C++ Threads

Lawrence Crowl

Google

June 2008

introduction

goals for the standard

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

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

- traditional notion of shared memory
  - all writes are instantly available to all threads
- 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; a.store( 3, release );`
  - another thread *acquires* those writes
    - `i = a.load( acquire ); i + v;`

memory fences

- most shared memory processors have them
- they imply global action
  - may inhibit more loosely coupled machines
  - may inhibit distributed shared memory
- a limited form already in the standard
- a more general form possible

dependency ordering

- load-dependent synchronization
- good for data structures rarely written
- in development

sequencing

- sequenced-before relation
  - provides intra-thread ordering
- acquire and release operations
  - provide inter-thread ordering
- together define the happens-before relation
  - between memory operations in one thread or 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

requirements on atomics

- static initialization
- reasonable implementation on current and near-future hardware
- semi-experts can write working code
- experts can write very efficient code
- provide a foundation for lock-free data structures

atomic types

- volatile does not mean atomic!
  - it still means 'device register'
- new standard atomic types
- functions for C compatibility
- methods and operators for ease of use

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 }

atomic template

- makes an atomic type from a non-atomic type
  - must be bitwise copyable and comparable
  - specialized for basic types and pointers
  - (specialized) for alignment and size

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

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
- new technology lets us make assignment illegal

compare and swap

- may fail spuriously!
- designed for use in a loop

```
expected = variable.load();
do desired = function( expected );
while ( ! variable.compare_swap(
              expected, desired ) );
```

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 system may not provide it
- programming is harder without it

consistency options

- sequential consistency
  - observed values consistent with a sequential ordering of all events in the system
- weaker models
  - difficult to understand
  - potentially much higher performance
- approach
  - sequentially consistent by default
  - weaker semantics explicitly

atomics and memory

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

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 do not mix with signals
  - must be able to test for lock free
- compile-time macros for basic types
  - always lock-free and address-free
  - never lock-free
- run-time function for each type

variables

adopt thread-local storage

- adopt existing practice
  - 6 vendors with few syntactic variations
- define new storage duration and class
  - `thread_local int var;`
- 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_local vector<int> var;`
- carefully define initialization to permit lazy allocation for dynamic libraries

initialization of static-duration variables

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

function-local static storage

- initialization implicitly synchronized
  - while not holding any locks
- made possible by a new algorithm
  - developed by Mike Burrows
  - contributed to the community by Google

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, no uses
vector<int> v(u); // okay, within unit
vector<int> w(e); // error, out of unit
```

destruction

- first terminate all threads
  - more later
- execute destructors in a concurrent reverse of initialization
- taking care to interleave namespace-scope variables with function-scope static variables
- same restrictions on use of variables outside current translation unit

threads

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.join();
    // 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
}
```

detached threads

- executing the destructor of a live thread “detaches” the thread
- may cause dangling references and race conditions
- a minority of the committee thinks “mistake”
- a majority of the committee thinks “existing practice”

exception example

- suppose g() throws an exception

```
extern int f(int), g(int);
int f(int a) {
    int b;
    std::thread t(
            [&]{ b = f(a); } );
    int c = g(a);
    t.join();
    return b+c;
}
```

scheduling

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

synchronize

mutexes

- exclusive (single reader/writer)
- recursive or not
- timed or not
- probably more a year or two later

locks

- hold a mutex within a given scope
- does mutex **`lock()`** in constructor
- does mutex **`unlock()`** in destructor

```
std::mutex m;

{
    std::unique_lock< std::mutex >
        l( m ); // m.lock()
    . . . . .
    // m.unlock()
}
```

condition variables

- threads may wait on a condition variable
  - giving up their lock on the mutex.
- threads may notify a condition variable
  - notified threads re-lock 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::mutex mx;
  std::condition_variable not_full,
                          not_empty;
public:
  void insert( int arg ) {
    std::unique_lock< std::mutex > lk(mx);
    while ( (head+1)%10 == tail )
      not_full.wait(lk);
    store[head] = arg; head = (head+1)%10;
    not_empty.notify();
  }
```

termination

voluntary

- return from outermost function
- unable to standardize a mechanism for cancellation
- the new **`quick_exit`** facility enables terminating the process without cooperation

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 ) );
    ```
  - ```
    std::rethrow_exception( copied );
    ```
    ```

input and output

- none of these points include I/O
- no mechanism to shut down a thread waiting on I/O
- partially due to weaknesses in operating system interfaces
- no resolution yet

and beyond

high-level later

- some work is 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

conclusions

the basics are on track

- memory model
- atomic operations
- non-automatic variables
- threads and synchronization

some features need care

- thread termination
- exception propagation

the real value comes later

- the standard provide the means to abstract over the basics
- the users will write some really great high-level facilities
- most programming should be done at this level

more information

- C++ standard website
  - http://www.open-std.org
  - WG21 - C++
  - WG papers
  - 2008
  - N2597
- questions?