C++ Threads

Lawrence Crowl

Google

13 April 2007

introduction

goals for the standard

- extend the language into concurrency
- interact with the computational environment
- enable expressive libraries for concurrency

goals for this talk

- outline the primary features
- expose issues under debate
- get feedback

disclaimers

- talks necessarily suppress details
- every detail in this talk is wrong
- the standard is not done
- its details will change
- compilers will not agree at first

why add threads?

- concurrency and parallelism is a pressing need
  - effective internet programming
  - effective use of multi-core processors on desktops
  - solution of very large problems
- a standard solution in C++
  - enables portable expression of such programs
  - creates a community of understanding
  - leverages the work of communities

standardize on the environment

- C++ threads = OS threads
  - heavyweight, pre-emptive, independent
- shared memory
- loosely based on POSIX and Windows
- not a replacement for other standards
  - MPI, OpenMP, automatic parallelization, etc.

core versus library

- core language changes
  - how do two threads share memory?
  - what operations are atomic?
  - how does this affect variables?
- standard library changes
  - how do programs create and schedule threads?
  - how do threads synchronize and terminate?
  - is that all there is?

memory

instant shared memory

- all writes are instantly available to all threads
- traditional notion of shared memory
- but there are problems
  - it implies faster-than-light communication
  - it does not match current hardware
  - it inhibits most serial optimizations
- therefore it is not viable

message shared memory

- writes are explicitly communicated
  - between pairs of threads
  - through a lock or an atomic variable
- the mechanism is acquire and release
  - one thread *releases* its memory writes
    - v = 32; atomic_store_release(&a, 3);
  - another thread *acquires* those writes
    - i = atomic_load_acquire(&a); i + v;

memory fences

- most shared memory processors have them
- they are not currently in the proposal
- we keep using them in examples
- they imply global action
  - may inhibit more loosely coupled machines
  - may inhibit distributed shared memory
- may or may not find their way into the standard

sequencing redefined

- sequence points are gone
- sequence now defined by ordering relations
  - sequenced-before
  - indeterminately-sequenced.
- a write/write or read/write pair to a location
  - that are not sequenced-before
  - that are not indeterminately-sequenced
  - results in undefined behavior

sequencing extended

- sequenced-before
  - provides intra-thread ordering
- acquire and release
  - provide inter-thread ordering
- defining the happens-before relation
  - between memory operations in different threads

data race condition

- a non-atomic write to a memory location in one thread
- a non-atomic read from or write to that same location in another thread
- with no happens-before relation between them
- is undefined behavior

memory location

- a non-bitfield primitive data object
- a sequence of adjacent bitfields
  - not separated by a structure boundary
  - not interrupted by the null bitfield
- avoids expensive atomic read-modify-write operations on bitfields

effect on optimization

- relatively rare optimizations are restricted
  - fewer speculative writes
  - fewer speculative reads
- relatively common optimization have special help
  - they may assume that loops terminate
  - nearly always true

atomics

atomic definition

- operations
  - on a single variable
  - appear to execute sequentially
- all threads
  - observe the same sequence of values
  - but may not read all those values

not volatile

- the volatile qualifier has a long history
- we chose not to change its meaning
- volatile remains a "device register" mechanism
- volatile does not indicate atomicity
- an atomic variable can also be volatile

requirements on atomics

- static initialization
- reasonable implementation on current hardware
- enable (relative) novices to write working code
- enable experts to write very efficient code
- provide a foundation for lock-free data structures

atomics and memory

- operations specify a memory ordering
  - acquire, release, acq_rel (both), relaxed (neither)
  - ordered – extra consistency semantics
- too little ordering will break programs
- too much ordering will slow them down
- most programmers should be conservative
  - experts argue about the ordering
  - usually the performance is adequate

consistency problem

- x and y are atomic and initially 0
  - thread 1: `x = 1;`
  - thread 2: `y = 1;`
  - thread 3: `if ( x == 1 && y == 0 )`
  - thread 4: `if ( x == 0 && y == 1 )`
- are both conditions exclusive?
  - that is, is there a total store order?
- the hardware/software system may not provide it
- programming is harder without it

consistency options

- sequential consistency
  - the observed facts are consistent with a sequential ordering of all events in the system
- weaker models
  - see Hans Boehm's talk, Saturday 9:30
- details and formalism are difficult

minimal atomics

- atomic flag type
  - static atomic_flag v1 = ATOMIC_FLAG_INIT;
  - if (atomic_flag_test_set(& v1))
  - atomic_flag_clear(& v1);
- sufficient to implement the rest of the atomics
- similar to patch-panel programming
- users will expect more

basic atomics

- atomic bool
  - load, store, swap, compare-and-swap
- atomic integers
  - load, store, swap, compare-and-swap,
  - fetch-and-{ add, sub, and, ior, xor }
- atomic void pointer
  - load, store, swap, compare-and-swap,
  - fetch-and-{ add, sub }

low-level atomics

- types are opaque
- operations are type-generic function macros in C
- operations overloaded functions in C++
- operations include ordering

```
static atomic_long a = { 1 };
long t = atomic_load_acquire( & a );
while ( ! atomic_compare_swap_relaxed( & a, t, t/2 ) );
atomic_fetch_ior_ordered( & a, 1 );
```

high-level atomics

- types are classes
- member functions and operators are fully ordered

```
atomic_long a = { 1 };
long l = a; a = 3;
while ( ! a.compare_swap( &a, &l, l+1 ) );
++a; a += 4; a &= 3;
```

atomic assignment

- default assignment operator is wrong
  - non-atomic load and store
- even atomic load and store is wrong
  - users would expect the whole assignment to be atomic
- correcting the problem is difficult
  - it cannot be disabled in C 90
  - disabling it in C++ 98 breaks C compatibility
  - help (possibly) on the way in C++ 0x
    - N2172 POD's Revisited (Revision 2)
    - N2210 Defaulted and Deleted Functions

atomic template

- makes an atomic type from a non-atomic type
  - must be bitwise copyable and comparable
- defined specializations for basic types and pointers
- suggested specializations for alignment and size

```
atomic< int * > aip = { 0 };
aip = ip;  aip += 4;
atomic< gnat > ag = { …. };
while ( ! ag.compare_swap( &ag, &g, g+4 ) );
atomic< circus > ac; // works, but not recommended
```

atomic freedom

- lock-free
  - robust to crashes
  - someone will make progress
- wait-free
  - operations complete in a bounded time
- address-free
  - atomicity does not depend on using the same address

lock-free atomics

- large atomics have no hardware support
  - necessarily implemented with locks
- locks and signals do not mix
  - must be able to test for lock free
- compile-time macros for basic types
  - always lock-free
  - never lock-free
- run-time function for each type

wait-free atomics

- hard to implement without direct hardware support
  - resulting programs end up being hardware-specific
  - therefore difficult to write portable programs
- few people seem to care
- property unspecified in standard
  - no requirement for it
  - no query about it

address-free atomics

- memory may not have a consistent address
  - processes sharing memory may not have the same address for that memory
  - memory may be mapped into the address space twice
- atomic operations must be address-free to work
  - one small tool for inter-process communication
- a lock-free operation is address-free
  - not clear we can say this in a standard way
  - but we will make our intent known

variables

affect on variables

- thread-local storage (TLS)
- dynamic initialization of static-duration variables
- destruction of static-duration variables

adopt thread-local storage

- adopt existing practice
  - 5 vendors with few syntactic variations
- define new storage duration and class
- `__thread int var = 3;`
- variable is unique to each thread
- variable is accessible from every thread
- variable address is not constant

extend thread-local storage

- existing practice supports only static initialization and trivial destructors
- extend practice to support dynamic initializers and destructors
  - __thread vector<int> var;
- carefully define initialization to permit lazy allocation
- operating system support can improve efficiency

initialization of static-duration variables

- dynamic initialization is tricky
  - no syntax to order most initializations
- without synchronization, potential data races
- with synchronization, potential deadlock
- no consensus yet

function-local static storage

- initialization implicitly synchronized
  - while not holding any locks
- initialization explicitly synchronized

```
int f() {
   // synchronized
   static int u protected = g();
   static int v = h();
   // unsynchronized
   static int w private = i();
   static nonPOD x private (3);
}
```

non-local static storage

- initialization implicitly synchronized
- concurrent initialization enabled
- the initialization may not use a dynamically-initialized object defined outside the translation unit

```
extern vector<int> e;
vector<int> u; // okay, default initialization
vector<int> v(u); // okay, within translation unit
vector<int> w(e); // error, outside translation unit
```

destruction

- do not destruct
  - programs probably will not be correct in the presence of destruction and threads
  - probably the single largest incompatibility
- do destruct
  - rules to concurrently reverse the concurrent initialization
  - requires cooperative shutdown

launching

fork and join

- very basic thread class
  - fork a function execution
  - void join operation

```
void f();

void bar()
{
    std::thread t1(f);  // f() executes in separate thread
    .....
    t1(); // wait for thread t1 to end
}
```

functors

- may also use function-like objects

```
struct c
{
   void operator()() const;
};

void bar()
{
   std::thread t2((c()));  // c() executes in separate thread
   ....
   t2.join(); // wait for thread t2 to end
}
```

scheduling

scheduling

- limited thread scheduling
  - yield
  - sleep
- standard access to non-standard underlying OS thread handles
  - for detailed control
- query for the hardware concurrency

synchronization

mutexes

- exclusive (single reader/writer)
- shared (multiple readers)
- convertible (between exclusive and shared)
- upgradeable (right to become the writer)

locks

- hold a mutex within a given scope
- represents the mutex acquire/release pair
- release occurs in the destructor for the lock

```
std::upgradable_mutex mut;

void foo() {
  std::upgradable_lock< std::upgradable_mutex >
      read_lock( mut );
  // ... do read operation
  if ( /* sometimes need to write */ ) {
    std::exclusive_lock< std::upgradable_mutex >
        write_lock( std::move( read_lock ) );
    // ... do write operation, what was read hasn't changed
  }
}
```

condition variables

- threads may wait on a condition variable
  - giving up their hold on the mutex.
- threads may notify a condition variable
  - notified threads re-aquire the mutex and
  - must reevaluate any condition
- benefits
  - easier to use than events
  - enables the monitor pattern

buffer example

- conditions represent extreme states

```
class buffer {
  int head, tail, store[10];
  std::exclusive_mutex mutex;
  std::condition not_full, not_empty;
public:
  buffer() : head( 0 ) , tail( 0 ) { }
  void insert( int arg ) {
    std::exclusive_lock< std::exclusive_mutex >
        scoped( mutex );
    while ( (head+1)%10 == tail )
      wake.wait();
    store[head] = arg;
    head = (head+1)%10;
    not_empty.notify();
  }
};
```

termination

voluntary

- return from outermost function
- some form of cooperative, synchronous, exception-based termination
- strong opposition to asynchronous termination or interrupts
- still debating the details

exceptions

- when the thread function exits via throw
  - call std::terminate?
  - propagate exception to joiner?
  - ignore the exception?
- provide a means to manually propagate
  - std::exception_ptr saved(std::current_exception());
  - std::exception_ptr copied(std::copy_exception(saved));
  - rethrow_exception(copied);

cancellation

- one thread requests another to cancel itself
- that cancellation will create an exception in the target thread
- when it is ready for one
  - when cancellations have not been disabled
  - only at certain synchronous cancellation points
- or it can test for a pending cancellation
- a thread can ignore the request

cancellation points

- void std::this_thread::cancellation_point()
- void std::this_thread::sleep(std::nanoseconds)
- void std::thread::join()
- void std::thread::operator()()
- void std::condition<Lock>::wait(Lock&)
- template<class Predicate> void std::condition<Lock>::wait(Lock&, Predicate)
- bool std::condition<Lock>::timed_wait(Lock&, std::nanoseconds)
- template<class Predicate> void std::condition<Lock>::timed_wait(Lock&, std::nanoseconds, Predicate)

and beyond

high-level later

- some work is (probably) being deferred to TR2
  - thread pools, groups, ...
  - value-based joins, futures, ...
  - parallel iterators, ...
- reasons for deferral add up
  - lack of pre-existing implementations
  - lack of solid definitions
  - lack of time to provide them

success?

- we can build the high-level TR2 facilities in a pure library
- which means you can build even higher-level facilities as well

futures as an example

- a future executes a function
  - making return value available later
  - propagating exceptions to joiners
- technology is
  - return values via N2096
  - exception propagation via N2179
  - cancellation via handles in N2178

missing lambda

- anonymous nested functions
  - are not yet in the standard
  - enable user-defined control constructs
  - that put synchronization and concurrency together
  - out of the way of computation
  - and separate the tasks of algorithms specification and performance tuning
- we may get lambda
  - time is running out

conclusions

the basics are on track

- memory model
- atomic operations
- non-automatic variables
- thread launching and synchronization

some features need work

- destruction of static-duration variables
- exception propagation
- cancellation
- high-level facilities
- lambda

the real value comes later

- the standard will define the basics
- the standard will provide the means to extend the basics
- the users will write some really great high-level facilities

more information

- C++ standard website
  - http://www.open-std.org
  - WG21 - C++
  - WG papers
  - 2007
  - N2169
- questions?