

❓ C++ - parallelization and synchronization

Jakub Yaghob





? The problem

? Race conditions

- ? Separate threads with shared state
- ? Result of computation depends on OS scheduling

? Race conditions – simple demo



? Linked list

? Shared state

List lst;

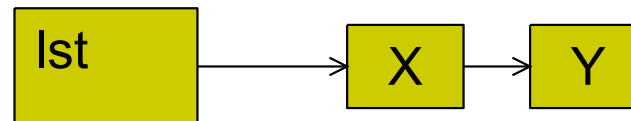
? Thread A

lst.push_front(A);

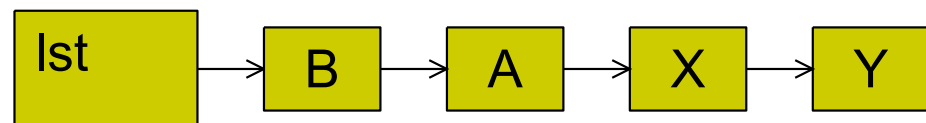
? Thread B

lst.push_front(B);

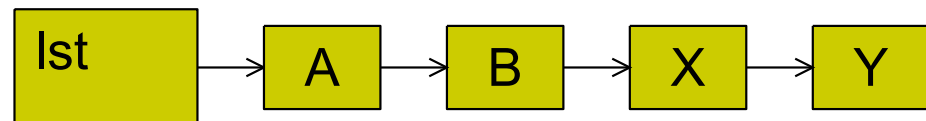
Initial state



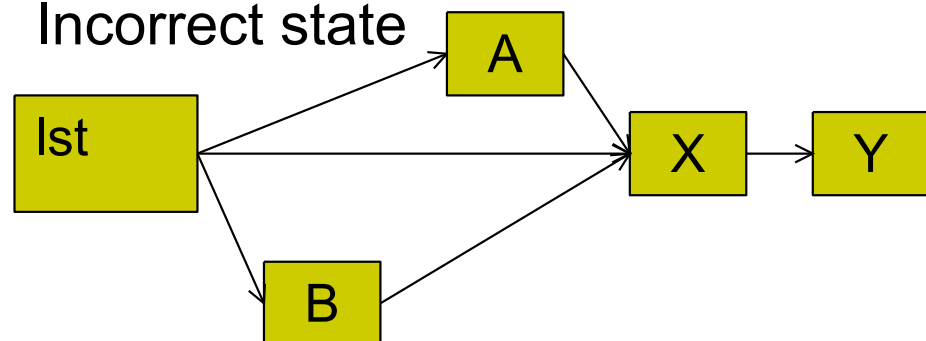
Correct state



Another correct state



Incorrect state



? Race conditions – advanced demo



```
struct Counter {  
    Counter():value(0) { }  
    int value;  
    void increment()  
    { ++value; }  
    void decrement()  
    { --value; }  
    int get()  
    { return value; }  
};
```

? Shared state

Counter c;

? Thread A

c.increment();
cout << c.get();

? Thread B

c.increment();
cout << c.get();

? Possible outputs

12, 21, 11



? C++ features

? C++ 11

- ? Atomic operations
- ? Low-level threads
- ? High-level futures
- ? Synchronization primitives
- ? Thread-local storage

? C++14 features

- ? Shared timed mutex

? C++17 features

- ? Parallel algorithms
- ? Shared mutex

? C++20 features

- ? Stop tokens
- ? Semaphore
- ? Coordination types



? Threads

? Low-level threads

- ? Header `<thread>`

- ? **thread** class

- ? Fork-join paradigm

- ? Namespace **this_thread**



? Threads

? Class thread

? Constructor

? **template <class F, class ...Args>
explicit thread(F&& f, Args&&... args);**

? Destructor

? If joinable() then **terminate()**

? **bool joinable() const noexcept;**

? **void join();**

? Blocks, until the thread *this has completed

? **void detach();**

? **id get_id() const noexcept;**

? **static unsigned hardware_concurrency();**



? Threads

? Class jthread

- ? Like thread, autostops+autojoins on destruction
- ? Provides stop token
 - ? Internal member of **std::stop_source** type
 - ? Constructor accepts function with **std::stop_token** as first argument
 - ? Destructor calls **request_stop**
 - ? Interface functions **get_stop_source**, **get_stop_token**, and **request_stop**



? Threads

? Namespace **this_thread**

? **thread::id get_id() noexcept;**

? Unique ID of the current thread

? **void yield() noexcept;**

? Opportunity to reschedule

? **sleep_for, sleep_until**

? Blocks the thread for relative/absolute timeout



? Threads

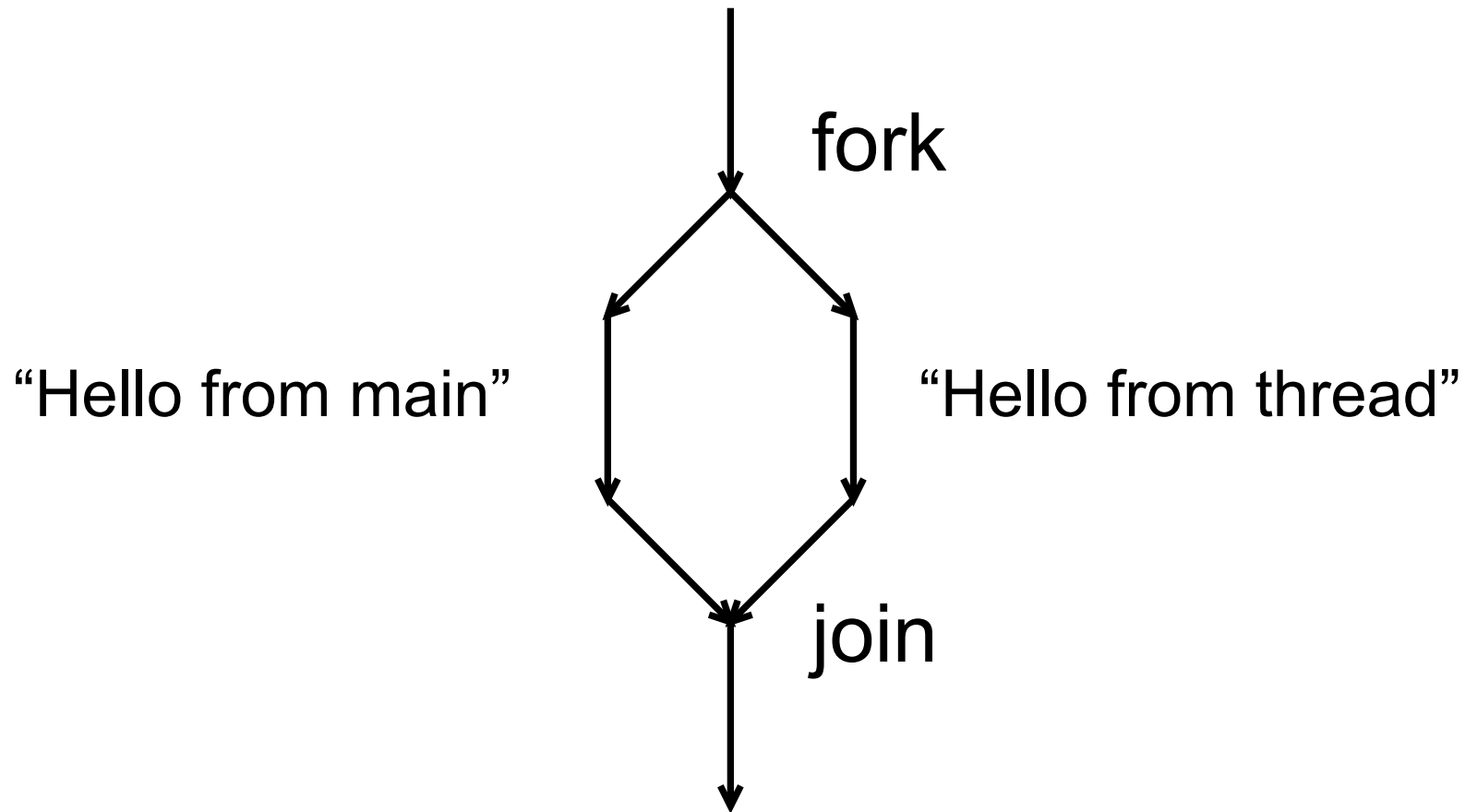
? Demo

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

```
void thread_fn() { std::cout << "Hello from thread" <<
    std::endl; }
```

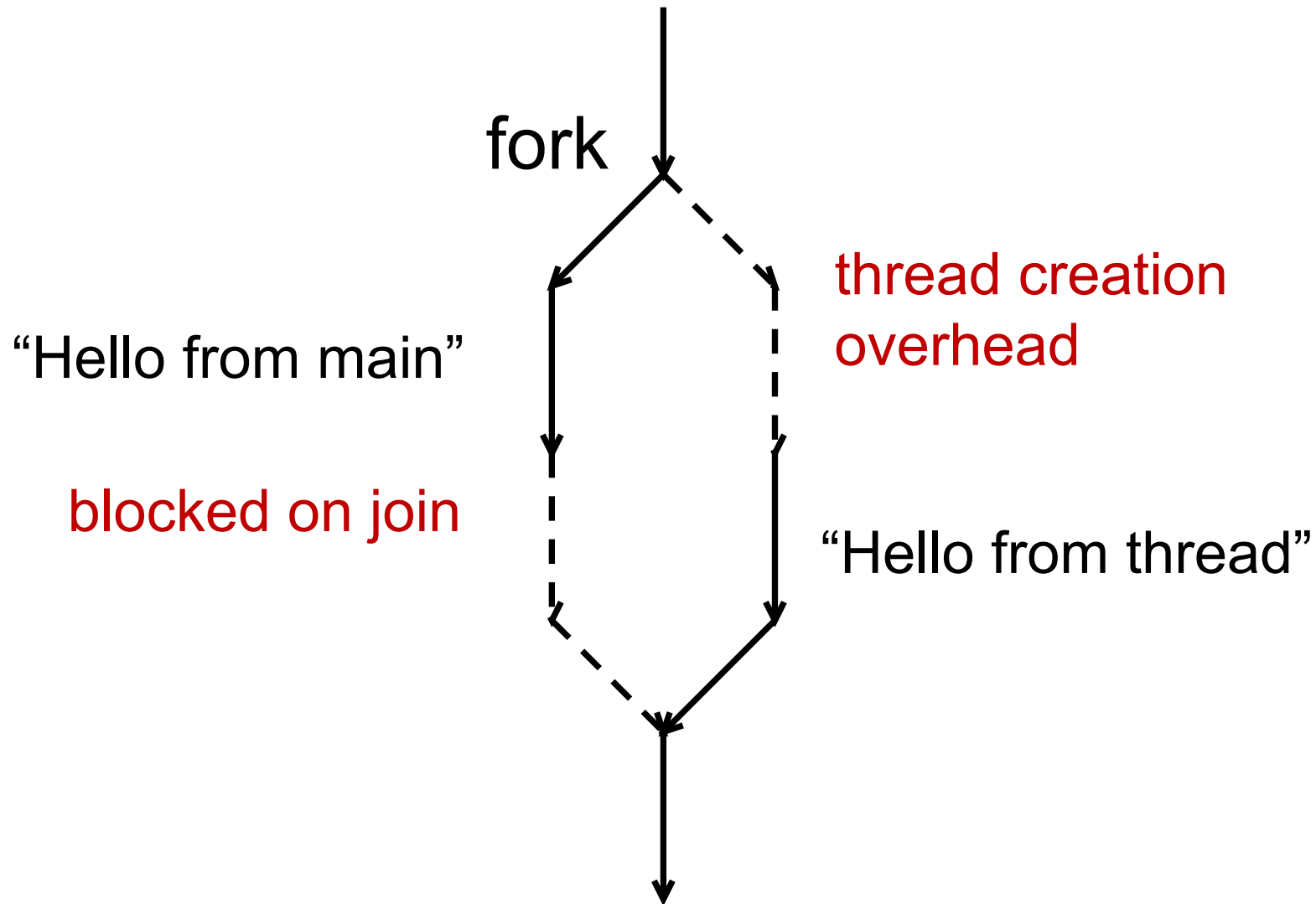
```
int main(int argc, char **argv) {
    std::thread thr(&thread_fn);
    std::cout << "Hello from main" << std::endl;
    thr.join();
    return 0;
}
```

? Threads

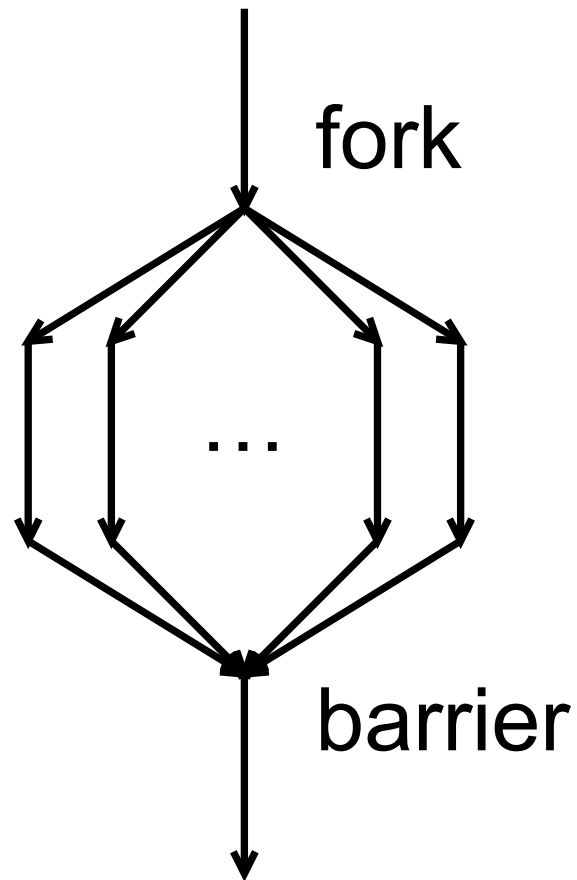




? Threads



? Threads





? Threads

? Demo

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

```
int main(int argc, char **argv) {
    std::vector<std::thread> workers;
    for(int i=0;i<10;++i)
        workers.push_back(std::thread([i]() {
            std::cout << "Hello from thread " << i << std::endl;
        }));
    std::cout << "Hello from main" << std::endl;
    for(auto &t : workers)
        t.join();
    return 0;
}
```



? Threads

? Passing arguments to threads

? By value

? Safe, but you MUST make deep copy

? By move (rvalue reference)

? Safe, as long as strict (deep) adherence to move semantics

? By const reference

? Safe, as long as object is guaranteed deep-immutable

? By non-const reference

? Safe, as long as the object is monitor



? Futures

? Futures

- ? Header <future>
- ? High-level asynchronous execution
- ? Future
- ? Promise
- ? Async
- ? Error handling



? Futures

? Shared state

? Consist of

- ? Some state information and some (possibly not yet evaluated) result, which can be a (possibly void) value or an exception

? Asynchronous return object

- ? Object, that reads results from an shared state

? Waiting function

- ? Potentially blocks to wait for the shared state to be made ready

? Asynchronous provider

- ? Object that provides a result to a shared state



? Futures

? Future

? **std::future<T>**

? Future value of type T

? Retrieve value via **get()**

? Waits until the shared state is ready

? **wait(), wait_for(), wait_until()**

? **valid()**

? **std::shared_future<T>**

? Value can be read by more than one thread

vyhlídka ten share state, takže
nejde volat duplikát a
jakkmile to jednou zavolám,
valid se nastaví na false.



? Futures

? Async

? `std::async`

? Higher-level convenience utility

? Launches a function potentially in a new thread

? Async usage

```
int foo(double, char, bool);
```

```
auto fut = std::async(foo, 1.5, 'x', false);
```

```
auto res = fut.get();
```

> tohle ale umožňuje i sériové volání...

> tohle bude `std::future<int>`

→ tedy to vykrmí, `.valid()` už potom bude `== false`.



? Futures

? Packaged task

? `std::packaged_task`

? How to implement async with more control

? Wraps a function and provides a future for the function result value, but the object itself is callable



? Futures

? Packaged task usage

```
std::packaged_task<int(double, char, bool)>  
    tsk(foo);  
auto fut = tsk.get_future();  
std::thread thr(std::move(tsk), 1.5, 'x', false);  
auto res = fut.get();
```



? Futures

? Promise

? `std::promise<T>`

? Lowest-level

? Steps

- ? Calling thread makes a promise
- ? Calling thread obtains a future from the promise
- ? The promise, along with function arguments, are moved into a separate thread
- ? The new thread executes the function and fulfills the promise
- ? The original thread retrieves the result



? Futures

? Promise usage

? Thread A

```
std::promise<int> prm;  
auto fut = prm.get_future();  
std::thread thr(thr_fnc, std::move(prm));  
auto res = fut.get();
```

? Thread B

```
void thr_fnc(std::promise<int> &&prm) {  
    prm.set_value(123);  
}
```



? Futures

? Constraints

- ? A default-constructed promise is inactive
 - ? Can die without consequence
- ? A promise becomes active, when a future is obtained via **get_future()**
 - ? Only one future may be obtained
- ? A promise must either be satisfied via **set_value()**, or have an exception set via **set_exception()**
 - ? A satisfied promise can die without consequence
 - ? **get()** becomes available on the future
 - ? A promise with an exception will raise the stored exception upon call of **get()** on the future
 - ? A promise with neither value nor exception will raise “broken promise” exception



? Futures

? Exceptions

? All exceptions of type `std::future_error`

? Has error code with enum type `std::future_errc`

? inactive promise

```
std::promise<int> pr;  
// fine, no problem
```

? too many futures

```
std::promise<int> pr;  
auto fut1 = pr.get_future();  
auto fut2 = pr.get_future();
```

? active promise, unused

```
std::promise<int> pr;  
auto fut = pr.get_future();  
// fine, no problem  
// fut.get() blocks indefinitely
```

```
// error "Future already  
retrieved"
```



? Futures

? satisfied promise

```
std::promise<int> pr;  
auto fut = pr.get_future();  
{ std::promise<int>  
  pr2(std::move(pr));  
  pr2.set_value(10);  
}  
auto r = fut.get();  
// fine, return 10
```

? too much satisfaction

```
std::promise<int> pr;  
auto fut = pr.get_future();  
{ std::promise<int>  
  pr2(std::move(pr));  
  pr2.set_value(10);  
  pr2.set_value(11);  
  // error "Promise already  
  satisfied"  
}  
auto r = fut.get();
```



? Futures

? exception

```
std::promise<int> pr;  
auto fut = pr.get_future();  
{ std::promise<int> pr2(std::move(pr));  
  pr2.set_exception(  
    std::make_exception_ptr(  
      std::runtime_error("bububu")));  
}  
auto r = fut.get();  
// throws the runtime_error
```



? Futures

? broken promise

```
std::promise<int> pr;  
auto fut = pr.get_future();  
{ std::promise<int> pr2(std::move(pr));  
  // error "Broken promise"  
}  
auto r = fut.get();
```



? Synchronization primitives

? Synchronization primitives

? Mutual exclusion

? Headers `<mutex>` and `<shared_mutex>`

? Condition variables

? Header `<condition_variable>`

? Semaphore

? Header `<semaphore>`



? Mutex

? Mutex

- ? A synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads
- ? **mutex** offers exclusive, non-recursive ownership semantics
 - ? 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 ownership of the **mutex**
 - ? A calling thread must not own the **mutex** prior to calling **lock** or **try_lock**
- ? The behavior of a program is undefined if a **mutex** is destroyed while still owned by some thread



? Mutex example

? Shared state

```
List lst;  
std::mutex mtx;
```

? Thread A

```
mtx.lock();  
lst.push_front(A);  
mtx.unlock();
```

? Thread B

```
mtx.lock();  
lst.push_front(B);  
mtx.unlock();
```



? Mutex variants

? Other **mutex** variants

? **timed_mutex**

- ? In addition, **timed_mutex** provides the ability to attempt to claim ownership of a **timed_mutex** with a timeout via the **try_lock_for** and **try_lock_until**

? **recursive_mutex**

- ? exclusive, recursive ownership semantics

- ? 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

? **recursive_timed_mutex**

- ? Combination



? Mutex variants

? Other mutex variants

? `std::shared_mutex`

? Additionally multiple threads can make shared lock using `lock_shared()`

? Either exclusive lock or shared lock

? `std::shared_timed_mutex`



? Mutex wrappers

? std::unique_lock

? Lock class with more features

? Timed wait, deferred lock

? std::lock_guard

? Scope based lock (RAII)

? Linked list demo, code for one thread

```
{  
    std::lock_guard<std::mutex> lk(mtx);  
    lst.push_front(X);  
}
```



? Mutex wrappers and others

? Shared lock wrapper

? `std::shared_lock`

? Calls `lock_shared` for the given shared mutex

? Variadic wrapper

? `template <typename ... MutexTypes> class scoped_lock;`

? Multiple locks at once, RAll, deadlock avoidance

? Interference size

? `std::size_t hardware_destructive_interference_size;`

? Size of a cache line



? Locking algorithms

? `std::lock`

- ? locks specified mutexes, blocks if any are unavailable, deadlock avoidance

? `std::try_lock`

- ? attempts to obtain ownership of mutexes via repeated calls to `try_lock`

// don't actually take the locks yet

```
std::unique_lock<std::mutex> lock1(mtx1, std::defer_lock);
```

```
std::unique_lock<std::mutex> lock2(mtx2, std::defer_lock);
```

// lock both unique_locks without deadlock

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



? Call once

? std::once_flag

? Helper object for **std::call_once**

? std::call_once

? invokes a function only once even if called from multiple threads

```
std::once_flag flag;  
void do_once() {  
    std::call_once(flag, [](){ do something only once });  
}  
std::thread t1(do_once);  
std::thread t2(do_once);
```



? Condition variable

? `std::condition_variable`

- ? Can be used to block a thread, or multiple threads at the same time, until
 - ? a notification is received from another thread
 - ? a timeout expires, or
 - ? a spurious wakeup occurs
 - ? Appears to be signaled, although the condition is not valid
 - ? Verify the condition after the thread has finished waiting
- ? Works with `std::unique_lock`
- ? **wait** atomically manipulates mutex, **notify** does nothing



? Condition variable example

```
std::mutex m;  
std::condition_variable cond_var;  
bool done = false; bool notified = false;
```

? Producer

```
for () {  
    // produce something  
    { std::lock_guard<std::mutex>  
      lock(m);  
      queue.push(item);  
      notified = true; }  
    cond_var.notify_one();  
}  
std::lock_guard<std::mutex> lock(m);  
notified = true;  
done = true;  
cond_var.notify_one();
```

? Consumer

```
std::unique_lock<std::mutex> lock(m);  
while(!done) {  
    while (!notified) {  
        // loop to avoid spurious wakeups  
        cond_var.wait(lock);  
    }  
    while(!queue.empty()) {  
        queue.pop();  
        // consume  
    }  
    notified = false;  
}
```



? Semaphore

? Counting semaphore

- ? `std::counting_semaphore`

- ? Constructor sets the count

- ? Manipulation

 - ? `acquire()`, `release(count=1)`

? Binary semaphore

- ? `std::binary_semaphore`



? Coordination types

? Latches

- ? Header <latch>

- ? Thread coordination mechanism

 - ? Block threads until an expected number of threads arrive

- ? Single use

? Barriers

- ? Header <barrier>

- ? Sequence of phases

 - ? Each call to **arrive()** decrements expected count, the thread can then **wait()**

 - ? When count==0, the completion function is called and all blocked threads are unblocked

 - ? Expected count is reset to the previous value

- ? Constructor

 - ? **barrier(ptrdiff_t expected, CompletionFunction f)**



? Stop tokens

- ? Asynchronously request to stop execution of an operation
 - ? Shared state among associated **stop_source**, **stop_token**, and **stop_callback**
 - ? **stop_token**
 - ? Interface for querying whether a stop request has been made or can ever be made
 - ? **bool stop_requested()**, **bool stop_possible()**
 - ? **stop_source**
 - ? Implements the semantics of making stop request
 - ? Creates **stop_tokens**
 - ? **stop_token get_token()**
 - ? Makes stop request
 - ? **request_stop()**
 - ? **stop_callback**
 - ? Invokes callback function when stop request has been made



? Thread-local storage

? Thread-local storage

- ? Added a new storage-class
- ? Use keyword **thread_local**
 - ? Must be present in all declarations of a variable
 - ? Only for namespace or block scope variables and to the names of static data members
 - ? For block scope variables **static** is implied
- ? Storage of a variable lasts for the duration of a thread in which it is created



? Parallel algorithms

? Parallelism

? In headers <algorithm>, <numeric>

? Parallel algorithms

? Execution policy in <execution>

? **seq** – execute sequentially

? **par** – execute in parallel on multiple threads

? **par_unseq** – execute in parallel on multiple threads, interleave individual iterations within a single thread, no locks

? **unseq** – execute in single thread+vectorized

? **for_each**

? **reduce, scan, transform_reduce, transform_scan**

? Inclusive scan – like **partial_sum**, includes i-th input element in the i-th sum

? Exclusive scan – like **partial_sum**, excludes i-th input element from the i-th sum

? No exceptions should be thrown

? Terminate



? Parallel algorithms

? Parallel algorithms

? Not all algorithms have parallel version

? adjacent_difference, adjacent_find, all_of, any_of, copy, copy_if, copy_n, count, count_if, equal, exclusive_scan, fill, fill_n, find, find_end, find_first_of, find_if, find_if_not, for_each, for_each_n, generate, generate_n, includes, inclusive_scan, inner_product, inplace_merge, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, merge, min_element, minmax_element, mismatch, move, none_of, nth_element, partial_sort, partial_sort_copy, partition, partition_copy, reduce, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, search, search_n, set_difference, set_intersection, set_symmetric_difference, set_union, sort, stable_partition, stable_sort, swap_ranges, transform, transform_exclusive_scan, transform_inclusive_scan, transform_reduce, uninitialized_copy, uninitialized_copy_n, uninitialized_fill, uninitialized_fill_n, unique, unique_copy



? C++ extension – executors

? Executors

? Now separate TS, not finished in C++23 timeframe, maybe in C++26?

? Executor

? Controls how a task (=function) is executed

? Direction

? One-way execution

? Does not return a result

? Two-way execution

? Returns future

? Then

? Execution agent begins execution after a given future becomes ready, returns future

? Cardinality

? Single

? One execution agent

? Bulk executions

? Group of execution agents

? Agents return a factory

? Thread pool

? Controls where the task is executed

? C++ extensions – concurrency



? Concurrency

? TS published, depends on executors TS

? Improvements to future

? `future<T2> then(F &&f)`

? Execute asynchronously a function f when the future is ready

? C++ extension – transactional memory



- ? TS v1 finished, never used
- ? TS v2 in progress
- ? Transactional memory
 - ? Atomic blocks
 - ? Transactional behavior
 - ? Exception inside the block leads to undefined behavior

```
unsigned int f()
{
    static unsigned int i = 0;
    atomic do {
        ++i;
        return i;
    }
}
```