bb}, e{ee}, val{v} { }
    V operator()() { return std::accumulate(b, e, val); }
};
double comp(vector<double>& v) { // spawn many tasks if v is large enough
    if (v.size() < 100) return std::accumulate(v.begin(), v.end(), 0.0); // to short to multi-thread
    auto f0{async(Accum<double, double>{&v[0], &v[v.size()/4], 0.0})};            // 0 125
    auto f1{async(Accum<double, double>{&v[v.size()/4], &v[v.size()/2], 0.0})};   // 125 250
    auto f2{async(Accum<double, double>{&v[v.size()/2], &v[v.size()*3/4], 0.0})}; // 250 375
    auto f3{async(Accum<double, double>{&v[v.size()*3/4], &v[v.size()], 0.0})};   // 375 500
    return f0.get()+f1.get()+f2.get()+f3.get(); // wait for the values as promised
}
int main () { vector<double> v;
    for (int i = 1; i <= 500; ++i) v.push_back(i); // fill the vector
    cout << "Sum = " << accumulate(v.begin(), v.end(), 0.0) << endl; // sequential 125250
    cout << "Sum = " << comp(v) << endl;                             // multi-thread 125250
}
```

Race Condition and Data Race: Example 2

Race Condition and Data Race: Practice Examples

Example 2


```
#include <functional>
#include <iostream>
#include <thread>
#include <vector>

struct Account { int balance{100}; }; // initially each account is loaded with Rs. 100
void addMoney(Account& to, int amount)
    { to.balance += amount; } // add amount to account with synchronization
int main() { Account account;

    std::vector<std::thread> vecThreads(100);
    for (auto& thr: vecThreads)
        thr = std::thread(addMoney, std::ref(account), 50); // add Rs. 50 to the account

    for (auto& thr: vecThreads) thr.join();
    std::cout << "account.balance: " << account.balance << std::endl; // final balance
}
```

- 100 threads are adding Rs. 50 to the same account using function addMoney but without synchronisation
- Final balance differs between Rs. 5000 and Rs. 5100 and we have a data race

Example 3

- Let us consider a function that transfers money from one account to another.
- In the single-threaded case, all is fine:

```
#include <iostream>

struct Account { int balance{100}; };  // initially each account is loaded with Rs. 100
void transferMoney(int amount, Account& from, Account& to) {
    if (from.balance >= amount) { // transfer is allowed only it there is enough fund
        from.balance -= amount;
        to.balance += amount;
    }
}
int main() { Account account1, account2;  // two accounts for mutual transfers
    transferMoney(50, account1, account2);  // two transfers between accounts
    transferMoney(130, account2, account1); // in sequential, total order

    std::cout << "account1.balance: " << account1.balance << std::endl;
    std::cout << "account2.balance: " << account2.balance << std::endl;
}
// account1.balance: 180 // 100 - 50 + 130
// account2.balance: 20  // 100 + 50 - 130
```

Example 3

Module M59

Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async
Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomics
Condition Variables
Futures and Promises
Async
Practice Examples
Module Summary

- Let us multi-thread the program

```
#include <iostream>
#include <functional> // ref
#include <thread>     // thread, this_thread::sleep_for
#include <chrono>     // chrono::nanoseconds
struct Account { int balance{100}; };  // initially each account is loaded with Rs. 100
void transferMoney(int amount, Account& from, Account& to) {
    if (from.balance >= amount) { // transfer is allowed only it there is enough fund
        from.balance -= amount;
        std::this_thread::sleep_for(std::chrono::nanoseconds(1)); // delay
        to.balance += amount;
    }
}
int main() { Account account1, account2;  // two accounts for mutual transfers
    // concurrent transfers between accounts
    std::thread thr1(transferMoney, 50, std::ref(account1), std::ref(account2));
    std::thread thr2(transferMoney, 130, std::ref(account2), std::ref(account1));
    thr1.join(); thr2.join();
    std::cout << "account1.balance: " << account1.balance << std::endl;
    std::cout << "account2.balance: " << account2.balance << std::endl;
}
// account1.balance: 50  // 100 - 50. thr2 fails to execute for low funds
// account2.balance: 150 // 100 + 50. thr2 fails to execute for low funds
```

- Correct result may be produced on occasions

Module Summary

- Understood synchronization issues in multi-thread programming in C++
- Studied various synchronization mechanisms through example
- Provided detail for self-study of synchronization mechanisms:
  - *Mutex*
  - *Lock*
  - *Atomics*
  - *Condition Variable*
  - *Future and Promises*
  - *Async*
- Explored use of the synchronization mechanisms to alleviate race condition and data race and left practice examples