Concurrency in C++

Back to Basics

Dheeraj Jha

Who Am I?

- Practicing software engineering for more than a decade.
- Core Team member of CppIndia
- Primarily using C++ as a programming language.
- Working in IoT domain and have interest in cloud technologies as well.
- Have interest in writing blogs and reading books.

Contact:

- www.jhadheeraj.com
- contact@jhadheeraj.com
- LinkedIn: @jhadheeraj
- Twitter: @dheerajjha03

Today we will discuss about

- Concurrency
- Concurrency vs Parallelism
- Concurrency with Process/thread
- Why use concurrency and associated cost?
- Launch a thread
- join() & detach()
- Passing arguments
- Thread synchronisation
- Some common patterns

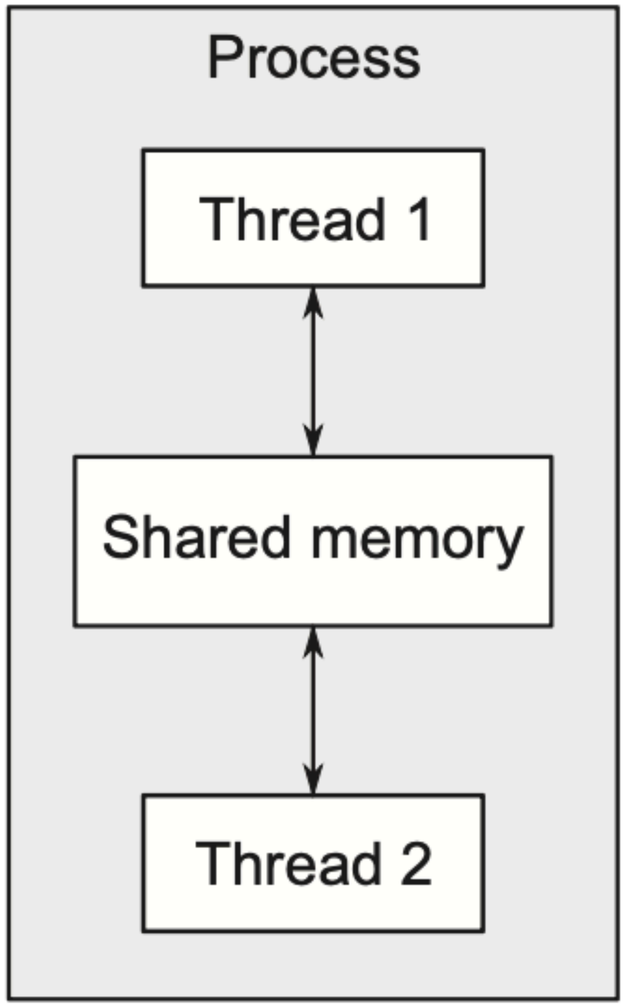
Concurrency

Concurrency vs Parallelism

- Concurrency is the ability of a program to have multiple tasks in progress at the same time, while parallelism is the ability of a system to perform multiple tasks simultaneously.
- Concurrency is a programming concept, while parallelism is a system-level concept.
- Concurrency can be achieved through techniques such as multi-threading or multi-tasking, while parallelism requires multiple processors or cores.
- Concurrency deals with managing and coordinating multiple tasks, while parallelism deals with actually performing those tasks simultaneously.
- Concurrency can improve the responsiveness and throughput of a program, while parallelism can improve the performance and speed of a system.

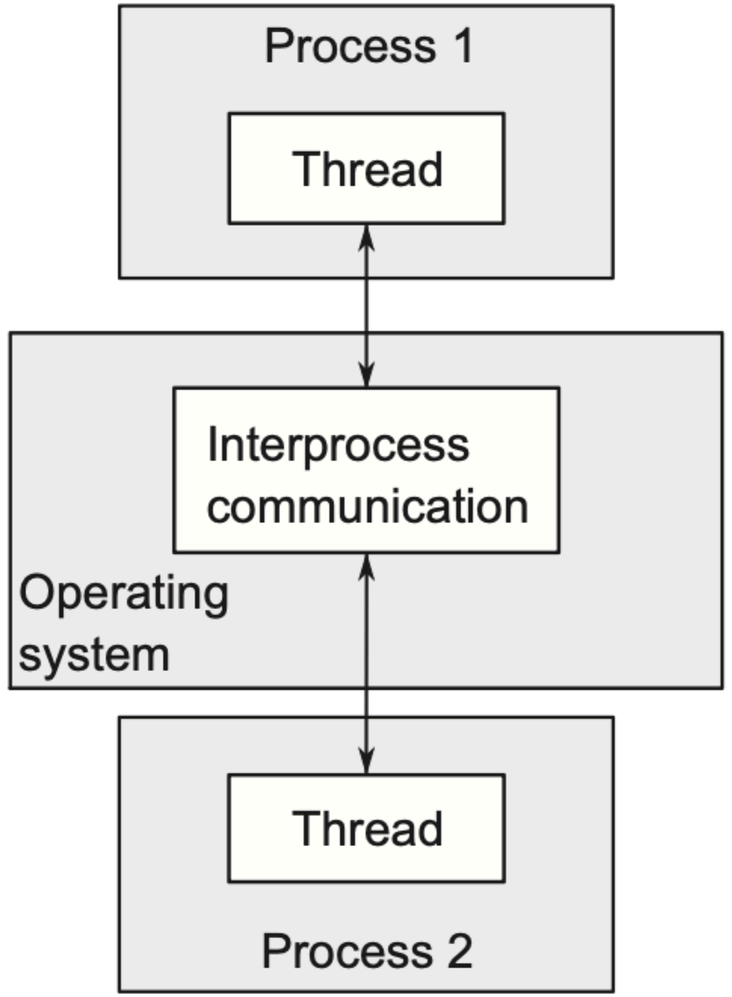
Concurrency With Multiple Processes

- Processes are independent units of execution that run separately from each other, each with its own memory space, resources, and system state.
- Processes can communicate with each other using inter-process communication (IPC) mechanisms such as pipes, sockets, or shared memory.
- Processes can have multiple threads, allowing them to perform concurrent execution and improve performance.
- Operating systems use schedulers to manage the execution of processes and allocate resources such as CPU time and memory.
- Processes can be isolated from each other, providing a level of security and protection from errors or crashes in other processes.



Concurrency With Multiple Threads

- Threads share the same memory space as the parent process, allowing them to access and modify shared data easily.
- Each thread has its own stack, which stores the thread's local variables and function call history.
- Threads can use synchronization mechanisms like mutexes or semaphores to protect shared data from concurrent access.
- Threads can improve the performance and responsiveness of a program, especially in multi-core systems.



Why use Concurrency?

- Separation of concerns
- Performance, specifically task and data parallelism.

Costs of Using Concurrency/Threads

- **Code complexity**: Threads can make the code more complex and harder to understand, which can lead to more bugs and maintenance issues.
- **Overhead**: Threads have an inherent overhead associated with them, such as allocating kernel resources and stack space, and adding the new thread to the scheduler, which can take time and lead to a performance hit.
- **Scalability**: Threads can quickly exhaust system resources if used excessively, which can lead to poor scalability and performance issues when dealing with high-demand servers that need to handle many connections.

Using Thread

Launching a Thread (1)

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

void do_some_work(){
    std::cout<<"Do something wonderful."<<std::endl;
}

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    std::thread my_thread(do_some_work);
    my_thread.join();
    return 1;
}
```

Launching a Thread (2)

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

void do_some_work(){
    std::cout<<"Do something wonderful."<<std::endl;
}

class background_task
{
public:
    void operator()() const
    {
        std::cout<<"Printing from object."<<std::endl;
        do_some_work();
    }
};

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    background_task f;
    std::thread my_thread(f);

    my_thread.join();
    return 1;
}
```

Launching a Thread (3)

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

void do_some_work(){
    std::cout<<"Do something wonderful."<<std::endl;
}

class background_task
{
public:
    void operator()() const
    {
        std::cout<<"Printing from object."<<std::endl;
        do_some_work();
    }
};

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;

    std::thread my_thread(background_task());    //Do you see any issue in this line?
    my_thread.join();
    return 1;
}
```

Launching a Thread (4)

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

void do_some_work(){
    std::cout<<"Do something wonderful."<<std::endl;
}

class background_task
{
public:
    void operator()() const
    {
        std::cout<<"Printing from object."<<std::endl;
        do_some_work();
    }
};

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    std::thread my_thread((background_task()));
    my_thread.join();
    return 1;
}
```

Launching a Thread (5)

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

void do_some_work(){
    std::cout<<"Do something wonderful."<<std::endl;
}

class background_task
{
public:
    void operator()() const
    {
        std::cout<<"Printing from object."<<std::endl;
        do_some_work();
    }
};

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    std::thread my_thread{background_task()};    //uniform initialization syntax
    my_thread.join();
    return 1;
}
```

Launching a Thread (6)

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

void do_some_work(){
    std::cout<<"Do something wonderful."<<std::endl;
}

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    std::thread my_thread([]{
        std::cout<<"Printing from Lambda"<<std::endl;
        do_some_work();
    });

    my_thread.join();
    return 1;
}
```

Wait for thread (join)

- Joining a thread means to wait for its completion before the execution of the main thread continues.
- When a thread is joined, the program will be blocked until the thread finishes its execution.
- A thread can only be joined once, after which it will be considered as non-joinable.
- Joining a thread ensures that the thread has completed its execution and that any resources it was using are freed before the main thread continues.
- Joining a thread can be useful to get the return value or output from a thread function.

Running Independent thread (detach)

- Detaching a thread means allowing it to run independently of the main thread and its execution state is not tracked by the program.
- The program will not be blocked when a thread is detached and the execution of the main thread continues concurrently with the detached thread.
- A detached thread continues execution even if the std::thread object that created it goes out of scope.
- Resources used by the thread will not be freed automatically when the thread finishes execution, the programmer needs to handle it.
- When a detached thread finishes execution, the program may not be aware of it and so it may not be able to retrieve its return value or output.

Running Independent thread (detach)

```cpp
void do_some_work(int& i){
    std::cout<<"print i: "<< i <<std::endl;
}

struct threadStruct{
    int& i;
    threadStruct(int& i_):i(i_){}
    void operator()(){
        for(int j=0; j<100000; j++){
            do_some_work(i);    //Potential access to dangling reference
        }
    }
};

void threadFun(){
    std::cout<<"threadFun() Enter"<<std::endl;

    int local_var{0};
    threadStruct obj(local_var);
    std::thread my_thread(obj);
    my_thread.detach();     // Don't wait for thread to finish
    std::cout<<"threadFun() Exit"<<std::endl;
}                           // New thread might still be running

```

Waiting in exceptional circumstances (1)

```cpp
struct threadStruct;
void threadFun(){
    std::cout<<"threadFun() Enter"<<std::endl;

    int local_var{0};
    threadStruct obj(local_var);
    std::thread my_thread(obj);
    try{
        //do something wonderful.
        throw;
    }catch (...){
        // Handle exception
    }
    my_thread.join();

    std::cout<<"threadFun() Exit"<<std::endl;
}

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    threadFun();
    return 1;
}
```

Waiting in exceptional circumstances (2)

```cpp
struct threadStruct;
void threadFun(){
    std::cout<<"threadFun() Enter"<<std::endl;

    int local_var{0};
    threadStruct obj(local_var);
    std::thread my_thread(obj);
    try{
        //do something wonderful.
        throw;
    }catch (...){
        // Handle exception
        my_thread.join();
    }
    my_thread.join();

    std::cout<<"threadFun() Exit"<<std::endl;
}

int main(){
    std::cout<<"Hello Test Thread"<<std::endl;
    threadFun();
    return 1;
}
```

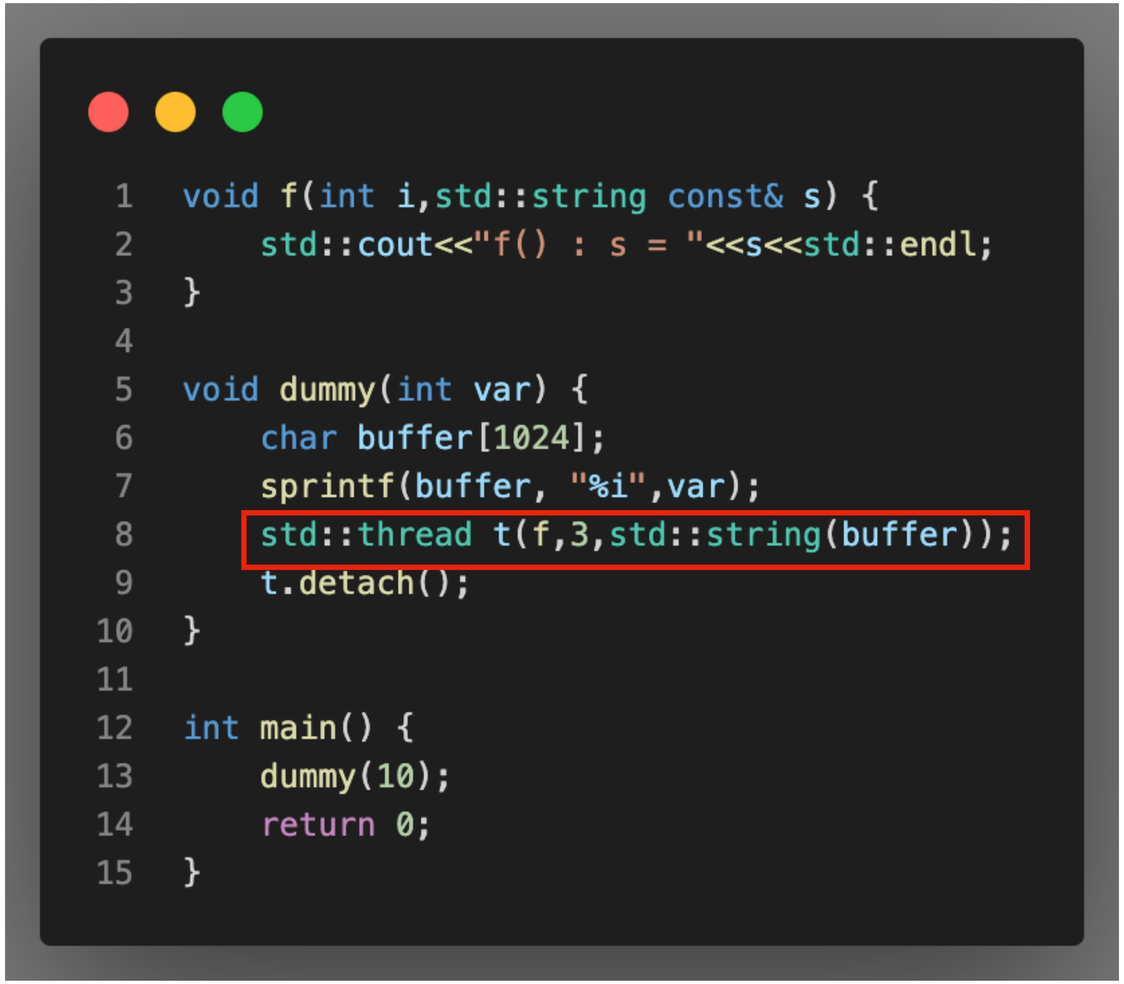
Waiting in exceptional circumstances (3)

```cpp
class thread_guard{
    std::thread& _t;
    public:
    explicit thread_guard(std::thread& t) : _t(t) {}
    ~thread_guard(){
        std::cout<<"~thread_guard()"<<std::endl;
        if (_t.joinable()){
            _t.join();
        }
        std::cout<<"~thread_guard(): exit"<<std::endl;
    }
    thread_guard (thread_guard const& ) = delete;
    thread_guard& operator=(thread_guard const& ) = delete;
};

void threadFun(){
    std::cout<<"threadFun() Enter"<<std::endl;

    int local_var{0};
    threadStruct obj(local_var);
    std::thread my_thread(obj);
    thread_guard th_guard(my_thread);
    try
    {
        // Do something wonderful.
        throw 1;
    }
    catch(...)
    {
        std::cerr << "Exception" << '\n';
    }
    std::cout<<"threadFun() Exit"<<std::endl;
}
```

Passing arguments to a thread function (1)

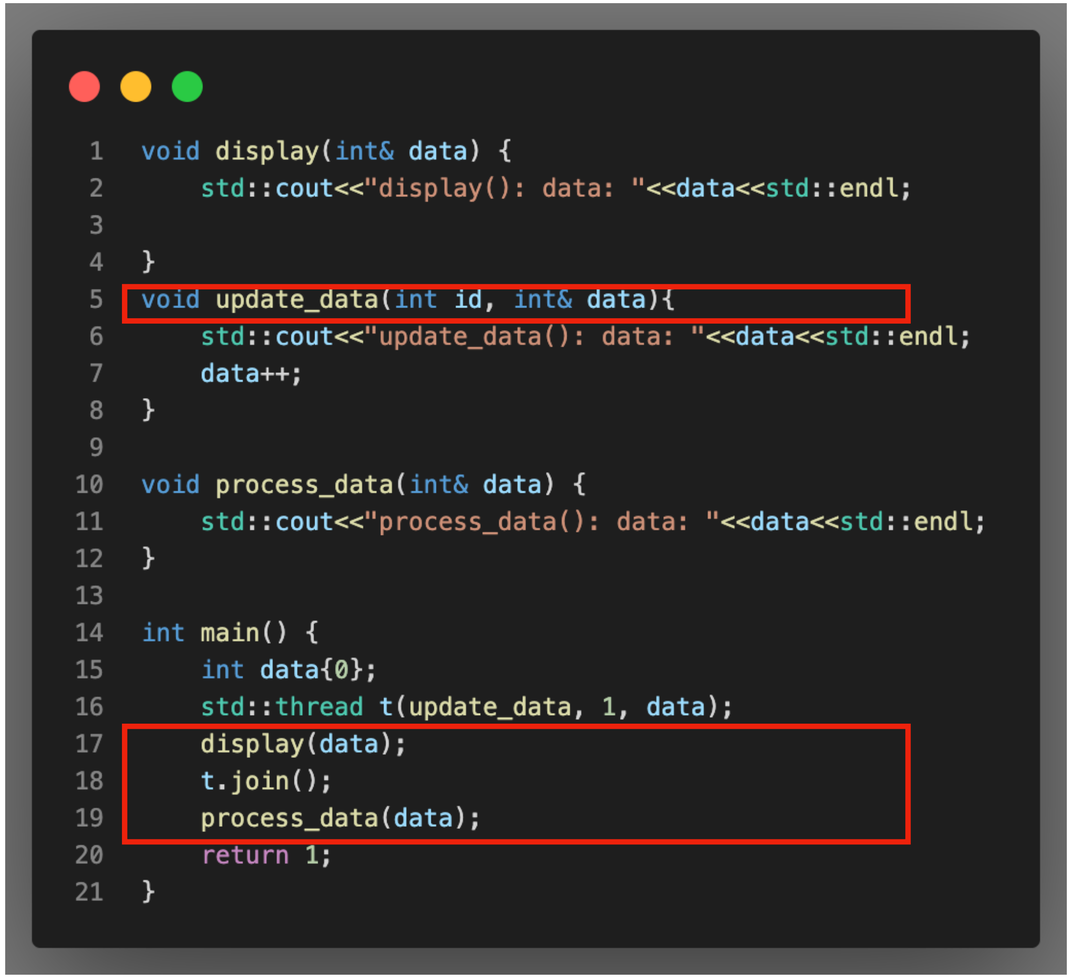
```
1  void f(int i,std::string const& s) {
2      std::cout<<"f() : s = "<<s<<std::endl;
3  }
4
5  void dummy(int var) {
6      char buffer[1024];
7      sprintf(buffer, "%i",var);
8      std::thread t(f,3,buffer);
9      t.detach();
10 }
11
12 int main() {
13     dummy(10);
14     return 0;
15 }
```

Passing arguments to a thread function (2)



```
1  void f(int i,std::string const& s) {
2      std::cout<<"f() : s = "<<s<<std::endl;
3  }
4
5  void dummy(int var) {
6      char buffer[1024];
7      sprintf(buffer, "%i",var);
8      std::thread t(f,3,std::string(buffer));
9      t.detach();
10 }
11
12 int main() {
13     dummy(10);
14     return 0;
15 }
```

Pass arguments by reference (1)



```cpp
void display(int& data) {
    std::cout<<"display(): data: "<<data<<std::endl;

}
void update_data(int id, int& data){
    std::cout<<"update_data(): data: "<<data<<std::endl;
    data++;
}

void process_data(int& data) {
    std::cout<<"process_data(): data: "<<data<<std::endl;
}

int main() {
    int data{0};
    std::thread t(update_data, 1, data);
    display(data);
    t.join();
    process_data(data);
    return 1;
}
```

Pass arguments by reference (2)

```cpp
void display(int& data) {
    std::cout<<"display(): data: "<<data<<std::endl;

}
void update_data(int id, int& data){
    std::cout<<"update_data(): data: "<<data<<std::endl;
    data++;
}

void process_data(int& data) {
    std::cout<<"process_data(): data: "<<data<<std::endl;
}

int main() {
    int data{0};
    std::thread t(update_data, 1, std::ref(data));
    display(data);
    t.join();
    process_data(data);
    return 1;
}
```

Identifying threads

- Thread identifiers are of type *std::thread::id*
- The identifier for a thread can be obtained from its associated std::thread object by calling the *get_id()* member function.
  - If the *std::thread* object doesn't have an associated thread of execution, the call to *get_id()* returns a default-constructed *std::thread::id* object, which indicates "not any thread."
- Alternatively, the identifier for the current thread can be obtained by calling *std::thread::get_id()*.
- Objects of type *std::thread::id* can be freely copied and compared
- Objects of type std::thread::id offer the complete set of comparison operators
- Instances of *std::thread::id* are often used to check whether a thread needs to perform some operation.

Synchronisation

Problems with sharing data between threads

- **Race conditions**: When multiple threads access and modify the same data simultaneously, it can lead to unpredictable results and data inconsistencies.
- **Deadlocks**: When two or more threads are blocked waiting for each other to release a resource, it can lead to a deadlock where none of the threads can proceed.
- **Livelocks**: A livelock is similar to a deadlock, but in this case, the threads are active but unable to make progress because they are constantly trying to acquire a resource that is already held by another thread.
- **Priority Inversion**: When a low-priority thread holds a resource that a high-priority thread needs, the high-priority thread may be blocked and unable to proceed, even though it has higher priority.
- **Starvation**: When a thread is unable to acquire a resource it needs because it is constantly being acquired by other threads, it can lead to starvation where the thread is unable to make progress.
- **Data Corruption**: When multiple threads access the same data without proper synchronization, it can lead to data corruption, where the data becomes inconsistent or incorrect.

Example of Race Condition

```cpp
int value{0};

void printEven()
{
    for (int i=0; i<10; i++){
        if (0 == value % 2) std::cout<<"Value: " << value << std::endl;
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}

void printOdd()
{
    for (int i=0; i<10; i++){
        if (0 != value % 2) std::cout<<"Value: " << value << std::endl;
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}

int main() {
    std::thread t1(printEven);
    std::thread t2(printOdd);

    t1.join();
    t2.join();
    return 0;
}
```

Synchronisation using Mutex

```cpp
int value{0};
std::mutex some_mutex;

void printEven()
{
    for (int i=0; i<10; i++){
        some_mutex.lock();
        if (0 == value % 2) std::cout<<"Value: " << value << std::endl;
        value++;
        some_mutex.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}

void printOdd()
{
    for (int i=0; i<10; i++){
        some_mutex.lock();
        if (0 != value % 2) std::cout<<"Value: " << value << std::endl;
        value++;
        some_mutex.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

std::mutex

- std::mutex is a standard C++ library class that provides a mutual exclusion mechanism, also known as a lock.
- It is used to ensure that only one thread can execute a specific section of code at a time, thus preventing race conditions and other synchronization issues.
- std::mutex class provides two main methods: lock() and unlock(), which can be used to acquire and release the lock respectively.
- std::mutex class also provides other methods like try_lock() which attempts to acquire the lock without blocking,
- std::mutex is a blocking mechanism, meaning that a thread that attempts to acquire a locked mutex will be blocked until the mutex is released by another thread. This can lead to performance problems if not used carefully.

Member types

Member type	Definition
`native_handle_type` (not always present)	*implementation-defined*

Member functions

(constructor)	constructs the mutex (public member function)
(destructor)	destroys the mutex (public member function)
operator= [deleted]	not copy-assignable (public member function)
Locking	
lock	locks the mutex, blocks if the mutex is not available (public member function)
try_lock	tries to lock the mutex, returns if the mutex is not available (public member function)
unlock	unlocks the mutex (public member function)
Native handle	
native_handle	returns the underlying implementation-defined native handle object (public member function)

std::lock_guard

```cpp
int value{0};
std::mutex some_mutex;

void printEven()
{
    for (int i=0; i<10; i++){
        std::lock_guard<std::mutex> guard(some_mutex);
        if (0 == value % 2) std::cout<<"Value: " << value << std::endl;
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}

void printOdd()
{
    for (int i=0; i<10; i++){
        std::lock_guard<std::mutex> guard(some_mutex);
        if (0 != value % 2) std::cout<<"Value: " << value << std::endl;
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

std::lock_guard

- std::lock_guard is a C++ Standard Template Library (STL) class that provides a convenient and safe way to manage a mutex lock.
- It is used to automatically acquire and release a lock (std::mutex) when the lock_guard object goes out of scope.
- The lock_guard class is non-copyable and non-movable, and it ensures that the lock is acquired at the time of construction and released at the time of destruction.
- std::lock_guard is an RAII (Resource Acquisition Is Initialization) wrapper for a std::mutex and it provides an exception-safe way to lock and unlock a mutex.
- Using lock_guard is simpler and more efficient than manually locking and unlocking a mutex, as it eliminates the possibility of forgetting to unlock the mutex, which could lead to deadlocks or other synchronization issues.

Member types

Member type	Definition
mutex_type	Mutex

Member functions

(constructor)	constructs a lock_guard, optionally locking the given mutex (public member function)
(destructor)	destructs the lock_guard object, unlocks the underlying mutex (public member function)
operator= [deleted]	not copy-assignable (public member function)

std::atomic<int>

```cpp
std::atomic<int> value(0); // Atomic variable to store the current value

void printEven()
{
    while (value < 10) {
        // Wait until value is even
        while (value % 2 != 0) {
            std::this_thread::yield();
        }

        std::cout << "Value: " << value << std::endl;
        value++;
    }
}

void printOdd()
{
    while (value < 10) {
        // Wait until value is odd
        while (value % 2 == 0) {
            std::this_thread::yield();
        }

        std::cout << "Value: " << value << std::endl;
        value++;
    }
}
```

std::atomic<int>

- std::atomic<int> is a C++ Standard Template Library (STL) class that provides atomic operations for integral types, such as int.
- It is a template class that wraps a value of the specified type and provides atomic operations on that value, such as load, store, compare_exchange and fetch_* operations.
- std::atomic<int> class provides thread-safe operations, which means that multiple threads can access and manipulate the atomic variable simultaneously without causing data race conditions.
- std::atomic<int> class guarantees that an operation on the atomic variable is indivisible and uninterruptible, so that the value of the variable remains consistent and predictable.
- std::atomic<int> class provides a way of achieving low-level synchronization without using locks, mutexes or other synchronizing objects. This can be useful in situations where lock contention is a bottleneck and where performance is critical.

std::this_thread::yield()

- **std::this_thread::yield()** is a function that tells the operating system to temporarily pause the execution of the current thread and allow other threads of the same priority to execute.
- It is useful when a thread is waiting for a shared resource, but it does not want to block the entire thread. Instead, it can call **yield**() to allow other threads to execute while it waits.
- Using **yield**() allows other threads to continue running while the current thread is waiting, rather than wasting CPU cycles in a blocked state.
- It does not block the execution of the thread for a specific period of time like **sleep_for**() function does.
- It is generally more efficient than using sleep, as it allows other threads to continue running while the current thread is waiting, but it could be less predictable than **sleep_for()** function. It is platform-dependent and the exact behavior of this function is not specified by the standard.

Common Concurrency Patterns

Publisher-Consumer Scenario

```cpp
std::queue<int> q;
std::mutex m;
std::condition_variable cv;

void producer()
{
    for (int i = 0; i < 10; i++) {
        std::unique_lock<std::mutex> lock(m);
        q.push(i);
        lock.unlock();
        cv.notify_one();
    }
}

void consumer()
{
    while (true) {
        std::unique_lock<std::mutex> lock(m);
        cv.wait(lock, []{return !q.empty();});
        std::cout << q.front() << std::endl;
        q.pop();
        lock.unlock();
    }
}
```

std::condition_variable

- std::condition_variable is a synchronization primitive that can be used to block a thread until notified by another thread that some condition has been met.
- std::condition_variable allows a thread to wait until a specific condition is met, and to be notified when that condition is met by another thread.
- std::condition_variable works in conjunction with std::unique_lock to synchronize access to shared data.
- std::condition_variable provides three main functions: wait(), notify_one(), and notify_all() which allows to wait, notify one or all waiting threads.
- std::condition_variable is useful in the producer-consumer problem, where one or more threads are producing data and one or more threads are consuming the data.

Member functions

(constructor)	constructs the object (public member function)
(destructor)	destructs the object (public member function)
operator= [deleted]	not copy-assignable (public member function)
Notification	
notify_one	notifies one waiting thread (public member function)
notify_all	notifies all waiting threads (public member function)
Waiting	
wait	blocks the current thread until the condition variable is awakened (public member function)
wait_for	blocks the current thread until the condition variable is awakened or after the specified timeout duration (public member function)
wait_until	blocks the current thread until the condition variable is awakened or until specified time point has been reached (public member function)
Native handle	
native_handle	returns the native handle (public member function)

std::unique_lock

- std::unique_lock is a general-purpose mutex ownership wrapper that is similar to std::lock_guard, but with added flexibility.
- std::unique_lock provides more flexibility than std::lock_guard, such as the ability to temporarily release ownership of the lock and the ability to transfer ownership of the lock to another unique_lock.
- std::unique_lock can be used to lock multiple mutexes simultaneously by using std::lock() function.
- std::unique_lock objects can be moved but not copied.
- std::unique_lock provides a member function named "owns_lock()" that can be used to check if the lock is acquired or not.

Member functions

(constructor)	constructs a `unique_lock`, optionally locking (i.e., taking ownership of) the supplied r (public member function)
(destructor)	unlocks (i.e., releases ownership of) the associated mutex, if owned (public member function)
`operator=`	unlocks (i.e., releases ownership of) the mutex, if owned, and acquires ownership of a (public member function)
Locking	
`lock`	locks (i.e., takes ownership of) the associated mutex (public member function)
`try_lock`	tries to lock (i.e., takes ownership of) the associated mutex without blocking (public member function)
`try_lock_for`	attempts to lock (i.e., takes ownership of) the associated *TimedLockable* mutex, return has been unavailable for the specified time duration (public member function)
`try_lock_until`	tries to lock (i.e., takes ownership of) the associated *TimedLockable* mutex, returns if tl been unavailable until specified time point has been reached (public member function)
`unlock`	unlocks (i.e., releases ownership of) the associated mutex (public member function)
Modifiers	
`swap`	swaps state with another **`std::unique_lock`** (public member function)
`release`	disassociates the associated mutex without unlocking (i.e., releasing ownership of) it (public member function)
Observers	
`mutex`	returns a pointer to the associated mutex (public member function)
`owns_lock`	tests whether the lock owns (i.e., has locked) its associated mutex (public member function)
`operator bool`	tests whether the lock owns (i.e., has locked) its associated mutex (public member function)

std::unique_lock vs std::lock_guard

unique_lock

- When you need to conditionally lock and unlock a mutex based on some condition.
- When you want to use the lock and unlock operations in a non-RAII style.
- When you want to use advanced features of a mutex such as timed lock operations.
- When you want to transfer ownership of a lock between unique_lock objects.
- When you need to use the lock in a scope and want to release the lock when the scope exits.

lock_guard

- When you want to use the RAII principle for automatically managing the lifetime of a lock.
- When you want to ensure that the lock is unlocked in case of exceptions.
- When you don't need the flexibility of transferring ownership or timed lock operations.
- When you want to simplify the coding and avoid accidentally forgetting to unlock the mutex.
- When you want to use a lock in a single scope and don't need to release the lock manually before the scope exits.

Reader-Writer Problem

- The reader-writer problem in concurrency refers to managing access to a shared location by multiple threads
- std::shared_mutex allows multiple threads to read a value simultaneously
- std::unique_lock is used to ensure only one thread can write to the location at a time
- Using both std::shared_mutex and std::unique_lock can effectively solve the readers/writers problem by managing read and write operations on the shared location
- This prevents conflicts or inconsistencies while accessing the shared location.

Barrier synchronization mechanism

- A barrier is a synchronization mechanism that **allows a group of threads to wait for each other to reach a certain point** before proceeding.
- This pattern is useful in situations where a set of threads need to work together to complete a task, and it is essential that they finish a specific step before moving on to the next one.
- (C++20) **std::barrier**: A barrier is a synchronization object that allows multiple threads to block until a certain number of threads have reached the barrier.
- (C++20) **std::counting_semaphore**: A counting semaphore is a synchronization object that allows multiple threads to block until a certain number of permits have been acquired.
- (C++20) **std::cyclic_barrier**: A cyclic barrier is a variation of a barrier that allows threads to block until a certain number of threads have reached the barrier, at which point all threads are released.
- **std::promise and std::future**: These C++11 classes allow a thread to pass a value or an exception to another thread, and can be used to implement a barrier pattern where one thread signals the other threads to proceed.
- (C++20) **std::latch**: A latch is a synchronization object that allows a thread to block until a certain number of threads have reached the latch

Conclusion

- Always use **RAII** (Resource Acquisition Is Initialization) for managing resources such as locks, threads and memory.
- Avoid using global variables and static variables as they can cause data races and undefined behavior.
- Use **std::atomic** or other synchronization mechanisms such as **std::mutex** and **std::lock_guard** to protect shared resources.
- Use **std::condition_variable** and **std::condition_variable_any** to synchronize between threads.
- Use move semantics and avoid copying when working with thread objects.
- Be aware of the overhead associated with launching a thread, and use thread pools and other techniques to manage the number of threads.
- Be aware of the performance implications of context switching, and use techniques such as lock-free data structures to reduce contention.
- Use the **std::future** and **std::promise** classes to pass data between threads safely and efficiently.
- **Always test your concurrent code thoroughly**, and use tools such as **ThreadSanitizer** and **AddressSanitizer** to detect and fix any issues.

Reference

- Back to Basics: Concurrency - Mike Shah - CppCon 2021(https://youtu.be/pfIC-kle4b0)
- Back to Basics: Concurrency - Arthur O'Dwyer - CppCon 2020 (https://youtu.be/F6Ipn7gCOsY)
- Modernes C++ (https://modernescpp.com/index.php/der-einstieg-in-modernes-c)
- Book: C++ Concurrency in Action by ANTHONY WILLIAMS

Questions?

Thank You