
# Synchronization

## Synchronization in C++

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

# License Information

# Introduction

For condition variables, see Section u06s06

- C++11 introduced synchronization primitives
  - Mutexes and condition variables
    - A semaphore could be implemented using a mutex and a condition variables
- C++14 and C++17 extended the support
  - For example, introduced recursive_mutex, shared_mutex, shared_lock, etc.
- C++20 introduced semaphores
  - Binary semaphores in binary_semaphore
  - Counting semaphores in counting_semaphore

C++20: Introduced semaphores, which are synchronization primitives that can be used to control access to a resource by multiple threads. binary_semaphore is a semaphore with only two states (locked/unlocked), and counting_semaphore allows a specified number of threads to access a resource.

# Mutexes in C++

For more operations see the reference documentation

- The complete library now defines several classes
  - For mutexes

| Type | Meaning |
| --- | --- |
| std::mutex | Mutual exclusion. Protect a shared data from being accessed by multiple threads. The semantic is exclusive and non-recursive. |
| std::recursive_mutex | As the a mutex but offer recursive ownership semantics. A recursive function can acquire the mutex more than once. |
| std::shared_mutex | As a mutex but allows a shared or an exclusive locks. |
| std::timed_mutex | As a mutex but with the possibility to use timeout during locking. |

Exclusive: Only one thread can lock the mutex at a time.
Non-recursive: The same thread cannot lock the mutex multiple times without unlocking it first.

```
std::mutex mtx;
mtx.lock();
// Critical section: code that accesses shared resources
mtx.unlock();
```

Similar to std::mutex but allows the same thread to lock the mutex multiple times.

Recursive: The same thread can lock the mutex multiple times without causing a deadlock.
Useful in scenarios where a function that locks a mutex might be called recursively.

```
std::recursive_mutex rec_mtx;
rec_mtx.lock();
// Critical section: code that accesses shared resources
rec_mtx.lock(); // Locking again within the same thread
// More critical section code
rec_mtx.unlock();
rec_mtx.unlock();
```

Allows multiple threads to hold a shared (read) lock or one thread to hold an exclusive (write) lock.

Similar to std::mutex but allows specifying a timeout for locking

Timeout: Provides functions to attempt to lock the mutex with a timeout, either for a specified duration or until a specified time point.

# Mutexes in C++

For more operations see the reference documentation

- Member functions are common (mutexes & locks)
  - Some of them (e.g., timed locks) are not always available

| Type | Meaning |
|---|---|
| lock | Locks (i.e., takes ownership of) the associated mutex. Blocks the calling thread until it can lock the mutex. |
| try_lock | Tries to lock (i.e., takes ownership of) the associated mutex without blocking Attempts to lock the mutex without blocking. Returns immediately with a success or failure status. |
| try_lock_for | Attempts to lock (i.e., takes ownership of) the associated mutex, returns if the mutex has been unavailable for the specified time duration. Attempts to lock the mutex, blocking for a specified duration. If the mutex is not available within that time, it returns with a failure status. |
| try_lock_until | Tries to lock (i.e., takes ownership of) the associated mutex, returns if the mutex has been unavailable until specified time point has been reached. Attempts to lock the mutex, blocking until a specified time point. If the mutex is not available by that time, it returns with a failure status. |
| unlock | Unlocks (i.e., releases ownership of) the associated mutex. Releases the lock on the mutex, making it available for other threads. |

# Mutex

The same thread must lock and unlock the mutex

- Mutex offers exclusive, non-recursive ownership semantics
- They implemented in the class std::mutex
  - A calling thread owns a mutex from the time that it successfully calls either lock or try_lock until it calls unlock
  - When a thread owns a mutex, all other threads will block (for calls to lock) or receive a false return value (for try_lock) if they attempt to claim the mutex
  - A calling thread must not own the mutex prior to calling lock or try_lock

# Mutex

- The behavior of a program is undefined if
  - A mutex is destroyed while still owned by any threads
  - A thread terminates while owning a mutex
- The class std::mutex is **neither** copyable nor movable

```
std::mutex mtx;
mtx.lock();
// Destroying mtx here would lead to undefined behavior
mtx.unlock();
```

```
std::mutex mtx;
std::thread t([&mtx] {
    mtx.lock();
    // Thread terminates without unlocking the mutex
});
t.join();
// Undefined behavior as the mutex is still locked
```

```
std::mutex mtx1;
std::mutex mtx2 = std::move(mtx1); // Error: std::mutex is non-movable
```

```
std::mutex mtx1;
std::mutex mtx2 = mtx1; // Error: std::mutex is non-copyable
```

# Example

Revisitation of the example in Section u05s04

```
std::mutex m;

void safe_print (int i) {
  m.lock ();
  std::cout << i;
  m.unlock ();
}
```

This slide provides an example of using std::mutex to synchronize access to the std::cout stream, ensuring that the output is not interleaved when multiple threads print to the console.

A std::mutex object m is declared to protect the critical section.

The safe_print function takes an integer i as an argument, locks the mutex m, prints the integer to the console, and then unlocks the mutex.

```
std::vector<std::thread> threadPool;

for (int i = 1; i <= 9; ++i) {
  threadPool.emplace_back([i] { safe_print(i); });
}

for (auto& t : threadPool) {
  t.join();
}
```

A vector of std::thread objects threadPool is created to hold the threads.

A loop iterates from 1 to 9, creating a new thread for each iteration that calls the safe_print function with the current value of i.

The new thread is added to the threadPool using emplace_back.

A loop iterates over each thread in the threadPool and calls join on each thread to wait for its completion.

Digits 1 to 9 are printed (unordered)

The digits 1 to 9 are printed to the console. The order of the digits is not guaranteed because the threads run concurrently, but the use of the mutex ensures that each digit is printed atomically (without interleaving).

# Recursive Mutex

A recursive mutex is a type of mutex that allows the same thread to lock it multiple times without causing a deadlock.

- A recursive mutex is a regular mutex that additionally allows a thread that holds the mutex to lock it again
  - Useful for **recursive** functions or functions that call each other and use the same mutex

Recursive mutexes are particularly useful in scenarios where a function that locks a mutex might call itself recursively or call another function that also locks the same mutex.

- Recursive mutexes are implemented in the class std::recursive_mutex
  - The unlock function must still be called once for each lock

Each call to lock must be matched with a corresponding call to unlock. If a thread locks a recursive mutex multiple times, it must unlock it the same number of times before the mutex is actually released.

Class:
Recursive mutexes are implemented in the C++ standard library as std::recursive_mutex.

# Example

Function f1 calls f2 (but f2 can be called directly)

```
std::recursive_mutex rm;
```

A std::recursive_mutex object rm is declared to protect the critical sections in both functions.

```
void f1() {
  rm.lock();
  std::cout << "foo\n";
  f2();
  rm.unlock();
}
```

The function f1 locks the recursive mutex rm, prints "foo" to the console, calls the function f2, and then unlocks the mutex.

Function f1 must protect IO and then calls function f2

```
void f2() {
  rm.lock();
  std::cout << "bar\n";
  rm.unlock();
}
```

The function f2 also locks the recursive mutex rm, prints "bar" to the console, and then unlocks the mutex.

Function f2 must also lock and unlock IO because can be called directly (not through f1)

A standard mutex will block f2 forever

Function f1 calls f2:
The function f1 calls f2, and both functions need to lock the same mutex to protect their critical sections.
Function f1 must protect IO:
The function f1 locks the mutex to protect the IO operation (printing "foo") and then calls f2.
Function f2 must also lock and unlock IO:
The function f2 must lock and unlock the mutex because it can be called directly by other code, not just through f1.
Standard Mutex Issue:
If a standard (non-recursive) mutex were used, f2 would block forever when called by f1 because f1 already holds the lock. The recursive mutex allows f2 to lock the mutex even though f1 already holds it.

# Shared Mutex

- Introduced in C++17
  - Are implemented in the class std::shared_mutex
- A shared mutex is a mutex that allows shared and exclusive locks A shared mutex is a type of mutex that allows multiple threads to hold a shared (read) lock simultaneously or one thread to hold an exclusive (write) lock.
  - An **exclusive** lock can be obtained by a single thread An exclusive lock can be obtained by a single thread at a time. This lock type is used when a thread needs to write to the shared resource.
    - Are granted by the member functions **lock** and **unlock**
  - A **shared** can be obtained by more than one thread A shared lock can be obtained by multiple threads simultaneously. This lock type is used when threads need to read from the shared resource.
    - Shared locks are granted by the member functions **lock_shared** and **unlock_shared**

# Shared Mutex

This slide provides additional details on the usage of shared mutexes, including the "try" versions of locking functions and their common use cases:

The "try" versions of locking functions attempt to acquire the lock without blocking. They return true if the lock was successfully acquired and false otherwise.

- It is also possible to use the “try” versions during locking and returns true or false
  - Functions **try_lock** try to get an exclusive lock
  - Function **try_lock_shared** try to get a shared lock
- Shared mutexes are mostly used to implement reader and writer locks

  Shared mutexes are commonly used to implement reader-writer locks, which allow multiple readers or a single writer to access a shared resource.
  - Readers use shared locks

    Use shared locks to read from the shared resource concurrently.
  - Writers use exclusive locks

    Use exclusive locks to write to the shared resource, ensuring exclusive access.

In POSIX these mutexes are called Reader-Writer Locks!

# Example

Reader and Writer Protocol

```
int value = 0;
std::shared_mutex sm;

std::vector<std::thread> tv;

for (int i = 0; i < 5; ++i)
  tv.emplace_back([&] {
    sm.lock_shared();
    ... use value ...
    sm.unlock_shared();
  })

for (int i = 0; i < 5; ++i)
  tv.emplace_back([&] {
    sm.lock();
    ++value;
    sm.unlock();
  })
```

A vector of std::thread objects tv is created to hold the reader and writer threads.

Run **reader threads** that can **share** the critical section

loop iterates 5 times, creating a new reader thread in each iteration. Each reader thread locks the shared mutex sm in shared mode, accesses the shared resource value, and then unlocks the shared mutex.

Reader threads can share the critical section, meaning multiple reader threads can hold the shared lock simultaneously.

Run **writer threads** that must acquire an **exclusive** lock on the critical section

Another loop iterates 5 times, creating a new writer thread in each iteration. Each writer thread locks the shared mutex sm in exclusive mode, modifies the shared resource value, and then unlocks the shared mutex.

Writer threads must acquire an exclusive lock on the critical section, meaning only one writer thread can hold the exclusive lock at a time, and no reader threads can hold the shared lock while the exclusive lock is held.

# Timed mutex

- The library std::timed_mutex, similarly to std::mutex, offers exclusive, non-recursive ownership semantics
- In addition, std::timed_mutex provides the ability to attempt to claim ownership of a std::timed_mutex with a timeout via the member functions
  - try_lock_for
  - try_lock_until

In addition to the basic locking and unlocking functions, std::timed_mutex provides the ability to attempt to claim ownership of the mutex with a timeout. This means that a thread can try to lock the mutex but will give up if it cannot acquire the lock within a specified time period.

Attempts to lock the mutex, waiting for a specified duration. Returns true if the lock was acquired, false otherwise.

```
std::timed_mutex tm;
    if (tm.try_lock_for(std::chrono::seconds(1))) {
      // Successfully locked the mutex within 1 second
      tm.unlock();
    } else {
      // Failed to lock the mutex within 1 second
    }
```

Attempts to lock the mutex, waiting until a specified time point. Returns true if the lock was acquired, false otherwise.

```
std::timed_mutex tm;
    auto timeout = std::chrono::steady_clock::now() + std::chrono::seconds(1);
    if (tm.try_lock_until(timeout)) {
      // Successfully locked the mutex before the timeout
      tm.unlock();
    } else {
      // Failed to lock the mutex before the timeout
    }
```

# RAII wrappers

Mutexes can be thought of as resources that must be acquired and freed with lock and unlock. Thus, they are easily subject to errors. Errors can be difficult to trace back.

- Mutexes can be thought of resources that must be acquired and freed with lock and unlock
- Thus, they are easily subject to errors
  - Errors can be difficult to trace back

This slide highlights the potential issues and errors that can arise when manually managing mutexes using lock and unlock functions:

$T_i$

```
m.lock();
...
m.unlock();
```

$T_j$ is buggy and make problems to the entire application, possibly locking also $T_i$ for ever

Manually managing mutexes with lock and unlock can lead to various errors, such as:
Forgetting to unlock a mutex.
Unlocking a mutex that was never locked.
Double locking a mutex.
Unlocking a mutex before it was locked.

$T_j$

```
// Incorrect usage: unlocking without locking
m.unlock();
...
m.unlock();
```

```
m.unlock();
...
m.unlock();
```

$T_j$

```
// Incorrect usage: double locking
m.lock();
...
m.lock();
```

```
m.lock();
...
m.lock();
```

$T_j$

```
// Incorrect usage: unlocking before locking
```

```
m.unlock();
...
m.lock();
```

# RAII wrappers

For more operations see the reference documentation

This slide introduces the RAII (Resource Acquisition Is Initialization) paradigm and explains how it can be used to manage mutexes more safely and conveniently:

- To avoid these problems, it is possible to use RAII paradigm, i.e., wrappers for mutexes

RAII is a programming idiom where resource acquisition and release are tied to the lifetime of an object. When an object is created, it acquires a resource, and when the object is destroyed, it releases the resource.

Using RAII wrappers for mutexes ensures that mutexes are properly locked and unlocked, reducing the risk of errors and making the code easier to maintain.

| Type | Meaning |
|---|---|
| std::lock_guard | A mutex wrapper that provides a convenient RAII-style mechanism for owning a mutex for the duration of a scoped block. |
| std::unique_lock | A RAII wrapper for std::mutex. A mutex wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables. |
| std::shared_lock | RAII wrapper for std::shared_mutex. A shared mutex wrapper allowing deferred locking, timed locking and transfer of lock ownership. |
| std::scoped_lock | RAII wrapper for owning zero or more mutexes (contestually) with no deadlock. |

A simple RAII wrapper that locks a mutex when it is created and unlocks the mutex when it goes out of scope.

```
std::mutex m;
{
   std::lock_guard<std::mutex> guard(m);
   // Mutex is locked
}
// Mutex is automatically unlocked when guard goes out of scope
```

A more flexible RAII wrapper that supports deferred locking, timed locking, recursive locking, transfer of lock ownership, and use with condition variables

```
std::mutex m;
   std::unique_lock<std::mutex> lock(m);
   // Mutex is locked
   lock.unlock();
   // Mutex is unlocked
   lock.lock();
   // Mutex is locked again
```

An RAII wrapper for std::shared_mutex that supports shared locking, deferred locking, timed locking, and transfer of lock ownership.

An RAII wrapper that can lock multiple mutexes at once, ensuring no deadlock occurs. It locks the mutexes when created and unlocks them when it goes out of scope.

```
std::mutex m1, m2;
  {
    std::scoped_lock lock(m1, m2);
    // Both mutexes are locked
  }
  // Both mutexes are automatically unlocked when lock goes out of scope
```

# RAII wrappers

This slide explains the behavior and features of RAII wrappers for mutexes:

- They call lock in the constructor and unlock in the destructor
  - They are movable to "transfer ownership" of the locked mutex
  - They also have the member functions **lock** and **unlock** to manually control the mutex

RAII Wrappers:
Constructor and Destructor:
RAII wrappers for mutexes automatically call lock on the mutex when the wrapper object is created (in the constructor) and call unlock when the wrapper object is destroyed (in the destructor).
This ensures that the mutex is properly locked and unlocked, reducing the risk of errors.

2. Movable:
Transfer Ownership:
RAII wrappers are movable, meaning you can transfer ownership of the locked mutex from one wrapper object to another. This is useful in scenarios where you need to pass the locked mutex to another scope or thread.

Manual Control:
Member Functions:
RAII wrappers also provide member functions lock and unlock to manually control the mutex if needed. This allows for more flexible control over the locking and unlocking process.

# Example

Lock guards

- When
  - A lock guard object is created, it attempts to take ownership of the mutex it is given
  - Control leaves the scope, the lock guard is destructed and the mutex is released

```
#include <iostram:
#include <thread>
#include <mutex>

std::mutex m;

void t () {
  const std::lock_guard<std::mutex> lock(m);
  ...
  return;
}
```

Creates a lock guard and acquires m

When a std::lock_guard object is created, it attempts to take ownership of the mutex it is given. This means it locks the mutex in its constructor.

When control leaves the scope in which the std::lock_guard object was created, the std::lock_guard object is destructed, and the mutex is automatically unlocked in its destructor.

The mutex m is automatically released when the lock gets out of scope

When the function t returns, the std::lock_guard object lock goes out of scope, and the mutex m is automatically released.

# Example

Unique locks with mutex

```
#include <iostram:
#include <thread>
#include <mutex>

std::mutex m;

void f1 () {
    std::unique_lock l{m};
    ...

}
```

Create a lock l on the mutex m
m.lock() is called for immediate locking

When a std::unique_lock object is created with a mutex, it immediately locks the mutex.

Alternative definition

This line creates a std::unique_lock object named l and locks the mutex m.

m.unlock() is called

When the function f1 returns, the std::unique_lock object l goes out of scope, and the mutex m is automatically unlocked.
When the std::unique_lock object goes out of scope, it automatically unlocks the mutex.

```
std::unique_lock<std::mutex> l{m};
```

The std::unique_lock ensures that the mutex is unlocked even if an exception is thrown within the critical section.
Annotation:
This guarantees that the mutex is properly managed and reduces the risk of deadlocks or resource leaks.

It also guarantees unlocking in case an exception is thrown

# Example

Unique locks with shared mutex

When a std::unique_lock object is created with a shared mutex, it immediately locks the shared mutex.

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

std::shared_mutex sm;

void f1 () {
   std::unique_lock l{sm};
   ...
}
```

The example demonstrates a simple function t that locks a mutex m using a std::lock_guard and then performs some operations in the critical section. When the function returns, the std::lock_guard ensures that the mutex is properly unlocked.

Create a lock l on the shared mutex sm
sm.lock() is called for immediate locking

sm.unlock() is called

When the std::unique_lock object goes out of scope, it automatically unlocks the shared mutex.

Alternative definition

```
std::unique_lock<std::shared_mutex> l{sm};
```

It also guarantees unlocking in case an exception is thrown

The std::unique_lock ensures that the shared mutex is unlocked even if an exception is thrown within the critical section.

# Example

Unique locks with deferred locking

This slide provides an example of using std::unique_lock with deferred locking to manage multiple mutexes and avoid deadlocks:

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

std::mutex m1;
std::mutex m2;

void update (int num) {
  std::unique_lock lock1{m1, std::defer_lock};
  std::unique_lock lock2{m2, std::defer_lock};

  std::lock(lock1);
  std::lock(lock2);

  num += (int) rand();

  return;
}
```

Don't actually take the locks on m1 and m2

These lines create std::unique_lock objects lock1 and lock2 for mutexes m1 and m2, respectively, but do not lock the mutexes immediately.
Deferred Locking:
Definition: std::unique_lock can be constructed with the std::defer_lock argument, which means the mutex is not locked immediately upon construction.

The std::lock function can be used to lock multiple std::unique_lock objects at once, ensuring that the locks are acquired without causing a deadlock.
This line locks both lock1 and lock2 in a way that avoids deadlocks by ensuring a consistent locking order.

std::lock(lock1,lock2);
lock both unique locks without deadlock (due to the **order**)

When the std::unique_lock objects go out of scope, they automatically unlock the mutexes they manage.

m1 ad m2 are **automatically** unlocked

# Example

Scoped locks
Part A

This slide explains the importance of maintaining a consistent locking order when using multiple mutexes to avoid deadlocks:

- When threads use multiple mutexes
  - It is easy to cause a deadlock if different threads try to lock the mutexes in a different order
  - The solution is to force a consistent order

When threads need to lock multiple mutexes, it is crucial to lock them in a consistent order to avoid deadlocks.

If different threads lock the mutexes in different orders, it can lead to a deadlock situation where each thread is waiting for a mutex held by another thread.

threadA can get the locks for m1 and m2
threadB gets a lock on m3 …
deadlock

The code example shows two threads, threadA and threadB, that lock three mutexes (m1, m2, and m3) in different orders.

```
std::mutex m1, m2, m3;

void threadA() {
  std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
  std::unique_lock l3{m3}, l2{m2}, l1{m1};
}
```

The order is not consistent with threadB

threadA locks m1, m2, and m3 in that order.

threadB locks m3, m2, and m1 in that order.

This inconsistent locking order can lead to a deadlock where threadA holds m1 and m2 and is waiting for m3, while threadB holds m3 and is waiting for m1.

Consistent Order:
To avoid deadlocks, ensure that all threads lock the mutexes in the same order.
Example:
Both threadA and threadB should lock the mutexes in the same order, such as m1, m2, and m3.

## Example

Scoped locks
Part B

This slide explains the use of std::scoped_lock to manage multiple mutexes and avoid deadlocks:

When threads need to lock multiple mutexes, it can be challenging to guarantee a consistent locking order, which can lead to deadlocks.
Solution:
std::scoped_lock can be used to lock multiple mutexes in a deadlock-free manner.

- When threads use multiple mutexes
  - Sometimes, it is not possible to guarantee a consistent order
  - The class **std::scoped_lock** takes any number of mutexes and locks them using a deadlock-free algorithm

std::scoped_lock takes any number of mutexes and locks them using a deadlock-free algorithm.

This line creates a std::scoped_lock object named l and locks the mutexes m1, m2, and m3.

```
std::mutex m1, m2, m3;

void threadA() {
  std::scoped_lock l{m1, m2, m3};
}
void threadB() {
  std::scoped_lock l{m3, m2, m1};
}
```

It is generally very time-inefficient.
Use with care.

Locks m1, m2, and m3 using std::scoped_lock.

Locks m3, m2, and m1 using std::scoped_lock.

Even though the order of mutexes is different in threadA and threadB, std::scoped_lock ensures that the locks are acquired in a deadlock-free manner.

Using std::scoped_lock can be time-inefficient, so it should be used with care.

# Semaphores in C++

For more operations see the reference documentation

- Semaphores in C++ have been introduced with C++20
  - They contain a counter initialized by the constructor
  - When the counter is zero, **acquire** blocks until the counter is incremented

Semaphores in C++20:
Definition:
Semaphores are synchronization primitives that control access to shared resources.
Counter:
Semaphores contain a counter that is initialized by the constructor.
Acquire and Release:
When the counter is zero, the acquire operation blocks until the counter is incremented (by a release operation).

| Type | Meaning |
|---|---|
| counting_semaphores | Synchronization primitive that can control access to a shared resource. Unlike a mutex a counting semaphore allows more than one concurrent access to the same resource. |
| binary_semaphores | A specialization form of the counting semaphore with the maximum value being 1. May be more efficiently than the default counting semaphore. |

Counting Semaphores:
Definition:
A counting semaphore allows more than one concurrent access to the same resource.
Usage:
Useful for managing a pool of resources where multiple threads can access the resources concurrently.

Definition:
A binary semaphore is a specialization of the counting semaphore with the maximum value being 1.
Usage:
Useful for scenarios where only one thread can access the resource at a time, similar to a mutex but potentially more efficient

# Semaphores in C++

For more operations see the reference documentation

- Member functions are common (mutexes & locks)
  - Unlike mutex semaphores **are not tied to threads**, i.e., acquiring a semaphore can occur on a different thread than releasing the semaphore
  - All operations on semaphore can be performed concurrently and without any relation to specific threads of execution

Unlike mutexes, semaphores are not tied to specific threads. This means that one thread can acquire a semaphore, and a different thread can release it.

All semaphore operations can be performed concurrently without any relation to specific threads of execution.

Increments the internal counter and unblocks any threads waiting to acquire the semaphore.

| Type | Meaning |
|---|---|
| release | Increments the internal counter and unblocks acquirers. |
| acquire | Decrements the internal counter or blocks until it can. |
| try_acquire | Tries to decrement the internal counter without blocking. |
| try_acquire_for | Tries to decrement the internal counter, blocking for up to a duration time. |
| try_acquire_until | Tries to decrement the internal counter, blocking until a point in time. |

# Example

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

Conters are set to 0 (non-signaled state)

The initial count of 0 means that if a thread tries to acquire() the semaphore immediately after this declaration, it will block because the semaphore's count is not greater than 0.
The thread will remain blocked until another thread calls release() on the semaphore, increasing its count to 1.

```
std::binary_semaphore m2t{0}, t2m{0};
```

Two binary semaphores m2t and t2m are initialized with a counter value of 0, indicating a non-signaled state.

m2t = main to thread
t2m = thread to main

The worker thread function tf waits for a signal from the main thread by calling m2t.acquire().
After receiving the signal, it prints a message, sleeps for 3 seconds, and then sends a signal back to the main thread by calling t2m.release().

```
void tf() {
  m2t.acquire();
  std::cout << "[thread] Got signal\n";
  std::this_thread::sleep_for(3s);
  std::cout << "[thread] Send signal\n";
  t2m.release();
}

int main() {
  std::thread thr(tf);
  std::cout << "[main] Send signal\n";
  m2t.release();
  t2m.acquire();
  std::cout << "[main] Got signal\n";
  thrWorker.join();
}
```

Take a look at it you will understand how it works.

Output will be:
main send signal
thread got signal
thread send signal
main got signal

# Conclusions

- As in all other standards, synchronization must be used with care
  - For each call to **lock** or **acquire**, **unlock** or **release** must be called exactly once
  - For mutex function **lock** and **unlock** must be called by the **same** thread
  - Mutexes and semaphores are usually neither copyable nor movable
    - It is not trivial to create dynamic data structure of them but using dynamic memory allocation
  - Pay attention to deadlock
    - Prefere RAII versions when possible