Concurrent programming in C++11

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

Motivation

- C++11 (ISO/IEC 14882:2011) offers its own concurrency model.
  - Minor revisions in C++14.
  - More expected for C++17.
- Any compliant implementation must supply it.
  - Solves inherent problems from **PThreads**.
  - Portable concurrent code: **Windows**, **POSIX**, . . .
- **Implications**:
  - Changes in the **language**.
  - Changes in the **standard library**.
- Influence on C11 (ISO/IEC 9899:2011).
- **Important**: Concurrency and parallelism are two **related** but **distinct** concepts.

Structure

- C++ **language offers**:
  - A new **memory model**.
  - **thread_local** variables.
- C++ **standard library offers**:
  - **Atomic types**.
    - Useful for portable **lock free programming**.
  - **Portable abstractions** for concurrency.
    - **thread**.
    - **mutex**.
    - **lock**.
    - **packaged_task**.
    - **future**.

2 Library overview

- Threads
- Access to shared data
- Waiting
- Asynchronous execution

Thread launching

- A **thread** represented by class **std::thread**.
  - Usually represents an OS thread.

Launching a thread from a function

```
void f1();
void f2();

void g() {
  thread t1{f1}; // Launches thread executing f1()
  thread t2{f2}; // Launches thread executing f2()

  t1.join(); // Waits until t1 terminates.
  t2.join(); // Waits until t2 terminates.
}
```

Shared objects

- Two threads may access to a **shared object**.
- Possibility for **data races**.

Access to shared variables

```
int x = 42;

void f() { ++x; }
void g() { x=0; }
void h() { cout << "Hello" << endl; }
void i() { cout << "Bye" << endl; }

void race() {
  thread t1{f}; thread t2{g};
  t1.join(); t2.join();

  thread t3{h}; thread t4{i};
  t3.join(); t4.join();
}
```

Argument passing

- **Simplified argument passing** without needing any **casts**.

Argument passing

```
void f1(int x);
void f2(double x, double y);

void g() {
  thread t1{f1, 10}; // Runs f1(10)
  thread t2{f1}; // Error
  thread t3{f2, 1.0} // Error
  thread t4{f2, 1.0, 1.0}; // Runs f2(1.0,1.0)
  // ...
  // Thread joins
```

Threads and function objects

- **Function object**: Object that can be invoked as a function.
- **operator ()** *overload/redefinition*.

Function object in a thread

```
struct myfunc {
  myfunc(int val) : x{val} {} // Constructor. Initializes object.
  void operator()() { do_something(x); } // Redefine operator()
  int x;
};

void g() {
  myfunc f1{10}; // Constructs object f1
  f1 (); // Invokes call operator f1.operator()
  thread t1{f1}; // Runs f1() in a thread
  thread t2{myfunc{20}}; // Construct temporal and invokes it
  // ...
  // Threads joins
```

2 Library overview
- Threads
- Access to shared data
- Waiting
- Asynchronous execution

Mutual exclusion

- **mutex** allows to control access with **mutual exclusion to a resource**.
  - **lock()**: **Acquires** associated lock.
  - **unlock()**: **Releases** associated lock.

Use of mutex

```
mutex m;
int x = 0;

void f() {
  m.lock();
  ++x;
  m.unlock();
}
```

Launching threads

```
void g() {
  thread t1(f);
  thread t2(f);
  t1.join();
  t2.join();
  cout << x << endl;
}
```

Problems with `lock()/unlock()`

- **Possible problems**:
  - Forgetting to **release a lock**.
  - **Exceptions**.
- Solution: **unique_lock**.
  - **Pattern**: **RAII** (Resource Acquisition Is Initialization).

Automatic lock

```
mutex m;
int x = 0;

void f() {
  // Acquires lock
  unique_lock<mutex> l{m};
  ++x;
} // Releases lock
```

Launching threads

```
void g() {
  thread t1(f);
  thread t2(f);
  t1.join();
  t2.join();

  cout << x << endl;
}
```

Acquiring multiple `mutex`

- **lock()** allows for acquiring simultaneously several **mutex**.
  - Acquires all or none.
  - If some is blocked it waits releasing all of them.

Multiple acquisition

```
mutex m1, m2, m3;

void f() {
  lock(m1, m2, m3);

  // Access to shared data

  // Beware: Locks are not released
  m1.unlock();
  m2.unlock();
  m3.unlock()
}
```

Acquiring multiple `mutex`

- Specially useful in cooperation with **unique_lock**

Multiple automatic acquisition

```
void f() {
  unique_lock l1{m1, defer_lock};
  unique_lock l2{m2, defer_lock};
  unique_lock l3{m3, defer_lock};

  lock(l1, l2, l3);
  // Access to shared data

} // Automatic release
```

2 Library overview

- Threads
- Access to shared data
- **Waiting**
- Asynchronous execution

Timed waiting

- Access to clock:

```
using namespace std::chrono;
auto t1 = high_resolution_clock::now();
```

- Time difference:

```
auto dif = duration_cast<nanoseconds>(t2−t1);
cout << dif.count() << endl;
```

- Specifying a wait:

```
this_thread::sleep_for(microseconds{500});
```

Condition variables

- Mechanism to synchronize threads when accessing shared resources.
  - **wait()**: Wait on a **mutex**.
  - **notify_one()**: Awakens a waiting thread.
  - **notify_all()**: Awakens all waiting threads.

Producer/Consumer

```
class request;

queue<request> q; // Requests queue
condition_variable cv; //
mutex m;

void producer();
void consumer();
```

Consumer

```
void consumer() {
  for (;;) {
    unique_lock<mutex> l{m};

    while (q.empty()) {
      cv.wait(l);
    }

    auto r = q.front();
    q.pop();
    l.unlock();

    process(r);
  };
}
```

- Effect of **wait**:
  1. Releases lock and waits a notification.
  2. Acquires the lock when awaken.

Producer

```
void producer() {
  for (;;) {
    request r = generate();

    unique_lock<mutex> l{m};
    q.push(r);

    cv.notify_one();
  }
}
```

- Effects of **notify_one()**:
  1. Awakes to one thread waiting on the condition.

2 Library overview

- Threads
- Access to shared data
- Waiting
- Asynchronous execution

Asynchronous execution and futures

- An asynchronous task allows simple launching of a task execution:
  - In a different thread of execution.
  - As a deferred task.

- A **future** is an object allowing that a thread can return a value to the code section that invoked it.

Asynchronous tasks invocation

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

int main() {
  std::future<int> r = std::async(task, 1, 10);
  other_task();
  std::cout << "Result= " << r.get() << std::endl;
  return 0;
}
```

Using futures

- **General idea**:
  - When a thread needs to pass a value to another thread it sets the value into a **promise**.
  - Implementation takes care that the value is available in the corresponding **future**.

- Access to the **future** through **f.get()**:
  - If a value has been assigned $\rightarrow$ it gets that value.
  - In other case $\rightarrow$ calling thread blocks until it is available.
  - Allows to transparently transfer exceptions among threads.

Class `thread`

- Abstraction of a *thread* represented through class **thread**.
- One-to-one correspondence with operating system thread.
- All threads in an application run in the same address space.
- Each thread has its own stack.
- **Dangers**:
  - Pass a pointer or a non-const reference to another thread.
  - Pass a reference through capture in lambda expressions.
- **thread** represents a link to a system thread.
  - Cannot be copied.
  - They can be moved.

Thread construction

- A thread is constructed from a function and arguments that mast be passed to that function.
  - Template with variable number of arguments.
  - Type safe.

Example

```
void f();
void g(int, double);

thread t1{f}; // OK
thread t2{f, 1}; // Error: Too many arguments

thread t3{g, 1, 0.5}; // OK
thread t4{g}; // Error: Missing arguments
thread t5{g, 1, "Hello"}; // Error: Wrong types
```

Construction and references

- Constructor of **thread** is a template with variable number of arguments.

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

- Arguments passing to a thread is by value.
- To force passing by reference:
  - Use a helper function for **reference_wrapper**.
  - Use lambdas and reference captures.

```
void f(record & r);

void g(record & s) {
  thread t1{f,s}; // Error if f takes a reference
  thread t2{f, ref(s)}; // OK. Reference to s
  thread t3{[&] { f(s); }}; // Reference to s
}
```

Two-phase construction

- Construction includes thread launching.
  - There is no separate operation to *start* execution.

Producer/Consumer

```
struct producer {
  producer(queue<request> & q);
  void operator()();
  // ...
};

struct consumer {
  consumer(queue<request> & q);
  void operator()();
  // ...
};
```

Stages

```
void f() {
  // Stage 1: Construction
  queue<request> q;
  producer prod{q};
  consumer cons{q};

  // Stage 2: Launching
  thread tp{prod};
  thread tc{cons};

  // ...
```

Empty thread

- Default constructor creates a thread without associated execution task.

```
thread() noexcept;
```

- Useful in combination with move constructor.

```
thread(thread &&) noexcept;
```

- An execution takes can be moved from a thread to another thread.
  - Original thread remains without associated execution task.

```
thread create_task();
thread t1 = create_task();
thread t2 = move(t1); // t1 is empty now
```

Thread identity

- Each thread has a unique identifier.
  - Type: **thread::id**.
  - If the **thread** is not associated with a thread **get_id()** returns **id{}**.
  - Current thread identifier is obtained with **this_thread::get_id()**.

- **t.get_id()** returns **id{}** if:
  - An execution task has not been assigned to it.
  - It has finished.
  - Task has been moved to another **thread**.
  - It has been detached (**detach()**).

Operations on thread::id

- Is an implementation dependent type, but it must allow:
  - Copying.
  - Comparison operators (**==**, **<**, ...).
  - Output to streams through operator **«**.
  - *hash* transformation through specialization **hash<thread::id>**.

Example

```
void print_id(thread & t) {
  if (t.get_id() == id{})
    cout << "Invalid thread" << endl;
  else {
    cout << "Current thread: " << this_thread::get_id() << endl;
    cout << "Received thread: " << t.get_id() << endl;
  }
}
```

Joining

- When a thread wants to wait for other thread termination, it may use operation **join()**.
  - **t.join()** → waits until **t** has finished.

Example

```
void f() {
  vector<thread> vt;
  for (int i=0; i< 8; ++i) {
    vt.push_back(thread(f,i));
  }

  for (auto & t : vt) { // Waits until all threads have finished
    t.join();
  }
}
```

Periodic tasks

Initial idea

```
void update_bar() {
  while (!task_has_finished()) {
    this_thread::sleep_for(chrono::seconds(1))
    update_progress();
  }
}

void f() {
  thread t{update_bar};
  t.join();
}
```

- Problems?

What if I forget `join`?

- When scope where thread was defined is exited, its destructor is invoked.
- **Problem**:
  - Link with operating system thread might be lost.
  - System thread goes on running but cannot be accessed.
- If **join()** was not called, destructor invokes **terminate()**.

Example

```
void update() {
  for (;;) {
    show_clock(steady_clock::now());
    this_thread::sleep_for(second{1});
  }
}

void f() {
  thread t{update};
}
```

- **terminate()** is called when exiting **f()**.

Destruction

- **Goal**: Avoid a thread to survive its **thread** object.
- **Solution**: If a **thread** is *joinable* its destructor invokes **terminate()**.
  - A **thread** is joinable if it is linked to a system thread.

Example

```
void check() {
  for (;;) {
    check_state();
    this_thread::sleep_for(second{10});
  }
}

void f() {
  thread t{check};
} // Destruction without join() -> Invokes terminate()
```

Problems with destruction

Example

```
void f();
void g();

void example() {
  thread t1{f};  // Thread running task f
  thread t2;  // Empty thread

  if (mode == mode1) {
    thread tg {g};
    // ...
    t2 = move(tg); // tg empty, t2 running g()
  }

  vector<int> v{10000}; // Might throw exceptions
  t1.join();
  t2.join();
}
```

- What if constructor of **v** throws an exception?
- What if end of example is reached with **mode==mode1**?

Automatic thread

- RAII pattern can be used.
  - Resource Acquisition Is Initialization.

A joining thread

```
struct auto_thread : thread {
  using thread::thread; // All thread constructors
  ~auto_thread() {
    if (joinable()) join();
  }
};
```

- Constructor acquires resource.
- Destructor releases resource.
- Avoids resource leakage.

Simplifying with RAII

- Simpler code and higher safety.

Example

```
void example() {
  auto_thread t1{f};  // Thread running task f
  auto_thread t2;  // Empty thread

   if (modo == mode1) {
     auto_thread tg {g};
     //  ...
     t2 = move(tg); // tg empty, t2 running g()
  }

  vector<int> v{10000}; // Might throw exceptions
}
```

Detached threads

- A thread can be specified to go on running after destructor, with **detach()**.
- Useful for task running as daemons.

Example

```
void update() {
  for (;;) {
    show_clock(steady_clock::now());
    this_thread::sleep_for(second{1});
  }
}

void f() {
  thread t{update};
  t.detach();
}
```

Problems with detached threads

- **Drawbacks**:
  - Control of active threads is lost.
  - Uncertain whether the result generated by a thread can be used.
  - Uncertain whether a thread has released its resources.
  - Access to objects that might have already been destroyed.

- Recommendations:
  - Avoid using detached threads.
  - Move threads to other scope (via return value).
  - Move threads to a container in a larger scope.

A hard to catch bug

- Problem: Access to local variables from a detached thread after destruction.

Example

```
void g() {
  double x = 0;
  thread t{[&x]{ f1(); x = f2();}}; // If g has finished −> Problem
  t.detach();
}
```

Operations on current thread

- Operations on current thread as global functions in name subspace **this_thread**.
  - **get_id()**: Gets identifier from current thread.
  - **yield()**: Allows potential selection of another thread for execution.
  - **sleep_until(t)**: Wait until a certain point in time.
  - **sleep_for(d)**: Wait for a given duration of time.
- Timed waits:
  - If clock can be modified, **wait_until()** is affected.
  - If clock can be modified, **wait_for()** is **not** affected.

Thread local variables

- Alternative to **static** as storage specifier: **thread_local**.
  - A variable **static** has a single shared copy for all threads.
  - A variable **thread_local** has a per thread copy.
- Lifetime: *thread storage duration*.
  - Starts before its first usage in thread.
  - Destroyed upon thread exit.
- Reasons to used thread local storage:
  - Transform data from static storage to thread local storage.
  - Keep data caches to be thread local (exclusive access).
    - Important in machines with separate caches and coherence protocols.

A function with computation caching

```
thread_local map<int, int> cache;

int compute_key(int x) {
  auto i = cache.find(x);
  if (i != cache.end()) return i->second;
  return cache[arg] = slow_and_complex_algorithm(arg);
}

vector<int> generate_list(vector<int> v) {
  vector<int> r;
  for (auto x : v) {
    r.push_back(compute_key(x));
  }
}
```

- Avoids need for synchronization.
- Some computations might be repeated in multiple threads.

4 Mutex objects and condition variables

- Mutex objects
- Condition variables

mutex classification

- Represent exclusive access to a resource.
  - **mutex**: Basic non-recursive *mutex*.
  - **recursive_mutex**: A *mutex* that can be acquired more than once from the same thread.
  - **timed_mutex**: Non-recursive *mutex* with timed operations.
  - **recursive_timed_mutex**: Recursive *mutex* with timed operations.
- Only a thread can own a *mutex* at a given time.
  - Acquire a *mutex* $\rightarrow$ Get exclusive access to object.
    - Blocking operation.
  - Release a *mutex* $\rightarrow$ Release exclusive access to object.
    - Allows another thread to get access.

Operations

- Construction and destruction:
  - Can be default constructed.
  - Cannot be neither copied nor moved.
  - Destructor may lead to undefined behavior if *mutex* is not free.
- Acquire and release:
  - **m.lock()**: Acquires *mutex* in a blocking mode.
  - **m.unlock()**: Releases *mutex*.
  - **r = m.try_lock()**: Tries to acquire *mutex*, returning success indication.
- Others:
  - **h = m.native_handle()**: Returns platform dependent identifier of type **native_handle_type**.

Example

Exclusive access

```cpp
mutex mutex_output;

void print(int x) {
  mutex_output.lock();
  cout << x << endl;
  mutex_output.unlock();
}

void print(double x) {
  mutex_output.lock();
  cout << x << endl;
  mutex_output.unlock();
}
```

Threads launch

```cpp
void f() {
  thread t1{print, 10};
  thread t2(print, 5.5};
  thread t3(print, 3);

  t1.join();
  t2.join();
  t3.join();
}
```

Errors in mutual exclusion

- In case of error exception **system_error** is thrown.
- Error codes:
  - **resource_deadlock_would_occur**.
  - **resource_unavailable_try_again**.
  - **operation_not_permitted**.
  - **device_or_resource_busy**.
  - **invalid_argument**.

```
mutex m;
try {
  m.lock();
  //
  m.lock();
}
catch (system_error & e) {
  cerr << e.what() << endl;
  cerr << e.code() << endl;
}
```

Deadlines

- Operations supported by **timed_mutex** and **recursive_timed_mutex**.

- Add acquire operations with indication of deadlines.
  - **r = m.try_lock_for(d)**: Try to acquire *mutex* for a duration **d**, returning success indication.
  - **r = m.try_lock_until(t)**: Try to acquire *mutex* until a point in time returning success indication.

4 Mutex objects and condition variables

- Mutex objects
- Condition variables

Condition variables

- Synchronizing operations among threads.
- Optimized for class **mutex** (alternative **condition_variable_any**)).
- Construction and destruction:
  - **condition_variable c{}**: Creates a condition variable
    - Might throw **system_error**.
  - Destructor: Destroys condition variable.
    - Requires no thread is waiting on condition.
  - Cannot be neither copied nor moved.
  - Before destruction all threads blocked in variable need to be notified.
    - Or they could be blocked forever.

Notification/waiting operations

- Notification:
  - **c.notify_one()**: Wakes up one of waiting threads.
  - **c.notify_all()**: Wakes up all waiting threads.
- Unconditional waiting (**l** of type **unique_lock<mutex>**):
  - **c.wait(l)**: Blocks until it gets to acquire lock **l**.
  - **c.wait_until(l,t)**: Blocks until it gets to acquire lock **l** or time **t** is reached.
  - **c.wait_for(l,t)**: Blocks until it gets to acquire lock **l** or duration **d** elapses.
- Waiting with predicates.
  - Takes as additional arguments a predicate that must be satisfied.

Revisiting producer/consumer

Predicate injection in wait

```
void consumer() {
  for (;;) {
    unique_lock<mutex> l{m};

    cv.wait(l, [this]{ return !q.empty();});

    auto r = q.front();
    q.pop();
    l.unlock();

    process(r);
  };
}
```

Summary

- C++ offers a concurrency model through a combination of language and library.
- Class **thread** abstracts an OS thread.
- Synchronization through a combination of **mutex** and **condition_variable**.
- **std::async** offers a high-level mechanism to run threads.
- **std::future** allows result and exceptions transfer among threads.
- **thread_local** offer portable support for thread local storage.

References

- *C++ Concurrency in Action. Practical multithreading.*
  Anthony Williams.
  Chapters 2, 3, and 4.

Concurrent programming in C++11

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid