An extremely brief introduction to C++11 threads for users of pthreads

Marc Paterno
Fermilab/SCD/ADSS/SSI

Contents

1 Major differences between pthreads and C++11

1. **pthreads** is a C library, and was not designed with some issues critical to C++ in mind, most importantly *object lifetimes* and *exceptions*.
2. **pthreads** provides the function `pthread_cancel` to cancel a thread. C++11 provides no equivalent to this.
3. **pthreads** provides control over the size of the stack of created threads; C++11 does not address this issue.[1]
4. C++11 provides the class `thread` as an abstraction for a thread of execution.
5. C++11 provides several classes and class templates for *mutexes*, *condition variables*, and *locks*, intending RAII[2] to be used for their management.
6. C++11 provides a sophisticated set of function and class templates to create callable objects and anonymous functions (lambda expressions) which are integrated into the thread facilities.

[1] On Linux, `setrlimit` is available, but affects all threads in the process.

[2] Resource allocation is initialization.

The use of RAII to control thread resources (including mutexes) cannot be over-emphasized. RAII is at the center of the design of the C++11 thread library and all of its facilities.

2 An example use of C++11 threads

2.1 Starting and joining threads

This example uses the GNU Scientific Library's implementation of QAGS to perform numeric integration of two functions simultaneously. We create two `thread` objects, each controlling a thread-of-execution. Note the ease with which we can pass function arguments to the function to be executed in the thread.

```
int main() {
  const std::size_t limit(100000);      // max intervals for QAGS
  const double low(0.0), high(1000.0); // range of integration
  const double abserr(1e-11);          // absolute error goal

  std::thread t1(integrate, hard, low, high, abserr, limit);
  std::thread t2(integrate, easy, low, high, abserr, limit);

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

Listing 2.1: The main program.

The call to `std::thread::join()` stalls the calling (main) thread-of-execution until the `thread` on which it is called finishes.

2.2 The "thread function"

There is nothing special about the function to be executed by a `thread`, *except* that it is good practice to prevent it from exiting on an exception, which would result in a call to `std::terminate`.

```
// FUNC is the kind of function GSL knows how to integrate. The
// void* is how GSL passes extra arguments; we will not need it.
typedef double (FUNC)(double, void*);
std::mutex G_COUT_MUTEX; // This is global, like std::cout.

// Calculate the integral of f from low to high, to absolute
// precision abs, limiting the workspace size to limit.
void integrate(FUNC* f, double low, double high, double abs,
               std::size_t limit) {
  { // This scope exists only to control the lifetime of lck.
    std::lock_guard<std::mutex> lck(G_COUT_MUTEX);
    std::cout << "Starting integration in thread "
              << std::this_thread::get_id() << std::endl;
```

```
  }
  Workspace w(limit);  // To be used by GSL's QAGS
  double result(0.0), error(0.0);
  gsl_function func {f, 0}; // struct defined by GSL
  gsl_integration_qags(&func, low, high, abs, 0.0, limit,
                       w.get(), &result, &error);

  std::lock_guard<std::mutex> lck(G_COUT_MUTEX);
  std::cout << "In thread: " << std::this_thread::get_id()
            << " result: " << result
            << " error: " << error
            << " intervals: " << w.size()
            << std::endl;
}
```

Listing 2.2: The thread function.

Note how the lifetimes of objects are used to control the acquisition and release of the mutex. C++ strictly defines the lifetimes of created objects; rely on them!

Note also GSL's technique for obtaining thread safety *without locks*: pass to a function all the data it uses, rather than using `static` data.

2.3 The sentry class `Workspace`

A *sentry* object is an object whose lifetime controls some resource. A sentry class is the class of such an object. This one encapsulates the workspace type for the GSL function we use.

```
class Workspace {
private:
  gsl_integration_workspace* ws_;
public:
  explicit Workspace(std::size_t lim);
  Workspace(Workspace const&) = delete;            // no copy
  Workspace& operator=(Workspace const&) = delete; // no assignment
  ~Workspace();
  std::size_t size() const;
  gsl_integration_workspace* get();
};

Workspace::Workspace(std::size_t lim) :
   ws_(gsl_integration_workspace_alloc(lim)) { }
Workspace::~Workspace() { gsl_integration_workspace_free(ws_); }
std::size_t Workspace::size() const { return ws_->size; }
gsl_integration_workspace* Workspace::get() { return ws_; }
```

Listing 2.3: The sentry class `Workspace`.

2.4 The functions we integrate

```
// These are the two functions we will integrate.
double easy(double x, void*) { return std::log(x)/std::sqrt(x); }
double hard(double x, void*) { return std::sin(100*x); }
```

Listing 2.4: The integrands `easy` and `hard`.

The unnamed `void*` second argument is forced upon us by the design of GSL; it is ignored.

2.5 The result

All this code is in one file: `ex04.cc`. To compile, you need a C++11 compiler (GCC 4.7.1 is close enough) and the GSL library.

Compile with:

```
g++ -O3 -std=c++11 -Wall -pedantic -Werror -o ex04 ex04.cc -lgsl
```

On Linux, you may also need to include `-lgslcblas -lpthread`; the requirement to name `-lpthread` appears to me to be a bug in GCC 4.7.1.

The result of execution is (extra line breaks added to fit on this page):

```
Starting integration in thread 0x100581000
Starting integration in thread 0x100781000
In thread: 0x100781000 result: 310.394 error: 9.83391e-12
                       intervals: 9
In thread: 0x100581000 result: 0.0199936 error: 9.99855e-12
                       intervals: 16346
```

On your machine, the printed value of `std::thread_id` is probably different.

Note that the thread we *started* first *finished* last.

3 Futures

The class `std::future` can be used to encapsulate a function run in its own thread of execution, and to obtain its return value (or an exception it throws). The function template `std::async` is used to create the `future`; the enumeration values `std::launch::async` and `std::launch::deferred` determine when the thread-of-execution begins.

```
#include <future>
#include <iostream>
#include <string>

int f() { return 1; }
int g(const char* msg) { throw std::string(msg); }

int main() {
  std::future<int> a = std::async(std::launch::deferred, f);
```

```
  std::future<int> b = std::async(std::launch::async, g, "cold");
  std::cout << a.get() << std::endl;
  try { std::cout << b.get() << std::endl; }
  catch (std::string& s)
    { std::cout << "Caught " << s << " from b" << std::endl; }
}
```

Listing 3.1: Simple use of `future`.

4 What I could not cover in 12 minutes

There are many other things of interest for multithreaded programming C++11. *Some* of them are:

- Additional mutex and lock types, and locking strategies, *e.g.*, `std::recursive_mutex`, `std::timed_mutex`; `std::unique_lock`; `std::defer_lock`, `std::try_to_lock`.
- Condition variables (some uses of POSIX condition variables are better replaced by `std::future`).
- Class templates `duration` and `time_point`, used in all time-related interfaces, *e.g.*, `std::this_thread::sleep_for` and `std::this_thread_sleep_until`.
- Atomic types and functions on atomic types.
- Memory fence functions to for memory-ordering between operations.
- Variadic templates (which enable the simple means of passing arguments to a thread function)
- Lambda expressions (anonymous closure objects), which can be used in place of functions.
- *rvalue* references, which enable perfect forwarding to a thread function
- Additional support for function objects, *e.g.*, from `std::function` and `std::bind`.

5 What's missing in GCC 4.7.1

GCC 4.7.1, when used with the `-std=c++11` flag, support *much* but not *all* of C++11. The full feature matrix is available at `http://gcc.gnu.org/gcc-4.7/cxx0x_status.html`.

The important items related to concurrency are:

- No support of thread-local storage.
- Very limited support for atomics.
- The new memory model is not yet implemented.

To access the `std::this_thread::sleep_until` and `std::this_thread::sleep_for` functions requires using `-D_GLIBCXX_USE_NANOSLEEP` on some platforms.

6 References

There are many online references available. Ones I have used include:

- http://en.cppreference.com/w/cpp/thread.
- The C++ committee public web site: http://www.open-std.org/jtc1/sc22/wg21.

My favorite book on the subject is **C++ Concurrency in Action**, by Anthony Williams, ISBN 1933988770.