Document Number: WG21/N1680=J16/04-0120
Date: 10 September 2004
Reply to: Andrei Alexandrescu
andrei@metalanguage.com
2216 NE 46th St. Apt. A
Seattle WA 98105 USA

Memory model for multithreaded C++

Andrei Alexandrescu Hans Boehm Kevlin Henney
Doug Lea Bill Pugh

Abstract

The C++ Standard defines single-threaded program execution. Fundamentally, multithreaded execution requires a much more refined memory and execution model. C++ threading libraries are in the awkward situation of specifying (implicitly or explicitly) an extended memory model for C++ in order to specify program execution. We propose integrating a memory model suitable for multithreaded execution in the C++ Standard. On top of that model, we propose a standard threading library.

1 Introduction

Many of today's applications make use of multithreaded execution. We expect such use to grow as the increased use of hardware multithreading (a.k.a. "hyperthreading") and multi-core processors will force or entice more and more applications to become multithreaded. C++ is commonly used as part of multithreaded applications, sometimes with either direct calls into an OS-provided threading library (e.g. POSIX threads (pthreads) [3] or Win32 threads) or with the aid of an intervening layer that provides a platform-neutral interface (e.g. Boost Threads). However, concurrent programming is inherently diverse, and includes high-performance parallel programming, asynchronous task processing, message-based and event-based systems, non-blocking, lock-free and optimistic data structures, transactional approaches, and so on.

Unfortunately, programming multithreaded applications of any kind in C++ remains a black art. Properties critical for reliable, efficient, and correct multithreaded execution are left unspecified. The Win32 threading model is notoriously underspecified (and by extension, any portable layer that relies on it, such as Boost Threads). POSIX threads are more rigorously defined, but pthreads only specify a C binding; C++ programs using pthreads enjoy vicarious correctness at best. The C++ Standard specifies program execution in terms of *observable behavior*, which in turn describes sequential execution on an implicitly single-threaded abstract machine. Thus, currently, many questions that programmers have about how to predict and control multithreaded code simply

do not have answers. Superstition, myths, bad advice, and "hey, it worked for me" stories are rampant.

Absent a clearly defined memory model as a common ground between the compiler, the hardware, the threading library, and the programmer, multithreaded C++ code is fundamentally at odds with compiler and processor-level optimizations [4]. Even relatively simple uses can fail unexpectedly due to unforeseen compiler optimizations. These optimizations are safe in a single-threaded context, but not with multiple threads. Additionally, the specification does not address at all the interactions of memory operations with atomic update operations or constructs such as locks built out of them. (The pthreads specification does not help much here, since locking semantics are not fully integrated with the semantics of other memory operations.)

These problems can be solved without introducing additional keywords or syntactic constructs. The main plan of attack is:

1. Specification of an abstract memory model describing the interactions between threads and memory.

2. Application of this model to existing aspects of the C++ specification to replace the current implicitly sequential semantics. This will entail new constraints on how compilers can emit and optimize code. In particular, this will entail a reworking of the specification of `volatile` to provide useful multithreaded semantics.

3. Introduction of a small number of standard library classes providing standardized access to atomic update operations (such as `compare_and_set`). These classes will have multithreaded semantics integrated with the above specifications for other memory operations. Thus, compilers will need to treat these as *intrinsics*. These operations form the low-level basis for modern multithreaded synchronization constructs such as locks, and are also required in the construction of efficient non-blocking data structures.

4. Definition of a standard thread library that provides similar functionality to pthreads and Win32 threads, but meshes with the rest of the C++ standard.

This is an ambitious proposal, and the current draft represents only the first step of a long process. In the remainder of this draft, we briefly describe the fundamental issues, some concrete steps in addressing them, and sketch out remaining work.

2 A Memory Model

A memory model describes the behavior of threads with respect to basic memory operations – mainly reads and writes of variables potentially accessible across multiple threads. The main questions addressed by a memory model include:

Atomicity: Which memory operations have indivisible effects?

Visibility: Under what conditions will the effects of a write action by one thread be seen by a read by another thread?

Ordering: Under what conditions are sequences of memory operations by one or more threads guaranteed to be visible in the same order by other threads?

Until recently, memory models had been described rigorously only for hardware systems, not programming languages. However, the work underlying the revised Java Memory Model specification [1] and related efforts have resulted in models that can be readily applied to C++. We anticipate that this part of the specification will mainly be a matter of adapting, not creating, a formal model. As such, the process of defining a sound memory model for C++ can reuse the years-long effort that was invested in defining, peer-reviewing, refining, and debugging the mentioned formal model. We believe such reuse to be tremendously beneficial in terms of correctness and time savings.

3 Memory Effects

A memory model defines categories of memory actions. For example actions that act as *acquire* and *release* operations guarantee visibility of a set of updates by one thread to another. The next step for a language specification is to map these notions to all of the memory-related constructions in the language. This process entails nailing down a large set of "small issues" that are necessary for programmers to be able to predict and control effects. Areas that we have so far identified include:

Atomicity A given platform may guarantee atomicity only for reads and writes of certain bit widths and alignments. The spec must permit these to vary, and must therefore provide some means for programs to query these properties.

Extra writes There are several cases in C++ in which compilers and machines have historically been permitted to issue writes that are not obvious from inspection of source code. The most notable examples involve structures with small fields. For example, given:

```
struct S { short a; char b; char c; } s;
```

an assignment such as `s.a = 0` might be executed as if the code were `*(int*)&s = 0` if a compiler infers from context that `b` and `c` are zero as well, as in the following example:

```
void Fun(S& s) {
  if (s.b == 0 && s.c == 0) {
      s.a = 0;
```

```
  }
}
```

Another instance of this problem is with hardware that doesn't support memory writes below a certain size (often, `sizeof(int)`). In that case, if `S` is "packed" such that `sizeof(S) == sizeof(int)`, the compiler must generate code for `s.a = 0` to read the entire `s` object in a register, mask a portion of it, and write back the entire object to memory. Effectively, simply writing the field `S::a` became a (possibly non-atomic) read-write operation as far as the neighboring fields `S::b` and `S::c` are concerned.

This would be unexpected at best in a multithreaded context in which the other fields were also being assigned concurrently. A spec must clearly define whether and when such compiler transformations remain legal.

Volatile data In the current language spec, the `volatile` qualifier is mainly used to indicate guaranteed order of reads and writes within single-threaded semantics—for example for device control registers, memory-mapped I/O, or opaque flow (as in `setjmp` or interrupts). In a multi-threaded language, it may be useful for `volatile` to take on the extra burden of constraining inter-thread visibility and ordering properties. There are a few options for the detailed semantics. In the simplest, `volatile` reads act as `acquire` and writes as `release`. This has the virtue of being relatively easy to use by programmers who are not intimately familiar with memory models. For example, the infamous "double-checked locking" idiom [4] works as expected under these rules if references are declared as `volatile` (and other lock-based rules below are followed). This has the disadvantage of imposing "heavier" constraints on the compiler and processor than necessary in very performance-sensitive applications. However, optimizers can often eliminate unnecessary operations (such as consolidating several consecutive `acquire` and `release` operations into one). To help optimizers in the few remaining situations, weaker forms could also be provided as operations on atomic variables, as discussed below.

Opaque calls One concern about moving to multithreaded specifications is that compilers may become overly conservative when compiling code with opaque function calls—flushing and reloading registers and/or issuing memory barriers in case the called function's effects depend on this. It may be desirable to allow programmers to control this using some kind of qualifier. Options include those with defaults in both directions; for example, assuming lack of effects unless a function is qualified as, say, `mutable`; versus assuming effects unless qualified with some extended form of `const`. Alternatively, or in addition, the spec could include a means for programmers to tell compilers that a certain program is either definitely single-threaded or definitely multithreaded, as a way of controlling certain optimizations. Further exploration of options and their consequences is

needed. These considerations are very related to an existing C++ standardization proposal [2].

4 Atomics

Atomic update operations (as well as associated memory barrier instructions, which impose memory ordering constraints on the processor) form the basis for essentially all modern multithreaded synchronization and coordination. While there is some diversity across architectures in the nature and style of these instructions, there is enough commonality in current and medium-term-future systems to define a small set of intrinsics that can be used for portable concurrent programming. There are several stylistic options here. One approach is to define three small intrinsified inlinable classes, one each holding a single value of type **int**, **long**, and (templated) pointer, and supporting operations such as:

```
namespace std {
  class atomic_int {
  public:
     int get();
     int set(int v);
     bool compare_and_set(int expected_value, int new_value);

     int weak_get();
     int weak_set(int v);
     bool weak_compare_and_set(int expected_value,
         int new_value);

     // other minor convenience functions, including:
     int get_and_increment();
     int get_and_add(int v);
     // ...
  };
}
```

The main attraction of this approach is that it appears to be implementable on essentially any platform. Even those machines without such primitives can emulate them using private locks. And even though some machines (such as PowerPC) support LL/SC (*load-linked, store-conditional*) instead of CAS (*compare-and-set*), in practice, nearly all usages of LL/SC are to perform CAS (the reverse is impossible), so there would rarely be motivation to resort to non-standardized, non-portable constructions even on these platforms.

The idea of the "weak" versions is to permit finer control of atomics and barriers than otherwise available using **volatile** or other constructions. For example, a **weak_set** need only perform a store ordering barrier, not a full *release*, which may be cheaper on some machines. (Details are yet to be fully worked out. Additionally, a refinement of this approach relying more heavily on

templates and traits will unify the interface appropriately.)

5 Thread Library

Currently, multithreaded C++ programs tend to rely primarily on one of a fairly small set of libraries for threading support: POSIX threads, Win32, ACE, and Boost. These possess many more similarities than differences. The opportunity arises to provide a standard library that conservatively abstracts over such packages.

Even if this is not done, such libraries must, to conform to the rest of this proposal, specify their basic locking primitives in terms of the memory model. All basic locks should and do provide semantics in accord with the basic acquire and release actions specified by the Standard. Compilers in turn must respect these semantics. The mechanics to ensure this would rely on how the opaque call issue mentioned above is resolved.

In this draft we do not even sketch out the APIs of this library. A future draft will include proposed classes developed with the involvement and feedback from users and developers of existing libraries.

References

[1] Tim Lindholm et al. Java Specification Request 133: Memory Model and Thread Specification Revision. Available at `http://www.jcp.org/jsr/detail/133.jsp`.

[2] Walter E. Brown et al. Toward Improved Optimization Opportunities in C++0X. Document WG21/N1664 = J16/04-0104; available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1664.pdf`, Jul 2004.

[3] IEEE Standard for Information Technology. *Portable Operating System Interface (POSIX) — System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. ANSI/IEEE 1003.1c-1995, 1995.

[4] Scott Meyers and Andrei Alexandrescu. C++ and The Perils of Double-Checked Locking. *Doctor Dobb's Journal*, Jul 2004.