

KIV/CPP – Programming in C++

08. Threads in C++, basic synchronization primitives, parallel algorithms

Martin Úbl

KIV ZČU

2025/2026

- threads in general and in depth are covered in other courses
- here we will cover only the necessary minimum
- getting an overview

Threads

- thread = an independently executed flow of instructions
- threads can typically run in parallel
 - either seemingly (they take turns – time-slicing, see ZOS, OS)
 - or truly (multiple CPU cores – see ZOS, OS, PPR)
- typically we sometimes want to synchronize threads
 - a thread waits for another thread's work to finish
 - a thread waits for a critical section
 - note: an area of code in which at most one thread may be present at one time
 - etc...
- for more, see KIV/ZOS, KIV/OS and KIV/PPR
- or KIV/UPP in the summer semester
- we will cover only the language side of things

Threads

- `std::thread`
- `#include <thread>`
- a class whose instances wrap exactly one thread instance
- does not support copy semantics (threads cannot simply be cloned)
- takes the function to be executed as the first parameter
 - ordinary function, method, lambda function, functor, ...
- the remaining parameters are the parameters passed to this function

```
void vlaknova_fce(int i) {  
    // ...  
}
```

```
std::thread thr(&vlaknova_fce, 4);
```

Threads

- class method as a parameter
- in effect, we just respect the `thiscall` convention and provide a pointer to the class instance as the first parameter

```
void Worker::work(int i) {  
    while (true) {  
        //  
    }  
}
```

```
Worker w1;  
std::thread thr(&Worker::work, &w1, 8);
```

- note: pay attention to the lifetime of object `w1`; the thread may theoretically still be running even after leaving the scope

Threads

- the `std::thread` destructor assumes that we have terminated everything correctly
 - this means reading the thread's return code
 - calling the `join()` method
 - as a rule, it is a good idea to call `joinable()` before that – this ensures that nobody has already read the return code
- without that, the destructor will complain (at runtime)
- `std::terminate` is called and the application crashes

```
std::thread thr(&do_some_work);
```

```
// ...
```

```
if (thr.joinable())  
    thr.join();
```

- the thread object can be “detached” from the resource
- `detach()`
- the destructor then does not require reading the return value
- however, correct termination may be a problem
 - e.g. we may want to return from the main function already, but the thread is still happily running

```
{  
    std::thread thr(&do_some_work);  
  
    thr.detach();  
} // the destructor does not complain
```

Threads

- `std::jthread`
- basically the same as `std::thread`
- but...
 - the destructor automatically performs `join()`
 - a standard way to terminate a thread using a shared interrupt token

```
std::jthread thr([] (std::stop_token st) {  
    while (!st.stop_requested())  
        work_work_work();  
});
```

```
// some work
```

```
thr.request_stop(); // stop the thread ASAP  
thr.join();         // not needed explicitly
```

- `std::jthread`
- naturally, a similar effect can be achieved by replacing the stop token with a reference to some shared variable and using an ordinary `std::thread`
- `std::jthread` only standardizes this flow

Threads

- `std::jthread`
- dependencies and sequences can be expressed more clearly
 - e.g. `task1` and `task2` can run in parallel, but `task3` and `task4` depend on their result:

```
{  
    std::jthread task1(&task_fnc1);  
    std::jthread task2(&task_fnc2);  
} // task1 and task2 run in parallel
```

```
// both tasks will be finished here
```

```
{  
    std::jthread task3(&task_fnc3);  
    std::jthread task4(&task_fnc4);  
} // task3 and task4 run in parallel
```

- *Futures*
- or when we do not want to get our hands dirty with thread management
- the *futures* and *promises* mechanisms are known from other languages
 - e.g. JavaScript
- `#include <future>`

- `std::async`
- runs the function supplied as a parameter
 - either asynchronously (`std::launch::async`)
 - or deferred (when someone requests the result) (`std::launch::deferred`)
- the `async` call itself finishes immediately
- an instance of `std::future` is returned (template type)
 - the template type is chosen according to the return value of the function being run

- `std::future`
- template type
- serves for signaling (and passing results) between threads
- essentially two basic methods:
 - `wait()` – blocks until the result is available
 - variants `wait_for()` and `wait_until()`
 - `get()` – obtains the result; if it is not available, calls `wait()`

Futures

- `std::async` and `std::future`

```
auto result = std::async(std::launch::async ,
    []() {
        int x;

        // ...

        return x;
    });

// some other work that does not
// require the result

int big_result = local_result + result.get();
```

- `std::async` can be somewhat limiting
- it may be necessary to use `std::thread` for better management
- e.g. *thread pooling* and similar approaches
- if we still want to use `std::future`, we need one more mechanism
 - *promises*
 - `std::promise`
 - also a template type
 - provides instances of `std::future`
 - the worker thread “sends” the result to the promise
 - the outer thread “retrieves” the result from the future
 - each thread can retrieve its own future instance (and thus wait for the result on its own)

Futures

- outer code – creates the promise, retrieves the future, waits for the result

```
std::promise<int> res_promise;  
std::future<int> local_future  
    = res_promise.get_future();  
  
std::thread thr(&ThreadFnc,  
               std::move(res_promise));  
  
// some work  
  
int mega_result = local_result  
                  + local_future.get();  
  
thr.join();
```

- worker thread – takes the promise, sets the result

```
void ThreadFnc(std::promise<int> pr) {  
    // work  
  
    pr.set_value(9001);  
  
    // other work?  
}
```

- set_value() signals the corresponding futures associated with the given promise

- summary of the first part
- `std::thread` – thread, the lowest abstraction, almost does not restrict the user - suitable for a long lifetime
- `std::jthread` – a variant of `std::thread`, expresses task dependencies better, but can be limiting
- `std::async` – asynchronous task processing (immediate or deferred) – suitable for shorter tasks that do not need more complex synchronization than synchronization over the computation result
- `std::promise` – a means for synchronizing a thread producing a result and threads consuming the result
- `std::future` – consumers of the result wait on this object and retrieve the result

- instance of the current thread: `std::this_thread`
- in the thread context, several static methods can be called on this object
 - `sleep_for()` – sleeps for the specified time
 - `sleep_until()` – sleeps until the specified time
 - `yield()` – gives up the time quantum (see other courses)
 - `get_id()` – obtains the thread identifier (not necessarily the real system ID)

```
using namespace std::chrono_literals;
```

```
std::this_thread::sleep_for(100ms);
```

- C++11 also added basic synchronization primitives
- essentially, they only wrap standard primitives
- C++ only adds further small restrictions to increase usage safety
- again – details including functionality are covered in other courses (KIV/ZOS, OS, PPR, UPP)
- here we will discuss mostly the language implications

Synchronization

- `std::mutex`
- `#include <mutex>`
- classic system mutex
 - mutual exclusion
 - critical section protection
 - has an owner – only whoever locked it can unlock it
- it has its own methods, but they should not be used directly
 - `lock()` – locks the mutex; if it is already locked, blocks until it is released
 - `unlock()` – unlocks the mutex; if it is not locked, or someone else locked it, throws an error
 - `try_lock()` – attempts to lock the mutex; if it fails, returns immediately and indicates failure
- what if we forget to unlock the mutex?

Synchronization

- `std::lock_guard`
- RAII wrapper over locks (and mutexes)
- the constructor locks
- the destructor unlocks

```
std::mutex mtx;
```

```
// for demonstration - scope
{
    std::lock_guard<std::mutex> lck(mtx);

    // critical section
}
```

Synchronization

- e.g. a method that is entirely a critical section

```
class Crit {  
    protected:  
        std::mutex mMtx;  
  
    public:  
        void work()  
        {  
            std::lock_guard<std::mutex> lck(mMtx);  
            // critical section  
        }  
  
        // ...  
};
```

Synchronization

- note: this is one of the few uses of the mutable keyword

```
class Crit {  
    protected:  
        mutable std::mutex mMtx;  
  
    public:  
        void work() const  
        {  
            std::lock_guard<std::mutex> lck(mMtx);  
            // critical section  
        }  
  
        // ...  
};
```

- other variants of RAII wrappers over locks
- `std::unique_lock` – similar to `lock_guard`, but it can be locked lazily and unlocked explicitly
- `std::shared_lock` – shared lock, e.g. for read access
 - in combination with `unique_lock`, it can implement a solution to the “readers-writer” problem
- `std::scoped_lock` – RAII wrapper over multiple locking (we need several locks at once)

- other mutex variants
- `std::recursive_mutex` – a mutex that a thread can lock multiple times
- `std::timed_mutex` – a mutex with a timeout
- `std::shared_mutex` – a shared mutex, implementation of the mentioned “readers-writer” problem
- their combinations

- group locking
- `std::lock`, `std::try_lock`
- several instances of lockable objects as input
- always locks all of them at once
- if any of them is not available, unlocks all of them and blocks until all are available
 - alternatively, `std::try_lock` does not block and returns an error

- condition variable
- `std::condition_variable`
- `#include <condition_variable>`
- again, merely a wrapper over a system condition variable
- it can be waited on
 - `wait()`
 - needs an instance of `std::unique_lock`; it actually contains a critical section
 - unblocks when someone signals (notifies) the same condition variable
 - or because of a *spurious wakeup* (see other courses)
 - provides a way to handle this in the form of an additional condition

- condition variable
- `std::condition_variable`
- it is possible to wait for a certain amount of time
 - `wait_for()`, `wait_until()`
 - returns whether the waiter was awakened due to signaling or timeout expiration
 - suitable, for example, for implementing a timeout over a deferred operation

- condition variable
- `std::condition_variable`
- it is possible to notify (signal) a waiting thread
 - `notify_one()` – wakes one waiter from the queue
 - `notify_all()` – wakes all waiters from the queue

Example - worker thread

```
std::mutex workMtx;
std::condition_variable workCv;
std::queue<Work> workQueue;
std::atomic<bool> running = true;

void process_inputs() {
    while (running) {
        std::unique_lock<std::mutex> lck(workMtx);

        workCv.wait(lck, []() { // waits for work or termination
            return !workQueue.empty() && running;
        });

        if (!running)           // was there a signal because
            break;              // of termination?

        Work w = workQueue.top(); // select work from the queue
        workQueue.pop();

        lck.unlock(); // now nobody can take the work from us,
                     // we can unlock
        process(w);   // and process it safely, so that
    }                 // we do not block the lock unnecessarily
}
```

- condition variable
- it is covered in detail by other courses
- it has many more implications and pitfalls than we mentioned

- `std::atomic`
- template data type with atomic operations
- uses properties of the given architecture
 - e.g. data type properties, transactional memory, bus locking, ...
- fallback to locks if HW support for the given atomic operation is not available

```
std::atomic<int> i;
```

```
i++;
```

Synchronization

- `std::call_once` + `std::once_flag`
- thread-safe one-time execution of an action
- a similar effect can also be achieved with a critical section and a bool flag or atomic variable
 - however, this approach guarantees execution using actually available system facilities in a portable way

```
std::once_flag fl;
```

```
for (int i = 0; i < 10; i++) {  
    std::call_once(fl, [](){  
        std::cout << "Hello_once!" << std::endl;  
    });  
}
```

- note: naturally, it makes sense mainly in a multithreaded environment

Parallel STL

- since C++17, the standard library provides parallel versions of algorithms from the `<algorithm>` library
- it is enough to add a `std::execution` policy as the first parameter
- assuming correct use, parallel processing can therefore be obtained almost for free

```
std::for_each(vec.begin(), vec.end(),  
              [](int a) {  
    // serial execution  
});
```

```
std::for_each(std::execution::par_unseq,  
              vec.begin(), vec.end(),  
              [](int a) {  
    // very likely parallel execution  
});
```

- *Execution policy*
- how will the given algorithm be executed?
- 4 predefined values
 - `std::execution::seq` – sequential (serial)
 - `std::execution::unseq` – unsequenced (serial)
 - `std::execution::par` – sequential parallel
 - `std::execution::par_unseq` – unsequenced parallel

- Parallel STL
- more in other courses
- with the mentioned simple trick, for selected types of problems it is possible to get a speedup essentially almost for free
- a problem may arise if a greater degree of synchronization is needed
- or when we want to have control over threads

- What else does C++ offer?
- atomic variables (`std::atomic`)
- semaphores (`std::counting_semaphore`, `std::binary_semaphore`)
- latches and barriers (`std::latch`, `std::barrier`)
- deferred locking
- obtaining information about the environment
 - number of CPU cores
 - alignment to avoid false sharing (cache usage), etc.
- and more...

- under the hood, these are still system facilities
- e.g. thread implementations on Linux/Mac are still pthreads
 - therefore it is still necessary to add the `-lpthread` flag (or an equivalent) when compiling
 - beware, it may also be necessary to install the `libtbb` package
- we have by no means covered the full topic of threads and synchronization
- we have only introduced, at a surface level, the language and library elements that can be used to implement a parallel program
- more about the principles of parallelization in KIV/ZOS, KIV/OS, KIV/PPR and KIV/UPP