Concurrent Programming in the D Programming Language

by Walter Bright
Digital Mars

Introduction

- What is sequential consistency across threads?
- What are the problems with it?
- D features that mitigate those problems

The Basic Problem

Many-core programming offers exciting and compelling advantages.

but

Programming languages are designed with single threaded view.

Sequential Consistency is assumed.

Sequential Consistency

- For statements A, B, C
- A is completed before B is started
- B is completed before C is started

```
A:  a = 3;
B:  if (a == 3) b = 4;
C:  c = a + b;
```

Result: c is 7

Doesn't Hold For MultiThreads

Initially, x == 0 and y == 0

Thread 1

```
x = 1;
y = 2;
```

Thread 2

```
if (y == 2)
    assert(x == 1);    // boom!
```

Double Checked Locking Bug

```
typedef struct S { int m; } S;

S* getValue()
{   static S* s = NULL;
    static Mutex lock;
    if (!s)
    {    mutex_acquire(&lock);
         if (!s)
         {    S t = (S *)malloc(sizeof(S));
              t->m = 3;
              s = t;
         }
         mutex_release(&lock);
    }
    return s; /* s->m can be garbage! */
}
```

Problem: Implicit Sharing

- Data is visible at all times from any thread
- Data can be cached on the CPU chip's thread-local memory caches
- Indeterminate when those caches get flushed
- Indeterminate when those caches get refreshed

Just Use Memory Barriers!

- Complex, difficult to understand
- Hard to verify they are correctly used
- Hard to devise tests for them
- Incorrect usage can work fine on one machine, not on the next
- Hard to track down bugs in them

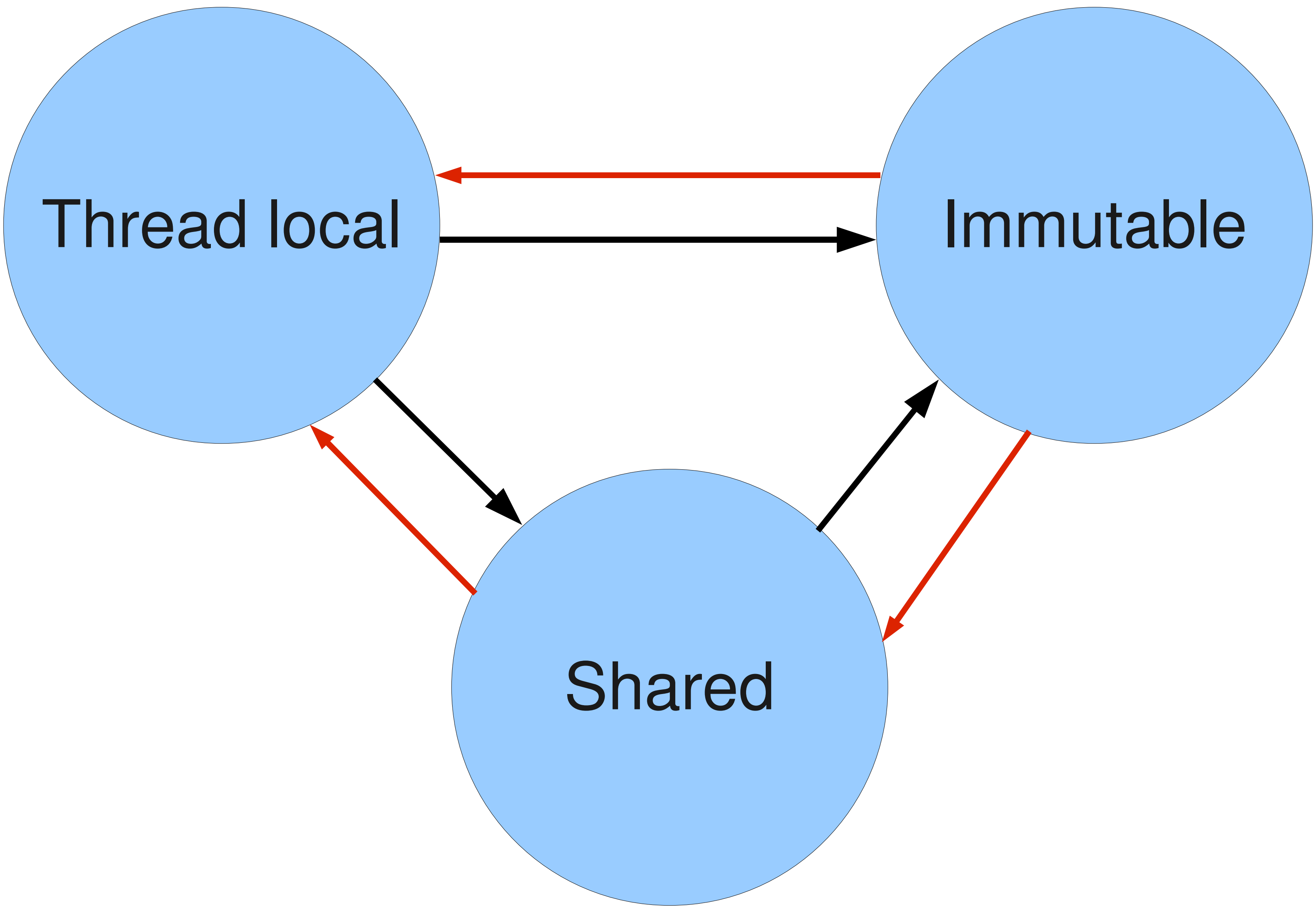
Even If You Understand It

- Unintentional sharing can happen with any static, global, or reference
- Very hard to find sharing in non-trivial code
- Impractical to verify that code does not have sharing bugs
- Code can have latent sharing bugs for years
- Relying on code reviews doesn't scale well

Perfect Problem for Language to Solve

- Redesigning the programmers will never work
- A language can offer guaranteed behavior
- Some problems can be defined out of existence
- D provides an opportunity for that

Types of Memory



Thread Local

- Default for globals, function locals, and allocated data
- No thread synchronization required
- May contain references to shared or immutable data

Immutable

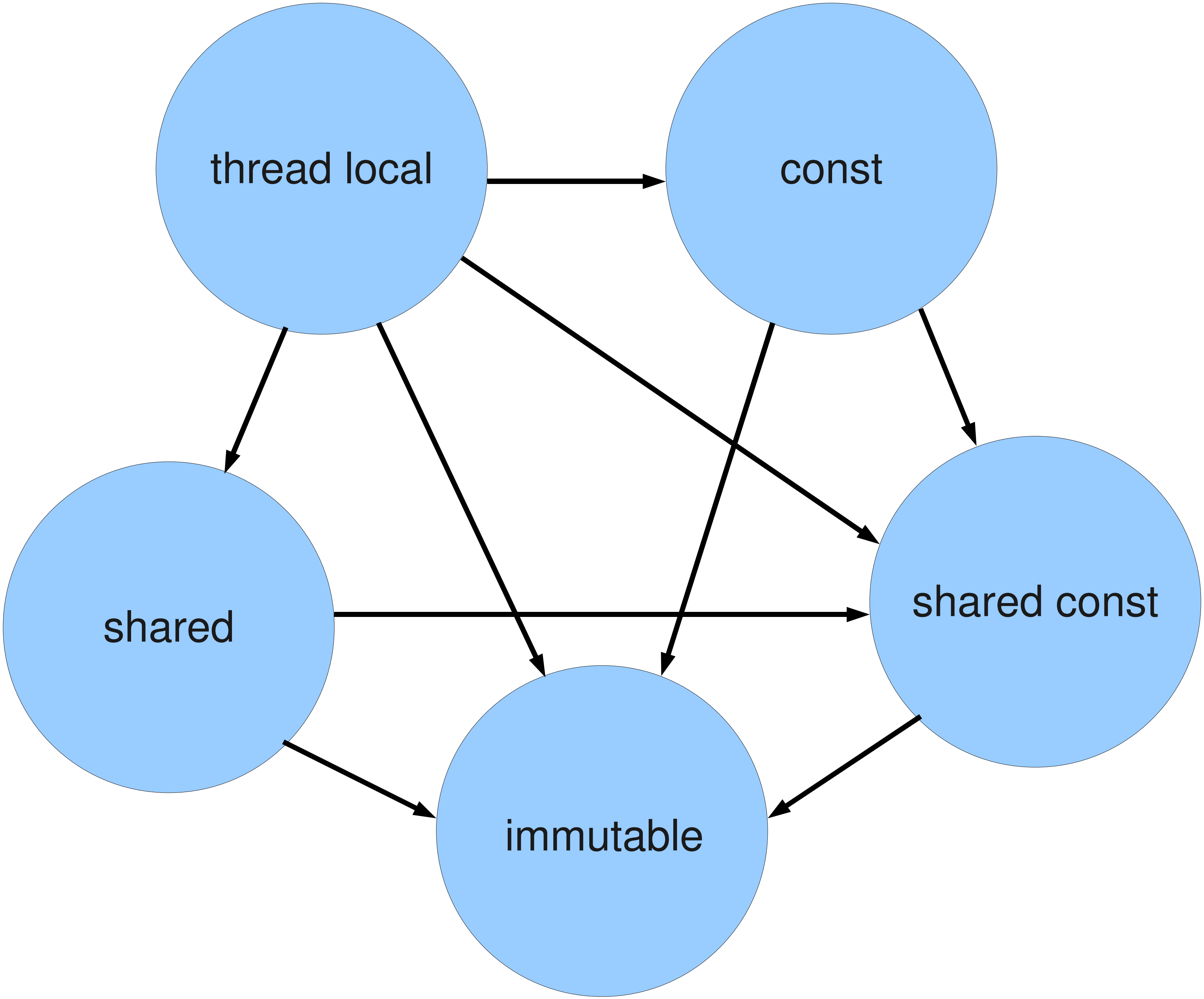
- Once set, it can never change
- Multiple threads can simultaneously access it
- No synchronization necessary
- Immutability is transitive
- Cannot refer to mutable data – either shared or thread local

Shared

- Mutable
- Accessible from multiple threads
- Synchronization required
- Shared-ness is transitive
- May contain references to immutable data
- Cannot refer to thread local data

D Type Qualifiers

- *<no qualifier>* : thread local
- shared : shared among threads
- immutable : can never change, implies shared
- const : read-only view of thread-local or immutable data
- shared const : read-only view of shared or immutable data


thread local
const
shared const
shared
immutable

It Looks Complicated But...

- Once shared, cannot escape being shared
- Once immutable, cannot escape being immutable
- Once const, cannot escape being const

It all follows from these three simple rules.

The Code

```
int x;                  // thread local
shared int y;     // multiple threads can read/write y
immutable int z = 7;     // z will always be 7
```

```
int* p;
const(int)* pc;              // cannot change the int p points to
shared(int)* ps;             // points to shared data
shared const(int)* psc;  // points to shared data
```

```
p = &x;       // ok
p = &y;       // error, p cannot point to shared
p = &z;       // error, p cannot point to immutable
*p = 4;       // ok
```

```
ps = &x;      // error, x is thread local
ps = &y;      // ok, y is shared
ps = &z;      // error, z is immutable
*ps = 4;      // ok
```

```
pc = &x;      // ok to point to mutable
pc = &y;      // error, y is shared
pc = &z;      // ok to point to immutable
*pc = 4;      // error, pointer to const
```

```
psc = &x;     // error, x is thread local
psc = &y;     // ok
psc = &z;     // ok
*psc = 4;     // error, pointer to const
```

Shared and Sequential Consistency

- Shared tells compiler not to reorder reads and writes of shared data
- Compiler (not the programmer) puts read and write barriers in for shared data access
- Double-checked locking bug is no longer possible since checked variable must be shared
- Code is portable because compiler takes care of memory barrier differences

One More Thing: Pure Functions

- Cannot read mutable static or global state
- Cannot write to static or global state
- Cannot have any side effects
- Parameters are values, const references or immutable references
- Only result is the return value

Therefore, pure functions **never** require synchronization

A Pure Function

```
int foo();
pure int bar();
int x;
immutable int y = 7;

pure int sum(int v, immutable(int)[] array)
{
    int i = x;          // error, cannot access mutable global state
    int s = y;          // ok, y is immutable
    foo();              // error, foo() is impure
    s += bar();         // ok
    foreach (e; array)
        s += e;         // ok, s is not visible outside of sum()
    return v + s;
}
```

There's a Catch

- Memory barriers are slow
- So accessing shared variables will be slow
- How to fix?

Minimize Shared Access

- Minimize use of shared data
- Maximize use of immutable data
- Maximize use of pure functions
- Cache shared data in thread local data

Which has another benefit...

Debugging Threading Problems

- Threading problems will be isolated to shared data, by definition
- Shared data will be explicitly marked as shared
- So the scope of the bug is already reduced to a smallish subset of the code, rather than the entire code base as in a language with implicit sharing

Manual Methods

- As D is a systems programming language, manual control of memory barriers is possible by explicitly casting away the shared attribute
- Of course, escaping from the type system also means accepting the responsibility of making the code correct

Double Checked Locking Fix

```
struct S { int m; };

shared(S)* getValue()
{
    static shared(S*) s = NULL;
    static shared(Mutex) lock;
    if (!s)
    {   mutex_acquire(&lock);
        scope(exit) mutex_release(&lock);
        if (!s)
        {   auto t = new(shared(S));
            t.m = 3;
            s = t;
        }
    }
    return s;
}
```

Conclusion

- Very hard to verify a program is free of sequential consistency bugs with implicit sharing and manual insertion of memory barriers
- Requiring explicit sharing, along with immutability and purity, and compiler insertion of memory barriers, greatly mitigates this

Acknowledgements

Thanks to Bartosz Milewski, Andrei Alexandrescu and Brad Roberts for reviewing this. Any remaining errors are solely my own.