# Introduction to Multi-Thread Programming (Part 2)

Yifan ZHU <i@zhuyi.fan>

October 8, 2021

1. High-Level C++ API
   - Introduction
   - Thread Management
   - OOP Style Locking
   - Atomic Variables
   - Thread Local Storage

# Introduction

With C++ 17, it is time to forget everything I mentioned in Part 1!

# Headers

- `#include <thread>`: create threads, control, statistics
- `#include <mutex>`: mutexes
- `#include <shared_mutex>`: rwlocks
- `#include <atomic>`: atomic variables

# Thread Classes

```cpp
#include <iostream>
#include <thread>
#include <chrono>

void foo()
{
    // simulate expensive operation
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

void bar()
{
    // simulate expensive operation
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::cout << "starting first helper...\n";
    std::thread helper1(foo);

    std::cout << "starting second helper...\n";
    std::thread helper2(bar);

    std::cout << "waiting for helpers to finish..." << std::endl;
    helper1.join();
    helper2.join();

    std::cout << "done!\n";
}
```

Figure: Thread Example

# Joinable?

```
std::thread t;
std::cout << "before starting, joinable: " <<
 ↪  std::boolalpha << t.joinable()
<< '\n';

t = std::thread(foo);
std::cout << "after starting, joinable: " <<
 ↪  t.joinable()
<< '\n';

t.join();
std::cout << "after joining, joinable: " <<
 ↪  t.joinable()
<< '\n';
```

# Raw Mutex

```
std::mutex mutex;
mutex.lock();
//....
mutex.unlock();
```

Good Enough. But what is the problem?

# RAII Lock Guard

```cpp
#include <thread>
#include <mutex>
#include <iostream>

int g_i = 0;
std::mutex g_i_mutex;  // protects g_i

void safe_increment()
{
    const std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;

    std::cout << "g_i: " << g_i << "; in thread #"
              << std::this_thread::get_id() << '\n';

    // g_i_mutex is automatically released when lock
    // goes out of scope
}

int main()
{
    std::cout << "g_i: " << g_i << "; in main()\n";

    std::thread t1(safe_increment);
    std::thread t2(safe_increment);

    t1.join();
    t2.join();

    std::cout << "g_i: " << g_i << "; in main()\n";
}
```

# RAII Lock Guard

- Automatically released when destructor is called.
- Exception same in scope.

# Lock Deference

```
void transfer(Box &from, Box &to, int num)
{
    // don't actually take the locks yet
    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // 'from.m' and 'to.m' mutexes unlocked in 'unique_lock' dtors
}
```

Figure: Unique Lock Example

Use `std::shared_lock` for shared mutexes.

# Deadlock Avoidance

```
    {
        std::lock(e1.m, e2.m);
        std::lock_guard<std::mutex> lk1(e1.m, std::adopt_lock);
        std::lock_guard<std::mutex> lk2(e2.m, std::adopt_lock);
// Equivalent code (if unique_locks are needed, e.g. for condition variables)
//        std::unique_lock<std::mutex> lk1(e1.m, std::defer_lock);
//        std::unique_lock<std::mutex> lk2(e2.m, std::defer_lock);
//        std::lock(lk1, lk2);
// Superior solution available in C++17
//        std::scoped_lock lk(e1.m, e2.m);
        {
            std::lock_guard<std::mutex> lk(io_mutex);
            std::cout << e1.id << " and " << e2.id << " got locks" << std::endl;
        }
        e1.lunch_partners.push_back(e2.id);
        e2.lunch_partners.push_back(e1.id);
    }
```

Figure: Deadlock Avoidance

# Tags

| Type | Effect(s) |
|---|---|
| `defer_lock_t` | do not acquire ownership of the mutex |
| `try_to_lock_t` | try to acquire ownership of the mutex without blocking |
| `adopt_lock_t` | assume the calling thread already has ownership of the mutex |

Figure: Tags

# Atomic Variables

## std::atomic

Defined in header <atomic>

```
template< class T >
struct atomic;                                   (1)  (since C++11)

template< class U >
struct atomic<U*>;                               (2)  (since C++11)
```

Defined in header <memory>

```
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)
```

Defined in header <stdatomic.h>

```
#define _Atomic(T) /* see below */                (5)  (since C++23)
```

Figure: Atomic Classes

# Atomic Variables

**Member functions**

| | |
|---|---|
| (constructor) | constructs an atomic object<br>(public member function) |
| **operator=** | stores a value into an atomic object<br>(public member function) |
| **is_lock_free** | checks if the atomic object is lock-free<br>(public member function) |
| **store** | atomically replaces the value of the atomic object with a non-atomic argument<br>(public member function) |
| **load** | atomically obtains the value of the atomic object<br>(public member function) |
| **operator T** | loads a value from an atomic object<br>(public member function) |
| **exchange** | atomically replaces the value of the atomic object and obtains the value held previously<br>(public member function) |
| **compare_exchange_weak**<br>**compare_exchange_strong** | atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not<br>(public member function) |
| **wait** (C++20) | blocks the thread until notified and the atomic value changes<br>(public member function) |
| **notify_one** (C++20) | notifies at least one thread waiting on the atomic object<br>(public member function) |
| **notify_all** (C++20) | notifies all threads blocked waiting on the atomic object<br>(public member function) |

Figure: Member Functions

# Atomic Variables

**Specialized member functions**

| | |
|---|---|
| `fetch_add` | atomically adds the argument to the value stored in the atomic object and obtains the value held previously (public member function) |
| `fetch_sub` | atomically subtracts the argument from the value stored in the atomic object and obtains the value held previously (public member function) |
| `fetch_and` | atomically performs bitwise AND between the argument and the value of the atomic object and obtains the value held previously (public member function) |
| `fetch_or` | atomically performs bitwise OR between the argument and the value of the atomic object and obtains the value held previously (public member function) |
| `fetch_xor` | atomically performs bitwise XOR between the argument and the value of the atomic object and obtains the value held previously (public member function) |
| `operator++` `operator++(int)` `operator--` `operator--(int)` | increments or decrements the atomic value by one (public member function) |
| `operator+=` `operator-=` `operator&=` `operator\|=` `operator^=` | adds, subtracts, or performs bitwise AND, OR, XOR with the atomic value (public member function) |

Figure: Specialized Functions

# Thread Local Storage

```cpp
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

thread_local unsigned int rage = 1;
std::mutex cout_mutex;

void increase_rage(const std::string& thread_name)
{
    ++rage; // modifying outside a lock is okay; this is a thread-local variable
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << "Rage counter for " << thread_name << ": " << rage << '\n';
}

int main()
{
    std::thread a(increase_rage, "a"), b(increase_rage, "b");

    {
        std::lock_guard<std::mutex> lock(cout_mutex);
        std::cout << "Rage counter for main: " << rage << '\n';
    }

    a.join();
    b.join();
}
```

Possible output:

```
Rage counter for a: 2
Rage counter for main: 1
Rage counter for b: 2
```

Figure: TLS Example

# Register Hook

```
struct Callback {
        Callback() { std::cout << "thread created" <<
↪   std::endl; }
        ~Callback() { std::cout << "thread ended" <<
↪   std::endl; }
};
static thread_local Callback callback {};
```