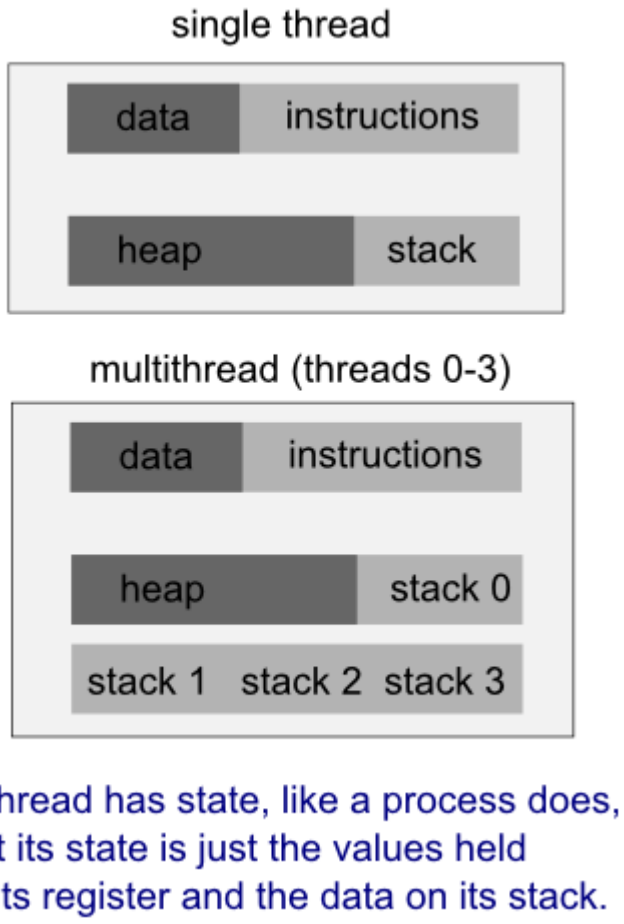
Multithreading in C++

It is an ability of a platform or application to create a process that exist of multiple threads of execution.

A thread of execution is smallest sequence of programming.

A **thread** uses the same address space of a process. A process can have multiple threads. A key difference between processes and threads is that multiple threads **share** parts of their state. Typically, multiple threads can read from and write to the same memory (no process can directly access the memory of another process). However, each thread still has its own stack of activation records and its own copy of CPU registers, including the stack pointer and the program counter, which together describe the **state** of the thread's execution.



❖ **Why multithreaded?**

Application that has more than one thread of execution within the application itself is called multhreaded application.

- For example, if we want to create a server that can serve as many concurrent connections as the the server can cope with, we can achieve such a task relatively easily if we devote a new thread to each connection.
- when we need multithreading is GUI application. GUI applications have one thread of execution (Main Thread) and do one operation at a time. This thread is either waiting for an event or processing an event. So, if the user triggers a time-consuming operation from the user interface, the interface freezes while the operation is in progress.

- **std::thread**

#include <thread>

The class thread represents a single thread of execution. Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object .

No two std::thread objects may represent the same thread of execution; std::thread is not CopyConstructible orCopyAssignable, although it is MoveConstructible and MoveAssignable.

❖ Member functions

(constructor)	constructs new thread object (public member function)
(destructor)	destructs the thread object, underlying thread must be joined or detached (public member function)
operator=	moves the thread object (public member function)

Observers

joinable	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
get_id	returns the *id* of the thread (public member function)
native_handle	returns the underlying implementation-defined thread handle (public member function)

hardware_concurrency [static]	returns the number of concurrent threads supported by the implementation (public static member function)

Operations

join	waits for a thread to finish its execution (public member function)
detach	permits the thread to execute independently from the thread handle (public member function)
swap	swaps two thread objects (public member function)

std::thread t_empty;

Creates an thread object but it executes nothing.

- std::thread::join

```
void join();          (since C++11)
```

Blocks the current thread until the thread identified by *this finishes its execution

Return value

(none)

Postconditions

joinable is false

Note:

1. Once a thread is started we **wait for this thread to finish by calling join()** function on thread object.

2. Double join will result into program termination.

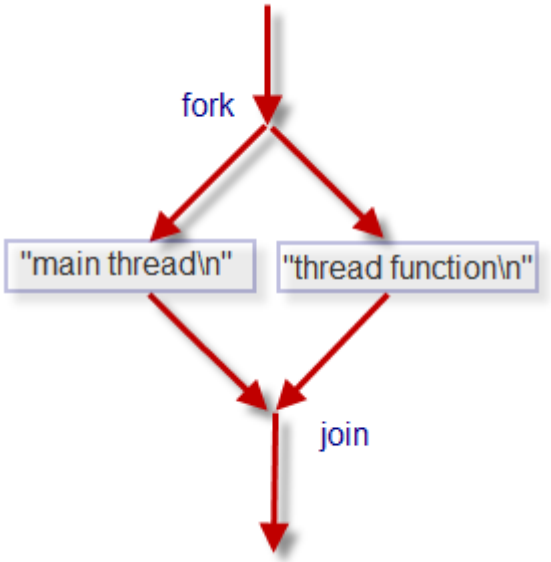
3. If needed we should **check thread is joinable** before joining. (using joinable() function)

e.g:

```
    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::thread::joinable

Checks if the thread object identifies an active thread of execution. Specifically, returns true if get_id() != std::thread::id(). So a default constructed thread is not joinable.

A thread that has finished executing code, but has not yet been joined is still considered an active thread of execution and is therefore joinable.

Return value

true if the thread object identifies an active thread of execution, false otherwise.

e.g:

```
std::thread t;

t = std::thread(foo);
if (t.joinable())
{
   t.join();
}
```

❖ std::thread::detach

Separates the thread of execution from the thread object, allowing execution to continue independently. Any allocated resources will be freed once the thread exits.

After calling detach *this no longer owns any thread.

❖ Postconditions

joinable is false

Note:

1. This is used to detach newly created thread from the parent thread.

2. Always check before detaching a thread that it is joinable otherwise we may end up double detaching and **double detach() will result into program termination**.

3. If we have detached thread and main function is returning then the detached thread execution is suspended.

BIG NOTES:

Either join() or detach() should be called on thread object, otherwise during thread object's destructor it will terminate the program.

Because inside destructor it checks if thread is still joinable? if yes then it terminates the program.

e.g:

```
   std::thread t(independentThread);
   t.detach();
```

```
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Exiting thread caller.\n";
```

❖ std::this_thread::yield

Provides a hint to the implementation to reschedule the execution of threads, allowing other threads to run.

❖ std::this_thread::sleep_for

Blocks the execution of the current thread for at least the specified ***sleep_duration***.

A steady clock is used to measure the duration. This function may block for longer than sleep_duration due to scheduling or resource contention delays.

e.g:

```
std::this_thread::sleep_for(2s);
```

❖ lambda functions

We can replace the thread_function() with lambda function (anonymous function) like this:

```
int main()
{
    std::thread t([]() {std::cout << "thread function\n";});
    std::cout << "main thread\n";
    t.join();     // main thread waits for t to finish
    return 0;
}
```

- **Different types for creating threads.**

❖ Function Pointer – this is the very basic form of creating threads.

```
int fun(int val)
{
    std::cout << val << "thread function\n";
}
int main()
```

```
{
    std::thread t(fun, 10);
    std::cout << "main thread\n";
    t.join();    // main thread waits for t to finish
    return 0;
}
```

❖ Lambda Function

```
int main()
{
    std::thread t([]() {std::cout << "thread function\n";});
    std::cout << "main thread\n";
    t.join();    // main thread waits for t to finish
    return 0;
}
```

or

```
int main()
{
    auto fun = [](int x) {std::cout << x << "thread function\n";};
    std::thread t(fun, x);
    std::cout << "main thread\n";
    t.join();    // main thread waits for t to finish
    return 0;
}
```

❖ Functor (Function Object)

```
class Base
{
public:
    void operator()(int x)
    {
        while(x > 0)
            std::cout << x << "thread\n";
    }
};
```

```
int main()
{
   std::thread t(Base(), 5);
   std::cout << "main thread\n";
   t.join();    // main thread waits for t to finish
   return 0;
}
```

❖ Non-static member function

```
class Base
{
public:
   void run(int x)
   {
      while (x > 0)
         std::cout << x << "thread\n";
   }
};
int main()
{
   Base b;
   std::thread t(&Base::run(), &b, 5);
   std::cout << "main thread\n";
   t.join();    // main thread waits for t to finish
   return 0;
}
```

❖ Static member function

```
class Base
{
public:
   static void run(int x)
   {
      while(x > 0)
         std::cout << x << "thread\n";
   }
};
int main()
```

```
{
   //Note here we don't have to pass the address of object as we are calling static function. No need to create object of base.
   std::thread t(&Base::run(), 5);
   std::cout << "main thread\n";
   t.join();    // main thread waits for t to finish
   return 0;
}
```

- **Carefully Pass Arguments to Threads**

❖ Passing simple arguments to a std::thread in C++11

To Pass arguments to thread's associated callable object or function just pass additional arguments to the std::thread constructor.
By default all arguments are copied into the internal storage of new thread.

```
void threadCallback(int x, std::string str)
{
   std::cout << "Passed Number = " << x << std::endl;
   std::cout << "Passed String = " << str << std::endl;
}
int main()
{
   int x = 10;
   std::string str = "Sample String";
   std::thread threadObj(threadCallback, x, str);
   threadObj.join();
   return 0;
}
```

❖ How not to pass arguments to threads in C++11

Don't pass addresses of variables from local stack to thread's callback function. Because it might be possible that local variable in Thread 1 goes out of scope but Thread 2 is still trying to access it

through it's address.
In such scenario accessing invalid address can cause unexpected behaviour.

```
void newThreadCallback(int* p)
{
   std::chrono::milliseconds dura(1000);
   std::this_thread::sleep_for(dura);
   *p = 19; // Till this point i may be goes out of scope.
}

int main()
{
   int i = 10;
   std::thread t(newThreadCallback, &i);
   t.detach();
   return 0;
}
```

Similarly be *careful while passing pointer to memory located on heap to thread*. Because it might be possible that some thread deletes that memory before new thread tries to access it.
In such scenario accessing invalid address can cause unexpected behaviour.

```
void newThreadCallback(int* p)
{
   std::chrono::milliseconds dura(1000);
   std::this_thread::sleep_for(dura);
   *p = 19; // At this point p might gets deleted.
}

int main()
{
   int* p = new int(10);
   std::thread t(newThreadCallback, p);
   t.detach();
   delete p;
   p = nullptr;
```

```
    return 0;
}
```

❖ How to pass references to std::thread in C++11

```
void threadCallback(int& x)
{
    x++;
}
int main()
{
    int x = 9;
    std::thread threadObj(threadCallback, std::ref(x));
    threadObj.join();
    return 0;
}
```

O/P: 10

❖ Why is a parameter to a C++ thread that is passed by a reference treated as a pass by value, unless it's defined explicitly as a std::ref?

Constructor of thread look like this,

constructor:

```
template< class Function, class... Args >
explicit thread(Function&& f, Args&&... args);
```

thread constructor accepts arguments as a rvalue reference.

This is to avoid race conditions when accessing the parameter, especially during its destruction.Imagine that the two threads need to have access to the same shared parameter object. Who is responsible for deleting that object? In general, either of the threads may exit before the other, especially in all kind of error or exceptional scenarios. If one certain thread deletes the object, then the other might be left with a stale reference. Special synchronizing steps between the threads would be needed to avoid troubles. Alas, for such synchronization the two threads must use some common data, and the only sensible way to pass (references to the) common data is the parameter object itself, which makes all this approach pretty convoluted and fragile. For example,

if the other thread is busy with some lengthy task, then the object-deleting thread might decide to exit anyway after a timeout, leaving the other thread with a blatant stale reference.

The cleanest and simplest solution is to use a dynamically allocated shared parameter object which is accessed via a *std::shared_ptr*. But guess what, *std::shared_ptr objects are passed by value, exactly what the thread mechanism is doing!* So the mechanism does the exact needed thing with the most straightforward approaches and discourages the use of error-prone fragile approaches.

Thread Synchronization

Threads should not be allowed to access the same object at one time in order prevent corruption of the object and therefore of the program.

- **Critical Sections**

To prevent more than one thread from accessing the same object simultaneously, the object can be given a CCriticalSection member variable. Threads which wish to use the object can call the Lock() method of the objects' CCriticalSection before doing so and call Unlock when they are finished. If any other thread tries to call the Lock() function while the object is being used, then that call will not return until the first thread calls Unlock(), effectively stalling the second thread until the object is free to be used.

- **Race Condition**

- Race condition is situation where two or more threads / process happened to change the common data at the same time.

- If there is race condition we have to protect it and protected section is called **critical section.**

Mutexes

> Mutex is short for *'mutually exclusive'*. A mutex object can be used in a similar way to a CCriticalSection object but can be ***shared across processes***. For cross-process sharing you must give the Mutex a name. All processes which share the object must refer to the mutex by the same name.

Mutexes may also provide deletion safety, where the process holding the mutex cannot be accidentally deleted.

Alternately, if the process holding the mutex is deleted (perhaps due to an unrecoverable error), the mutex can be automatically released.

A mutex may be recursive: a process is allowed to lock it multiple times without causing a deadlock.

❖ C++ 11 Mutex

Defined in header <mutex>

There are two important methods on a mutex: lock() and unlock(). As their names indicate, the first one enable a thread to obtain the lock and the second releases the lock.

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

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

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

};

This wrapper works well in most of the cases, but when an exception occurs in the decrement method, you have a big problem. Indeed, if an exception occurs, the unlock() function is not called and so the lock is left in a blocked state.

❖ Automatic management of locks

It exists a good solution to avoid forgetting to release the lock: **std::lock_guard**. This class is a simple smart manager for a lock. When the std::lock_guard is created, it automatically calls lock() on the mutex. When the guard gets destructed, it also releases the lock. You can use it like this:

A **mutex** is a synchronization object. You acquire a lock on a mutex at the beginning of a section of code, and release it at the end, in order to ensure that no other thread is accessing the same data at the same time. A mutex typically has a lifetime equal to that of the data it is protecting, and that one mutex is accessed by multiple threads.

A **lock object** is an object that encapsulates that lock. When the object is constructed it acquires the lock on the mutex. When it is destructed the lock is released.

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

   void increment() {
      std::lock_guard<std::mutex> guard(mutex);
      counter.increment();
   }

   void decrement() {
      std::lock_guard<std::mutex> guard(mutex);
      counter.decrement();
   }
       };
```

❖ std::mutex::lock

```
void lock();
```

Lock mutex

The calling thread locks the `mutex`, blocking if necessary:

- If the mutex isn't currently *locked* by any thread, the calling thread *locks* it .
- If the mutex is currently locked by another thread, execution of the calling thread is blocked until *unlocked* by the other thread
- If the mutex is currently locked by the same thread calling this function, it produces a *deadlock* (with *undefined behavior*).

❖ Return value

(none) Its **void**

e.g.

```
 mtx.lock();
std::cout << "thread #" << id << '\n';
mtx.unlock();
```

❖ std::mutex::try_lock

`bool try_lock();`

`Attempts to lock the mutex,  Its similar to lock function just return type is diff.`

`TOPIC:`

`std::mutex::try_lock() On Mutex In C++11 Threading`

`0. try_lock()` **`Tries to lock the mutex. Returns immediately.`** `On successful lock acquisition returns true otherwise returns false.`

`1. If try_lock() is not able to lock mutex, then it` **`doesn't get blocked`** `that's why it is called non-blocking. 2. If try_lock is called again by the same thread which owns the mutex, the behavior is undefined. It is a dead lock situation with undefined behavior. (if you want to be able to lock the same mutex by same thread more than one time the go for recursive_mutex)`

❖ Return value

`true` if the function succeeds in *locking* the `mutex` for the thread.

`false` otherwise.

E.g.

```
if (mtx.try_lock()) {   // only increase if currently not locked:
   ++counter;
   mtx.unlock();
}
```

❖ std::lock_guard

- `It is very light weight wrapper for owning mutex on scoped basis.`
  - `It acquires mutex lock the moment you create the object of lock_guard.`
  - `It automatically removes the lock while goes out of scope.`
  - `You can not explicitly unlock the lock_guard.`
  - `You can not copy lock_guard.`
  - `Bit faster tha other kind of locks.`

`e.g:`

```
void decrement() {
        std::lock_guard<std::mutex> guard(mutex);
        counter.decrement();
}
```

❖ std::recursive_mutex

Defined in header `<mutex>`

`class recursive_mutex;` (since C++11)

The `recursive_mutex` class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

Link : [Recursive Mutex In C++ Threading - YouTube]

NOTES:

0. It is same as mutex but, **Same thread can lock one mutex multiple times** using recursive_mutex.

1. If thread T1 first call lock/try_lock on recursive mutex m1, then m1 is locked by T1, now as T1 is running in recursion T1 can call lock/try_lock any number of times there is no issue.

2. But if T1 have aquired 10 times lock/try_lock on mutex m1 then thread T1 will have to unlock it 10 times otherwise no other thread will be able to lock mutex m1. It means recursive_mutex keeps count how many times it was locked so that many times it should be unlocked.

3. How many time we can lock recursive_mutex is not defined but when that number reaches and if we were calling lock() it will return std::system_error OR if we were calling try_lock() then it will return false.

BOTTOM LINE:

0. It is similar to mutex but have extra facility that it can be locked multiple time by same thread.

1. If we can avoid recursive_mutex then we should because it brings overhead to the system.

2. It can be used in loops and recursive functions.

- A calling thread *owns* a recursive_mutex for a period of time that starts when it successfully calls either lock or try_lock. During this period, the thread may make additional calls to lock or try_lock. The period of ownership ends when the thread makes a matching number of calls to unlock.
- When a thread owns a recursive_mutex, all other threads will block (for calls to lock) or receive a false return value (for try_lock) if they attempt to claim ownership of the recursive_mutex.
- The maximum number of times that a recursive_mutex may be locked is unspecified, but after that number is reached, calls to lock will throw std::system_error and calls to try_lock will return false.

eg.1

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

std::recursive_mutex m1;
int buffer = 0;
void recursion(char c, int loopFor)
{
   if (loopFor < 0)
      return;
   m1.lock();
   cout << "Thread ID:" << c << " " << buffer++ << endl;
   recursion(c, --loopFor);
   m1.unlock();
}
int main()
```

```
{
   tread t1(recursion, '1', 10);
   tread t2(recursion, '2', 10);
   t1.join();
   t2.join();
   return 0;
}
```

e.g.2

```
std::recursive_mutex  g_num_mutex;
for (int i = 0; i < 3; ++i) {
   g_num_mutex.lock();
   ++g_num;
   std::cout << "lock acquired" << std::endl;
   g_num_mutex.unlock();
}
```

O/P:

lock acquired

lock acquired

lock acquired

❖ std::timed_mutex

Defined in header `<mutex>`

`class timed_mutex;` (since C++11)

The timed_mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

NOTES:

0. std::timed_mutex is blocked till timeout_time or the lock is aquired and returns true if success otherwise false.

1. Member Function:

a. lock

b. try_lock

c. try_lock_for ---\ These two functions makes it different from mutex.

d. try_lock_until ---/

e. unlock

EXAMPLE: try_lock_for(); Waits until specified timeout_duration has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

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

int myAmt = 0;
std::timed_mutex m;

void increment(int i)
{
   if (m.try_lock_for(std::chrono::seconds(1))) // We can use here m.try_lock_until also
   {
      ++myAmt;
      std::this_thread::sleep_for(std::chrono::seconds(2));
      cout << "Thread ID:" << i << " Entered " << endl;
      m.unlock();
   }
   else
      cout << "Thread" << i << "couldnt enter" << endl;
}
int main()
{
   thread t1(increment, 1);
   thread t2(increment, 2);
   t1.join();
```

```
    t2.join();
    cout << myAmt << endl;
    return 0;
}
```

O/P:

Thread 2 couldnt enter

Thread 1 enter

1

If we change code in increment function like

```
if (m.try_lock_for(std::chrono::seconds(2)))
{
    ++myAmt;
    std::this_thread::sleep_for(std::chrono::seconds(1));
    cout << "Thread ID:" << i << " Entered " << endl;
    m.unlock();
}
```

Then O/P:

Thread 1 enter

Thread 2 enter

2

Thread 2

the ability to attempt to claim ownership of a timed_mutex with a timeout via the try_lock_for() and try_lock_until() methods.

❖ std::timed_mutex::try_lock_for

Tries to lock the mutex. Blocks until specified timeout_duration has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

If timeout_duration is less or equal timeout_duration.zero(), the function behaves like try_lock().

```
using Ms = std::chrono::milliseconds;
```

```
std::timed_mutex mutex;
for (int i = 0; i < 3; ++i) {
    if (mutex.try_lock_for(Ms(100))) {
        stream << "success ";
        mutex.unlock();
    }
    else {
        stream << "failed ";
    }
}
```

❖ std::timed_mutex::try_lock_until

Tries to lock the mutex. Blocks until specified timeout_time has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

If timeout_time has already passed, this function behaves like try_lock().

```
std::timed_mutex test_mutex;

void f()
{
    auto now = std::chrono::steady_clock::now();
    test_mutex.try_lock_until(now + std::chrono::seconds(10));
}

int main()
{
    std::lock_guard<std::timed_mutex> l(test_mutex);
    std::thread t(f);
    t.join();
}
```

❖ std::shared_mutex

Defined in header <shared_mutex>

class shared_mutex; (since C++17)

The shared_mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a shared_mutex has two levels of access:

- **shared** - several threads can share ownership of the same mutex.
- **exclusive** - only one thread can own the mutex.

Shared mutexes are usually used in situations when multiple readers can access the same resource at the same time without causing data races, but only one writer can do so.

E.g:

```
// Multiple threads/readers can read the counter's value at the same time.
unsigned int get() const {
  std::shared_lock<std::shared_mutex> lock(mutex_); // Or mutex_.lock_shared();
  return value_;                                    // mutex_.unlock_shared();
}

// Only one thread/writer can increment/write the counter's value.
void increment() {
  std::unique_lock<std::shared_mutex> lock(mutex_);
  value_++;
}

// Only one thread/writer can reset/write the counter's value.
void reset() {
  std::unique_lock<std::shared_mutex> lock(mutex_);
  value_ = 0;
}
```

❖ std::shared_lock

Defined in header <shared_mutex>

```
template< class Mutex >
class shared_lock;
```

(since C++14)

The class shared_lock is a general-purpose shared mutex ownership wrapper allowing deferred locking, timed locking and transfer of lock ownership. Locking a shared_lock locks the associated shared mutex in shared mode (to lock it in exclusive mode, std::unique_lock can be used)

The shared_lock class is movable, but not copyable.

❖ std::unique_lock

```
The class unique_lock is a general-purpose mutex ownership wrapper allowing
deferred locking, time-constrained attempts at locking, recursive locking,
transfer of lock ownership, and use with condition variables.
1. The class unique_lock is a mutex ownership wrapper.
2. It Allows:
a. Can Have Different Locking Strategies
b. time-constrained attempts at locking (try_lock_for, try_lock_until)
c. recursive locking d. transfer of lock ownership (move not copy)
e. condition variables.

Locking Strategies
1. defer_lock:do not acquire ownership of the mutex.
2. try_to_lock: try to acquire ownership of the mutex without blocking.
3. adopt_lock:assume the calling thread already has ownership of the mutex.


e.g :
```

```
// 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);
```

❖ Diff Between lock_guard and unique_lock

- `lock_guard and unique_lock are pretty much the same thing; lock_guard is a restricted version with a limited interface.`
- `A lock_guard always holds a lock from its construction to its destruction.`
- `A unique_lock` ***`can be created without immediately locking, can unlock at any point`*** `in its existence, and can transfer ownership of the lock from one instance to another.`
- `So you always use lock guard, unless you need the capabilities of unique_lock.`
- `You can not explicitly unlock the lock_guard or You can not copy lock_guard.`
- **`lock_gaurd is bit faster than unique lock.`** `So if we don't want extra functionality that unique_lock offers, we should always use lock_guard.`

`e.g:`

1. std::unique_lock<std::mutex> lk(mutex, std::defer_lock); //lock defered
   std::lock(lk); // Lock aquired

```
2. std::unique_lock<std::mutex> lk(mutex); // Lock aquired
```

❖ std::recursive_timed_mutex

Defined in header `<mutex>`

class recursive_timed_mutex; (since C++11)

The recursive_timed_mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

In a manner similar to std::recursive_mutex, recursive_timed_mutex provides exclusive, recursive ownership semantics. In addition, recursive_timed_mutex provides the ability to attempt to claim ownership of arecursive_timed_mutex with a timeout via the `try_lock_for` and `try_lock_until` methods.

❖ std::shared_timed_mutex

Defined in header `<shared_mutex>`

`class shared_timed_mutex;` (since C++14)

The shared_timed_mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a shared_timed_mutex has two levels of access:

- ***shared*** - several threads can share ownership of the same mutex.
- ***exclusive*** - only one thread can own the mutex.

Shared mutexes are usually used in situations when multiple readers can access the same resource at the same time without causing data races, but only one writer can do so.

Example to understand shared & exclusive mutex:

Think of a lockable object as a *blackboard* (lockable) in a class room containing a *teacher*(writer) and many *students* (readers).
While a teacher is writing something (exclusive lock) on the board:

1. Nobody can read it, because it's still being written, and she's blocking your view => *If an object is exclusively locked, shared locks cannot be obtained*.
2. Other teachers won't come up and start writing either, or the board becomes unreadable, and confuses students => *If an object is exclusively locked, other exclusive locks cannot be obtained*.

When the students are reading (shared locks) what is on the board:

1. They all can read what is on it, together => *Multiple shared locks can co-exist*.

The teacher waits for them to finish reading before she clears the board to write more => *If one or more shared locks already exist, exclusive locks cannot be obtained*.

❖ std::latch

- Introduced in C++ 20.

- std::latch is a single-use counter that allows threads to wait for the count to reach zero.
- Create the latch with a non-zero count
- One or more threads decrease the count
- Other threads may wait for the latch to be signalled.
- When the count reaches zero it is permanently signalled and all waiting threads are woken

.

❖ std::lock() In C++11

It is used to lock multiple mutex at the same time.
IMPORTANT: styntax: std::lock(m1, m2, m3, m4);
1. All arguments are locked via a sequence of calls to lock(), try_lock(), or unlock() on each argument.
2. Order of locking is not defined (it will try to lock provided mutex in any order and ensure that there is no dead lock).
3. It is a blocking call.

[Example:0] – **No deadlock**.

Thread 1	Thread 2
std::lock(m1,m2);	std::lock(m1,m2);

[Example:1] -- **No deadlock**.

Thread 1	Thread 2
std::lock(m1, m2);	std::lock(m2, m1);

[Example:2] -- **No deadlock**.

Thread 1	Thread 2
std::lock(m1, m2, m3, m4);	std::lock(m3, m4); std::lock(m1, m2);

[Example:3] -- Yes, the below can **deadlock**.

Thread 1	Thread 2
std::lock(m1,m2);	std::lock(m3,m4);
std::lock(m3,m4);	std::lock(m1,m2);

e.g.

```
// the call to std::lock() locks the two mutexes
std::lock(lhs.m, rhs.m);
```

```
// two std::lock_guard instances are constructed one for each mutex.
std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
```

- The **std::adopt_lock** parameter is supplied in addition to the mutex to indicate to the **std::lock_guard** objects that the mutexes are already locked, and they should just adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor.

- Although **std::lock** can help us to avoid deadlock in those cases where we need to acquire two or more locks together, it doesn't help if they're acquired separately. In that case we have to rely on our discipline as developers to ensure we don't get deadlock.
- Deadlock can happen just by having each thread call **join()** on the **std::thread** object for the other.

❖ std::scoped_lock() In C++17

- The class scoped_lock is a mutex wrapper that provides a deadlock-avoiding convenient RAII-style mechanism for owning one or more mutexes for the duration of a scoped block.
- When a ***scoped_lock*** object is created, it attempts to take ownership of the mutexes it is given. When control leaves the scope in which the scoped_lock object was created, the scoped_lock is destructed and the mutexes are released, in reverse order. If several mutexes are given, ***deadlock avoidance*** algorithm is used as if by ***std::lock***.
- The scoped_lock class is ***non-copyable***.

"std::scoped_lock" should be used instead of "std::lock_guard"

std::scoped_lock basically provides the same feature as *std::lock_guard*, but is more generic: It can lock several mutexes at the same time, with a deadlock prevention mechanism.

The equivalent code to perform simultaneous locking with *std::lock_guard* is significantly more complex. Therefore, it is simpler to use *std::scoped_lock* all the time, even when locking only one mutex

```
void f2(std::mutex& m1, std::mutex& m2) {
   std::lock(m1, m2);
   std::lock_guard<std::mutex> lock1{ m1, std::adopt_lock }; // Noncompliant
   std::lock_guard<std::mutex> lock2{ m2, std::adopt_lock }; // Noncompliant
// Do some work
}
```

```
void f2(std::mutex& m1, std::mutex& m2) {
std::scoped_lock lock{ m1, m2 }; // Compliant, and more simple
// Do some work
}
```

❖ std::call_once

The std::call_once function, introduced in C++11, ensures a callable is called exactly one time, in a thread safe manner.

e.g:

1.

```
std::once_flag flag;
void do_something() {
   std::call_once(flag, []() {std::cout << "Happens once" << std::endl; });

   std::cout << "Happens every time" << std::endl;
}
```

2.

```
once_flag single_flag;
Singleton* Singleton::getInstance()
{
   call_once(single_flag, [] {
         m_single = new Singleton();
      });
   return m_single;
}
```

Call_once will make sure the callable will be called only once ad only by one thread.

❖ once_flag

The class std::once_flag is a helper structure for std::call_once.

An object of type std::once_flag that is passed to multiple calls to std::call_once allows those calls to coordinate with each other such that only one of the calls will actually run to completion.

std::once_flag is neither copyable nor movable.

Condition Variable

❖ std::condition_variable

The thread that intends to modify the variable has to

1.acquire a std::mutex (typically via std::lock_guard)

2.perform the modification while the lock is held

3.execute notify_one or notify_all on the std::condition_variable (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

Any thread that intends to wait on std::condition_variable has to

1.acquire a std::unique_lock<std::mutex>, on the same mutex as used to protect the shared variable

2.execute wait, wait_for, or wait_until. The wait operations atomically release the mutex and suspend the execution of the thread.

3.When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

std::condition_variable works only with **std::unique_lock<std::mutex>**; this restriction allows for maximal efficiency on some platforms. std::condition_variable_any provides a condition variable that works with anyBasicLockable object, such as std::shared_lock.

Condition variables permit concurrent invocation of
the wait, wait_for, wait_until, notify_one and notify_allmember functions.

The class std::condition_variable is a StandardLayoutType. It is
not CopyConstructible, MoveConstructible,CopyAssignable, MoveAssignable.

e.g:

1.

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

```
std::condition_variable cv;
std::mutex cv_m;
int n = 0;
const int LIM = 15;
void print_odds()
{
   while (n < LIM)
   {
      std::unique_lock<std::mutex> lk(cv_m);
      cv.wait(lk, []() {return n % 2 == 1; });
      std::cout << "odd: " << n++ << '\n';
      cv.notify_one();
   }
}

void print_evens()
{
   while (n < LIM)
   {
      std::unique_lock<std::mutex> lk(cv_m);
      cv.wait(lk, []() {return n % 2 == 0; });
      std::cout << "even: " << n++ << '\n';
      cv.notify_one();
   }
}

int main()
{
   std::thread t1(print_odds);
   std::thread t2(print_evens);
   t1.join();
   t2.join();
}
```

2.

```
#include <iostream>
```

```
#include <condition_variable>
#include <thread>

std::condition_variable cv;
std::mutex cv_m;
int balanceAmt = 0;
//const int LIM = 15;
void withdrawAmt(int amt)
{
   std::unique_lock<std::mutex> ul(cv_m);
   cv.wait(ul, [] {return balanceAmt > 0; });
      if (balanceAmt >= amt)
      {
         balanceAmt = balanceAmt - amt;
         cout << amt << "withdraw" << endl;
      }
      else
      {
         cout << amt << "Not sufficient amount to withdraw" << endl;
      }
   cout << "Balance Amt : " << balanceAmt;
}

void deositeAmt(int amt)
{
   std::unique_lock<std::mutex> ul(cv_m);
   balanceAmt = balanceAmt + amt;
   cout << amt << "deposited" << endl;
   cv.notify_one();
}

int main()
{
   std::thread t1(withdrawAmt, 200);
   std::thread t2(deositeAmt, 500);
   t1.join();
   t2.join();
```

```
    return 0;
}
```

❖ std::condition_variable_any

The condition_variable_any class is a generalization of std::condition_variable. Whereas std::condition_variable works only on std::unique_lock<std::mutex>, ***condition_variable_any can operate on any lock that meets the BasicLockable requirements.***

If the lock is std::unique_lock, std::condition_variable may provide better performance.

The difference is the parameter to the wait() functions. All the wait functions in std::condition_variable take a lock parameter of type std::unique_lock<std::mutex>&, whereas the wait functions for std::condition_variable_any are all templates, and take a lock parameter of type Lockable&, where Lockable is a template parameter.

This means that std::condition_variable_any can work with user-defined mutex and lock types, and with things like boost::shared_lock --- anything that has lock() and unlock() member functions.

Semaphores

Semaphores are used when a limited supply of resources are being requested by multiple threads. The collection of resources as a whole receives requests from threads and allocates one of its resources per request. If all of the resources are being used, then the thread must wait until one becomes available. For instance, four threads may each wish to make a connection when only two connections can be open at one time.

When number of resources a Semaphore protects is greater than 1, it is called a Counting Semaphore. When it controls one resource, it is called a Boolean Semaphore. A boolean semaphore is equivalent to a mutex.

❖ Binary Semaphores

The semaphore discussed previously is called a **counting sempahore**. Another kind of semaphore is the **binary semaphore**; This is exactly like a counting semaphore except for the following:

1.the semaphore value is restricted to 0 and 1.

2.P succeeds only when the semaphore value is 1.

3.V does not change the semaphore value when it is 1. (Thus successive Vs are lost.)

❖ std::counting_semaphore, std::binary_semaphore

- A semaphore represents a number of available “slots”.
- If you acquire a slot on the semaphore then the count is decreased until you release the slot. Attempting to acquire a slot when the count is zero will either block or fail.
- A thread may release a slot without acquiring one and vice versa.
- A binary semaphore has 2 states: 1 slot free or no slots free. It can be used as a mutex.
- As well as blocking sem.acquire(), there are also sem.try_acquire(), sem.try_acquire_for() and sem.try_acquire_until() functions that fail instead of blocking

e.g:

```
std::counting_semaphore slots(5);
void func()
{
slots.acquire();
do_stuff();
// at most 5 threads can be here
slots.release();
}
```

❖ Binary Semaphores Vrs Mutex:

There is an ambiguity between binary semaphore and mutex. We might have come across that a mutex is binary semaphore. But they are not! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore. Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex). Semaphore is signaling mechanism (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

- Mutex is costly operation due to protection protocols associated with it.
- A Mutex controls access to a single shared resource. It provides operations to acquire() access to that resource and release() it when done.

- A Semaphore controls access to a shared pool of resources. It provides operations to Wait()until one of the resources in the pool becomes available, and Signal() when it is given back to the pool.

General Questions:

1. **Can a thread acquire more than one lock (Mutex)?**

Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

2. **Can a mutex be locked more than once?**

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a recursive mutex can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

3. **What happens if a non-recursive mutex is locked more than once.**

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks.

4. **Are binary semaphore and mutex same?**

No. We suggest to treat them separately, as it is explained signalling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with mutex. We will cover these in later article.

5. **Is it necessary that a thread must block always when resource is not available?**

Not necessary. If the design is sure 'what has to be done when resource is not available', the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

For example POSIX pthread_mutex_trylock() API. When mutex is not available the function returns immediately whereas the API pthread_mutex_lock() blocks the thread till resource is available.

Advance multithreading techniques

❖ std::future , std::promise and Returning values from Thread

A std::future object can be used with asych, std::packaged_task and std::promise. In this article will mainly focus on using std::future with std::promise object.

Many times we encounter a situation where we want a thread to return a result.

Suppose in our application we created a thread that will compress a given folder and we want this thread to return the new zip file name and its size in result.
Now to do this we have 2 ways,
1.) Old Way : Share data among threads using pointer
Pass a pointer to the new thread and this thread will set the data in it. Till then in main thread keep on waiting using a condition variable. When new thread sets the data and signals the condition variable, then main thread will wake up and fetch the data from that pointer.
To do a simple thing we used a condition variable, a mutex and a pointer i.e. 3 items to catch a returned value.
Now suppose we want this thread to return 3 different values at different point of time then problem will become more complex. Could there be a simple solution for returning the value from threads.
The answer is yes using std::future, checkout next solution for it.

2.) C++11 Way : Using std::future and std::promise
std::future is a class template and its object stores the future value.
Now what the hell is this future value.
[showads ad=inside_post]
Actually a **std::future** object internally stores a value that will be assigned in future and it also provides a mechanism to access that value i.e. using get() member function. But if somebody tries to access this associated value of future through get() function before it is available, then get() function will block till value is not available.
std::promise is also a class template and its object promises to set the value in future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.
A **std::promise** object shares data with its associated **std::future** object.
Lets see step by step,
Create a std::promise object in Thread1.

```
std::promise<int> promiseObj;
```

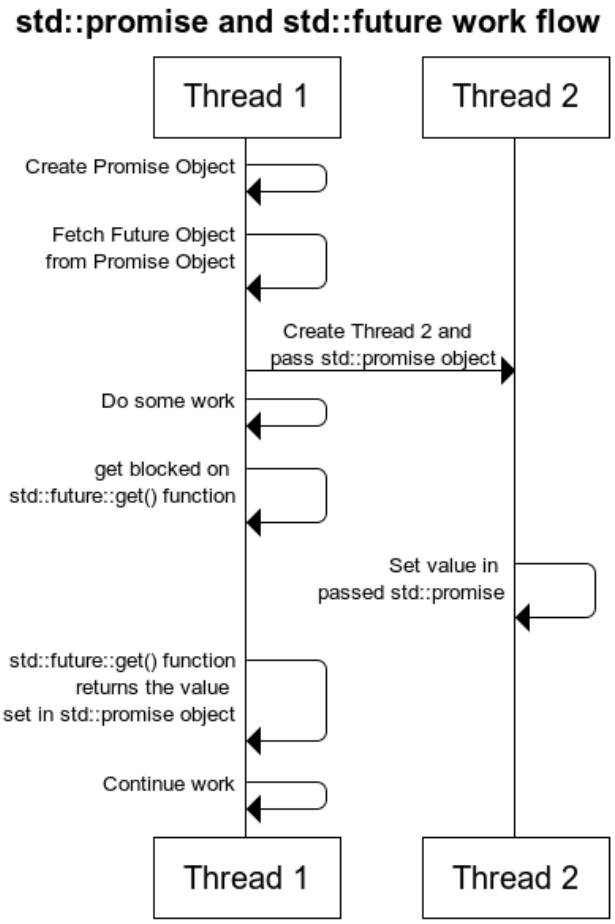
As of now this promise object doesn't have any associated value. But it gives a promise that somebody will surely set the value in it and
once its set then you can get that value through associated std::future object.
But now suppose Thread 1 created this promise object and passed it to Thread 2 object. Now how Thread 1 can know that when Thread 2 is going to set the value in this promise object?
The answer is using std::future object.

Every std::promise object has an associated std::future object, through which others can fetch the value set by promise.
So, Thread 1 will create the std::promise object and then fetch the std::future object from it before passing the std""promise object to thread 2 i.e.
std::future<int> futureObj = promiseObj.get_future();

Now Thread 1 will pass the promiseObj to Thread 2.
Then Thread 1 will fetch the value set by Thread 2 in std::promise through std::future's get function,
int val = futureObj.get();

But if value is not yet set by thread 2 then this call will get blocked until thread 2 sets the value in promise object i.e.
promiseObj.set_value(45);

Check out complete flow in following Diagram,




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

```
#include <future>
void initiazer(std::promise<int>* promObj)
{
   std::cout << "Inside Thread" << std::endl;    promObj->set_value(35);
}
int main()
{
   std::promise<int> promiseObj;
   std::future<int> futureObj = promiseObj.get_future();
   std::thread th(initiazer, &promiseObj);
   std::cout << futureObj.get() << std::endl;
   th.join();
   return 0;
}
```

If std::promise object is destroyed before setting the value the calling get() function on associated std::future object will throw exception.
A part from this, if you want your thread to return multiple values at different point of time then just pass multiple std::promise objects in thread and fetch multiple return values from thier associated multiple std::future objects.
Basically std::promise is sent to the called thread and once the value is ready we set that value in promise object, now at calling thread side we get that value using std::future object which was created using std::promise object before sending it to the called thread. And this is how we receive value from one thread to another in synchronization.

❖ std::shared_future

A shared_future object behaves like a future object, except that it can be copied, and that more than one shared_future can *share* ownership over their end of a *shared state*. They also allow the value in the *shared state* to be retrieved multiple times once *ready*.

The class template std::shared_future provides a mechanism to access the result of asynchronous operations, similar to std::future, except that multiple threads are allowed to wait for the same shared state. Unlike std::future, which is only moveable (so only one instance can refer to any particular asynchronous result), std::shared_future is copyable and multiple shared future objects may refer to the same shared state.
e.g:

```
void threadFunction1(shared_future<void> fu)
{
```

```
    std::cout << "Thread 1 start" << std::endl;
    /* This will wait till promise value is not set*/
    while (fu.wait_for(std::chrono::milliseconds(1)) == std::future_status::timeout)
    {
        std::cout << "Doing Some Work" << std::endl;
    }
}
void threadFunction2(shared_future<void> fu)
{
    std::cout << "Thread 2 start" << std::endl;
    /* This will wait till promise value is not set*/
    while (fu.wait_for(std::chrono::milliseconds(1)) == std::future_status::timeout)
    {
        std::cout << "Doing Some Work" << std::endl;
    }
}


int main()
{

    std::promise<void> pr;
    std::future<void> fu1 = pr.get_future();
    shared_future<void> sf = fu1.share();

    thread t1(threadFunction1, sf);
    thread t2(threadFunction2, sf);
    while (true)
    {
        char ch = getchar();
        if (ch == '0')
        {
            pr.set_value();
            t1.detach();
            t2.detach();
            break;
            exit(0);
```

```
        }
    }
    if(t1.joinable())
        t1.join();
    if (t2.joinable())
        t2.join();

    return 0;
}
```

❖ std::async

what is std::async()

std::async() is a function template that accepts a callback(i.e. function or function object) as an argument and potentially executes them asynchronously.

```
template <class Fn, class... Args>
future<typename result_of<Fn(Args...)>::type> async(launch policy, Fn&& fn, Args&&... args);
```

std::async returns a **std::future<T>,** that stores the value returned by function object executed by **std::async()**. Arguments expected by function can be passed to std::async() as arguments after the function pointer argument.

First argument in std::async is launch policy, it control the asynchronous behaviour of std::async.

We can create std::async with 3 different launch policies i.e.

std::launch::async

It guarantees the asynchronous behaviour i.e. passed function will be executed in seperate thread.

std::launch::deferred

Non asynchronous behaviour i.e. Function will be called when other thread will call get() on future to access the shared state.

std::launch::async | std::launch::deferred

Its the default behaviour. With this launch policy it can run asynchronously or not depending on the load on system. But we have no control over it.

If we do not specify an launch policy. Its behaviour will be similar to **std::launch::async | std::launch::deferred**.

We are going to use std::launch::async launch policy in this article.

We can pass any callback in std::async i.e.

- Function Pointer
- Function Object

- Lambda Function

Need of std::async()

Suppose we have to fetch some data (string) from DB and some from files in file-system. Then I need to merge both the strings and print.
In a single thread we will do like this,

```
using namespace std::chrono;
std::string fetchDataFromDB(std::string recvdData)
{
    // Make sure that function takes 5 seconds to complete
    std::this_thread::sleep_for(seconds(5));
    //Do stuff like creating DB Connection and fetching Data
    return "DB_" + recvdData;
}
std::string fetchDataFromFile(std::string recvdData)
{
    // Make sure that function takes 5 seconds to complete
    std::this_thread::sleep_for(seconds(5));
    //Do stuff like fetching Data File
    return "File_" + recvdData;
}
int main()
{
    // Get Start Time
    system_clock::time_point start = system_clock::now();
    //Fetch Data from DB
    std::string dbData = fetchDataFromDB("Data");
    //Fetch Data from File
    std::string fileData = fetchDataFromFile("Data");
    // Get End Time
    auto end = system_clock::now();
    auto diff = duration_cast <std::chrono::seconds> (end - start).count();
    std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;
    //Combine The Data
    std::string data = dbData + " :: " + fileData;
    //Printing the combined Data
    std::cout << "Data = " << data << std::endl;
    return 0;
```

}

Output:

Total Time Taken = 10 Seconds

Data = DB_Data::File_Data

As both the functions **fetchDataFromDB()** & **fetchDataFromFile()** takes 5 seconds each and are running in a single thread so, total time consumed will be 10 seconds.

Now as fetching data from DB and file are independent of each other and also time consuming. So, we can run them in parallel.

One way to do is create a new thread pass a promise as an argument to thread function and fetch data from associated std::future object in calling thread.

The other easy way is using std::async.

Calling std::async with function pointer as callback

Now let's modify the above code and call fetchDataFromDB() asynchronously using std::async() i.e.

```
std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");
// Do Some Stuff
//Fetch Data from DB
// Will block till data is available in future<std::string> object.
std::string dbData = resultFromDB.get();
```

std::async() does following things,

- It automatically creates a thread (Or picks from internal thread pool) and a promise object for us.
- Then passes the std::promise object to thread function and returns the associated std::future object.
- When our passed argument function exits then its value will be set in this promise object, so eventually return value will be available in std::future object.

Now change the above example and use std::async to read data from DB asyncronously i.e.

```
int main()
{
    // Get Start Time
    system_clock::time_point start = system_clock::now();
    std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");
    //Fetch Data from File
    std::string fileData = fetchDataFromFile("Data");
```

```
    //Fetch Data from DB
    // Will block till data is available in future<std::string> object.
    std::string dbData = resultFromDB.get();
    // Get End Time
    auto end = system_clock::now();
    auto diff = duration_cast <std::chrono::seconds> (end - start).count();
    std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;
    //Combine The Data
    std::string data = dbData + " :: " + fileData;
    //Printing the combined Data
    std::cout << "Data = " << data << std::endl;
    return 0;
}
```

Now it will take 5 seconds only.

```
Total Time Taken = 5 Seconds
Data = DB_Data::File_Data
```

Calling std::async with Function Object as callback

```
/** Function Object */
struct DataFetcher
{
    std::string operator()(std::string recvdData)
    {
        // Make sure that function takes 5 seconds to complete
        std::this_thread::sleep_for(seconds(5));
        //Do stuff like fetching Data File
        return "File_" + recvdData;
    }
};
//Calling std::async with function object
std::future<std::string> fileResult = std::async(DataFetcher(), "Data");
```

Calling std::async with Lambda function as callback

```
//Calling std::async with lambda function
std::future<std::string> resultFromDB = std::async([](std::string recvdData) {
   std::this_thread::sleep_for(seconds(5));
   //Do stuff like creating DB Connection and fetching Data
   return "DB_" + recvdData;
   }, "Data");
```

Other key points about multithreading

❖ Sleep VS Wait In Threading

SLEEP :

1. 1. “I’m done with my timeslice, and please don’t give me another one for at least n milliseconds.” The OS doesn’t even try to schedule the sleeping thread until requested time has passed. 1. It will keep the lock and sleep.
2. 2. Sleep is directly to thread, it is a thread function.

WAIT :

1. “I’m done with my timeslice. Don’t give me another timeslice until someone calls notify().” As with sleep(), the OS won’t even try to schedule your task unless someone calls notify() (or one of a few other wakeup scenarios occurs).
2. It releases the lock and wait.
3. Wait is on condition variable, it is like there is a condition variable in a thread and wait is applied to that CV but it ends up puting thread in waiting state.

❖ Are The Arguments Passed To A C++11 Thread's Constructor Pass By Vale Or Pass By Reference?

Thread function arguments are always pass by value, i.e., they are always copied into the internal storage for threads. Any changes made by the thread to the arguments passed does not affect the original arguments.

```
void ChangeCurrentMissileTarget(string& targetCity)
```

```
{
    targetCity = "Metropolis";
    cout << " Changing The Target City To " << targetCity << endl;
}

int main()
{
    string targetCity = "Star City";
    thread t1(ChangeCurrentMissileTarget, targetCity);
    t1.join();
    cout << "Current Target City is " << targetCity << endl;
    return 0;

}
```

o/p:

No matcing function if we make string argument as string targetCity then output will be,

Changing The Target City To Metropolis
Current Target City is Star City
Note that the "targetCity" variable is not modified.

So how we can pass parameter as reference, we need to create thread like this,

```
thread t1(ChangeCurrentMissileTarget, std::ref(targetCity));
```

❖ Transfer Ownership Of C++11 Threads

Yes. We can transfter ownership of threads to others thread.

std::thread object owns a resource, where the resource is a current thread of execution. You can call std::move to move the ownership of the underlying resource from one std::thread object to another.

The question is – why would you want to do that? Here's a scenario:You want to write a function that creates a thread but does not want to wait for it to finish. Instead it wants to pass the thread to another function which will wait for the thread to finish and execute some action once the execution is done.

```
void FireHTTPGet()
```

```
{
    std::this_thread::sleep_for(std::chrono::milliseconds(5000));
    cout << "Finished Executing HTTP Get" << endl;
}

void ProcessHTTPResult(thread t1)
{
    t1.join();
    cout << "HTTP Get Thread Finished Executing - Processing Result Data!" << endl;
}

int main()
{
    thread t11(FireHTTPGet);
    thread t12(ProcessHTTPResult, std::move(t11));
    //Do bunch of other processing without waiting for t11 to finish - instead now we've shouldered
off the
    // responsibility of monitoring t11 thread to t12.
    //Finally wait for t12 to finish
    t12.join();
    return 0;
}
```

❖ Thread Local Storage

A thread_local object comes into existence when a thread starts and is destroyed when the thread ends. Each thread has its own instance of a thread-Local object.

```
thread_local int globalVar = 0;
mutex mu;

void PrettyPrint(int valueToPrint)
{
    lock_guard<mutex> lock(mu);
    cout << "Value of globalVar in thread " << this_thread::get_id() << " is " << globalVar << endl;
}
```

```
void thread_Local_Test_Func(int newVal)
{
   globalVar = newVal;
   PrettyPrint(globalVar);
}

int main()
{
   globalVar = 1;
   thread t1(thread_Local_Test_Func, 5);
   thread t2(thread_Local_Test_Func, 20);
   t1.join();
   t2.join();
   cout << "Value of globalVar in MAIN thread is " << globalVar << endl;

   return 0;
}
```

o/p:

Value of globalVar in thread 17852 is 5
Value of globalVar in thread 29792 is 20
Value of globalVar in MAIN thread is 1

Output will be if globalVar was not declared thread_local ?
Value of globalVar in thread 27200 is 5
Value of globalVar in thread 31312 is 20
Value of globalVar in MAIN thread is 20

❖ How Can You Retrieve Results From A Thread?

Answer :

- Passing reference to a result variable to the thread in which the thread stores the results
- Store the result inside a class memeber variable of a function object which can be retrieved once the thread has finished executing.

❖ Thread hardware_concurrency() function in C++

Thread::hardware_concurrency is an in-built function in C++ std::thread. It is an **observer function** which means it observes a state and then returns the corresponding output. This function returns **the number of concurrent threads** supported by the available hardware implementation. This value might not always be accurate.

Parameters: This function does not accept any parameters.

Return Value: It returns a non-negative integer denoting the **number of concurrent threads** supported by the system. If the value is either not computable or not well defined it returns 0.

e.g:

```
int main()
{
   unsigned int con_threads;

   // calculating number of concurrent threads
   // supported in the hardware implementation
   con_threads = thread::hardware_concurrency();

   cout << "Number of concurrent threads supported are: "
      << con_threads << endl;

   return 0;
}
```

- ## Deadlock avoidance

- While locking 2 or more mutexex, ***lock the them in the same order***.
- **Use std::lock** which is a function that can **lock two or more mutexes at once without risk of deadlock**.

- ## C++11 : How to Stop or Terminate a Thread

C++11 does not provides a direct method to stop a running thread and that's because that thread might have some resources to release or close before exit i.e.

- What if a thread has acquired a lock and we kill that thread suddenly, then who's gonna release that lock ?
- What if a thread has opened a file to write the text and we stopped that thread, then who's gonna close that file ?
- What if thread has allocated memory on heap and before it could delete that memory, we stopped the thread. Then who's gonna prevent that memory leak.

Therefore there is no direct function to close the thread. But we can notify the thread to exit and we can implement out thread in a such a way that after some interval or at some checkpoints it should

check, if I am requested to exit or not. If yes then it should exit gracefully , by releasing all the resources.

```
#include <thread>
#include <iostream>
#include <assert.h>
#include <chrono>
#include <future>
void threadFunction(std::future<void> futureObj)
{
   std::cout << "Thread Start" << std::endl;
   while (futureObj.wait_for(std::chrono::milliseconds(1)) == std::future_status::timeout)
   {
      std::cout << "Doing Some Work" << std::endl;
      std::this_thread::sleep_for(std::chrono::milliseconds(1000));
   }
   std::cout << "Thread End" << std::endl;
}
int main()
{
   // Create a std::promise object
   std::promise<void> exitSignal;
   //Fetch std::future object associated with promise
   std::future<void> futureObj = exitSignal.get_future();
   // Starting Thread & move the future object in lambda function by reference
   std::thread th(&threadFunction, std::move(futureObj));
   //Wait for 10 sec
   std::this_thread::sleep_for(std::chrono::seconds(10));
   std::cout << "Asking Thread to Stop" << std::endl;
   //Set the value in promise
   exitSignal.set_value();
   //Wait for thread to join
   th.join();
   std::cout << "Exiting Main Function" << std::endl;
   return 0;
}
```

Move constructors

A move constructor of class T is a non-template constructor whose first parameter is T&&, const T&&, volatile T&&, or const volatile T&&, and either there are no other parameters, or the rest of the parameters all have default values.

❖ Syntax

class_name (class_name &&)	(1) (since C++11)
class_name (class_name &&) = default;	(2) (since C++11)
class_name (class_name &&) = delete;	(3) (since C++11)

❖ Explanation

1. Typical declaration of a move constructor.
2. Forcing a move constructor to be generated by the compiler.
3. Avoiding implicit move constructor.

The move constructor is called whenever an object is initialized from xvalue of the same type, which includes

- initialization, T a = std::move(b); or T a(std::move(b));, where b is of type T;
- function argument passing: f(std::move(a));, where a is of type T and f is void f(T t);
- function return: return a; inside a function such as T f(), where a is of type T which has a move constructor.

Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc), rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state. For example, moving from a std::string or from astd::vector may result in the argument being left empty. However, this behaviour should not be relied upon.

e.g:

1.

```
struct A
{
   std::string s;
   A() : s("test") { }
   A(const A& o) : s(o.s) { std::cout << "move failed!\n"; }
   A(A&& o) noexcept : s(std::move(o.s)) { } // Move Constructor
};
```

2. A typical move constructor definition would look like this:

```
//C++11
MemoryPage(MemoryPage&& other) : size(0), buf(nullptr)
{
   // pilfer other’s resource
   size = other.size;
   buf = other.buf;
   // reset other
   other.size = 0;
   other.buf = nullptr;
}
```

Resource Acquisition Is Initialization (RAII)

Resource Acquisition Is Initialization, or just RAII, is a programming idiom that is used to manage the life cycle of a resource automatically by binding it to the lifetime of an object.

We acquire resources in constructor and release them in the destructor.

e.g.

```
class RAII
{
   int* ptr;
public:
   RAII(int* ptr_val) : ptr(ptr_val) {}
   ~RAII() { delete ptr; }
};
```

What is resource?

- Something that must be acquired before use.
- In limited supply.

e.g.Memory, mutex, files, sockets.

RAII classes in C++:

std::string, std::unique_lock, std::unique_ptr, std::shared_ptr

Resource Acquisition Is Initialization (RAII) provides a class that does the **join()** in its destructor,

```
class thread_RAII {
thread& t;
public:
   thread_RAII(thread& th) :t(th) { cout << "thread_RAII constructor\n"; }
   ~thread_RAII() {
      if (t.joinable())
      {
         cout << "if joinable(), then t.join()\n";
         t.join();
      }
      cout << "thread_RAII destructor\n";
   }
private:
```

```
    // copy constructor
    thread_RAII(const thread_RAII& thr) ;
    // copy-assignment operator
    thread_RAII& operator=(const thread_RAII& thr);
};

int main()
{
    thread t(task);
    thread_RAII raii(t);
    //Dont need to join thread
}
```