ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

RARELY UPDATE DATA

```
class data_cache
{
   std::map<std::string, data_entry> entries;
   mutable std::shared_mutex mutex;

public:
   data_entry find(const std::string& key) const
   {
      std::shared_lock lock(mutex);
      const auto iter = entries.find(key);
      return iter == entries.end() ? data_entry{} : iter->second;
   }

   void update_or_add(const std::string& key, const data_entry& data)
   {
      std::lock_guard lock(mutex);
      entries[key] = data;
   }
};
```

PROTECTED LAZY INITIALIZATION

```
std::shared_ptr<resource> resource_ptr;

void foo()
{
  if(!resource_ptr)
  {
    resource_ptr.reset(new resource);
  }
  resource_ptr->do_something();
}
```

PROTECTED LAZY INITIALIZATION

```
std::shared_ptr<resource> resource_ptr;
std::mutex mutex;

void foo()
{
   std::unique_lock lock(mutex);    ???
   if(!resource_ptr)
   {
      resource_ptr.reset(new resource);
   }
   lock.unlock();
   resource_ptr->do_something();
}
```

PROTECTED LAZY INITIALIZATION

```
std::shared_ptr<resource> resource_ptr;
std::mutex mutex;

void foo()
{
   if(!resource_ptr)
   {
      std::lock_guard lock(mutex);
      if(!resource_ptr)
      {
         resource_ptr.reset(new resource);
      }
   }
   resource_ptr->do_something();
}
```

Ok???

PROTECTED LAZY INITIALIZATION

```
std::shared_ptr<resource> resource_ptr;
std::mutex mutex;

void foo()
{
   if(!resource_ptr)
   {
      std::lock_guard lock(mutex);
      if(!resource_ptr)
      {
         resource_ptr.reset(new resource);
      }
   }
   resource_ptr->do_something();
}
```

Undefined behavior

PROTECTED LAZY INITIALIZATION

```
std::shared_ptr<resource> resource_ptr;
std::once_flag flag;

void init_resource()
{
  resource_ptr.reset(new resource);
}

void foo()
{
  std::call_once(flag, init_resource);
  resource_ptr->do_something();
}
```

WAIT FOR CONDITION

```
bool flag;
std::mutex mutex;

void wait_for_flag()
{
   std::unique_lock lock(mutex);
   while(!flag)
   {
      lock.unlock();
      lock.lock();
   }
}
```

Very bad and slow

WAIT FOR CONDITION

```
bool flag;
std::mutex mutex;

void wait_for_flag()
{
   std::unique_lock lock(mutex);
   while(!flag)
   {
      lock.unlock();
      std::this_thread::sleep_for(100ms);
      lock.lock();
   }
}
```

Bit better

STD::CONDITION_VARIABLE

```
std::mutex mutex;
std::queue<data_chunk> data_queue;
std::condition_variable condition;
```

```
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        const data_chunk = prepare_data();
        {
            std::lock_guard guard(mutex);
            data_queue.push(data);
        }
        condition.notify_one();
    }
}
```

```
void data_processing_thread()
{
    while(true)
    {
        std::unique_lock lock(mutex);
        condition.wait(
            lock,
            [](){return !data_queue.empty();}
        );
        auto data = data_queue.front();
        queue.pop();
        lock.unlock();
        process(data);
        if(is_last_chunk(data))
        {
            break;
        }
    }
}
```

Spurious wake

```
std::condition_variable::notify_all();
```

STD::CONDITION_VARIABLE

```
template <typename T>
class threadsafe_queue
{
    std::queue<T> queue;
    std::mutex mutex;
    std::conditional_variable condition;

public:
    void push(T new_value)
    {
        std::lock_guard lock(mutex);
        queue.push(new_value);
        condition.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock lock(mutex);
        condition.wait(lock, [this]{return !queue.empty();});
        value = queue.front();
        queue.pop();
    }
};
```

STD::CONDITION_VARIABLE

```cpp
threadsafe_queue<data_chunk> data_queue;
```

```cpp
void data_preparation_thread()
{
   while(more_data_to_prepare())
   {
      const data_chunk = prepare_data();
      data_queue.push(data);
   }
}
```

```cpp
void data_processing_thread()
{
   while(true)
   {
      data_chunk data;
      data_queue.wait_and_pop(data);

      process(data);

      if(is_last_chunk(data))
      {
         break;
      }
   }
}
```

STD::ASYNC

- Task based programming
- Allows to get result of asynchronous operation by **std::future**
- Allows to catch and handle exceptions
- Avoids oversubscription by "thread scheduler"

STD::ASYNC

```
int foo()
{
   //some operations
   ...
   return result_of_operations;
}

int main()
{
   std::future<int> future = std::async(foo);
   do_something();

   const int result = future.get();
   std::cout << "Result = " << result << '\n';

   return 0;
}
```

foo can execute either synchronously or asynchronously

Can be called once

STD::ASYNC

```
int foo()
{
   //some operations
   ...
   throw CustomException{};
}

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

   try
   {
      const int result = future.get();
      std::cout << "Result = " << result << '\n';
   }
   catch(CustomException&)
   {
      std::cout << "Catch custom exception\n";
   }
}
```

STD::ASYNC

```
int main()
{
   auto future1 = std::async(std::launch::deferred, []{ baz(); });
   auto future2 = std::async(std::launch::async, []{ bar(); });
   auto future3 = std::async(std::launch::async | std::launch::deferred,
                             []({ foo(); });

   future1.get();
   future2.get();
   future3.get();

   return 0;
}
```

STD::PACKAGED_TASK

```
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';
}

int main()
{
   task_lambda();

   return 0;
}
```

STD::PACKAGED_TASK

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

   return 0;
}
```

STD::PACKAGED_TASK

```
int main()
{
   std::packaged_task task([]{throw CustomException();});
   auto future = task.get_future();
   std::thread thread(std::move(task));
   thread.detach();

   try
   {
      future.wait();
   }
   catch(CustomException&)
   {
      std::cout << "Catch custom exception\n";
   }

   return 0;
}
```

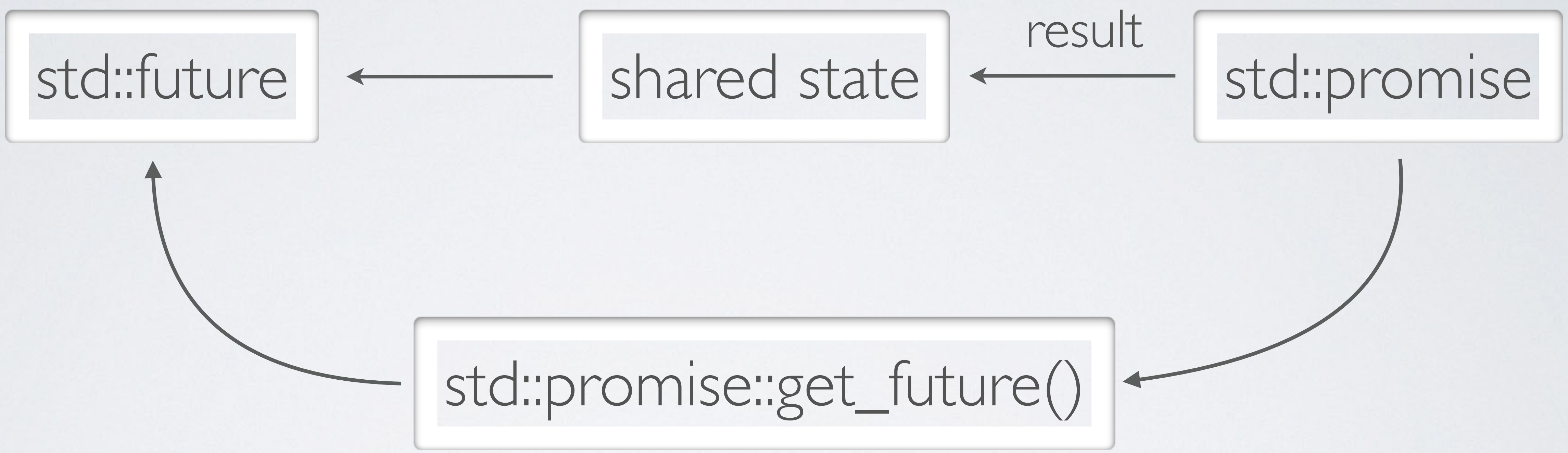
can catch exceptions too

STD::PACKAGED_TASK

```cpp
std::mutex mutex;
std::deque<std::packaged_task<void()>> tasks;

void gui_thread()
{
    while(!gui_shutdown_message_reveiced())
    {
        get_and_process_gui_message();

        std::packaged_task<void()> task;

        {
            std::lock_guard guard(mutex);

            if(tasks.empty())
            {
                continue;
            }

            task = std::move(tasks.front());
            tasks.pop_front();
        }

        task();
    }
}
```

```cpp
template <typename Func>
auto post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f);
    auto result = task.get_future();

    std::lock_guard guard(mutex);
    tasks.push_back(std::move(task));

    return result;
}
```

STD::PROMISE


std::future
shared state
result
std::promise
std::promise::get_future()

STD::PROMISE

```
void accumulate(std::vector<int> vector, std::promise<int> promise)
{
   int sum = std::accumulate(vector.begin(), vector.end(), 0);
   promise.set_value(sum);
}

int main()
{
   std::vector<int> numbers = {1, 2, 3, 4, 5};
   std::promise<int> promise;
   auto future = promise.get_future();

   std::thread thread(accumulate, numbers, std::move(promise));
   thread.detach();

   std::cout << "result = " << future.get();

   return 0;
}
```

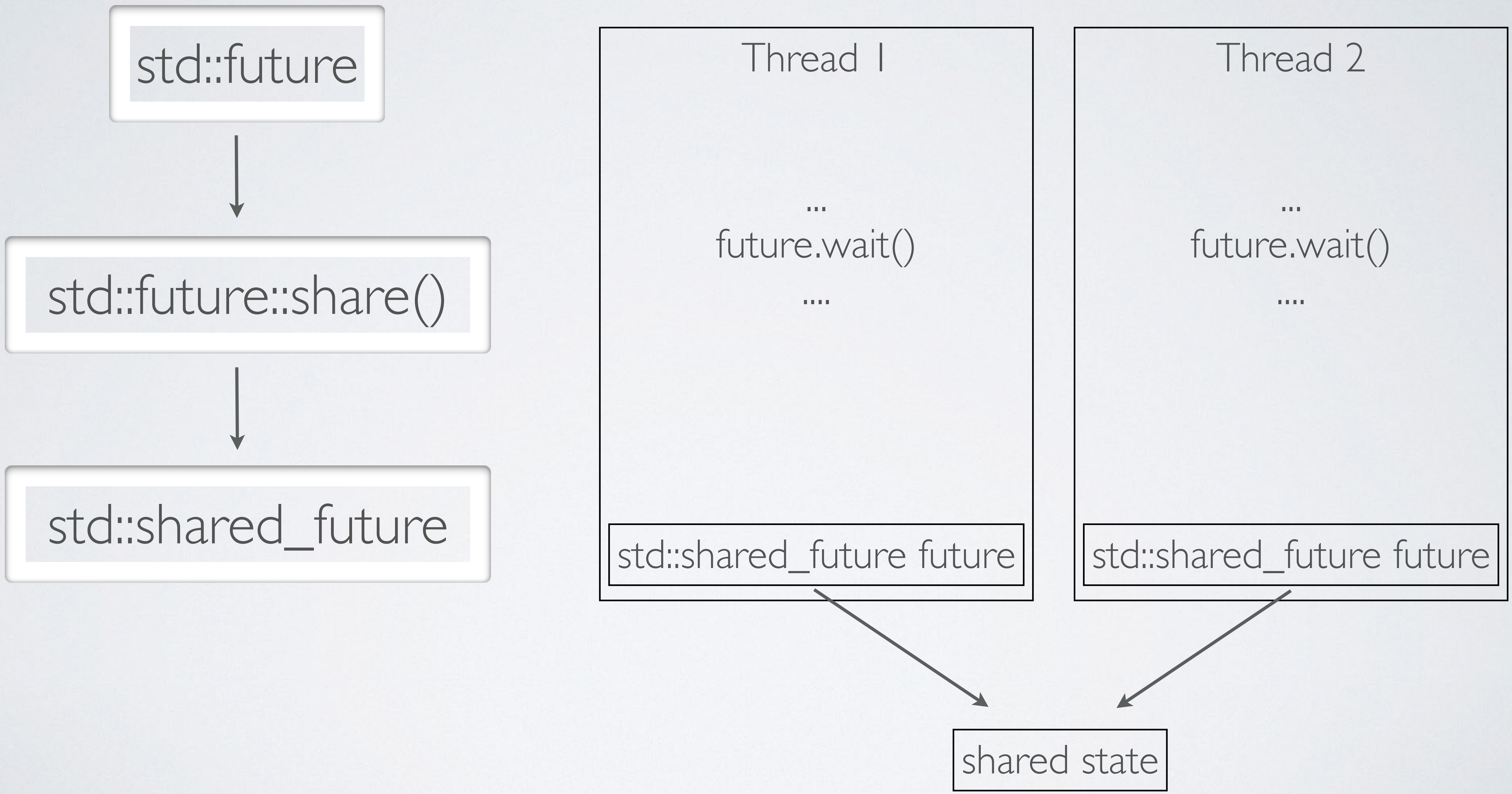
STD::PROMISE

```
void process_connections(connection_set& connections)
{
    while(!done(connections))
    {
        for(auto connection : connections)
        {
            if(connection->has_incoming_data())
            {
                data_packet data = connection->incoming();
                std::promise<payload_type>& promise = connection->get_promise(data.id);
                promise.set_value(data.payload);
            }

            if(connection->has_outgoing_data())
            {
                outgoing_packet data = connection->top_of_outgoing_queue();
                connection->send(data.payload);
                data.promise.set_value(true);
            }
        }
    }
}
```

STD::PROMISE

```
extern std::promise<double> promise;

try
{
   promise.set_value(calculate_value());
}
catch(...)
{
   promise.set_exception(std::current_exception());
}
```

STD::SHARED_FUTURE



STD::FUTURE DESTRUCTION

Destructor may block current thread if all of the following are true:

- The shared state was created by a call to **std::async** (with **std::launch::async** policy).
- The shared state is not yet ready.
- This was the last reference to the shared state.

STD::FUTURE DESTRUCTION

```
int main()
{
   std::packaged_task task([]{ std::this_thread::sleep_for(5s); });

   {
      auto future = task.get_future();
      std::thread thread(std::move(task));
      thread.detach();
   }

   return 0;
}
```

don't block thread

STD::FUTURE DESTRUCTION

```
int main()
{
    {
        auto future = std::async(std::launch::deferred,
                                 []{ std::this_thread::sleep_for(5s); });
    } //don't block thread

    {
        auto future = std::async(std::launch::async,
                                 []{ std::this_thread::sleep_for(5s); });
    } //block thread

    {
        auto future = std::async([]{ std::this_thread::sleep_for(5s); });
    } //may block thread

    return 0;
}
```


Multithreaded programming
Theory
Actual