Threads and Asynchronous Programming; Boost ASIO

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Autumn 2020

Parallel Programming

"Concurrent execution of instructions at the same time."

- **shared memory**
  - processes
  - **threads**
- distributed memory

Parallel Programming

"Concurrent execution of instructions at the same time."

- **shared memory**
  - processes
  - **threads**
- distributed memory

- more difficult than sequential programming
  - deadlocks
  - data consistency
  - extremely hard to debug
  - awareness of the memory model required
- since C++11 the memory model is defined by the standard:
  http://en.cppreference.com/w/cpp/language/memory_model

Threads

- `#include <thread>`
- lightweight synopsis:

```
struct thread {
    thread(); // does nothing
    template< typename F, typename... Args >
    thread( F, Args &&... ); // starts a new thread
    void join(); // waits until it ends
    ...
};
```

- arguments are copied
- if references are needed, they need to be wrapped in `std::ref`/`std::cref`
  - similar to `std::bind`

Threads

- the main thread has to wait for all created threads
  - ... unless the thread is detached
- threads cannot be copied
  - the ownership can be moved
- not RAII-friendly class
  - `join` has to be called manually
  - `std::terminate` is called otherwise
  - explanation and discussion: https://akrzemi1.wordpress.com/2012/11/14/not-using-stdthread/
  - (`thread` is a low-level abstraction)
  - main point: write your own RAII wrappers
- add flag `-pthread` to the compiler on POSIX systems

Threads

```
int fibonacci( int n ) {...}
void write( int n ) {
    std::cout << fibonacci( n ) << std::endl;
}
int main() {
    std::thread t1( write, 14 );
    std::thread t2( write, 40 );
    t1.join();
    t2.join();
}
```

Reminder – Nearly Nothing is Atomic

```
volatile int val;
void inc( int n ) { for ( int i = 0; i != n; i++ ) val++; }
void dec( int n ) { for ( int i = 0; i != n; i++ ) val--; }
int main() {
    std::thread t1( inc, 1000000 );
    std::thread t2( dec, 1000000 );
    t1.join(); t2.join();
    std::cout << "Value is " << val << "\n";
}
```

```
for i in `seq 4`; do ./nonatomic; done
```

produces something like:

```
Value is -43996
Value is 10625
Value is -177065
Value is 13246
```

Working with Memory

- access to memory needs to be guarded
  - **mutual exclusion** devices (`#include <mutex>`)
    - simple `std::mutex`
    - `std::recursive_mutex`
    - `std::timed_mutex`
    - `std::shared_mutex` *(C++17)*
  - RAII-style mechanisms
    - simple `std::lock_guard`
    - `std::unique_lock`
  - deadlock prevention
    - `std::lock`
    - `std::scoped_lock` *(C++17)*
  - atomic primitives (`#include <atomic>`)
    - advanced topic
- thread synchronization
  - condition variables (`#include <condition_variable>`)

Mutex Example

in most cases `std::mutex` should be manipulated using RAII helpers:

- `std::lock_guard` – locked at construction, unlocked in destructor, no other operations
- `std::unique_lock` – also supports explicit unlocking, can be moved, used with condition variables

Mutex Example

in most cases `std::mutex` should be manipulated using RAII helpers:

- `std::lock_guard` – locked at construction, unlocked in destructor, no other operations
- `std::unique_lock` – also supports explicit unlocking, can be moved, used with condition variables

```
std::mutex mutex;
std::vector< Item > work; size_t next = 0;
Item *getNextItem() {
    std::lock_guard< std::mutex > lock( mutex );
    return work.size() <= next ? nullptr : &work[next++];
}
void worker() {
    while ( Item *item = getNextItem() )
        item->doWork();
}
```

Locking Multiple Mutexes – `std::lock`

- bad idea (why?)

```
void worker() {
    std::lock_guard lock1( some_mutex );
    std::lock_guard lock2( other_mutex );
}
```

Locking Multiple Mutexes – `std::lock`

- bad idea (why?)

```
void worker() {
    std::lock_guard lock1( some_mutex );
    std::lock_guard lock2( other_mutex );
}
```

- `std::lock` ensures mutexes are locked in some reasonable order such that circular waiting is avoided

```
void transferMoney( Account &from, Account &to, int amount )
{
    std::lock( from.mutex, to.mutex );
    std::lock_guard< std::mutex >
        lf( from.mutex, std::adopt_lock ),
        lt( to.mutex,   std::adopt_lock );
    from.withdraw( amount );
    to.deposit( amount );
}
```

Locking Multiple Mutexes – `std::scoped_lock` (C++17)

- `std::scoped_lock` is variadic and can lock multiple mutexes in reasonable order

```
void transferMoney( Account &from, Account &to, int amount )
{
    std::scoped_lock guard( from.mutex, to.mutex );
    from.withdraw( amount );
    to.deposit( amount );
}
```

- note: automatic class template deduction happens here

Condition Variables

`std::condition_variable`

- block a thread until a shared variable (*condition*) is modified
  - the condition variable is protected by a mutex
- the thread that wants to modify the variable:
  - locks the mutex (e.g. using `std::lock_guard`)
  - modifies the variable
  - unlocks the mutex
  - executes `notify_one` or `notify_all` on the condition variable
- the thread that wants to wait on the condition variable:
  - acquires a `std::unique_lock` on the mutex
  - executes `wait`, `wait_for`, or `wait_until` (this releases the mutex)
  - is then awakened by
    - notification of the condition variable
    - timeout (in the `wait_for` or `wait_until` case)
    - a spurious wakeup (need to check the condition!)

Condition Variable Example – Barrier

```
struct Barrier {
  Barrier(int w) : _workers(w), _arrived(0) {}
  void wait() {
    std::unique_lock< std::mutex > lk(_mtx);
    if (++_arrived == _workers) {
      lk.unlock();
      _cv.notify_all();
    } else {
      _cv.wait(lk, [this]{ return _arrived == _workers; });
    }
  }
private:
  const int _workers;
  int _arrived;
  std::condition_variable _cv;
  std::mutex _mtx;
};
```

Note on Concurrent Memory Access

- concurrent access to the same memory location is undefined behaviour unless any synchronization mechanism is used
- for now, the only synchronization mechanism is mutex
- using the `volatile` specifier is not enough
  - does not say anything about other memory locations

Note on Concurrent Memory Access

- concurrent access to the same memory location is undefined behaviour unless any synchronization mechanism is used
- for now, the only synchronization mechanism is mutex
- using the `volatile` specifier is not enough
  - does not say anything about other memory locations
  - does not ensure order of writes visible to other threads is the same as they were performed in
    - actual order can differ thanks to memory model

Note on Concurrent Memory Access

- concurrent access to the same memory location is undefined behaviour unless any synchronization mechanism is used
- for now, the only synchronization mechanism is mutex
- using the `volatile` specifier is not enough
  - does not say anything about other memory locations
  - does not ensure order of writes visible to other threads is the same as they were performed in
    - actual order can differ thanks to memory model
  - `volatile` just forbids compiler to optimize out loads and stores from/to the variable
  - might be enough in special cases on microcontrollers:
    - e.g. the main routine waits for a value change from an interrupt
    - beware of bit-width and atomicity

Atomic Operations, `std::atomic`

- in some cases using mutexes gives too much performance penalty
- atomic operations can be used for simple actions (exchange of values, addition to integer...)

Atomic Operations, `std::atomic`

- in some cases using mutexes gives too much performance penalty
- atomic operations can be used for simple actions (exchange of values, addition to integer…)
- `std::atomic` can be used to represent an atomic value that supports these operations
  - specialization for integers that provides atomic arithmetic operations
  - might not be actually atomic, on platforms that do not support it it can use locks
- `std::atomic_flag` – a very simple atomic boolean, can be only cleared or set to `true` in which case it returns its previous value
  - useful for signalling from signal handlers

Atomic Operations, `std::atomic`

- in some cases using mutexes gives too much performance penalty
- atomic operations can be used for simple actions (exchange of values, addition to integer…)
- `std::atomic` can be used to represent an atomic value that supports these operations
  - specialization for integers that provides atomic arithmetic operations
  - might not be actually atomic, on platforms that do not support it it can use locks
- `std::atomic_flag` – a very simple atomic boolean, can be only cleared or set to **`true`** in which case it returns its previous value
  - useful for signalling from signal handlers
- overall, atomics and lock-free programming go far beyond the scope of this lecture

Asynchronous Programming

- modern approach, widely used e.g. in JavaScript
  - avoid using threads directly
  - increase system performance (mainly in systems with a lot of IO)
- standard library implementation is (as of C++17) quite poor
  - allows only for async tasks, result has to be obtained synchronously
  - no composition of asynchronous tasks
  - standard implementation offers only heavy threads
- other approaches
  - Folly (Facebook Open-Source Library)
  - boost::future
  - C++2z standard proposals

Future in the Standard Library

- `#include <future>`
- mainly used to run a work in thread and obtain result later
- `std::promise< T >`
  - for storing a result of asynchronous computation (value or exception)
  - `get_future()` – obtain a `future` through which the result can be awaited and obtained

Future in the Standard Library

- `#include <future>`
- mainly used to run a work in thread and obtain result later
- `std::promise< T >`
  - for storing a result of asynchronous computation (value or exception)
  - `get_future()` – obtain a `future` through which the result can be awaited and obtained
- `std::future< T >`
  - `get()` – returns value or throws an exception (may block)
  - `wait()` – wait for fulfillment of the promise (may block)
  - `wait_for()` – wait for fulfillment or timeout (may block)

Future in the Standard Library

- `#include <future>`
- mainly used to run a work in thread and obtain result later
- `std::promise< T >`
  - for storing a result of asynchronous computation (value or exception)
  - `get_future()` – obtain a `future` through which the result can be awaited and obtained
- `std::future< T >`
  - `get()` – returns value or throws an exception (may block)
  - `wait()` – wait for fulfillment of the promise (may block)
  - `wait_for()` – wait for fulfillment or timeout (may block)
- `std::async( Function, Args... )`
  - runs a function with given args asynchronously (in a thread)
  - returns a `std::future` with the result
  - the returned future's destructor blocks!

std::async Example

```
Config cfg;
// std::future<int>
auto handle = std::async( std::launch::async,
     [&] { return cfg.load( "app.conf" ); } );
doSomething();
try {
    // wait until config is loaded
    int result = handle.get();
} catch ( std::exception &e ) {
    // if cfg.load throws
    std::cerr << e.what() << std::endl;
}
```

std::async Caveat

- what is wrong here?

```
void f() { ... }
void g() { ... }

int main() {
        std::async(std::launch::async, f);
        std::async(std::launch::async, g);
}
```

`std::async` Caveat

- what is wrong here?

```
void f() { ... }
void g() { ... }

int main() {
        std::async(std::launch::async, f);
        std::async(std::launch::async, g);
}
```

- the `std::future` returned by `std::async` gets immediately destructed and blocks until the function returns

Boost ASIO

- multi-platform asynchronous IO
  - network connection
  - serial line
  - file handles (only on Windows)

Boost ASIO

- multi-platform asynchronous IO
  - network connection
  - serial line
  - file handles (only on Windows)
- main principle
  - create `boost::asio::io_service` (`io_context` in newer versions)
  - assign work to service (e.g. reading from a socket)
  - once there is something to read, a callback is invoked
  - service can do its work in a separate thread or in the main loop of a GUI program

Boost ASIO

- multi-platform asynchronous IO
  - network connection
  - serial line
  - file handles (only on Windows)
- main principle
  - create `boost::asio::io_service` (`io_context` in newer versions)
  - assign work to service (e.g. reading from a socket)
  - once there is something to read, a callback is invoked
  - service can do its work in a separate thread or in the main loop of a GUI program
- advantages:
  - asynchronous IO can lead to performance gain
  - removes response sending and error handling hell
  - can be also used in synchronous manner (C++ wrapper for sockets)
- disadvantage: quite complex

Boost ASIO Example

see `web_client.cpp`

- *extremely* simple web client
- compile with `-lboost_system -pthread`
- `class WebClient`
- `work` tells the `io_service` it cannot stop (it still has work to do) until the destructor of `work` is called
- everything works asynchronously
  - `get` resolves the URL and calls `resolve_handler`
  - `resolve_handler` (re)opens the TCP socket and calls `connect_handler`
  - `connect_handler` sends a request and calls `read_handler`
  - `read_handler` reads from the socket and calls itself