Multi-Threading in C++
Multi-Threading in C++

In C++, it is allowed to run multiple threads simultaneously that use the same memory.

- Multiple threads may *read* from the same memory location
- All other accesses (i.e. read-write, write-read, write-write) are called *conflicts*
- Conflicting operations are only allowed when threads are *synchronized*
- This can be done with *mutexes* or *atomic operations*
- Unsynchronized accesses (also called *data races*), deadlocks, and other potential issues when using threads are undefined behavior!

All conflicting operations must be synchronized in some way!
Threads Library (1)

The header <thread> defines the class std::thread
- Using this class is the best way to use threads platform-independently
- May require additional compiler flags depending on the actual underlying implementation
- Use CMake to determine these flags in a platform-independent way
- For gcc and clang on Linux this will usually be -pthread

```cpp
cmake_minimum_required(VERSION 3.21)
project(sample)

find_package(Threads REQUIRED)
add_executable(sample main.cpp)
target_link_libraries(sample PUBLIC Threads::Threads)
```
The constructor of `std::thread` can be used to start a new thread
- Syntax: `thread(Function&& f, Args&&... args)`
- The function `f` will be invoked in a new thread with the arguments `args`
- The thread will terminate once `f` returns
- The default constructor can be used to create an empty thread object

The member function `join()` must be used to wait for a thread to finish
- `join()` must be called exactly once for each thread
- `join()` must be called *before* an `std::thread` object is destroyed
- When the destructor of an `std::thread` is called, the program is terminated if the associated thread was not joined
Example

```cpp
#include <thread>

void foo(int a, int b);

int main() {
    // Pass a function and args
    std::thread t1(foo, 1, 2);
    // Pass a lambda
    std::thread t2([]() {
        foo(3, 4);
    });

    foo(5, 6);

    t2.join();
    t1.join();
}
```
Example

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

void safe_print(std::string_view s);

int main() {
    {
        std::thread t1([]() { safe_print("Hi\n"); });
        t1.join();
    }
    // Everything is fine, we called t1.join()
    {
        std::thread t2([]() {});
    }
    // Program terminated because t2.join() was not called
}```
std::thread is movable but not copyable

- Moving transfers all resources associated with the running thread
- Only the moved-to thread can be joined
- The moved-from thread object is empty (not associated with any thread)

Example

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

void safe_print(std::string_view s);

int main() {
    std::thread t1([]() { safe_print("Hi\n"); });
    std::thread t2 = std::move(t1); // t1 is now empty
    t2.join(); // OK, thread originally started in t1 is joined
}
Threads Library (6)

std::thread can be used in standard library containers

```cpp
#include <thread>
#include <vector>

void safe_print(int i);

int main() {
    std::vector<std::thread> threadPool;
    for (int i = 1; i <= 9; ++i) {
        threadPool.emplace_back([i]() { safe_print(i); });
    }
    // Digits 1 to 9 are printed (unordered)
    for (auto& t : threadPool) {
        t.join();
    }
}
```
Other Functions of the Thread Library

The thread library also contains other useful functions that are closely related to starting and stopping threads:

• `std::this_thread::sleep_for()`: Stop the current thread for a given amount of time
• `std::this_thread::sleep_until()`: Stop the current thread until a given point in time
• `std::this_thread::yield()`: Let the operating system schedule another thread
• `std::this_thread::get_id()`: Get the (operating-system-specific) id of the current thread
Mutual Exclusion (1)

*Mutual exclusion* is a straightforward way to synchronize multiple threads

- Threads acquire a lock on a mutex object before entering a critical section
- Threads release their lock on the mutex when leaving a critical section

High-level programming model

- The resource (usually a class) that requires protection from data races owns a mutex object of the appropriate type
- Threads that intend to access the resource acquire a suitable lock on the mutex *before* performing the actual access
- Threads release their lock on the mutex *after* completing the access
- Usually locks are simply acquired and released in the member functions of the class
Mutual Exclusion (2)

The standard library defines several useful classes that implement mutexes in the <mutex> and <shared_mutex> headers

- `std::mutex` – regular mutual exclusion
- `std::recursive_mutex` – recursive mutual exclusion
- `std::shared_mutex` – mutual exclusion with shared locks

The standard library provides RAII wrappers for locking and unlocking mutexes

- `std::unique_lock` – RAII wrapper for exclusive locking
- `std::shared_lock` – RAII wrapper for shared locking

The RAII wrappers should *always* be preferred for locking and unlocking mutexes

- Makes bugs due to inconsistent locking/unlocking much more unlikely
- Manual locking and unlocking may be required in some rare cases
- Should still be performed through the corresponding functions of the RAII wrappers
std::unique_lock can be used to lock a mutex in exclusive mode

- The constructor acquires an exclusive lock on the mutex
- Constructor syntax: `unique_lock(mutex_type& m)`
- Blocks the calling thread until the mutex becomes available
- The destructor releases the lock automatically
- Can be used with any mutex type from the standard library

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

std::mutex printMutex;
void safe_print(int i) {
    std::unique_lock lock(printMutex); // lock is acquired
    std::cout << i;
} // lock is released
```
std::unique_lock (2)

std::unique_lock provides additional constructors

- unique_lock(mutex_type& m, std::defer_lock_t t) – Do not immediately lock the mutex
- unique_lock(mutex_type& m, std::try_to_lock_t t) – Do not block when the mutex cannot be locked

std::unique_lock provides additional member functions

- lock() – Manually lock the mutex
- try_lock() – Try to lock the mutex, return true if successful
- operator bool() – Check if the std::unique_lock holds a lock on the mutex


std::unique_lock (3)

Example

```cpp
#include <mutex>

std::mutex mutex;

void foo() {
    std::unique_lock lock(mutex, std::try_to_lock);
    if (!lock) {
        doUnsynchronizedWork();

        // block until we can get the lock
        lock.lock();
    }

    doSynchronizedWork();

    // release the lock early
    lock.unlock();

    doUnsynchronizedWork();
}
```
std::unique_lock (4)

std::unique_lock is movable to transfer ownership of a lock on a mutex

```cpp
#include <mutex>

class MyContainer {
    private:
        std::mutex mutex;
    
    public:
        class iterator { /* ... */ }; 

        iterator begin() {
            std::unique_lock lock(mutex);

            // compute the begin iterator constructor args
            // keep the lock for iteration
            return iterator(std::move(lock), ...);
        }
};
```
The following code will deadlock since `std::mutex` can be locked at most once.

```cpp
#include <mutex>

std::mutex mutex;

void bar() {
    std::unique_lock lock(mutex);

    // do some work...
}

google() {
    std::unique_lock lock(mutex);

    // do some work...

    bar(); // INTENTIONALLY BUGGY, will deadlock
}
Recursive Mutexes (2)

std::recursive_mutex implements recursive ownership semantics

- The same thread can lock an std::recursive_mutex multiple times without blocking
- Other threads will still block if an std::recursive_mutex is currently locked
- Can be used with std::unique_lock just like a regular std::mutex
- Useful for functions that call each other and use the same mutex

```cpp
#include <mutex>

std::recursive_mutex mutex;
void bar() {
    std::unique_lock lock(mutex);
}
void foo() {
    std::unique_lock lock(mutex);
    bar(); // OK, will not deadlock
}
```
std::shared_lock (1)

std::shared_lock can be used to lock a mutex in shared mode
• Constructors and member functions analogous to std::unique_lock
• Multiple threads can acquire a shared lock on the same mutex
• Shared locking attempts block if the mutex is locked in exclusive mode
• Only usable in conjunction with std::shared_mutex

We have to adhere to some contract to write well-behaved programs
• Shared mutexes are mostly used to implement read/write-locks
• Only read accesses are allowed when holding a shared lock
• Write accesses are only allowed when holding an exclusive lock
std::shared_lock (2)

Example

```cpp
#include <shared_mutex>

class SafeCounter {
    private:
        mutable std::shared_mutex mutex;
        size_t value = 0;

    public:
        size_t getValue() const {
            std::shared_lock lock(mutex);
            return value; // read access
        }

        void incrementValue() {
            std::unique_lock lock(mutex);
            ++value; // write access
        }
};
```
Working with Mutexes

We usually have to make mutexes `mutable` within our data structures

- The RAII wrappers require mutable references to the mutex
- `const` member functions of our data structure usually also need to use the mutex

Using mutexes without care can easily lead to deadlocks within the system

- Usually occurs when a thread tries to lock another mutex when it already holds a lock on some mutex
- Can in some cases be avoided by using `std::recursive_mutex` (if we are locking the same mutex multiple times)
- Requires dedicated programming techniques when multiple mutexes are involved
Avoiding Deadlocks (1)

The following example will lead to deadlocks

```cpp
std::mutex m1, m2, m3;
void threadA() {
    // INTENTIONALLY BUGGY
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    // INTENTIONALLY BUGGY
    std::unique_lock l3{m3}, l2{m2}, l1{m1};
}
```

Possible deadlock scenario

- threadA() acquires locks on m1 and m2
- threadB() acquires lock on m3
- threadA() waits for threadB() to release m3
- threadB() waits for threadA() to release m2
Avoiding Deadlocks (2)

Deadlocks can be avoided by always locking mutexes in a *globally* consistent order

- Ensures that one thread always “wins”
- Maintaining a globally consistent locking order requires considerable developer discipline
- Maintaining a globally consistent locking order may not be possible at all

```cpp
std::mutex m1, m2, m3;
void threadA() {
    // OK, will not deadlock
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    // OK, will not deadlock
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
```
Avoiding Deadlocks (3)

Sometimes it is not possible to guarantee a globally consistent order

- The `std::scoped_lock` RAII wrapper can be used to safely lock any number of mutexes
- Employs a deadlock-avoidance algorithm if required
- Generally quite inefficient in comparison to `std::unique_lock`
- Should only be used as a last resort!

```cpp
std::mutex m1, m2, m3;
void threadA() {
    // OK, will not deadlock
    std::scoped_lock l{m1, m2, m3};
}
void threadB() {
    // OK, will not deadlock
    std::scoped_lock l{m3, m2, m1};
}
```
Condition Variables (1)

A condition variable is a synchronization primitive that allows multiple threads to wait until an (arbitrary) condition becomes true.

- A condition variable uses a mutex to synchronize threads
- Threads can *wait* on or *notify* the condition variable
- When a thread waits on the condition variable, it blocks until another thread notifies it
- If a thread waited on the condition variable and is notified, it holds the mutex
- A notified thread must check the condition explicitly because *spurious wake-ups* can occur
The standard library defines the class std::condition_variable in the header <condition_variable> which has the following member functions:

- **wait()**: Takes a reference to a std::unique_lock that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable.
- **notify_one()**: Notify a single waiting thread, mutex does not need to be held by the caller.
- **notify_all()**: Notify all waiting threads, mutex does not need to be held by the caller.
Condition Variables Example

One use case for condition variables are worker queues: Tasks are inserted into a queue and then worker threads are notified to do the task.

```cpp
std::mutex m;
std::condition_variable cv;
std::vector<int> taskQueue;

void pushWork(int task) {
    {
        std::unique_lock l{m};
        taskQueue.push_back(task);
    }
    cv.notify_one();
}

void workerThread() {
    std::unique_lock l{m};
    while (true) {
        while (!taskQueue.empty()) {
            int task = taskQueue.back();
            taskQueue.pop_back();
            l.unlock();
            // [...] do actual work here
            l.lock();
        }
        cv.wait(l);
    }
}
```
Atomic Operations

Mutual exclusion may be inefficient for synchronization

- Very coarse-grained synchronization
- May require communication with the operating system

Modern hardware also supports *atomic operations* for synchronization.

- The memory order of a CPU determines how *non-atomic* memory operations are allowed to be reordered
- In C++ all non-atomic conflicting operations have undefined behavior even if the memory order of the CPU would allow it!
- There is one exception: Special atomic functions are allowed to have conflicts
- The compiler usually knows your CPU and generates “real” atomic instructions only if necessary
Atomic Operations Library (1)

C++ provides atomic operations in the atomic operations library

- Implemented in the `<atomic>` header
- `std::atomic<T>` is a class that represents an atomic version of the type T
- Can be used (almost) interchangeably with the original type T
- Has the same size and alignment as the original type T
- Conflicting operations are only allowed on `std::atomic<T>` objects

`std::atomic` on its own does not provide any synchronization at all

- Simply makes conflicting operations possible and defined behavior
- Exposes the guarantees of specific memory models to the programmer
- Suitable programming models must be used to achieve proper synchronization
Atomic Operations Library (2)

\texttt{std::atomic} has several member functions that implement atomic operations:

- \texttt{T load()}: Loads the value
- \texttt{void store(T desired)}: Stores \texttt{desired} in the object
- \texttt{T exchange(T desired)}: Stores \texttt{desired} in the object and returns the old value

If \texttt{T} is an integral type, the following operations also exist:

- \texttt{T fetch_add(T arg)}: Adds \texttt{arg} to the value and returns the old value
- \texttt{T fetch_sub(T arg)}: Same for subtraction
- \texttt{T fetch_and(T arg)}: Same for bitwise and
- \texttt{T fetch_or(T arg)}: Same for bitwise or
- \texttt{T fetch_xor(T arg)}: Same for bitwise xor
Example (without atomics)

```cpp
#include <thread>

int main() {
    unsigned value = 0;
    std::thread t([&]() {
        for (size_t i = 0; i < 10; ++i)
            ++value; // UNDEFINED BEHAVIOR, data race
    });

    for (size_t i = 0; i < 10; ++i)
        ++value; // UNDEFINED BEHAVIOR, data race

    t.join();

    // value will contain garbage
}
```
Atomic Operations Library (4)

Example (with atomics)

```cpp
#include <atomic>
#include <thread>

int main() {
    std::atomic<unsigned> value = 0;
    std::thread t([&]() {
        for (size_t i = 0; i < 10; ++i)
            value.fetch_add(1); // OK, atomic increment
    });

    for (size_t i = 0; i < 10; ++i)
        value.fetch_add(1); // OK, atomic increment

    t.join();

    // value will contain 20
}
Semantics of Atomic Operations

C++ may support atomic operations that are not supported by the CPU

- `std::atomic<T>` can be used with any trivially copyable type
- In particular also for types that are much larger than one cache line
- To guarantee atomicity, compilers are allowed to fall back to mutexes

The C++ standard defines precise semantics for atomic operations

- Every atomic object has a totally ordered *modification order*
- There are several *memory orders* that define how operations on different atomic objects may be reordered
- The C++ memory orders do not necessarily map precisely to memory orders defined by a CPU
Modification Order (1)

All modifications of a *single* atomic object are totally ordered
- This is called the *modification order* of the object
- All threads are guaranteed to observe modifications of the object in this order

Modifications of *different* atomic objects may be unordered
- Different threads may observe modifications of multiple atomic objects in a different order
- The details depend on the *memory order* that is used for the atomic operations
Modification Order (2)

Example

```cpp
std::atomic<int> i = 0, j = 0;
void workerThread() {
    i.fetch_add(1); // (A)
    i.fetch_sub(1); // (B)
    j.fetch_add(1); // (C)
}
void readerThread() {
    int iLocal = i.load(), jLocal = j.load();
    assert(iLocal != -1); // always true
}
```

Observations

- Reader threads will never see a modification order with (B) before (A)
- Depending on the memory order, multiple reader threads may see any of (A), (B), (C), or (A), (C), (B), or (C), (A), (B)
The atomics library defines several memory orders

- All atomic functions take a memory order as their last parameter
- The two most important memory orders are `std::memory_order_relaxed` and `std::memory_order_seq_cst`
- `std::memory_order_seq_cst` is used by default if no memory order is explicitly supplied
- You should stick to this default unless you identified the atomic operation to be a performance bottleneck

```cpp
std::atomic<int> i = 0;
i.fetch_add(1); // uses std::memory_order_seq_cst
i.fetch_add(1, std::memory_order_seq_cst);
i.fetch_add(1, std::memory_order_relaxed);
```
Memory Order (2)

std::memory_order_relaxed

- Roughly maps to a CPU with weak memory order
- Only consistent modification order is guaranteed
- Atomic operations of different objects may be reordered arbitrarily

```cpp
std::atomic<int> i = 0, j = 0;
void threadA() {
    while (true) {
        i.fetch_add(1, std::memory_order_relaxed); // (A)
        i.fetch_sub(1, std::memory_order_relaxed); // (B)
        j.fetch_add(1, std::memory_order_relaxed); // (C)
    }
}
void threadB() { /* ... */ }
void threadC() { /* ... */ }
```

Observations

- `threadB()` may observe (A), (B), (C)
- `threadC()` may observe (C), (A), (B)
Memory Order (3)

std::memory_order_seq_cst
- Roughly maps to a CPU with strong memory order
- Guarantees that all threads see all atomic operations in one globally consistent order

```cpp
std::atomic<int> i = 0, j = 0;
void threadA() {
    while (true) {
        i.fetch_add(1, std::memory_order_seq_cst); // (A)
        i.fetch_sub(1, std::memory_order_seq_cst); // (B)
        j.fetch_add(1, std::memory_order_seq_cst); // (C)
    }
}
void threadB() { /* ... */ }
void threadC() { /* ... */ }
```

Observations
- `threadB()` may observe (C), (A), (B)
- `threadC()` will then also observe (C), (A), (B)
Compare-And-Swap Operations (1)

Compare-and-swap operations are one of the most useful operations on atomics

- **Signature:** `bool compare_exchange_weak(T& expected, T desired)`
- Compares the current value of the atomic to `expected`
- Replaces the current value by `desired` if the atomic contained the expected value and returns `true`
- Updates `expected` to contain the current value of the atomic object and returns `false` otherwise

Often the main building block to synchronize data structures without mutexes

- Allows us to check that no modifications occurred to an atomic over some time period
- Can be used to implement “implicit” mutual exclusion
- Can suffer from subtle problems such as the A-B-A problem
Compare-And-Swap Operations (2)

Example: Insert into a lock-free singly linked list

```cpp
#include <atomic>

class SafeList {
    private:
        struct Entry {
            T value;
            Entry* next;
        };

        std::atomic<Entry*> head;

        Entry* allocateEntry(const T& value);

    public:
        void insert(const T& value) {
            auto* entry = allocateEntry(value);
            auto* currentHead = head.load();
            do {
                entry->next = currentHead;
            } while (!head.compare_exchange_weak(currentHead, entry));
        }
};
```
Compare-And-Swap Operations (3)

std::atomic actually provides two CAS versions with the same signature

- `compare_exchange_weak` – weak CAS
- `compare_exchange_strong` – strong CAS

Semantics

- The weak version is allowed to return false, even when no other thread modified the value
- This is called “spurious failure”
- The strong version may use a loop internally to avoid this
- General rule: If you use a CAS operation in a loop, always use the weak version
std::atomic_ref (1)

std::atomic can be unwieldy

- std::atomic is neither movable nor copyable
- As a consequence it cannot easily be used in standard library containers

std::atomic_ref allows us to apply atomic operations to non-atomic objects

- The constructor takes a reference to an arbitrary object of type T
- The referenced object is treated as an atomic object during the lifetime of the std::atomic_ref
- std::atomic_ref defines similar member functions to std::atomic

Data races between accesses through std::atomic_ref and non-atomic accesses are still undefined behavior!
std::atomic_ref (2)

Example

```cpp
#include <atomic>
#include <thread>
#include <vector>

int main() {
    std::vector<int> localCounters(4);
    std::vector<std::thread> threads;

    for (size_t i = 0; i < 16; ++i) {
        threads.emplace_back([&]() {
            for (size_t j = 0; j < 100; ++j) {
                std::atomic_ref ref(localCounters[i % 4]);
                ref.fetch_add(1);
            }
        });
    }

    for (auto& thread : threads) {
        thread.join();
    }
}
```