Concurrency in C++11

- Introduction
- Basics: futures, threads, “tasks”
- Synchronization: mutexes, locks, atomics

// This is code
assert(true && "You can read C++");

This is a link: http://en.cppreference.com/w/
Dmitry N. Petrov is...

- Staff software engineer at Intel Labs
- R&D: C/C++ since 2001, Java since 2004
- Compilers & development tools
- Electronic design automation
- System programming
C++ at a glance

- “Minimum overhead”
  - “High-level assembler with classes”
- Almost backward compatible with ANSI C (C89)
- Latest standard: C++11
  - Earlier known as C++0x
- Well-known desktop and server applications, embedded software, ...
- Complex compared to modern “productivity languages” (Java, C#, scripting languages, …)
Concurrenty in C++ before C++11

- Essentially sequential abstract machine
  - No concurrent memory model definition
  - Required compiler-specific tricks and non-portable APIs to enable concurrency
- OpenMP, MPI
- POSIX (pthread), WinAPI, …
- Boost.Thread
  - C++11 concurrency API is based on Boost.Thread
Some important C++11 features

- Range-based for

```cpp
std::vector<std::string> strings;
// ...
for (const std::string & s : strings) {
    std::cout << s << std::endl;
}
```

- 'auto': type inference for local variables

```cpp
auto x = foo(a, b, c);
```

- Lambdas: anonymous functor objects

```cpp
auto f = [](int i) { return i + 1; };
std::function<int(int)> ff = [](int i) -> int { return i + 1; };
std::cout << f(41) << std::endl;
```
Good Bedroom Reading

Anthony Williams

C++ Concurrency in Action: Practical Multithreading

C++11 concurrency from the Boost.Thread maintainer
See also

- Herb Sutter posts on concurrency and C++11
  - [http://herbsutter.com/category/concurrency/](http://herbsutter.com/category/concurrency/)
- C++ Concurrency
  - Channel9 Herb Sutter video from C++ and Beyond 2012
Before you start coding…

• Make it work. Make it right. Make it fast.
  • Premature optimization is the root of all evil
• Stop worrying and use the library
• Performance is measurable
  • Local optimizations can be global “pessimizations”
  • Profile your code
• Debugging & testing concurrent code is hard
  • Defensive coding, paired programming, code review, …
“Hello, world!”: now with C++11 concurrency

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

int main(int argc, char ** argv) {
    auto hello = std::async([] { std::cout << "Hello, " ; });
    auto world = std::async([] { std::cout << "world!"; return 42; });
    hello.wait();
    int magic = world.get();
}
```

- This program will print something
  - Usually “Hello, world!”
- hello and world are “futures”
  - Asynchronous or deferred operations with or without result
  - `std::future<T>`
- `[] { ... }` – lambda without captured values and arguments
- `std::async` submits task to run-time
std::future<T>

- High-level abstraction for deferred computation
- wait() - waits for the completion
- get() - waits for the completion and returns result (or re-throws an exception)
- ~future() disposes shared state
- std::async(f, args)
  std::async(policy, f, args)
  - Exact behavior determined by the run-time library
  - policy: std::launch: { async, deferred[, sync, any] }
<thread>:
std::thread

- std::thread – single thread of execution
- join() - waits until thread completion
- detach() - let the thread execute independently

**NB:** thread MUST be joined or detached before dtor

**NB:** if a thread function throws an exception, program is terminated
“Hello, world!”: now with threads

```cpp
#include <iostream>
#include <thread>

int main(int argc, char ** argv) {
    std::thread hello([] { std::cout << "Hello, "; });
    std::thread world([] { std::cout << "world!"; });
    hello.join();
    world.join();
}

• This will also print something (usually “Hello, world!”)
```
<thread>:
std::this_thread

• yield() – suspends current thread

• std::thread::id get_id()

• sleep_for(duration)
sleep_until(abs_time)

• No standard way to interrupt or terminate a thread from outside

• Boost.Thread (look for EXTENSION):
http://www.boost.org/doc/libs/1_56_0/doc/html/thread/thread_management.html
See also

- `std::packaged_task<R(Args)>`
  - Something (callable + arguments) that can be invoked (asynchronously)

- `std::shared_future<T>`
  - Future that can be shared (e.g., between multiple consumer threads)
Threads vs Tasks

- **Thread**: system-level resource, managed by OS
  - This is std::thread
  - Problem: oversubscription – OS threads are expensive
    - Usually threads are long-lived objects organized in execution pools
- **Task**: abstract “something” that should be executed [asynchronously]
  - Somewhat close to std::future
- Unlike Java, C++ standard library provides no abstraction for “task execution engine” and related parallel programming patterns
  - Vendor-specific & proprietary libraries are widely used
  - Intel TBB: [https://www.threadingbuildingblocks.org/](https://www.threadingbuildingblocks.org/)
# Intel TBB code example

```cpp
#include "tbb/tbb.h"

void process_file(const std::string & name) {
    // ...
}

void process_files(const std::vector<std::string> & file_names, std::size_t n) {
    parallel_for(blocked_range<std::size_t>(0, n),
                 [&file_names](const blocked_range<std::size_t> & r) {
                    for (std::size_t i : r) {
                        process_file(file_names.at(i));
                    }
                 });
}
```

- ` [&file_names]... ` – capture `file_names` by reference
- See TBB user guide at [https://www.threadingbuildingblocks.org/](https://www.threadingbuildingblocks.org/)
Synchronization

- You should already know something about...
  - Data races
  - Mutexes
  - Locks
  - Deadlocks & livelocks
  - Atomic operations

- SC-DRF memory model
  - Sequentially Consistent for Data Race-Free code
  - Related operations are seen in the same order
    - Unless there is a data race
  - Required by language standard since C++11
Reordering

- Basic operations: READ(x), WRITE(x)
  - 'x' is “memory location”
- Compiler can reorder operations
- Processor can reorder operations
- Memory (cache, etc) can reorder operations

- Without synchronization you can't make any assumptions about shared mutable objects
Memory locations

- Values of fundamental data types occupy exactly one memory location, regardless of their size
  
  ```
  char foo, bar; // Two different memory locations
  ```

- Adjacent bit fields of non-zero width share the same memory location
  
  ```
  struct gizmo_42_regs {
      unsigned r0 : 4;  // 'r0' and 'r1' share the same
      unsigned r1 : 4;  // "memory location"
      unsigned _0 : 0;  // ===== separator =====
      unsigned r2 : 4;  // 'r2' and 'r3' share the same
      unsigned r3 : 4;  // "memory location"
  };
  ```
<mutex>: std::mutex

- Good old mutex
- lock()
- try_lock()
- unlock()

- std::lock(L1, ..., Ln) – lock L1, ..., Ln using deadlock avoidance algorithm
class thread_safe_stack {
    std::vector<std::string> data;
    std::mutex data_mtx;

public:
    // ...

    void push(const std::string & x) {
        data_mtx.lock();
        data.push_back(x);
        data_mtx.unlock();
    }

    std::string pop() {
        data_mtx.lock();
        std::string top;
        if (!data.empty()) {
            top = data.back();
            data.pop_back();
        }
        data_mtx.unlock();
        return top;
    }
};
RAII & Object lifetime

- RAII = “Resource Allocation Is Initialization”
  - Classes have constructor(s) and destructor
  - Resources are usually disposed in dtor
  - Resource lifetime = owner object lifetime

```c
void foo() {
    // ...
    my_class x; // Default ctor: 'my_class::my_class()'
    // ...
    // 'x' goes out of scope
    // => (implicit) dtor: 'my_class::~my_class()'
}
```

- Java: try-finally
- C#: using
RAII & Exception safety

- Exceptions complicate reasoning about possible execution paths
- Think about invariants for methods
- RAII
  - All resources allocated should be owned by someone
  - Dtors will be invoked during call stack unwinding
<mutex>:
std::lock_guard

- RAII object representing lock
- Locks in ctor, unlocks in dtor
class thread_safe_stack {
    std::vector<std::string> data;
    std::mutex data_mtx;

public:

    // ...

    void push(std::string x) {
        std::lock_guard<std::mutex> lock(data_mtx);
        data.push_back(x);
    }

    std::string pop() {
        std::lock_guard<std::mutex> lock(data_mtx);
        std::string top;
        if (!data.empty()) {
            top = data.back();
            data.pop_back();
        }
        return top;
    }
};

data_mtx is locked here
data_mtx is locked here
<mutex>:
std::unique_lock

- RAII object representing a lock
- (Usually) locks in ctor, unlocks in dtor
- Can defer lock with special ctor
- Can unlock before dtor
- std::unique_lock(mutex)
- lock(), try_lock(), unlock()

Use std::lock_guard unless you really need to unlock “in the middle” / defer locking
**<condition_variable>: std::condition_variable**

- Event notification mechanism
  - E.g., “there are messages ready for processing”
- `wait(std::unique_lock<std::mutex>& lock)`
  1. Releases lock (that’s why `std::unique_lock`)
  2. Blocks current execution thread T
  3. Waits for notification
  4. Reacquires lock, resumes thread T
- `wait(std::unique_lock<std::mutex>& lock, pred)`
  - waits until a predicate is satisfied.
  Equivalent to:
  ```cpp
  while(!pred()) { wait(lock); }
  ```
<condition_variable>:
std::condition_variable

- notify_one() – notifies some waiting thread
- notify_all() – notifies all waiting threads

**NB:** lost notifications are similar to deadlocks
  - Use notify_all() before “make it fast” phase
Example: producer / consumer

```cpp
std::mutex fifo_mutex;
std::queue<Sample> data_fifo;
std::condition_variable data_rdy;

void producer_thread() {
    while (true) {
        const Sample s = produce();
        std::lock_guard<std::mutex> lk(fifo_mutex);
        data_fifo.push(s);
        data_rdy.notify_all();
        if (is_last(s)) break;
    }
}

Sample produce();
void process(const Sample &);
bool is_last(const Sample &);

void consumer_thread() {
    while (true) {
        std::unique_lock<std::mutex> lk(fifo_mutex);
        data_rdy.wait(lk,
            [] { return !data_fifo.empty(); });
        const Sample s = data_fifo.front();
        data_fifo.pop();
        lk.unlock();
        process(s);
        if (is_last(s)) break;
    }
}
```
See also

- `std::shared_lock`
- `std::defer_lock`
- `std::adopt_lock`
To lock or not to lock?

• Minimize shared mutable state
  • Tasks, actors, …
• Provide good (concurrency-aware) API
  • High-level operations (transactions), not steps
    /* Ex.1 */ if (!q.empty()) { x = q.front(); q.pop(); process(x); }
    /* Ex.2 */ if (q.try_pop_front(x)) { process(x); }
• Think about invariants
• Lock at proper granularity
  • Again, think about invariants
Concurrent programming patterns

- Active object
  - Replace locking operation with message submission
- Reactor
  - Receive message, spawn new task
- SPMD
  - Parallel for
  - Parallel reduce
  - Recursive fork-join
- Pipeline (special case of “Task dependency graph”)
  - $F \rightarrow G \rightarrow H$
  - Operations F, G, H are independent
Coding for concurrency

• Localize (and eliminate) mutability
  • Immutable objects are thread-safe by default
  • C++: ‘const’ discipline matters

• Minimize dependencies between tasks
  • Maximize determinism

• Group data by tasks

• Separate business logic from orchestration
  • Better testability

• Use library solutions when possible
  • Thread-safe containers, logging, …
Atomics

- **C/C++: ‘volatile’ != atomic**
- `std::atomic<T>`
  - `compare_exchange_weak(expected, desired)`, `compare_exchange_strong(expected, desired)`
  - Basic read-modify write operation known as CAS (Compare-And-Swap)
  - `operator ++`, `operator --`, `operator +=`, …
- `std::atomic_flag`
  - `test_and_set()`, `clear()`
Example: test-and-set mutex

class tas_mutex {
    std::atomic_flag flag;

public:
    tas_mutex()
        : flag(ATOMIC_FLAG_INIT)
    {} 

    void lock() {
        while(flag.test_and_set());
    }

    void unlock() {
        flag.clear();
    }
};
More on atomics

Herb Sutter: atomic<> Weapons
(Channel9 video from C++ and Beyond 2012)

Data contention

• Data propagation through memory hierarchy can be very slow
  • Cache ping-pong in loops
  • Atomics (and mutexes) count, too
• “False sharing”
  • Caches operate in cache lines
  • E.g.: naïve matrix multiplication
• Data proximity
  • Organize your data by tasks
  • Use extra padding to test for false sharing
  • TBB: cache_aligned_allocator<T>
CAS and the “ABA problem”

- Thread T writes A to shared variable X
- Thread T goes to sleep
- Someone writes B to X
- ...  
- Someone writes A to X
- Thread T wakes up
- Thread T sees value A of shared variable X
- Thread T happily continues

Did anyone tell you shared state is bad?
Workarounds for ABA problem

- Tagged state reference
  - \(<\text{Value}, \text{Modification counter}>\)
  - Used in Boost.Lockfree

- Special case: memory management in dynamic data structures
  - Use GC
  - Use smart node reclamation strategy

- Minimize shared state
  - Tasks, actors, …

Boost.Lockfree: since 1.53

CAF: C++ Actor Framework
http://actor-framework.org/