Threads I

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Outline

- parallel programming
- threads
- working with memory
- asynchronous programming

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
  - knowledge of memory model required

Threads

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

```
struct thread {
    thread(); // do nothing
    template< typename F, typename... Args >
    thread( F, Args &&... ); // start new thread
    void join(); // wait until it ends
    ...
};
```

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
- add flag `-pthread` to the compiler

Threads

08_thread.cpp

```
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();
}
```

Working with memory

- access to the memory needs to be guarded
  - **mutual exclusion** devices
    - 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::lock_guard` *(C++17)*
  - atomic primitives
    - *some next lecture*
- thread synchronization
  - conditional variables

Working with memory

mutex – 08_mutex.cpp

- idea of safe output stream
- better approach can be found in the study materials

```
std::mutex mutex;

template< typename T >
void safeCout( T &&value ) {
    std::lock_guard< std::mutex > lock( mutex );
    std::cout << std::forward< T >( value );
}
```

Working with memory

conditional variable – 08_cv.cpp

```
struct Barrier {
  Barrier( int w ) : _w( w ), _a( 0 ) {}
  void wait() {
    std::unique_lock< std::mutex > lk( _m );
    if ( ++_a == _w ) {
      lk.unlock();
      _cv.notify_all();
    } else
      _cv.wait(lk, [this]{ return _a == _w; });
  }
private:
  int _w; // workers
  int _a; // arrived
  std::conditional_variable _cv;
  std::mutex _m;
};
```

Working with memory

deadlock prevention – 08_deadlock.cpp

```
void transferMoney( Account &from,
                    Accout &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 );
}
```

Working with memory

- 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.
  - `i++` is **NOT** atomic
  - does not say anything about other memory locations

Working with memory

- 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.
  - `i++` is **NOT** atomic
  - does not say anything about other memory locations
  - it is sufficient when using MSVC

Working with memory

- 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.
  - `i++` is **NOT** atomic
  - does not say anything about other memory locations
  - it is sufficient when using MSVC
    - and not targeting ARM
    - and not using flag `/volatile:iso`

Asynchronous programming

`08_async.cpp`

- `#include <future>`
- modern approach
- avoid using “heavy” threads
- advantages
  - can return value
  - can rethrow exceptions
- disadvantages
  - no native handle
  - threads cannot be detached

Asynchronous programming

```
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;
}
```

Task

Implement a simple thread pool which accepts only non-parametrized tasks. Tasks will be enqueued and the thread pool will execute tasks if it has free slots for threads.

- tasks will not return any value
- tasks will not throw any exception
- the thread pool will have limit for threads