# Multithreading

- **What is multithreading/terminology**
- **Working with std::thread**
- **Why do we need Synchronization**
- **Synchronization examples**
- **Problems with synchronization**
- **Advices for designing concurrent code**
- **Questions**

# Multithreading

- **Thread**
  - Work which can be scheduled to execute on one core
  - A thread is contained inside a process
  - Each thread has its own callstack
  - Threads in the same process share the same resources

- **Process**
  - Used to start a separate program
  - Each process has at least one thread
  - If there is only one thread in a process the program is not multithreaded
  - As an example, you can start *make* then make starts *clang* (*make* starts a new process to run *clang*)

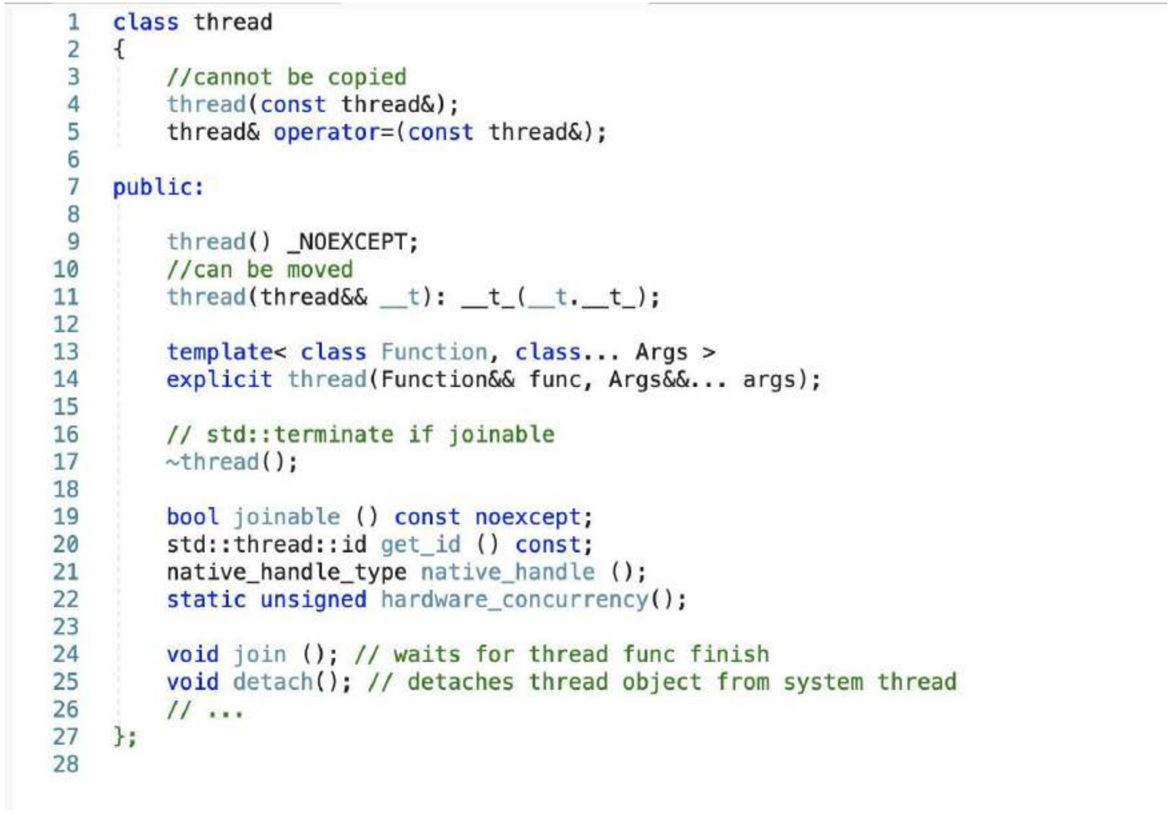
## Thread creation example

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

 void task()
 {
     std::cout << “task" << std::endl;
 }

 int main()
 {
     std::thread t(task);
     t.join();
 }
```

## Background task example

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

void background_task()
{
    while (true)
    {
        std::cout << "background_job" << std::endl;
    }
}

int main()
{
    std::thread background_thread(background_task);

    background_thread.detach();

    // emulate another useful work
    std::this_thread::sleep_for(std::chrono::seconds(2));
}
```

# std::thread



```
class thread
{
    //cannot be copied
    thread(const thread&);
    thread& operator=(const thread&);

public:

    thread() _NOEXCEPT;
    //can be moved
    thread(thread&& __t): __t_(__t.__t_);

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

    // std::terminate if joinable
    ~thread();

    bool joinable () const noexcept;
    std::thread::id get_id () const;
    native_handle_type native_handle ();
    static unsigned hardware_concurrency();

    void join (); // waits for thread func finish
    void detach(); // detaches thread object from system thread
    // ...
};

```

## Problems with data passed to another thread

```
#include <thread>

class some_resource { /* creates an array in heap */ };

void background_task(int& data)
{
    while (true)
    {
        std::cout << data.getValueByIndex(3) << std::endl;
    }
}

void background_task_launcher()
{
    some_resource resource;
    std::thread task([&]()
        {
            background_task(resource);
        });

    task.detach();
}

int main()
{
    background_task_launcher();
}
```

## Make std::threads unjoinable on all paths

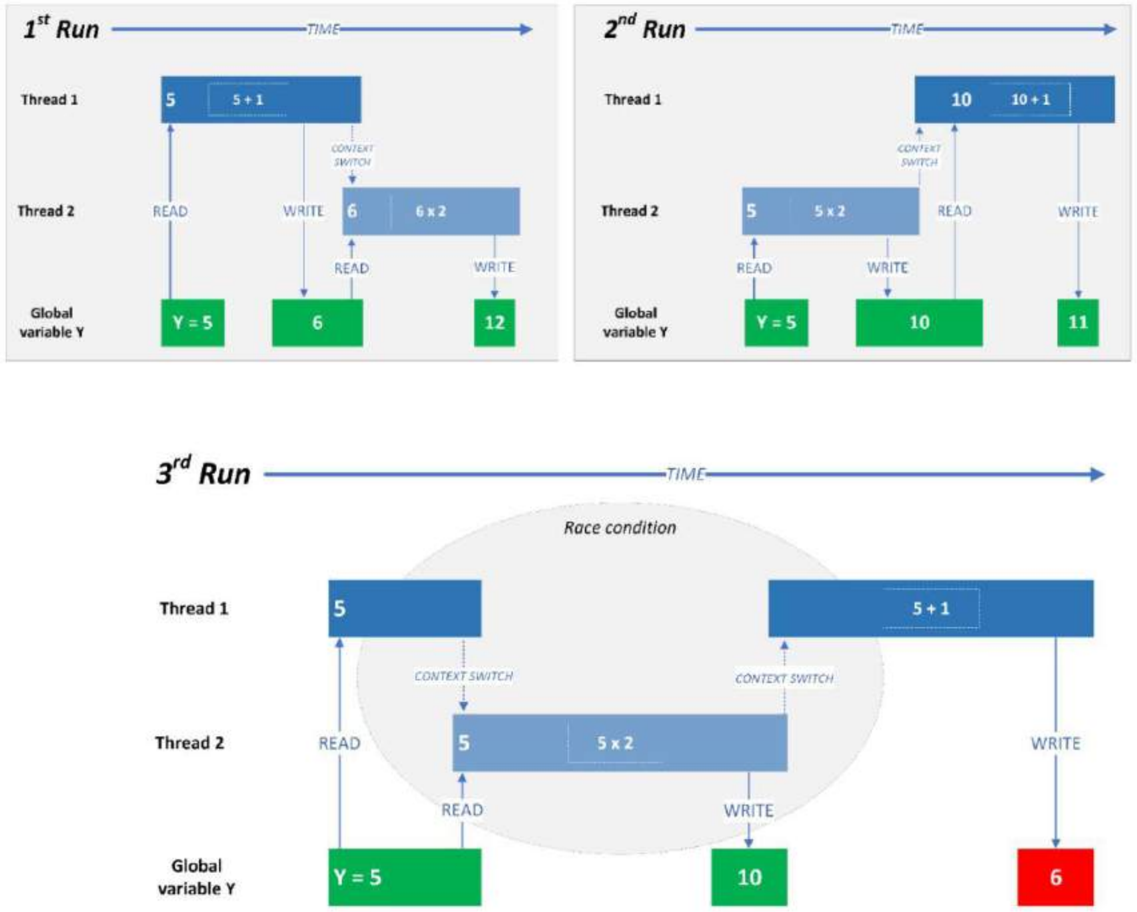
```
#include <thread>

void background_task(int& data)
{
    while (true)
    {
        std::cout << ++data << std::endl;
    }
}

void background_task_launcher()
{
    auto some_local_state = 0;
    std::thread task([&]()
        {
            background_task(some_local_state);
        });

    throw std::runtime_error("oops");

    task.detach();
}

int main()
{
    background_task_launcher();
}
```

# Synchronization

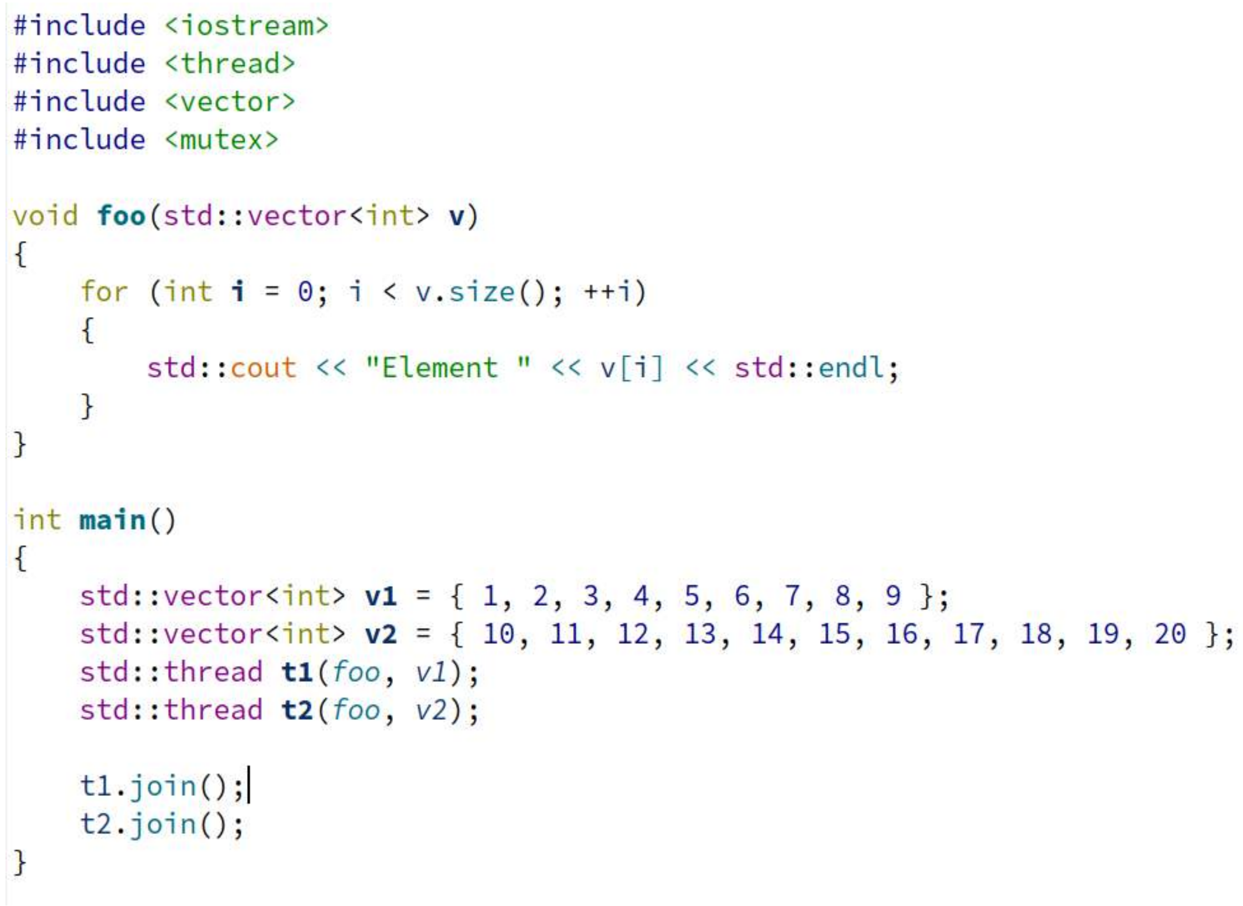
- Race conditions and Data Races
- Concept of invariant
- Problems with sharing data between threads
- Protecting data with mutexes
- C++ Thread support library
- Thread race issues

## Race condition example

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

uint32_t Y = 5;

void foo()
{
    Y += 1;
}

void bar()
{
    Y *= 2;
}

int main()
{
    std::thread t1(foo);
    std::thread t2(bar);

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

    std::cout << Y << std::endl;
}
```


1st Run
TIME
Thread 1
5
5 + 1
CONTEXT SWITCH
Thread 2
READ
WRITE
6
6 x 2
READ
WRITE
Global variable Y
Y = 5
6
12
2nd Run
TIME
Thread 1
10
10 + 1
CONTEXT SWITCH
Thread 2
5
5 x 2
READ
WRITE
READ
WRITE
Global variable Y
Y = 5
10
11
3rd Run
TIME
Race condition
Thread 1
5
5 + 1
CONTEXT SWITCH
CONTEXT SWITCH
Thread 2
READ
5
5 x 2
WRITE
READ
WRITE
Global variable Y
Y = 5
10
6

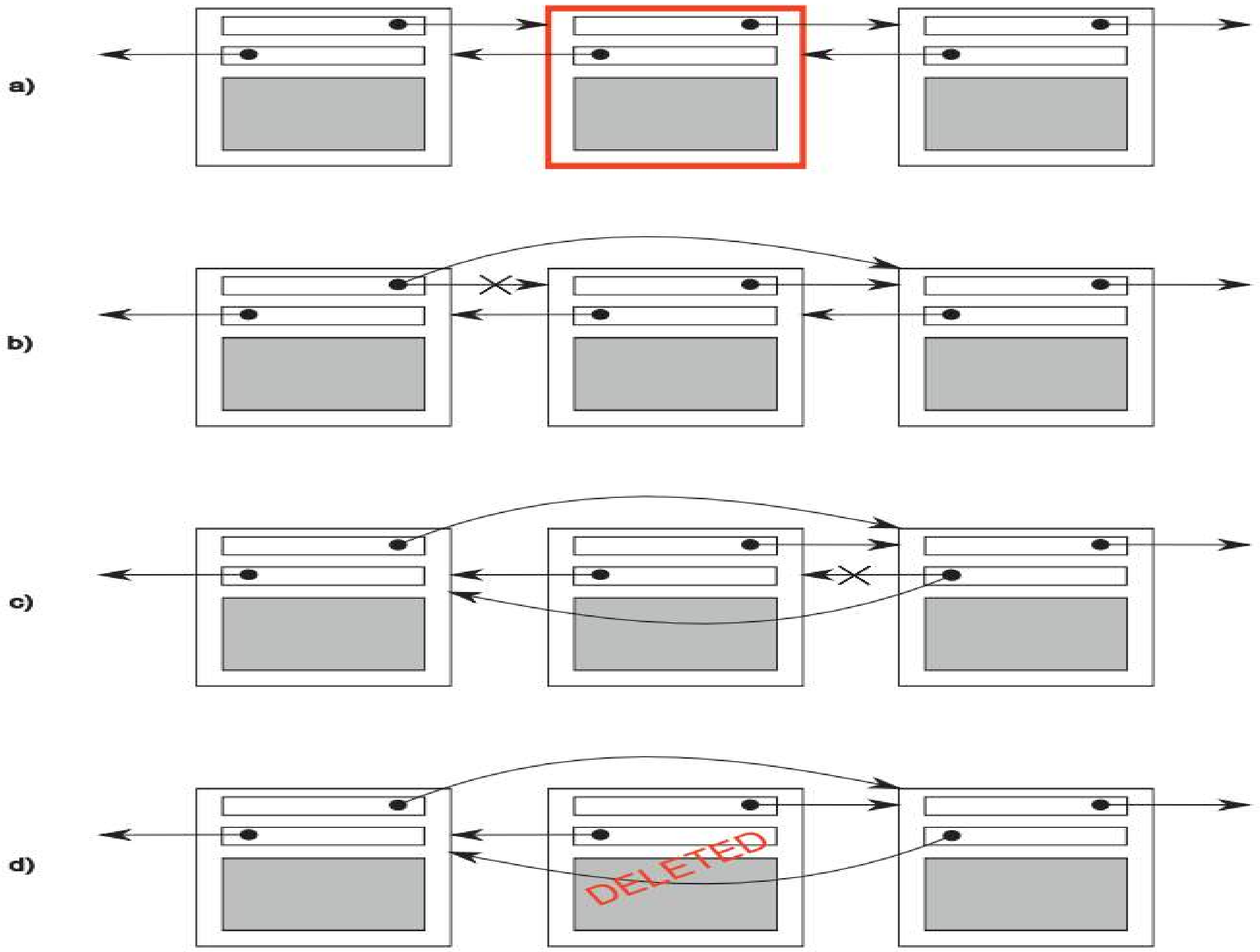
## Multiple threads executing the critical section



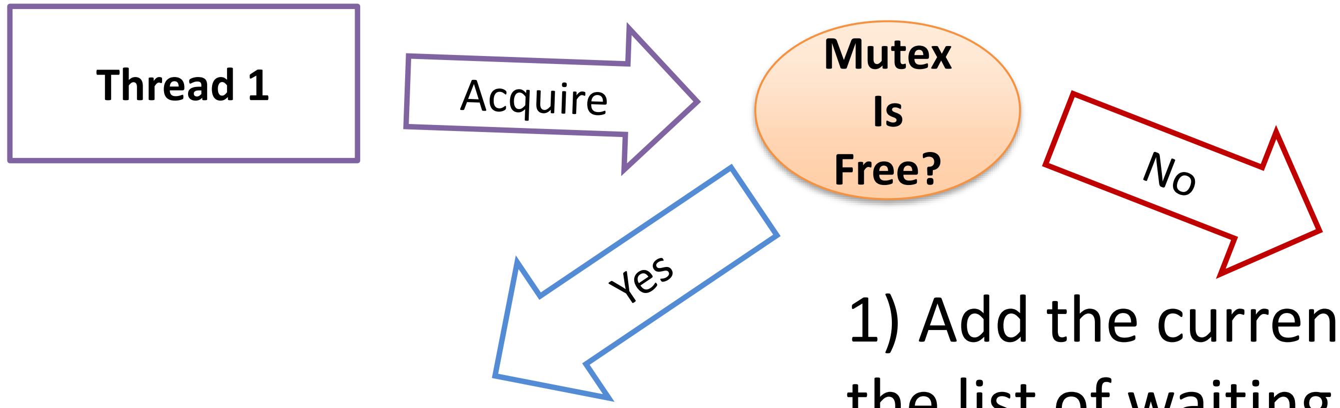
```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

void foo(std::vector<int> v)
{
    for (int i = 0; i < v.size(); ++i)
    {
        std::cout << "Element " << v[i] << std::endl;
    }
}

int main()
{
    std::vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    std::vector<int> v2 = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
    std::thread t1(foo, v1);
    std::thread t2(foo, v2);

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

# Data Race example

```cpp
#include <list>

std::list<int> some_list;

void add_to_list(int new_value)
{
    some_list.push_back(new_value);
}

void remove_from_list(int value)
{
    some_list.remove(value);
}

int get_list_front()
{
    return some_list.front();
}

int get_list_back()
{
    return some_list.back();
}
```


a)
b)
c)
d)
DELETED

# Mutex



1) Mark as "acquired"
2) return

1) Add the current thread to the list of waiting threads of mutex

2) Exclude the thread from planning (DispatcherReadyList)

```
#include <mutex>

std::list<int> some_list;
std::mutex some_mutex;

void add_to_list(int new_value)
{
    std::lock_guard guard(some_mutex);
    some_list.push_back(new_value);
}

void remove_from_list(int value)
{
    std::lock_guard guard(some_mutex);
    some_list.remove(value);
}

int get_list_front()
{
    std::lock_guard guard(some_mutex);
    return some_list.front();
}

int get_list_back()
{
    std::lock_guard guard(some_mutex);
    return some_list.back();
}
```

## C++ Thread support library

- **Mutual exclusion**

  mutex, recurcive_mutex, shared_mutex also each has a timed version
- **Generic mutex management**

  lock_guard, scoped_lock, unique_lock, shared_lock
- **Futures**

promise, packaged_task, future, shared_future, async, launch
- **Call once**

once_flag, call_once
- **Condition variables**

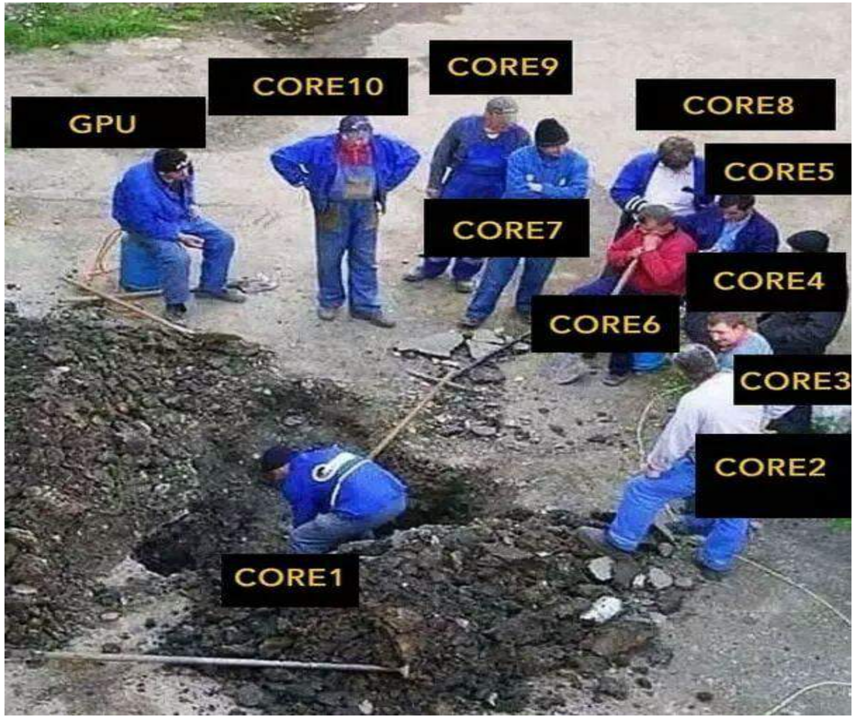
condition_variable.
- **Semaphores**

counting_semaphore, binary_semaphore

# Optimization with std::shared_lock

```
#include <list>
#include <shared_mutex>

std::list<int> some_list;
std::shared_mutex some_mutex;

void add_to_list(int new_value)
{
    std::lock_guard guard(some_mutex);
    some_list.push_back(new_value);
}

void remove_from_list(int value)
{
    std::lock_guard guard(some_mutex);
    some_list.remove(value);
}

int get_list_front()
{
    std::shared_lock guard(some_mutex);
    return some_list.front();
}

int get_list_back()
{
    std::shared_lock guard(some_mutex);
    return some_list.back();
}
```

# Multithreading issues



## Deadlock, livelock

- A **deadlock** is a situation that occurs in OS when any thread enters a waiting state because another waiting thread is holding the demanded resource. Deadlock is a common problem in multi-threading where several threads share a specific type of mutually exclusive resource.

- A **Livelock** is a situation where a request for an exclusive lock is denied repeatedly, as many overlapping shared locks keep on interfering each other. The processes keep on changing their status, which further prevents them from completing the task. This further prevents them from completing the task.

# Simple deadlock example

```cpp
std::mutex resourceX;
std::mutex resourceY;

void thread_A_func()
{
    while (true)
    {
        std::unique_lock lockX(resourceX);

        std::this_thread::yield(); // emulate some work

        std::unique_lock lockY(resourceY);

        std::cout << "thread_A working" << std::endl;
    }
}

void thread_B_func()
{
    while (true)
    {
        std::unique_lock lockY(resourceY);

        std::this_thread::yield(); // emulate some work

        std::unique_lock lockX(resourceX);

        std::cout << "thread_B working" << std::endl;
    }
}
```

# Real life deadlock example

```cpp
class DataHolder
{
public:
    // ...
    DataHolder() = default;

    DataHolder(DataHolder&& other) noexcept
    {
        swap(*this, other);
    }

    DataHolder& operator=(DataHolder&& other) noexcept
    {
        swap(*this, other);
        return *this;
    }

    // ...
private:
    void swap(DataHolder& lhs, DataHolder& rhs) const
    {
        std::lock_guard lhs_lock(lhs.m_useful_data_lock);
        std::lock_guard rhs_lock(rhs.m_useful_data_lock);

        std::swap(lhs.m_useful_data, rhs.m_useful_data);
    }

private:
    std::list<int> m_useful_data;
    std::mutex m_useful_data_lock;
};
```

```cpp
void swap(DataHolder& lhs, DataHolder& rhs) const
{
    std::lock_guard lhs_lock(lhs.m_useful_data_lock);
    std::lock_guard rhs_lock(rhs.m_useful_data_lock);

    std::swap(lhs.m_useful_data, rhs.m_useful_data);
}

DataHolder data_holder_1;
DataHolder data_holder_2;

void thread_A()
{
    data_holder_1 = std::move(data_holder_2);

    // call to swap(data_holder_1, data_holder_2)
}

void thread_B()
{
    data_holder_2 = std::move(data_holder_1);

    // call to swap(data_holder_2, data_holder_1)
}

// The case
// thread_A -> std::lock_guard data_holder_1.lhs_lock
//
// context_switch
//
// thread_B -> std::lock_guard data_holder_2.lhs_lock
//
// the idea that LHS lock in thread_A it's a RHS lock in thread_B and vice versa
```

## Real life deadlock solution

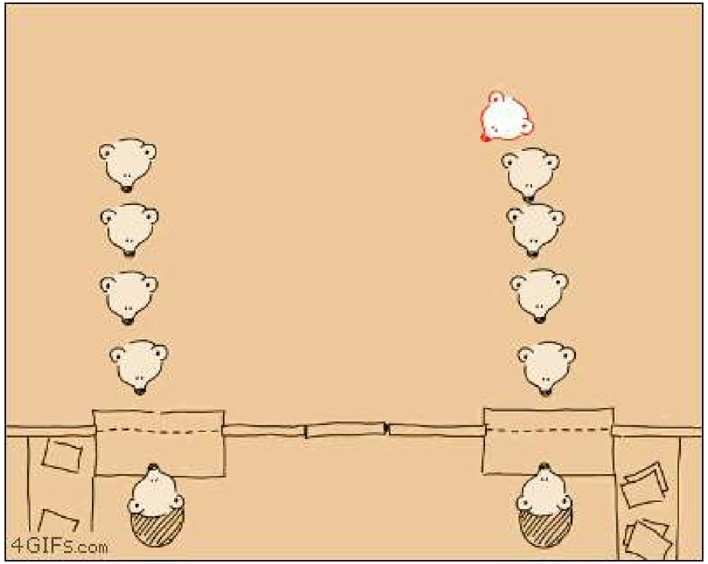
```
class DataHolder
{
public:
    // ...
    DataHolder(DataHolder&& other) noexcept
    {
        swap(*this, other);
    }

    DataHolder& operator=(DataHolder&& other) noexcept
    {
        swap(*this, other);
        return *this;
    }

private:
    void swap(DataHolder& lhs, DataHolder& rhs) const
    {
        std::scoped_lock lock(lhs.m_useful_data_lock, rhs.m_useful_data_lock);

        std::swap(lhs.m_useful_data, rhs.m_useful_data);
    }
    // ...
private:
    std::list<int> m_useful_data;
    std::mutex m_useful_data_lock;
};
```

# Simple livelock

```cpp
std::mutex resourceX;
std::mutex resourceY;

void thread_A_func()
{
    while (true)
    {
        std::unique_lock lockX(resourceX);

        std::this_thread::yield(); // emulate some work

        std::unique_lock lockY(resourceY, std::try_to_lock);

        if (lockY.owns_lock() == false) continue;

        std::cout << "thread_A working" << std::endl;
    }
}

void thread_B_func()
{
    while (true)
    {
        std::unique_lock lockY(resourceY);

        std::this_thread::yield(); // emulate some work

        std::unique_lock lockX(resourceX, std::try_to_lock);
        if (lockX.owns_lock() == false) continue;

        std::cout << "thread_B working" << std::endl;
    }
}
```

## Starvation as unique case of livelock

- Starvation is a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.



- **Difference Between Deadlock, Starvation, and Livelock**
- A deadlock is a situation that occurs in OS when any process enters in a waiting state because the demanded resource is being held by another waiting process.
- A livelock, on the other hand, is almost similar to a deadlock, except that the states of the processes which are involved in a livelock always keep on changing to one another, none progressing.
- So, Livelock is a unique case of resource starvation.
- **Summary:**
- Definition: A Livelock is a situation where a request for an exclusive lock is denied repeatedly, as many overlapping shared locks keep on interfering each other.
- Livelock occurs when the total number of allowed processes in a specific system should be defined by the total number of entries in the process table
- A deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource.
- A real-world example would be traffic, which is going only in one direction.
- An example of Livelock would be two people who meet face-to-face in a corridor, and both of them move aside to let the other pass.
- Starvation is a situation where all the low priority processes got blocked, and the high priority processes proceed.

# Protecting shared data during initialization

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;
void foo()
{
    std::unique_lock<std::mutex> lk(resource_mutex);
    if (!resource_ptr)
    {
        resource_ptr.reset(new some_resource);
    }
    lk.unlock();
    resource_ptr->do_something();
}

// data race?
void foo()
{
    if (!resource_ptr)
    {
        std::unique_lock<std::mutex> lk(resource_mutex);
        if (!resource_ptr)
        {
            resource_ptr.reset(new some_resource);
        }
        lk.unlock();
    }

    resource_ptr->do_something();
}
```

## The solution once_flag, call_once

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;

void foo()
{
    std::call_once(resource_flag, []()
    {
            resource_ptr.reset(new some_resource);
    });

    resource_ptr->do_something();
}
```

# Condition Variables STL

```
std::mutex mutex;
std::condition_variable cv;
int x = 0;
bool ready = false;

void producer() {
    std::unique_lock<std::mutex> lock(mutex);

    for (; x < 100;)
    {
        cv.wait(lock, [=] { return !ready; });

        ++x;
        ready = true;
        cv.notify_one();
    }
}
void consumer()
{
    std::unique_lock<std::mutex> lock(mutex);

    for (; x < 100;)
    {
        cv.wait(lock, [=] { return ready; });

        std::cout << x << std::endl;;
        ready = false;
        cv.notify_one();
    }
}
```

## Semaphore example

```
std::binary_semaphore smphSignalMainToThread(0), smphSignalThreadToMain(0);

void ThreadProc()
{
    // wait for a signal from the main by attempting to decrement the semaphore
    // this call blocks until the semaphore's count is increased from the main
    smphSignalMainToThread.acquire();

    std::cout << "[thread] Got the signal\n“;

    std::this_thread::sleep_for(std::chrono::seconds(3));

    std::cout << "[thread] Send the signal\n";
    // signal the main back
    smphSignalThreadToMain.release();
}

int main()
{
    std::thread thrWorker(ThreadProc);

    std::cout << "[main] Send the signal\n";

    // signal the worker thread to start working by increasing the semaphore's count
    smphSignalMainToThread.release();

    // wait until the worker thread is done doing the work by attempting to decrement the semaphore's count
    smphSignalThreadToMain.acquire();

    std::cout << "[main] Got the signal\n";
    thrWorker.join();
}
```

# Task based programming

```
int do_some_work()
{
    return 50;
}

int main()
{
    std::future<int> thread_result = std::async(do_some_work);

    std::cout << thread_result.get() << std::endl;
}

 // launch new physical thread
 std::future<int> thread_result = std::async(std::launch::async, do_some_work);

 // create deferred task (will be executed in the thread which call get() or wait())
 std::future<int> thread_result = std::async(std::launch::deferred, do_some_work);

 // by default behavior is implementation dependent (the scheduler can decide that there are too many threads already exist and it shouldn't create
a new one)
 std::future<int> thread_result = std::async(std::launch::async | std::launch::deferred, do_some_work);

// if the function passed to std::async raise an exception, the return std::future object holds thrown exception
```

## std::promise to transfer data between threads

```
std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };

void accumulate(std::vector<int>::iterator first,
    std::vector<int>::iterator last,
    std::promise<int> accumulate_promise)
{
    int sum = std::accumulate(first, last, 0);
    accumulate_promise.set_value(sum);

    // emulate some other useful work
    std::this_thread::sleep_for(std::chrono::seconds(5));
}

int main()
{
    std::promise<int> accumulate_promise;
    std::future<int> accumulate_future = accumulate_promise.get_future();

    std::thread work_thread(accumulate, numbers.begin(), numbers.end(), std::move(accumulate_promise));

    std::cout << "result=" << accumulate_future.get() << '\n';
    work_thread.join();
}

// note that it is highly recommended to use std::promise<void> to signal about an event which occurs only once
```

# std::packaged_task usage example

```cpp
int f(int x, int y) { return std::pow(x, y); }

void task_lambda()
{
    std::packaged_task<int(int, int)> task([](int a, int b)
    {
        return std::pow(a, b);
    });

    std::future<int> result = task.get_future();

    task(2, 9);

    std::cout << "task_lambda:\t" << result.get() << '\n';
}

void task_bind()
{
    std::packaged_task<int()> task(std::bind(f, 2, 11));
    std::future<int> result = task.get_future();

    task();

    std::cout << "task_bind:\t" << result.get() << '\n';
}

void task_thread()
{
    std::packaged_task<int(int, int)> task(f);
    std::future<int> result = task.get_future();

    std::thread task_td(std::move(task), 2, 10);
    task_td.join();

    std::cout << "task_thread:\t" << result.get() << '\n';
}

int main()
{
    task_lambda();
    task_bind();
    task_thread();
}
```

# A little bit about asynchronous programming 1

```
class server
{
public:
    server(boost::asio::io_context& io_context, short port)
        : acceptor_(io_context, tcp::endpoint(tcp::v4(), port))
    {
        do_accept();
    }

private:
    void do_accept()
    {
        acceptor_.async_accept(
            [this](boost::system::error_code ec, tcp::socket socket)
            {
                if (!ec)
                {
                    std::make_shared<session>(std::move(socket))->start();
                }

                do_accept();
            });
    }

    tcp::acceptor acceptor_;
};
```

## A little bit about asynchronous programming 2

```
class session : public std::enable_shared_from_this<session>
{
public:
    session(tcp::socket socket) : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }

private:
    void do_read()
    {
        auto self(shared_from_this());
        socket_.async_read_some(boost::asio::buffer(data_, max_length),
            [this, self](boost::system::error_code ec, std::size_t length)
            {
                if (!ec)
                {
                    do_write(length);
                }
            });
    }

    void do_write(std::size_t length)
    {
        auto self(shared_from_this());
        boost::asio::async_write(socket_, boost::asio::buffer(data_, length),
            [this, self](boost::system::error_code ec, std::size_t /*length*/)
            {
                if (!ec)
                {
                    do_read();
                }
            });
    }

    tcp::socket socket_;
    enum { max_length = 1024 };
    char data_[max_length];
```

## Tips

- Have valid reasons for introducing multi-threading
- Where it is avoidable, do not have multiple threads writing to the same data structure.
- Where this cannot be avoided, have a lock so that the latecoming thread will wait until is complete.
- Avoid having global variables except constants
- Make sure that threads release the resource as soon as they are finished with them.
- Threads are moderately expensive to create. If you're spinning them up on demand and destroying them often, the overhead can be considerable. Use a thread pool.
- If you have no resource acquisition delays in your program, your number of threads should try to match the number of processors. If you have a four core machine, the only reason to run more than four threads in an application is when the scenario requires a thread to go into a wait state while acquiring resources. Thread context switches are expensive.

## Questions