Dmitry N. Petrov, Intel Labs

Concurrency in C++11

Who's that?

- Staff software engineer at IL
- R&D: C/C++ since 2001, Java since 2004
- C & C++ compilers
- Distributed computing middleware
- Electronic design automation tools

Overview

- Hitchhiker's guide to C++11
- Practical concurrent programming in C++11
  - C++11 threads
  - High-level synchronization objects
  - Coding guidelines
- Advanced topics
  - Low-level synchronization objects
  - Lock-free & wait-free algorithms
- Some exercises in-between

C++ today

- Language with a long history
  - C++98, C++03, C++ TR1, **C++11**, C++14, C++17, …
- C++ is complex
- No standard ABI, no modules, no reflection, …
- Real-life C++: standard library + Boost [+ Qt] + …
- Still, many applications are written in C++
  - Microsoft, Apple, Google, Adobe, Mozilla, MySQL, …
- See:
  - http://en.cppreference.com/w/ (std library reference)
  - http://herbsutter.com/ (Herb Sutter)
  - http://www.boost.org/ (Boost)
  - http://yosefk.com/c++fqa/ (C++ criticism)

C++11: "Not your father's C++"

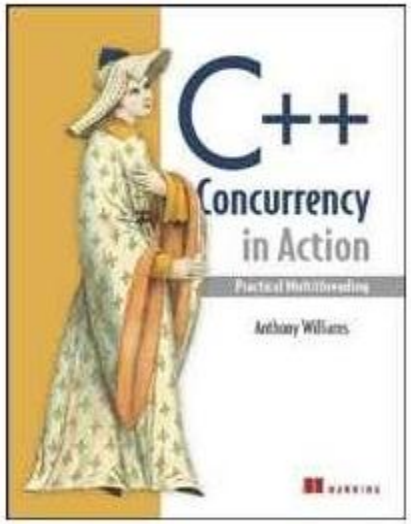
- nullptr (null pointer, non-ambiguous, use instead of NULL macro)
- Lambdas, aka anonymous functions
  - [incr](int x) { return x + incr; } *// captures 'incr' by value*
- Local type inference with 'auto'
  - const auto get_magic_number = [] { return 42; };
- Range-based for loop
  - for (const A& xi : a_vec) …
- Rvalue-references and "move semantics"
- Variadic templates
- Smarter "smart pointers"
  - std::unique_ptr<T>, std::shared_ptr<T>, std::weak_ptr<T>
- And many others. See more about C++11 at:
  - http://www.stroustrup.com/C++11FAQ.html

C++ and concurrency

- From C: libpthread, OpenMP, MPI,
  - Also: Apache Portable Runtime, libtask, etc
- Before C++11:
  - Boost.Thread : very close to C++11 threads
  - Concurrency support in Qt
  - MFC
  - Also: ZThreads, POCO, OpenThreads, etc
- Since C++11:
  - **Memory model that recognizes concurrency**
  - <thread>, <mutex>, <condition_variable>, <future>, <atomic>
- Higher-level libraries
  - Intel TBB https://www.threadingbuildingblocks.org/
  - AMD Bolt, Microsoft PPL, other vendors
  - Actors: Theron http://www.theron-library.com/
  - Join calculus: Boost.Join http://code.google.com/p/join-library/

[Williams]

- Anthony Williams
  C++ Concurrency in Action
  Practical Multithreading
- From the maintainer of Boost.Thread
  - and the mastermind of C++11 threads
- Not just about C++



<thread>

- std::thread(func, arg1, arg2, …, argN);
  - Creates a thread that runs 'func(arg1, arg2, …, argN)'
  - func is any Callable; lambdas and functor objects are fine
- join();
- detach();
- std::thread::id get_id();

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

int main() {
   std::thread hello([] { std::cout << "Hello, world!\n"; });
   hello.join();
}
```

Thread management

- bool joinable()
- Thread owns underlying system resource
  - native_handle_type native_handle()
- If a thread goes out of scope without join or detach, program terminates
  - **NB:** Boost.Thread detaches by default
  - Use std::move to pass ownership
    - …if you really need it
- If the thread body throws an exception, program terminates
  - **Why?**
  - Do not let exceptions escape from thread body
- unsigned std:thread::get_hardware_concurrency()

std::this_thread

- yield()
- std::thread::id get_id()
- sleep_for(const std::chrono::duration &)
- sleep_until(const std::chrono::time_point &)

<mutex>

- std::mutex a_mutex;
  - lock(), unlock() – "Lockable"
- std::lock_guard<std::mutex> a_lock;
  - RAII object representing scope-based lock
  - Simplifies reasoning about locks
- std::unique_lock<std::mutex>
  - RAII, but more flexible
  - Locks mutex in ctor, unlocks in dtor (if owned)
  - lock(), unlock()
  - bool try_lock()
  - bool try_lock_for(const std::chrono::duration &)
  - bool try_lock_until(const std::chrono::time_point &)

Example: thread-safe singleton

- If you possess the sacred knowledge…
  - … please, keep it to yourself for a while
- First try:

```
A* A::instance = nullptr;

A * A::get_instance() {
    if (!instance)
        instance = new A();
    return instance;
}
```

- Of cause, you can easily spot a problem

Example: thread-safe singleton

- With mutex:

```
A * A::instance = nullptr;
std::mutex A_mutex;

A * A::get_instance() {
    std::unique_lock<std::mutex> lock(A_mutex);
    if (!instance)
        instance = new singleton();
    lock.unlock();
    return instance;
}
```

Example: thread-safe singleton

- Real programmers know standard library

```
A * A::instance;
std::once_flag A_flag; // in <mutex>

A * A::get_instance() {
   // Faster than mutex
   std::call_once(A_flag,
        []() { instance = new A(); });
   return instance;
}
```

Example: thread-safe singleton

- That's how you should really do it:

```
A* A::get_instance() {
    static A instance;
    return &instance;
}
```

- Initialization is guaranteed to be thread-safe. Performed 1st time when A::get_instance is called.
- Underlying implementation is equivalent to std::once_flag / std::call_once
- **Caution!** Guaranteed to work only with C++11.
  - Memory model matters

<condition_variable>

- Event to wait for, periodically
  - "Next train has arrived"
  - E.g., "new data available in the input queue"
- std::condition_variable()
- notify_one(), notify_all()
- wait(std::unique_lock<std::mutex> & lk)
  - **NB:** unique_lock!
  - Releases lock, blocks tread until notification
  - Upon notification: reacquires lock
- wait(std::unique_lock<std::mutex> & lk, pred)
  - E.g.: cv.wait(lk, []{ return !input.empty(); });
  - while (!pred()) { wait(lk); }
- wait_for(lk, duration, pred)
- wait_until(lk, time_point, pred)

Caution: lost wake-ups

- Similar to deadlocks, but for conditions
- Thread ends up waiting for condition indefinitely
- Do not over-optimize notifications
- Always use notify_all
- Specify a timeout when waiting
- Performance penalty is negligible compared to the cost of lost wake-up

Example: producer/consumer

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

void producer_thread() {
    while (should_produce()) {
        const sample s = produce();
        std::lock_guard<std::mutex> lk(
            fifo_mutex);
        data_fifo.push(s);
        data_rdy.notify_all();
    }
}
```

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

Programming katas

- Thread-safe lock-based queue
  - Think about proper interface
  - What'd change if the queue is bounded? (size <= N)
- Read/Write mutex
  - read_lock() / read_unlock()
    - No thread can acquire read lock while any thread holds write lock (multiple readers are ok)
    - size, empty, …
  - write_lock() / write_unlock()
    - No thread can acquire write lock while any thread holds write lock or read lock
    - push, pop, …

<future>

- Future: one-off shared result (or exception)

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

int main() {
   std::future<void> hello = std::async([] { std::cout << "Hello, world!\n"; });
   std::future<int> magic = std::async([] { return 42; });
   hello.wait();
   std::cout << "Magic number is " << magic.get() << "\n";
}
```

- std::async(std::launch, …)
  - std::launch – bit mask
  - std::launch::async, std::launch::deferred
- wait_for(duration), wait_until(time_point)

std::shared_future<T>

- This is how you really should pass around "something running"
- std::shared_future<T> std::future<T>::share()

N3451: ~future Must Never Block

- … except for std::async:
  - **Underlying future state CAN cause the returned std::future destructor to block**
- E.g.:

```
{
        std::async(std::launch::async, [] { f(); });
        // OOPS, can block here (and will)
        std::async(std::launch::async, [] { g(); });
}
```

- This is almost good:

```
{
        auto f1 = std::async(std::launch::async, [] { f(); });
        auto f2 = std::async(std::launch::async, [] { g(); });
}
```

- Will be fixed in C++14

std::packaged_task<R(Args...)>

- Defined in <future>
- Stores callable & return value (or exception)
  - Building block for futures, thread pools, etc.
- Template argument is a function signature
  - std::deque<std::packaged_task<void()>> tasks;
- std::future<R> get_future()
- void operator()(Args...)
  - **NB:** packaged_task is Callable
- reset()

std::promise<T>

- Defined in <future>
- Underlying storage for std::future<T>
  - Arbitrary workflow (not just 1 function)
  - Use multiple returned values in different threads
- std::future<T> get_future()
- set_value(const T&)
- set_exception(std::exception_ptr p)
  - std::current_exception()
  - std::copy_exception(E)

Programming katas

- Concurrent quick sort
  - Use futures to synchronize between subtasks
  - Good example for learning hard way about parallel application performance
- Concurrent find + grep
  - Use Boost.Filesystem to crawl directory tree
  - Use <regex> to match regular expressions

Before you start coding…

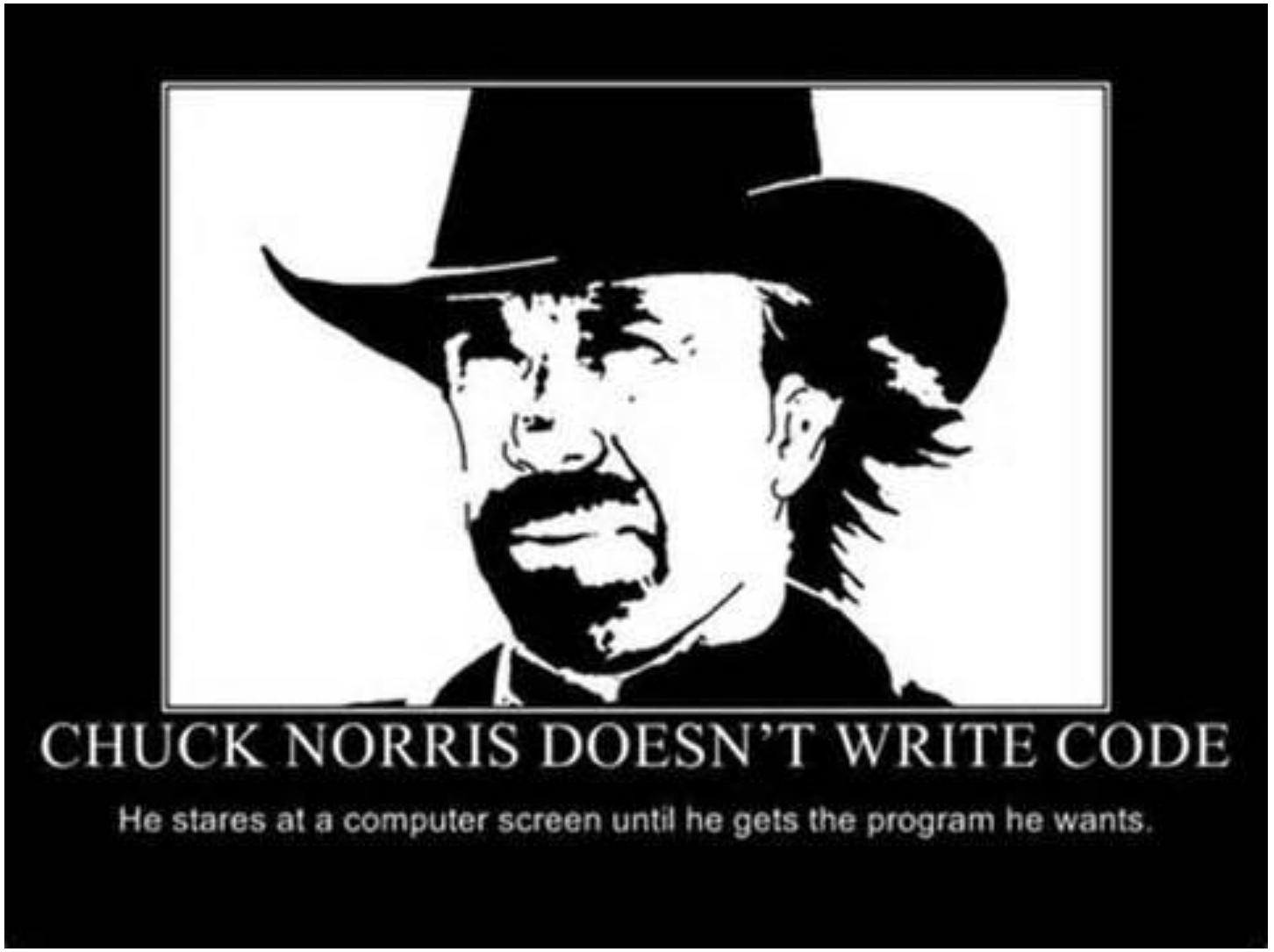
- Make it work. Make it right. Make it fast.
  - Premature optimization is the root of all evil
- Stop worrying and use the library
- Performance is measurable
  - Local optimization can be global "pessimization"
- Performance vs maintenance
- Code review, paired programming, …

Dark side of parallel programming

- **Dan Appleman:**
  "[multithreading used incorrectly] can destroy your reputation and your business because it has nearly infinite potential to **increase your testing and debugging costs**."

- **Jeff Atwood:**
  We're pretty screwed if the only way to get more performance out of our apps is to make them heavily multithreaded

But you still need that performance, right?



Worth mentioning yet again

- Minimize shared mutable state
  - Think high-level (e.g., actors)
  - Async agents with messaging
- Provide proper interfaces
  - High-level operations, not steps
  - **Think about class invariants**
- Lock at the proper granularity
  - Again: class invariants
  - Use lock hierarchies to avoid deadlocks
- Keep in mind exception safety
  - Leaked lock = deadlock

Exception safety

- Do not leave threads hanging in the air
  - Use RAII object for thread group join to prevent thread leak
  - Unfortunately, not a part of <thread>
  - Boost.Thread: boost::thread_group
- Remember about exception safe C++ coding
  - Use RAII
  - Rule of three
    - Constructor, destructor, copy assignment
    - C++11: rule of 3…5: move constructor, move copy assignment
  - Use smart pointers
  - Use copy-and-swap

Threads vs Tasks

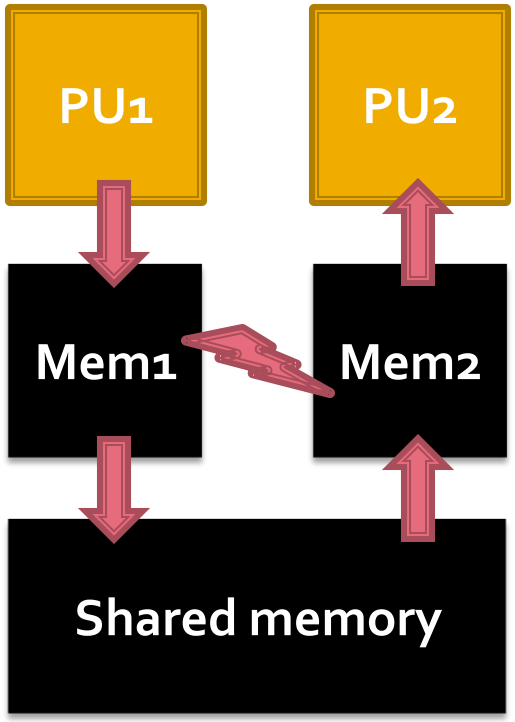
- Oversubscription
  - Threads cost system resources
  - Separate tasks (application logic) from threads
  - Tasks make promises
- Parallel for_each / transform / …
  - Fixed number of uniform tasks
- (Recursive) fork-join
- Pipeline
  - Effectively: get value, then do something else, then…
  - Can become part of C++14
- Tasks dependency graph
- Concurrent stack / queue for spawned tasks

Separate concerns between tasks

- E.g., Model / View / Controller
- Locking is evil
  - Turn mutex into an active object + message queue
  - E.g.: concurrent logging
- You're doing it wrong if…
  - Lots of shared data
  - Lots of communications between threads
- And that's bad, because of…

Data contention

- Data propagation through memory hierarchy can be **very** slow
  - Cache ping-pong in loops
  - **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>



Thread pools & data contention

- Bottleneck: task queue
- Solution: per-thread task queue + scheduler
- Work stealing
  - Balancing load between threads

- **Do not reinvent thread pools.**
  - Use the library. E.g., Intel TBB ☺

Threads & memory allocation

- Today standard malloc is thread-aware
  - But still is a performance bottleneck
- Custom allocators are not uncommon in critical apps
  - Concurrent malloc: use thread-private heaps
  - Do not lock global heap that often

Before we continue…

- Memory model: SC-DRF
  - Sequentially Consistent for Data Race Free programs
  - sequenced-before (sb)
  - synchronizes-with (sw)
  - happens-before (hb)
  - Memory locations & objects
  - Reordering

Lock-free: what's that?

- **Wait-free**
  - Every thread can progress in finite time
- **Lock-free**
  - At least one thread can progress in finite time
- **Obstruction-free**
  - One thread executing in isolation can progress in finite time
- WF < LF < OF

- **Can you spot the catch here?**

<atomic>

- **NB:** ‘volatile’ IS NOT atomic!
- Low-level synchronization mechanism
- Can control degree of synchronization
  - std::memory_order
  - Details: a bit later
- std::atomic<T>
  - std::atomic_bool
  - std::atomic_int
  - std::atomic<T*>
  - And so on (several specializations and typedefs)
  - Atomic Read-Modify-Write (RMW) operations
- bool is_lock_free()

std::atomic<T>::compare_exchange_*

- bool compare_exchange_weak(
  T& expected,
  T desired,
  std::memory_order order = std::memory_order_seq_cst
) volatile;
  - Can fail spuriously (when *this == expected)
  - Allows better performance
- bool compare_exchange_strong(
  T& expected,
  T desired,
  std::memory_order order = std::memory_order_seq_cst
) volatile;
- CAS: general-purpose atomic RMW
- **Caution!** The ABA problem

The ABA Problem

- Especially relevant for dynamic data structures
  - Stacks, lists, queues, …
- Thread 1 sees 'x': A
- Thread 1 uses 'x' to compute 'y'
  - E.g., 'y = x->next'
- Thread 2 changes 'x' to B
- Thread 2 changes 'x' back to A
  - E.g., allocate a new node in the memory location we've just freed
- Thread 1 sees 'x': A. compare_exchange_* works fine.
  - But 'y' can be wrong!
  - E.g., 'x->next' is different

Solutions to ABA

- Modification counter
  - Atomic pair: data + counter
  - Implementation is not always lock-free
    - Needs atomic double word CAS
  - Can use bit stealing
    - E.g., if you know that address space is 48-bit and the pointer is 64-bit
    - **Complicates debugging.** You've been warned.

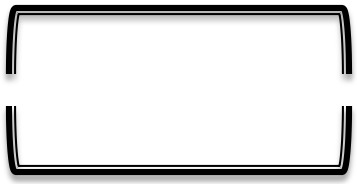
std::memory_order

- **Sequential consistency**
  - memory_order_seq_cst
- **Acquire-release ordering**
  - memory_order_acq_rel
  - memory_order_consume
  - memory_order_acquire
  - memory_order_release
- **Relaxed ordering**
  - memory_order_relaxed

4 words:
Do not use them

Sequential consistency

- Default ordering for atomics
- Load(x): **acquire**
- Store(x): **release**
- Partial order on acquires and releases
  - Can't move code above **acquire**
  - Can't move code below **release**
  - **acquire** can't move above **release**
- Requires additional synchronization



Cost of SC-DRF

- x86 & x86-64: even stronger guarantees
- IA64:
  store requires memory barrier
- POWER, ARM v7:
  load & store require memory barrier
- ARM v8: required to support sequentially consistent load & store

Relaxed ordering

- Only ‘hb’ within a given thread
  - Can reorder independent operations
- No implicit synchronization between threads
- Still SC “by default” in x86, x86-64, ARM v8
- Important for other archs
  - You should clearly understand ‘hb’
  - Better provide a special class that encapsulates corresponding semantics

In-between…

- memory_order_acq_rel
  - **acquire** can move above **release**
- memory_order_acquire
- memory_order_release
  - Explicit acquire/release
- memory_order_consume
  - 'Load(x)' performs acquire for 'x', and locations 'x' depends on

Fence

- std::atomic_thread_fence(memory_order)
- Explicit memory barrier
- Synchronizes ALL non-atomic and relaxed atomic accesses

Refresher on mutexes

- std::atomic_flag
  - Intentionally simplified atomic flag, minimal overhead
  - Building block for synchronization primitives

```
class tas_mutex {
    std::atomic_flag flag;
public:
    tas_mutex(): flag(ATOMIC_FLAG_INIT) {}
    void lock() {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock() {
        flag.clear(std::memory_order_release);
    }
};
```

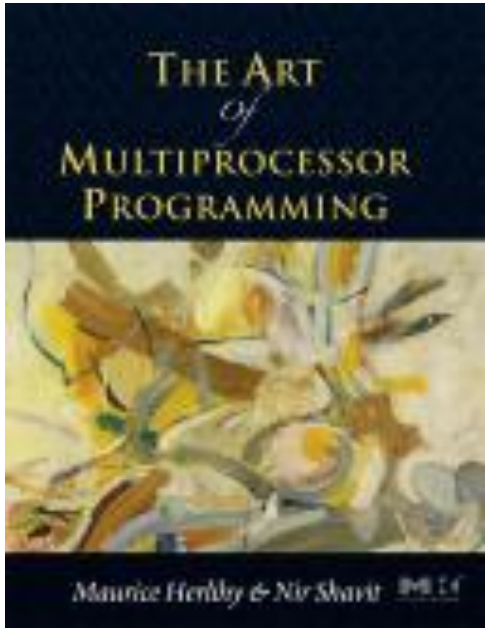
- WF? LF? OF?
- **Exercise:** ttas_mutex

Other mutex implementations

- Peterson's algorithm
  - Two threads, alternating priority
  - Generalized to Filter algorithm for N threads
- Lambert's algorithm ("Lambert's bakery")
  - Thread gets a numbered token "at the doorway"
  - First come – first served
- Adaptive backoff
  - Avoid pinging the cache by waiting for a period of time
- Array mutexes, queue mutexes, hierarchy mutexes…
- No mutex to rule them all

[Herlihy & Shavit]

- Maurice Herlihy, Nir Shavit
  The Art of Multiprocessor Programming
- Maurice Herlihy, 1991:
  Wait-free Synchronization
- Dijkstra Prize 2003
- Gödel Prize 2004
- Comprehensive study of design and implementation of concurrent data structures
- Great bedroom reading



Boost.Lockfree

- **Do not reinvent the bicycle**
  - http://www.boost.org/doc/libs/1_55_0/doc/html/lockfree.html
- boost::lockfree::stack<T>
  - Lock-free stack
- boost::lockfree::queue<T>
  - Lock-free queue
- boost::lockfree::spsc_queue<T>
  - Wait-free SPSC queue (aka ringbuffer)
- boost::lockfree::detail::tagged_ptr<T>
  - Atomic pair: <pointer, modification counter>

TANSTAAFL

(+) Maximize concurrency
(+) Responsiveness
(+) Robustness
(–) More complex algorithms
(–) Livelocks are still possible
- But usually are short-lived

(–) Complex effects on performance
- Individual operations can be slower
- Still has cache ping-pong

Lock-free algorithms

- Understanding locks is important for understanding lock-free
  - Mutex: "N threads agree on some boolean value"
  - Lock-free algorithm: "N threads agree on some value"
- Atomic variables for important data
  - Chose depending on **class invariants**
  - Invariant holds when algorithm starts (object is created)
  - No thread can make a step that would break the invariant
- CAS loop for synchronization
  - If I see the actual state, I can make the intended atomic modification
  - Otherwise I should update my data and check again
  - **Remember about ABA**
  - [Herlihy & Shavit]: "Consensus object"
- At least one thread will progress
  - Other threads still can wait indefinitely
- **Prototype with memory_order_seq_cst**

Wait-free algorithms

- More complex
  - And slower in sequential case
- Redistribute work between threads
  - Postpone modifications
- Herlihy: universal wait-free construct
  - Based on "Consensus object for N threads"
  - Threads cooperatively agree on numbering of announced actions
- Specific implementations are often limited by the number of concurring threads
  - E.g., SPSC queue

lock_free_stack

```
#include <atomic>

template <class T>
class lock_free_stack {
    struct node {
        T data;
        node * next;
        node(const T & data_) : data(data_) {}
    };

    std::atomic<node *> head;

public:
    void push(const T &);
    bool pop(T &);
};
```

lock_free_stack<T>::push

- Push is easy

```
template <class T>
void lock_free_stack<T>::push(const T & x) {
    const node * new_node = new node(x);
    new_node->next = head.load();
    while(!head.compare_exchange_weak(
        new_node->next, new_node));
}
```

lock_free_stack<T>::pop

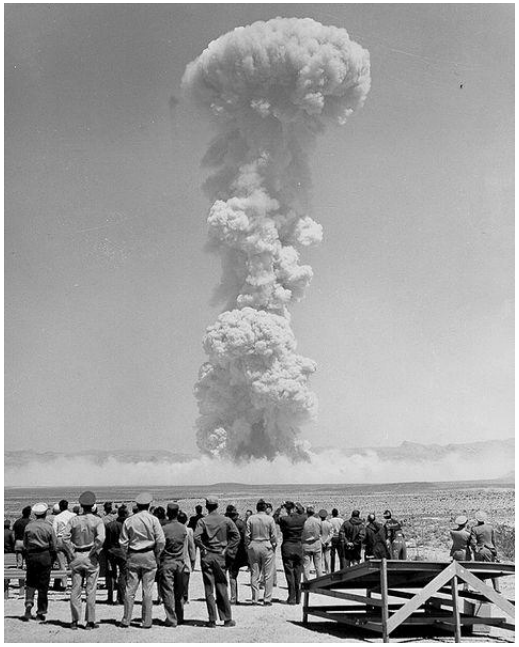
- Pop that leaks memory

```
template <class T>
bool lock_free_stack<T>::pop(T & result) {
    node * old_head = head.load();
    while (old_head &&
           !head.compare_exchange_weak(
                   old_head, old_head->next));
    if (!old_head) return false;
    result = old_head->data;
    return true;
}
```

- **You can't simply 'delete old_head'. Why?**

Memory management in lock-free data structures

- Hazard pointers
  - Remember "hazard pointers"
  - If node is being tagged with "hazard"…
    - … leave it until later
  - Periodically revisit objects left until later
  - Can have hidden memory leak
  - Similar to local heap with reclamation list
  - Patented (by IBM)
    - … but ok for GPL and other similar licenses



- Reference counting
  - Per-node: internal (threads visiting node) and external (links to node)
- **And again, remember about ABA**

Wait-free: going lazy

- Postponed removal: nodes have ‘valid’ flag
- **Invariant:** every ‘valid’ node is reachable
- Atomic <node * next, bool valid> pair with CAS and ‘try_set_flag’
  - C++: stealing a bit from an atomic pointer
- May need to retry the traversal if CAS fails
  - We can be traversing an invalidated “branch”
- Mutators (add, remove, …) erase invalid nodes before continuing