C/C++ Thread Safety Annotations

Le-Chun Wu <lcwu at google.com> *Modified:June 9, 2008*

Objective

This project creates a set of C/C++ program annotations that (1) allow developers to document multi-threaded code so that maintainers can avoid introducing thread safety bugs, and (2) help program analysis tools identify potential thread safety issues. We add a new GCC analysis pass that uses the source annotations to identify thread safety issues and emit compiler warnings.

Background

Multi-threading is an increasingly important technique to boost performance on multi-core/multiprocessor systems. Unfortunately, multi-threaded programming is hard: timing-dependent bugs, such as data races and deadlocks, are very difficult to expose in testing and hard to reproduce and isolate once discovered. Proper documentation of synchronization policies and thread safety guarantees is probably one of the most useful techniques to manage multi-threaded code and avoid concurrency bugs. In practice, programmers' intended synchronization policies, such as lock acquisition order and lock requirement for shared variables and functions, are often documented in comments. Comments help maintainers avoid introducing errors, but it is hard for program analysis tools to use the information to tell programmers when they have violated their synchronization policies and identify potential thread safety issues. Therefore this project creates program annotations for C/C++ to help developers document locks and how they need to be used to safely read and write shared variables. We design and implement a new GCC pass that uses the annotations to identify and warn about the issues that could potentially result in race conditions and deadlocks.

Overview

There are many styles of synchronization in multi-threaded programming. The annotations used here only focus on the mutex lock-based synchronization. The annotations are implemented in GCC's "attribute" language extension. The following is a list of C macro definitions using the proposed new attributes. We define these macros here to simplify the examples and discussions in this document. It is also a common practice for people to use the macros instead of the raw GCC attributes for code portability and compatibility.

```
#define GUARDED_BY(x)          __attribute__ ((guarded_by(x)))
#define GUARDED_VAR            __attribute__ ((guarded))
#define PT_GUARDED_BY(x)       __attribute__ ((point_to_guarded_by(x)))
#define PT_GUARDED_VAR         __attribute__ ((point_to_guarded))
#define ACQUIRED_AFTER(...)    __attribute__ ((acquired_after(__VA_ARGS__)))
#define ACQUIRED_BEFORE(...)   __attribute__ ((acquired_before(__VA_ARGS__)))
#define LOCKABLE               __attribute__ ((lockable))
#define SCOPED_LOCKABLE        __attribute__ ((scoped_lockable))
#define EXCLUSIVE_LOCK_FUNCTION(...)    __attribute__
((exclusive_lock(__VA_ARGS__)))
#define SHARED_LOCK_FUNCTION(...)       __attribute__
((shared_lock(__VA_ARGS__)))
```

```
#define EXCLUSIVE_TRYLOCK_FUNCTION(...) __attribute__
((exclusive_trylock(__VA_ARGS__)))
#define SHARED_TRYLOCK_FUNCTION(...)    __attribute__
((shared_trylock(__VA_ARGS__)))
#define UNLOCK_FUNCTION(...)            __attribute__ ((unlock(__VA_ARGS__)))
#define LOCK_RETURNED(x)                __attribute__ ((lock_returned(x)))
#define LOCKS_EXCLUDED(...)             __attribute__
((locks_excluded(__VA_ARGS__)))
#define EXCLUSIVE_LOCKS_REQUIRED(...)   __attribute__
((exclusive_locks_required(__VA_ARGS__)))
#define SHARED_LOCKS_REQUIRED(...)      __attribute__
((shared_locks_required(__VA_ARGS__)))
#define NO_THREAD_SAFETY_ANALYSIS       __attribute__
((no_thread_safety_analysis))
```

Note that the annotations proposed here are not expressive enough to handle fine-grained locking relationship between locks and the guarded variables, e.g. when each individual element (or a group of elements) of a linked list/hash table is guarded by a different lock. While most of the proposed annotations are designed for documenting synchronization policies, some are simply created to help program analysis tools. Detailed explanation of the annotations and their usage are discussed in the next section.

Detailed Design

Variable Annotations

The following annotations are used to specify synchronization policies, such as which variables are guarded by which locks and the acquisition order of locks.

- **GUARDED_BY(*lock*)** and **GUARDED_VAR**
  These two annotations document a shared variable/field that needs to be protected by a lock. GUARDED_BY specifies a particular lock should be held when accessing the annotated variable. GUARDED_VAR only indicates a shared variable should be guarded (by any lock). GUARDED_VAR is primarily used when the client cannot express the name of the lock. The lock argument in GUARDED_BY (or in any other annotations mentioned below that take lock arguments) can be a variable, a class member, or even an expression specifying an object. The following example shows how the annotation is used.

```
Mutex mu1;
int x GUARDED_BY(mu1);
class Foo {
  private:
    Mutex mu3_;
    int count_ GUARDED_BY(mu3_);
    int data_ GUARDED_VAR;
};
```

- **PT_GUARDED_BY(*lock*)** and **PT_GUARDED_VAR**
  These two annotations document a memory location pointed to by a pointer that should be guarded by a lock when dereferencing the pointer. Similar to GUARDED_BY and GUARDED_VAR annotations, PT_GUARDED_BY takes a lock argument while PT_GUARDED_VAR does not, and PT_GUARDED_VAR is primarily used when the client cannot express the name of the lock. Use of these annotations can help analysis tools which don't perform pointer/alias analysis. A pointer to a shared memory location could itself be a shared variable. For example, if a shared global pointer q, which is

guarded by mu1, points to a shared memory location that is guarded by mu2, q should be annotated as follows:

```
int *q GUARDED_BY(mu1) PT_GUARDED_BY(mu2);
```

Note that the currently proposed annotations cannot express a pointer that can point to multiple shared variables guarded by different locks. (We could potentially extend PT_GUARDED_BY to take multiple locks but then the analysis would be less precise.)

- **ACQUIRED_AFTER(...)** and **ACQUIRED_BEFORE(...)**
  These two annotations document the acquisition order between locks that can be held simultaneously by a thread. The arguments specify the locks that need to be either acquired before the annotated lock (for ACQUIRED_AFTER) or after the annotated lock (for ACQUIRED_BEFORE). For any two locks that require an acquisition order between them, only one of them needs to be annotated although it is OK to annotate both of them as long as they are consistent. The acquisition_order relation specified by these annotations is transitive and establishes a partial order on the locks. In the following example if a thread can hold both mutex locks mu1 and mu3 concurrently at some point of program execution, mu1 must be acquired first.

```
Mutex mu1;
Mutex mu2 ACQUIRED_AFTER(mu1);
Mutex mu3 ACQUIRED_AFTER(mu2);
```

Class/Type Annotations

The following annotations are used for classes and types. Besides documenting whether a class/type defines a lockable type, these annotations are primarily used by the analysis tools for sanity and integrity checks. For example, when processing annotations, an analysis tool can check whether the arguments of GUARDED_BY and ACQUIRED_AFTER are of LOCKABLE type. Or it can check whether the constructors of a scoped lock class are annotated as lock primitives with a lock argument. (See below.)

- **LOCKABLE**
  This annotation documents if a class/type is a lockable type (e.g. a mutex lock class).

- **SCOPED_LOCKABLE**
  This annotation documents if a class is a scoped lock type. A scoped lock object acquires a lock at construction and releases it when the object goes out of scope.

Examples of how these annotations are used can be found later in this section.

Function Annotations

The following annotations are used for functions and class methods. They (1) specify the lock and unlock primitives and/or (2) document the lock requirements for the annotated functions. These annotations also allow the analysis tools to check whether certain locks should (or should not) be held at call sites, and whether accesses to shared data are properly guarded in the callee bodies without interprocedural analysis.

- **EXCLUSIVE_LOCK_FUNCTION(...)**, **SHARED_LOCK_FUNCTION(...)**, and **UNLOCK_FUNCTION(...)**
  These annotations are used for specifying lock and unlock primitives. The optional arguments specify the locks that are acquired or released by the primitives. As mentioned earlier, the lock arguments can be variables, class members, and

expressions specifying an object. Here they can also be formal parameters or parameter positions (1-based) of the lock/unlock primitives. For the lock and unlock methods of a lockable class, the annotations don't need to take any argument. Here are several examples of how these annotations are used.

```
    Mutex mu1;
    int pthread_mutex_lock(Mutex *mutex) EXCLUSIVE_LOCK_FUNCTION(mutex);
    int pthread_mutex_unlock(Mutex *mutex) UNLOCK_FUNCTION(1);
    int my_mutex_lock(int i, Mutex *foo, Mutex *bar)
EXCLUSIVE_LOCK_FUNCTION(foo, 3, mu1);

    class LOCKABLE Mutex {
      public:
        void Lock() EXCLUSIVE_LOCK_FUNCTION();
        void ReaderLock() SHARED_LOCK_FUNCTION();
        void Unlock() UNLOCK_FUNCTION();
    };
```

Since a scoped lock acquires a lock in its constructor(s) and releases the lock in the destructor, the constructors and destructor of a scoped lock class need to be annotated as lock/unlock primitives.

```
    class SCOPED_LOCKABLE MutexLock {
      public:
        MutexLock(Mutex *mu) EXCLUSIVE_LOCK_FUNCTION(mu);
        ~MutexLock() UNLOCK_FUNCTION();
      private:
        Mutex *mu_;
    };
```

Note that if the lock/unlock primitives are declared in a header file that you cannot change, you can re-declare the primitives with proper annotations in your source.

- **EXCLUSIVE_TRYLOCK_FUNCTION(*succ_retval*, ...)** and **SHARED_TRYLOCK_FUNCTION(*succ_retval*, ...)**
  These annotations are used for specifying trylock primitives. Similar to the annotations for lock primitives, trylock annotations take optional arguments that specify the locks the primitives try to acquire. In addition, EXCLUSIVE_TRYLOCK and SHARED_TRYLOCK take a mandatory integer or boolean argument (*succ_retval*) that specifies the return value of a successful lock acquisition. (For example, pthread_mutex_trylock() returns 0 when it successfully grabs the lock, while some other implementation of trylock might return true on successful acquisition.) The following shows examples of their usage.

```
    int pthread_mutex_trylock(Mutex *mutex) EXCLUSIVE_TRYLOCK_FUNCTION(0,
mutex);

    class LOCKABLE Mutex {
        ...
      public:
        bool TryLock() EXCLUSIVE_TRYLOCK_FUNCTION(true);
        bool ReaderTryLock() SHARED_TRYLOCK_FUNCTION(true);
        ...
    };
```

- **EXCLUSIVE_LOCKS_REQUIRED(...)**, **SHARED_LOCKS_REQUIRED(...)**, and **LOCKS_EXCLUDED(...)**
  These annotations document lock requirements of functions/methods. If a function expects certain locks to be held before it is called, it needs to be annotated with

EXCLUSIVE_LOCKS_REQUIRED and/or SHARED_LOCKS_REQUIRED. The lock arguments specify the required locks. On the other hand, if a function acquires non-reentrant locks in its body, those locks need to be documented using LOCKS_EXCLUDED to indicate that they cannot be held when entering this function. For example, if function foo needs to be guarded by mutex locks mu1 and mu2 when it is called, and it acquires a non-reentrant lock mu3 in its body, it should be annotated as shown below.

```
    int x GUARDED_BY(mu1);
    int y GUARDED_BY(mu2);
    int z GUARDED_BY(mu3);

    // Forward declaration for foo is necessary as GCC attributes can only
    // be used in declarations, not in definitions.
    int foo(int a) EXCLUSIVE_LOCKS_REQUIRED(mu1)
SHARED_LOCKS_REQUIRED(mu2)
                   LOCKS_EXCLUDED(mu3);

    int foo(int a)
    {
      int res;
      x = a + y;
      mu3.Lock();
      z = x * a;
      res = z;
      mu3.Unlock();
      return res;
    }
```

These annotations allow the analysis tools to avoid interprocedural analysis when analyzing lock requirements. Note that they are intended to be applied to internal/private functions/methods, not to public APIs. A properly designed module or class shouldn't export its locking requirements.

- **LOCK_RETURNED(*lock*)**
  If a function/method returns a lock without acquiring it, the returned lock needs to be documented using LOCK_RETURNED annotation, as shown in the following example. This annotation is created simply to help the analysis tools to recognize and canonicalize locks during analysis.

```
    class Foo {
      private:
        Mutex mu_;
        int data GUARDED_BY(mu_);
        Mutex *get_lock() LOCK_RETURNED(mu_) { return &mu_; }

      public:
        void incData(int a) LOCKS_EXCLUDED(mu_) {
          get_lock()->Lock();
          data += a;
          get_lock()->Unlock();
        }
    };
```

- **NO_THREAD_SAFETY_ANALYSIS**
  This annotation is simply an escape hatch for analysis tools to ignore the annotated function in thread safety analysis. For example, if function Foo is guaranteed to be

accessed by a single thread, or somehow we were unable to document/annotate the locking policy in function Foo and the compiler kept emitting annoying warning messages which can be ignored, we can prevent the thread safety analysis on function Foo by annotating its declaration as shown below:

```
int Foo(int a, int b) NO_THREAD_SAFETY_ANALYSIS;
```

GCC Analysis Pass

We have modified GCC to handle the proposed thread safety attributes, and designed and implemented a new analysis pass that uses the annotations to detect and warn about potential issues that could result in data races and deadlocks. The thread safety issues detected by the analysis pass include:

- Accesses to shared variables and function calls are not guarded by proper (read or write) locks
- Locks are not acquired in the specified order
- A cycle in the locking order
- Try to acquire a lock that is already held by the same thread
    - Useful when mutex locks are non-reentrant
- Locks are not acquired and released in the same routine (or in the same block scope)
    - Having critical sections starting and ending in the same routine is a good practice

The new GCC pass is basically driven by a data-flow analysis. It computes "live" lock sets at each program point and uses the annotations to decide when to add/remove locks to/from the sets. Using the live lock sets and annotations on shared variables and functions, the analyzer is able to verify whether the variables and functions are well protected. For more detailed information, please see the comments and code in tree-threadsafe-analyze.[ch].

A new warning flag -Wthread-safety is added to control the thread safety analysis. To enable the analysis, use the following flags in the GCC command line. The flag -O is needed here because the analysis requires SSA which is entered only when the optimization is enabled.

```
-Wthread-safety -O
```

We also add a set of new warning flags to provide a finer-grain control of thread safety analysis. These flags are only effective (and enabled by default) when the flag -Wthread-safety is used. They are primarily used in the negative form to disable individual thread safety issue detection.

- -Wthread-unguarded-var
  Warn about shared variables not properly protected by locks specified in the annotations

- -Wthread-unguarded-func
  Warn about function calls not properly protected by locks specified in the annotations

- -Wthread-mismatched-lock-acq-rel
  Warn about mismatched lock acquisition and release, e.g. acquiring a lock without releasing it in the same function

- -Wthread-mismatched-lock-order
  Warn about lock acquisition order inconsistent with what specified in the annotations

- -Wthread-reentrant-lock

Warn about a lock being acquired recursively

A new built-in macro __SUPPORT_TS_ANNOTATION__ is defined when the thread safety analysis is enabled. Programmers can use this built-in macro to make their annotation macro definitions portable as in the following example.

```
#if defined(__GNUC__) && defined(__SUPPORT_TS_ANNOTATION__)

#define LOCKABLE            __attribute__ ((lockable))
#define SCOPED_LOCKABLE     __attribute__ ((scoped_lockable))
#define GUARDED_BY(x)       __attribute__ ((guarded_by(x)))
#define GUARDED_VAR         __attribute__ ((guarded))
#define PT_GUARDED_BY(x)    __attribute__ ((point_to_guarded_by(x)))
#define PT_GUARDED_VAR      __attribute__ ((point_to_guarded))
#define ACQUIRED_AFTER(...) __attribute__ ((acquired_after(__VA_ARGS__)))
#define ACQUIRED_BEFORE(...) __attribute__ ((acquired_before(__VA_ARGS__)))

#else

#define LOCKABLE
#define SCOPED_LOCKABLE
#define GUARDED_BY(x)
#define GUARDED_VAR
#define PT_GUARDED_BY(x)
#define PT_GUARDED_VAR
#define ACQUIRED_AFTER(...)
#define ACQUIRED_BEFORE(...)

#endif // defined(__GNUC__) && defined(__SUPPORT_TS_ANNOTATION__)
```

Examples

Here are several annotated code examples along with the compiler outputs when they are compiled with the thread safety analysis enabled:

- **designdoc_ex1.cc**

```
 1 #include "annotations.h"
 2
 3
 4 Mutex mu1;
 5 Mutex mu2 ACQUIRED_AFTER(mu1);
 6 Mutex mu3 ACQUIRED_AFTER(mu2);
 7
 8 int gx  GUARDED_BY(mu1) = 3;
 9 float gy GUARDED_BY(mu2) = 5.0;
10 int *gp GUARDED_BY(mu3) PT_GUARDED_BY(mu1);
11 int gw GUARDED_VAR = 4;
12
13 void func1(void) LOCKS_EXCLUDED(mu1, mu2, mu3);
14
15 class Foo {
16  private:
17   Mutex mu_ ACQUIRED_AFTER(mu2);
18   int a_ GUARDED_BY(mu_);
```

```
 19   float b_ GUARDED_BY(mu_);
 20
 21  public:
 22
 23   Foo() : a_(1) { b_ = 5.0; }
 24
 25   ~Foo() {}
 26
 27   void incrementA(int i)
 28   {
 29     a_ += i;
 30   }
 31
 32   float decrementB(float f) LOCKS_EXCLUDED(mu_)
SHARED_LOCKS_REQUIRED(mu1)
 33   {
 34     float res;
 35     mu_.Lock();
 36     if (gx > 2)
 37       b_ -= f;
 38     else
 39       b_ -= 2 * f;
 40     res = b_;
 41     mu_.Unlock();
 42     return res;
 43   }
 44 };
 45
 46 void func1(void)
 47 {
 48   int la;
 49   float *p PT_GUARDED_BY(mu2) = &gy;
 50   Foo foo;
 51
 52   gp = &gx;
 53
 54   if (gx > 3) {
 55     la = gx + gw;
 56   }
 57   else {
 58     *p = foo.decrementB(gy);
 59   }
 60
 61   foo.incrementA(la);
 62
 63   if (la < 10) {
 64     *gp = 7;
 65   }
 66 }
```

The compiler output:

```
$ g++ -Wthread-safety -O -c designdoc_ex1.cc
designdoc_ex1.cc: In member function 'void Foo::incrementA(int)':
designdoc_ex1.cc:29: warning: Reading variable 'a_' requires lock 'mu_'
designdoc_ex1.cc:29: warning: Writing to variable 'a_' requires lock 'mu_'
designdoc_ex1.cc: In function 'void func1()':
```

```
designdoc_ex1.cc:52: warning: Writing to variable 'gp' requires lock 'mu3'
designdoc_ex1.cc:54: warning: Reading variable 'gx' requires lock 'mu1'
designdoc_ex1.cc:55: warning: Reading variable 'gx' requires lock 'mu1'
designdoc_ex1.cc:55: warning: Access to variable 'gw' requires a lock
designdoc_ex1.cc:58: warning: Reading variable 'gy' requires lock 'mu2'
designdoc_ex1.cc:58: warning: Calling function 'decrementB' requires lock
'mu1'
designdoc_ex1.cc:58: warning: Access to memory location pointed to by
variable 'p' requires lock 'mu2'
designdoc_ex1.cc:64: warning: Reading variable 'gp' requires lock 'mu3'
designdoc_ex1.cc:64: warning: Access to memory location pointed to by
variable 'gp' requires lock 'mu1'
```

- **designdoc_ex2.cc**

```
 1 #include "annotations.h"
 2
 3 Mutex mu1;
 4 Mutex mu2 ACQUIRED_AFTER(mu1);
 5
 6 int x GUARDED_BY(mu1);
 7 int a GUARDED_BY(mu2);
 8
 9 void foo()
10 {
11   mu2.Lock();
12   mu1.Lock();
13   if (x > 2)
14     a = x + 1;
15   else
16     a = x - 1;
17   mu1.Unlock();
18   mu2.Unlock();
19 }
```

The compiler output:

```
$ g++ -Wthread-safety -O -c designdoc_ex2.cc
designdoc_ex2.cc: In function 'void foo()':
designdoc_ex2.cc:12: warning: Lock 'mu1' is acquired after lock 'mu2'
(acquired at line 11) but is annotated otherwise
```

- **designdoc_ex3.cc**

```
 1 #include "annotations.h"
 2
 3 class Foo {
 4  private:
 5   Mutex mu_;
 6   Mutex mu2_;
 7   int a_ GUARDED_BY(mu_);
 8   int b_ GUARDED_BY(mu2_);
 9
10  public:
11   bool func(int y)
12   {
13     int x;
14     x = y - 2;
15     if (mu_.TryLock()) {
16       a_ = x * 3;
```

```
 17       if (mu2_.TryLock()) {
 18         b_ += y;
 19         mu2_.Unlock();
 20       }
 21       mu_.Unlock();
 22       return true;
 23     }
 24     mu_.Unlock();
 25     return false;
 26   }
 27 };
 28
 29 Mutex mu1 ;
 30
 31 Foo *foo GUARDED_BY(mu1);
 32
 33 int main()
 34 {
 35   bool a, b, c;
 36   a = mu1.TryLock();
 37   b = !a;
 38   c = !b;
 39   if (c)
 40     {
 41       return 1;
 42     }
 43   foo->func(2);
 44   mu1.Unlock();
 45   return 0;
 46 }
```

The compiler output:

```
$ g++ -Wthread-safety -O -c designdoc_ex3.cc
designdoc_ex3.cc: In member function 'bool Foo::func(int)':
designdoc_ex3.cc:24: warning: Try to unlock 'mu_' that was not acquired
designdoc_ex3.cc: In function 'int main()':
designdoc_ex3.cc:43: warning: Reading variable 'foo' requires lock 'mu1'
designdoc_ex3.cc:44: warning: Try to unlock 'mu1' that was not acquired
designdoc_ex3.cc:36: warning: Lock 'mu1' (acquired at line 36) is not
released at the end of its scope in function 'main'
```

Known Issues

As mentioned in the overview, the annotations proposed in this design only focus on the mutex lock-based synchronization. Also they are not expressive enough to handle fine-grained locking relationship. For example, the annotations cannot express the lock requirements for elements of a hash map when each of them (or a group of them) is guarded by a different lock.

The lock variables/expressions that can be specified in the annotations are limited by GCC frontend's capability of attribute argument handling. For example, the code like the following:

```
class Foo {
  ...
 private:
  Mutex mu;
```

```
  friend class Bar;
};

class Bar {
 public:
  explicit Bar(Foo *foo);
  ...
 private:
  Foo *foo;
  void UseFoo() EXCLUSIVE_LOCKS_REQUIRED(foo->mu);
};
```

will produce a GCC error like:

```
error: invalid use of non-static data member 'Bar::foo'
```

Another known issue with the current implementation is that PT_GUARDED_BY and PT_GUARDED_VAR annotations don't support smart pointers. They can only be applied to "real" pointers. For example, if we have code like the following:

```
Mutex mu;
boost::scoped_ptr a PT_GUARDED_BY(mu);

void foo()
{
  *a = 5;
}
```

the current implementation will emit the following warning message:

```
warning: 'point_to_guarded_by' attribute ignored for a non-pointer
```