Thread Safety Annotations for Clang

DeLesley Hutchins <delesley@google.com>

Outline of Talk

Google™

- **Why...** we need thread annotations.
- **What...** the annotations are, and what they do.
- **How...** thread annotations are implemented in clang.
- **Huh?** (Current challenges and future work).

Why...

...we need thread safety annotations

The problem with threads... Google™

- Everybody wants to write multi-threaded code.
  - ... multi-core ... Moore's law ... power wall ... etc.
- Threading bugs (i.e. race conditions) are insidious.
- Race conditions are hard to see in source code:
  - Caused by interactions with other threads.
  - Not locally visible when examining code.
  - Not necessarily caught by code review.
- Race conditions are hard to find and eliminate:
  - Bugs are intermittent.
  - Hard to reproduce, especially in debugger.
  - Often don't appear in unit tests.

Real World Example:

- Real world example at Google.
- Several man-weeks spent tracking down this bug.

```
// a global shared cache
class Cache {
public:
  // find value, and pin within cache
  Value* lookup(Key *K);
  // allow value to be reclaimed
  void release(Key *K);
};

Mutex CacheMutex;
Cache GlobalCache;
```

Example (Part II): A Helper...

```
// Automatically release key when variable leaves scope
class ScopedLookup {
public:
  ScopedLookup(Key* K)
    : Ky(K), Val(GlobalCache.lookup(K))
  { }
  ~ScopedLookup() {
    GlobalCache.release(Ky);
  }
  Value* getValue() { return Val; }

private:
  Key*   Ky;
  Value* Val;
};
```

Example (Part III): The Bug

- Standard threading pattern:
  - lock, do something, unlock...

```
void bug(Key* K) {
  CacheMutex.lock();
  ScopedLookup lookupVal(K);
  doSomethingComplicated(lookupVal.getValue());
  CacheMutex.unlock();
  // OOPS!
};
```

The Fix

```
void bug(Key* K) {
  CacheMutex.lock();
  {
    ScopedLookup lookupVal(K);
    doSomethingComplicated(lookupVal.getValue());
    // force destructor to be called here...
  }
  CacheMutex.unlock();
};
```

Annotation Example:

Google™

```
Mutex CacheMutex;
Cache GlobalCache GUARDED_BY(CacheMutex);

class ScopedLookup {
public:
  ScopedLookup(Key* K) EXCLUSIVE_LOCKS_REQUIRED(CacheMutex)
    : Ky(K), Val(GlobalCache.lookup(K))
  { }
  ~ScopedLookup() EXCLUSIVE_LOCKS_REQUIRED(CacheMutex) {
    GlobalCache.release(Ky);
  }
  ...
};
```

Reporting the bug:

- Now we get a warning:

```
void bug(Key* K) {
  CacheMutex.lock();
  ScopedLookup lookupVal(K);
  doSomethingComplicated(lookupVal.getValue());
  CacheMutex.unlock();
  // Warning: ~ScopedLookup requires lock CacheMutex
};
```

What...

...the annotations are, and what they do

Some History

- Thread safety annotations:
  - Annotate code to specify locking protocol.
  - Verify protocol at compile time.
- Currently implemented within GCC.
  - Original implementation done by Le-Chun Wu
  - See "annotalysis" branch.
- Used in a number of projects at Google.
  - Replaces informal coding style guidelines.
  - Annotations used to be specified in comments.
- Currently porting the analysis to clang.
  - Initial development done by Caitlin Sadowski

Thread Safety Annotations

- Works a lot like type-checking.
  - Annotations associate mutexes with data
    ... defines the threading interface of a class.
  - Machine checking of annotations at compile time.
  - Catch common errors
    (e.g. failure to acquire lock before method call)

- Reference:
  - *Type-based race detection for Java*
    Flanagan and Freund, 2000

Annotation overview

- Acquiring and releasing locks:

```
LOCKABLE
EXCLUSIVE_LOCK_FUNCTION,    SHARED_LOCK_FUNCTION
EXCLUSIVE_TRYLOCK_FUNCTION, SHARED_TRYLOCK_FUNCTION
UNLOCK_FUNCTION
```

- Guarded data:

```
GUARDED_BY, PT_GUARDED_BY
```

- Guarded methods:

```
EXCLUSIVE_LOCKS_REQUIRED,  SHARED_LOCKS_REQUIRED
LOCKS_EXCLUDED
```

- Deadlock detection:

```
ACQUIRED_BEFORE, ACQUIRED_AFTER
```

- And a few misc. hacks...

Defining a Mutex...

- `LOCKABLE` attribute declares mutex classes.
- Other attributes declare lock and unlock functions.

```
class LOCKABLE Mutex {
public:
  // read/write lock
  void lock()               EXCLUSIVE_LOCK_FUNCTION();
  // read-only lock
  void lock_shared()        SHARED_LOCK_FUNCTION();
  void unlock()             UNLOCK_FUNCTION();
  // return true if lock succeeds
  bool try_lock()           EXCLUSIVE_TRYLOCK_FUNCTION(true);
  bool try_lock_shared()    SHARED_TRYLOCK_FUNCTION(true);
};
```

Lock functions, ctd.

- Some methods may acquire another mutex.

```
class MyObject {
public:
  Mutex Mu;
  void lock()   EXCLUSIVE_LOCK_FUNCTION(Mu) { Mu.lock(); }
  void unlock() UNLOCK_FUNCTION(Mu)         { Mu.unlock();}
};

void foo() {
  MyObject Obj1;
  MyObject Obj2;
  Obj1.lock();  // acquires lock Obj1.Mu
  Obj2.lock();  // acquires lock Obj2.Mu
}
```

Protecting data

Google™

- A guard declares the protecting mutex for a data member.

```
class MyObject {
public:
  Mutex Mu;
  int a  GUARDED_BY(Mu);
  int *b PT_GUARDED_BY(Mu);
};

void foo(MyObject &Obj) {
  Obj.a  = 0;        // Warning: requires lock Obj.Mu
  Obj.b  = &Obj.a;   // OK
  *Obj.b = 1;        // Warning: requires lock Obj.Mu
}
```

Guarded methods

Google™

- Methods and functions can also be guarded.
  - `*_LOCKS_REQUIRED` -- must hold lock when calling
  - `LOCKS_EXCLUDED` -- cannot hold lock when calling. (For non-reentrant mutexes.)

```
void foo(MyObject &Obj) EXCLUSIVE_LOCKS_REQUIRED(Obj.Mu) {
  Obj.a = 0;  // OK
}

void bar(MyObject &Obj) LOCKS_EXCLUDED(Obj.Mu) {
  Obj.lock();
  Obj.a = 0;
  Obj.unlock();
}
```

Deadlock detection

- Declaring mutex order:

```
class MyObject {
  Mutex Mu1;
  Mutex Mu2 ACQUIRED_AFTER(Mu1);
};

void foo(MyObject &Obj) {
  Obj.Mu2.lock();
  Obj.Mu1.lock(); // Warning: Mu2 acquired before Mu1
  ...
}
```

How...

...annotations are implemented in Clang

Implementation overview

- Basic algorithm
- Implementation subtleties
  - Parsing
  - Substitution
  - Expression equality
- Limitations of the analysis
- Discussion: gcc vs. clang

Basic Algorithm

Google™

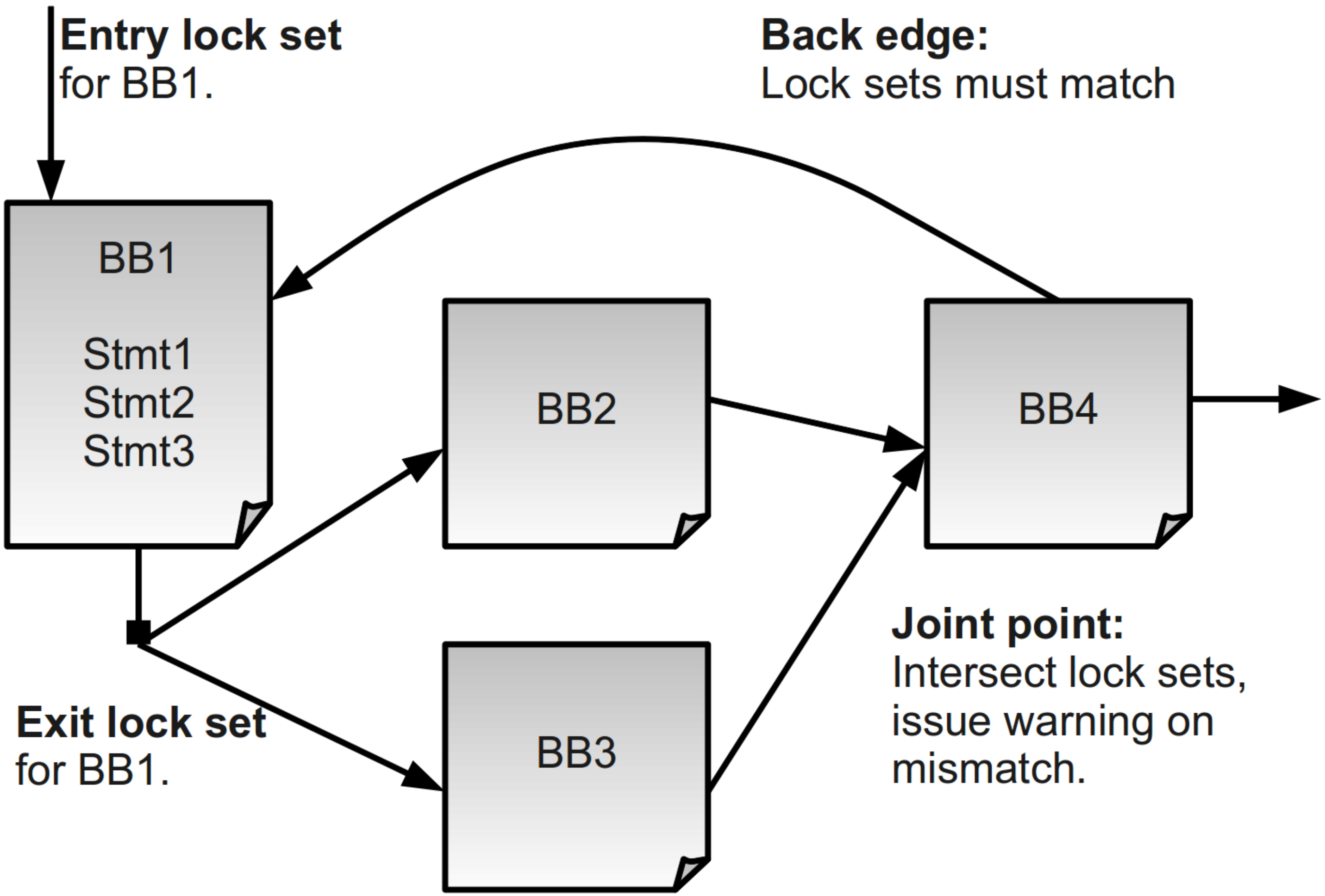
- Traverse the control flow graph.
- Maintain a set of currently held locks.

- On function call:
  - If lock function: add lock to set, check order
  - If unlock function: remove lock from set.
  - If guarded function: check if lock is in set.

- On load or store:
  - If guarded variable: check if lock is in set.

- Current implementation:
  - `lib/Analysis/ThreadSafety.cpp`

Join points and branches



CFG Example:

```
Mutex Mu1, Mu2;

void foo() {
  Mu1.lock();
  if (...) {
    Mu2.lock();
    // Warning: Mu2 was not unlocked at end of scope
  }
  while (...) {
    Mu1.unlock();
    doSomeIO();
    Mu1.lock();  // OK
  }
  // Warning: Mu1 was not unlocked at end of function
}
```

Subtleties: parsing

- Thread safety annotations use gcc attributes.
- Extend lexical scope to attributes.
- Late parsing of attributes.

```
class MyObject {
public:
  int a                    GUARDED_BY(this->Mu);
  void foo(MyObject &O) EXCLUSIVE_LOCKS_REQUIRED(O.Mu);

private:
  Mutex Mu;
};
```

Subleties: substitution

- A lock is identified by an expression.
- Subsitute arguments for parameters in scope.

```
class MyObject {
public:
  Mutex Mu;
  int a                    GUARDED_BY(this->Mu);
  void foo(MyObject &O)  EXCLUSIVE_LOCKS_REQUIRED(O.Mu);
}

void bar(MyObject &O1, MyObject &O2) {
  O1.a = 1;      // substitute &O1 for this, get (&O1)->Mu
  O1.foo(O2);    // substitute O2 for O, get O2.Mu
}
```

Subtleties: expression equality

Google™

- Need to compare lock expressions for equality. (Substitution frequently creates minor variations.)

```
(&Obj)->Mu        == Obj.Mu?
Obj               == *&Obj?
Obj.getMutex()    == Obj.getMutex()?     (Yes)
ObjArray[i+1].Mu  == ObjArray[1+i].Mu?   (No)
```

- Varying variables: (We could really use SSA here.)

```
void foo(ListNode *N) {
  N->lock();
  N = N->next();
  N->unlock();   // Oops!
}
```

Limitations: control flow

- The following will not pass the analyzer:

```
void foo() {
  if (threadsafe) Mu.lock();
  ...
  if (threadsafe) Mu.unlock();
}
```

- Or worse: (yes, people do this.)

```
void foo() {
  for (int i = 0; i < 10; ++i) MutexArray[i].lock();
  ...
  for (int i = 0; i < 10; ++i) MutexArray[i].unlock();
}
```

Limitations: aliasing

- Aliasing causes problems:

```
class ScopedMutex {
  Mutex *Mu;
  ScopedMutex(Mutex *M) EXCLUSIVE_LOCK_FUNCTION(M)
    : Mu(M)
  { Mu->lock(); }
  ~ScopedMutex() UNLOCK_FUNCTION(Mu) { Mu->unlock(); }
};

void foo(Mutex *M) {
  ScopedMutex SMu(M);
  // Warning: lock M is not released at end of function
  // Warning: releasing lock Smu.Mu that was not acquired
}
```

Current GCC implementation

- GCC implementation has many problems.

- Evil parser hacks to resolve scoping issues.
- Analysis operates on GIMPLE.
  - Lowering to GIMPLE introduces artifacts.
  - Original C++ semantics are lost. E.g.
    - missing type information.
    - virtual method calls.
    - control flow graph oddities.
- Some optimizations run before the analysis.
- Lowering algorithm changes with each gcc release.

Clang implementation

Google™

- Advantage: much better organized code base.
  - E.g. altering the parser.
- Advantage: accurate representation of C++ AST
  - No lowering artifacts!
- Disadvantage: accurate representation of C++ AST
  - No SSA.
  - Difficult to identify loads and stores.
  - Very complicated AST
    - GCC: function call
    - Clang: function, constructor, new, destructor, etc.
  - Hard to compare expressions.

Huh?

Current challenges and future work.

Looking forward

- Move analysis to static analyzer.
- Use C++11 attributes.
- Integrate thread safety attributes with type system.
  - Type checking?
  - Templates?

- Some questions to think about.
- We welcome advice from the clang community!

Move to static analyzer

- Clang AST is much harder to analyze than a compiler intermediate language.
- Several capabilities are not provided by the AST.
  - No SSA form.
    - (Varying variables problem)
  - Hard to identify loads and stores.
    - a + 1;   // load from a.
    - b = &a; // no load.
    - foo(a);   // depends on declaration of foo.
- Static analyzer provides some of these capabilities.
  - Also better support for aliasing.
  - Also better support for more complex control flow.

Integrate with type system

Google™

- `PT_GUARDED_BY` is a hack, and easy to break.

```
int *a PT_GUARDED_BY(Mu);
int *b = a;   // OK. Only looked at pointer.
*b = 1;       // No warning!
```

- Attributes should be associated with types, not declarations, using C++11 attributes. E.g.

```
int [[guarded_by(Mu)]] *a;
```

- Casting away the guard is like casting away **`const`**.

- **Question**: How invasive would this be to clang type checking?

Templates! (Oh no.)

Google

- It would be nice to use attributes with templates, e.g.

```
std::vector<int [[guarded_by(Mu)]]>
```

- Attributes should have *erasure* property:
  - Removing them should not affect run-time behavior.
- Different instantiations should share implementation.

`std::vector<int [[guarded_by(Mu)]]>` (same impl. as)

`std::vector<int>`

- But... attributes should still be visible.

`int [[guarded_by(Mu)]]& operator[]()` versus

`int& operator[]()`

- **Question**: How do we implement this?

Conclusion

- Thread safety attributes solve a real problem.
- Lots of work still needs to be done.
  - Current implementation is pre-alpha right now.

- E-mail suggestions to:
  `delesley@google.com`