THE NEW C++ STANDARD AND MULTITHREADING

Miroslav Virius
Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze
Miroslav.Virius@fjfi.cvut.cz

ABSTRACT:
Previous C++ standards did not concern the multithreading, neither on the core language level, nor on the standard library level. The new standard, published in 2012, defines several facilities supporting the multithreading. This article gives a short synopsis of these new C++ features and compares them with the multithreading support in Java and .NET languages.

KEYWORDS:
C++11, multithreading, thread class, atomic operations, future, promise, mutex, C++ Standard Library

1 INTRODUCTION

The C++ programming language was first published in the middle of 80's as an object-oriented extension to the C language [1]. The international standard of this programming language was prepared since 1989 to 1997 and its first version was released in 1998 [3]; this standard is known as C++98. The revised version of the standard was published in 2003 and this known as C++03 [3]. There are minor differences between these two C++ versions. The C++03 can be considered as technical correction of the C++98.

1.1 Known issues of the C++03

It was easy to see that the C++ standard [4] is not a finished product. The most apparent failure was that the incomplete standard libraries that did not support some of the most common containers, e.g. the hash tables or the tuples. It did not support the regular expressions, too, and of course it did not support the multithreading.
Note that in the time of the C++ origin, the multithreading was used in special tasks and the multithreading support was not common in programming languages.

1.2 Why is the multithreading necessary now

Since 70's up to the beginning of the new millennium, the speed of the computers grew nearly linearly due to the so called Moore's law [2], thus the programmers were not forced to use the multithreading. Now it seems that the miniaturization of the electronic chips and the ways to increase the computer speed this way has reached its limits posted by the laws of the quantum mechanics. There is a new way to increase the throughput of the processors – to equip the processor with two or more cores. The drawback of this solution is evident: To exploit the power of the contemporary processors, the programmer must be able to use multithreading in his programs.

1.3 Proprietary solutions

The programmers used several ways to solve the missing multithreading support in C++. In early 90's, the most common way was to use the multithreading support offered by the operating system API. Nevertheless, this is very low level solution and it is not portable. Thus, many higher level proprietary multithreading libraries emerged. Some of the most

common are the Boost Multithreading Library [6], the Intel Threading Building Blocks (also known as TBB, [7]), OpenMP by OpenMP Architecture Review Board [8] and many others.
The basic drawback of these solutions is the limited portability, too: Even though the above mentioned libraries are multi-platform ones, the selection of one multithreading library affects the application architecture and the change of the library may lead to multiple changes in the application source code.

2 MULTITHREADING SUPPORT IN THE C++11 STANDARD LIBRARY

The support of the multithreading in C++11 is based on he following tools:

1. The C++ memory model.
2. The C++ program execution model.
3. The C++ Standard Library class `std::thread` representing single thread of execution and related tools.
4. The `std::mutex`, `std::recursive_mutex` and `std::lock_guard` template classes and related tools for mutual exclusive access to resources.
5. The `std::future` template class and related tools representing the expected result of the computation in another thread.
6. The atomic operations library.

3 THE C++ MEMORY AND EXECUTION MODEL

3.1 The Memory Model

The standard [4] explicitly describes the situations, in which the concurrent access to some memory locations is threading safe. This is the case of the fields in the structs, e.g., and especially of the adjacent bit fields, that may be – under some circumstances – part of the same memory unit.

3.2 Conforming implementations

The program execution model of the C++ programming language described in the standard [5] defines an abstract parameterized nondeterministic automaton. The standard imposes only one requirement on the conforming implementations: They are required to hold the observable behavior of the abstract machine described by the standard.
Note that the parameters of the abstract automaton are the implementation-defined aspects, operations, constants etc. An example of such a parameter of the abstract machine is the value of the `sizeof(int)` expression [5].

3.3 Program execution model in the multithreaded environment

The standard imposes some requirements on the implementation concerning the multithreaded execution. Most of them are trivial, but of course they must be listed in the standard. An example of such a statement is the requirement that the implementations should ensure that all unblocked threads eventually make progress. The main purpose of this part of the standard is to define the concepts and the terms used to describe the multithreading support in the standard library.

4 THE THREAD

The `std::thread` class is declared in the `<thread>` header. As well as in many similar libraries, it – in fact – encapsulates one unique thread of execution of the underlying operating system. It provides the mechanism to create new thread and to manage its execution.

4.1 Basic thread features

Here are some basic considerations about the `thread` objects in the C++ Standard Library.

- No two `thread` objects may represent the same thread of execution in the underlying operating system.
- The thread may join another thread, i.e. to wait until the other thread ends his run.
- A thread of execution may be detached – i.e. it may be represented by no `thread` object in the program.
- The `thread` object may be in a state that does not represent any thread of execution in the underlying operating system. It is if the thread is waiting for another thread, e.g., after being constructed by the default constructor etc.
- The `thread` object exposes the access to the handle of the underlying thread of execution, if it is used in the operating system, by the `native_handle()` member function.

4.2 Construction of the thread object

The constructor of the `thread` class is defined using the variadic template:

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

This declaration describes in fact the declaration of a set of constructors with at least one parameter, but in fact with an unlimited number of parameters. The first parameter determines the function to be executed as a body of the thread; the additional parameters will be submitted to the function `F`. The constructed thread starts immediately with the end of the constructor invocation.
The constructor

```
thread() noexcept;
```

constructs a new thread object that represents no thread of execution.

4.3 Management of the thread execution

The `join()` member function causes the current thread to wait for the completion of another thread. The `swap()` member function allows to interchange the status of the `*this` thread with the given one.
In the namespace `std::this_thread` is declared the function `yield()`, that offers the opportunity to run another thread of execution.
The function `sleep_until()` blocks the current thread (i.e. the thread in which it is called) up to a given time point. The `sleep_for()` blocks the current thread for a given time interval.

5 SYNCHRONIZATION

All nontrivial multithreaded programs require the synchronization of the access to the resources shared across the application threads in order to prevent the race condition. The synchronization tools are declared in the `<mutex>` header in C++11 standard library. The basic locking device is represented by the `mutex`, `recursive_mutex`, `timed_mutex` and `recursive_timed_mutex` classes.

5.1 Mutex

The mutex class represents objects that may be owned exclusively by one thread.
If `m` is an object of type `std::mutex` or `std::timed_mutex`, the statement

```
m.lock()
```

blocks the calling thread until the ownership of the mutex `m` can be obtained for the calling thread. When the calling thread gets the ownership of the mutex, it can be rescheduled. Note that the calling thread may not own the mutex `m`.
The `try_lock()` method tries to acquire the lock and returns the value indicating whether it succeeded. It is a non-blocking operation.
The ownership of the mutex of type `mutex` or `std::timed_mutex` by the thread ends, if the owning thread calls the `unlock()` method.

5.2 Other mutex classes

The `std::recursive_mutex` class represents a recursive mutex with exclusive ownership semantics, i.e.

- only one thread may acquire the ownership of this mutex.
- On the other hand, the thread that owns a `recursive_mutex` object, may call `lock()` and acquire additional levels of ownership. The `recursive_mutex` object is released when the `unlock()` corresponding to any `lock()` has been called.

The `timed_mutex` and class `recursive_timed_mutex` classes provide the `try_lock_for()` and `try_lock_until()` methods. The first one blocks the calling thread up to moment it acquires the lock or up to the end of the given time interval – what comes first. The second one blocks the calling thread up to moment it acquires the lock or up to the given time point – what comes first. Both methods return `true` when they succeed.

5.3 Support classes and functions

The Standard C++ Library provides several support tools for the lock management. The template class `lock_guard<>` implements the RAII idiom on the locks. The template parameter is a mutex-type class; instances of this class may own the mutex and the destructor of the `lock_guard<>` unlocks it.
The template function

```
template <class L1, class L2, class... L3>
int try_lock(L1&, L2&, L3&...);
```

expects arguments representing lockable objects. It calls `try_lock()` on each argument. If a call to `try_lock()` returns `false`, this call fails and `unlock()` is called to all the previously locked arguments. The function

```
template <class L1, class L2, class... L3>
void lock(L1&, L2&, L3&...);
```

tries the sequence of calls to `lock()`, `try_lock()`, or `unlock()` on each argument.

5.4 Condition variables

It is necessary to have a tool to block a thread until an event occurs or a condition is met. This is provided by the so called condition variables in the C++ standard library. Class `condition_variable` provides a condition variable that can wait only on an object of type `unique_lock<mutex>`. Class `condition_variable_any` provides a general condition variable that can wait on objects of user-supplied lock types.
Condition variables permit concurrent invocation of member functions like `wait()`, `wait_for()`, `wait_until()`, `notify_one()` and `notify_all()`. The meaning of these functions is similar to analogous functions in other programming languages. Note that the `wait_for()` function waits until the notification comes or until a given time interval elapses, the `wait_until()` function waits until the notification comes or until a given time point is reached. The `notify..()` functions "wakes up" the waiting thread or threads.

5.5 Atomic operations

The atomic operations are low-level constructs that enable the programmer to do some elementary operations on memory locations without the overhead of mutexes. They are declared in the `<atomic>` header in the Standard C++ Library. The `atomic<>` template allows to construct an atomic type based on the template parameter; there are specializations of this template for integral and pointer types in this header. These specializations define atomic operations on atomic types such as the arithmetic operators.

6 FUTURES

The promise and the future concepts are interchanged sometimes, but in the C++ standard library, they have slightly different meaning. Both are constructs that allow retrieving in one thread the result from a function that has run in the same thread or in another thread. The `future` class and related tools are declared in the `<future>` header in the C++ Standard Library.

6.1 Shared state

The main communication tool used by these tools is the *shared state*. It consists of some state information and of the result of the computation. The result may be a value, it may be void and it may be an exception. The result in the shared state may be not yet evaluated; a function in the C++ program may wait (for a given time interval, until a time point is reached, or without any limit) until the shared state contains the evaluated result.

6.2 The asynchronous return object

The term *asynchronous return object* denotes an object that reads results from a shared state. A *waiting function* of an asynchronous return object is a function that potentially blocks to wait for the shared state to contain the result.

6.3 The promise class

The `promise` template class represents the *asynchronous provider*. It is an object that provides a result to a shared state. The constructor of this class constructs a `promise` object and a shared state. The `set_value()` method sets the result to the associated shared state. The `get_future()` member function returns a `future<R>` object with the same shared state as the calling `promise` object.
The promise can share its shared state with other providers.

6.3 The future class

The class template future defines a type for asynchronous return objects which do not share their shared state with other asynchronous return objects. The default constructor of the future class constructs an object without shared state at all. The `wait()` member function blocks until the shared state contains the result. The `get()` member function blocks until the shared state contains the result and then returns the value it contains. If the shared state contains an exception as a result of the computation, this member function throws this exception. The `valid()` member function checks whether the result is available and not yet retrieved by the `get()` method.

The class template `packaged_task` provides a type that wraps a function or callable object so that its return value is stored in a future when it is invoked.

7 THE COMPARISON WITH OTHER PROGRAMMING LANGUAGES

We have discussed the basic tools for the multithreading in C++11 in the previous sections. Now, let us compare it with the tools in other programming languages, viz. in Java and the programming languages for the .NET platform.

On the basic level, defined by the thread class and thread synchronization tools, the C++11 Standard Library offers the same possibilities as the Java or the .NET Languages. It does not provide the constructs like the `suspend()` and `resume()` methods, that are deprecated in the above mentioned languages.

The RAII (Resource Acquisition Is an Initialization) idiom for the locks is supported by the `lock_guard<>` and some other classes; this is a considerable advantage of the C++ implementation.

On the other hand, the Java and .NET languages support higher level concepts like thread pooling that have no analogs in the C++11. The same holds for the "easy parallelization" tools like functions implementing the parallel loops or parallel invocation of sets of functions based on thread pools.

This last point shows that the threading support in the C++ Standard Library is not yet finished and will probably be subject of revisions in future releases.

ACKNOWLEDGEMENT

This work has been supported by the MŠMT grant SGS 11/167.

BIBLIOGRAPHY

[1] Stroustrup, B.: *The Design and Evolution of C++*. AT&T 1994. ISBN 0-201-54330-3.

[2] Moore, G.: Cramming more components onto integrated circuits. *Electronics*, Volume 38, No. 8, 1965

[3] International standard ISO/IEC 14882:1998. *Information technology — Programming languages — C++*. First edition. Genève: ISO, 1998.

[4] *International standard ISO/IEC 14882:2003. Information technology — Programming languages — C++*. Second edition. Genève: ISO, 2003.

[5] *International standard ISO/IEC 14882:2011. Information technology — Programming languages — C++*. Third edition. Genève: ISO, 2011.

[6] *Boost C++ Libraries.* Available at < http://www.boost.org/> (citation April 2, 2012).

[7] *Intel® Threading Building Blocks (Intel® TBB)*. Available at <http://software.intel.com/en-us/articles/intel-tbb/> (citation April 2, 2012).

[8] *OpenMP*. Available at <http://openmp.org/wp/about-openmp/> (citation April 2, 2012).