Introduction to concurrency

Caveat programmer!

- The final two sections are an introduction to the concurrency facilities offered by C++11
- This is **not** a course on how to:
  - design, write, debug or test concurrent applications
  - get maximum performance from the machine
- Things that aren't covered include:
  - decomposing the problem domain
  - typical concurrency designs and patterns
  - context switching and grain size
  - the influence of caches
  - mixing I/O and CPU-bound tasks

Concurrency spectrum

program at the highest level you can on this spectrum!

- Message passing
  - multi-box: Hadoop, Google MapReduce
  - multi-process: MPI, Erlang
- Shared memory
  - parallel libraries: OpenMP, Threading Building Blocks
  - futures and thread pools: std::future, std::async, etc
  - atomics: std::atomic, lock-free, compare-and-swap
  - raw threads: std::thread, std::mutex, etc
  - memory fences: std::memory_order_acquire, etc

direct support in C++11 libraries for these approaches

increasing order of complexity and effort

Futures and std::async

any "callable object": function, lambda or functor

```
// C++11
std::future<int> f = std::async([]{ return long_calc(); });
// do other stuff
int i = f.get();     // return result or block until it's ready
                    // trying to read f again is an error
                    // use std::shared_future<int> instead
```

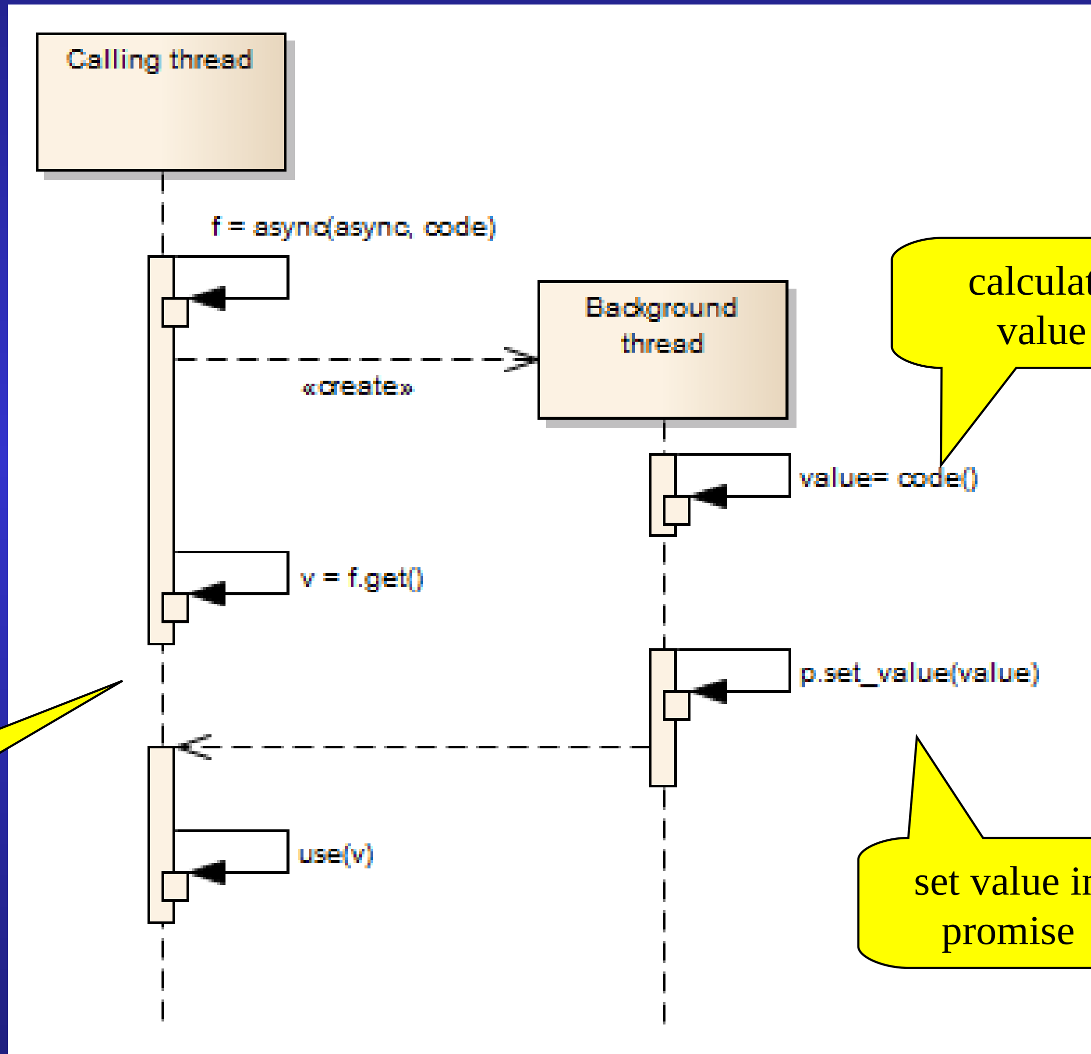
- Futures and promises are two ends of a one-shot message queue
  - promise is the sending end of the queue
  - future returns the value or rethrows exception when get() is called
- Destructor of future waits until task is finished
- Async is not a thread pool (likely to appear in TS)

std::async and background thread

```
void thread1()
{
  auto f
   = std::async(
          std::launch::async,
          code);

  auto v = f.get();
  use(v);
}
```

request background thread

block until promise is set



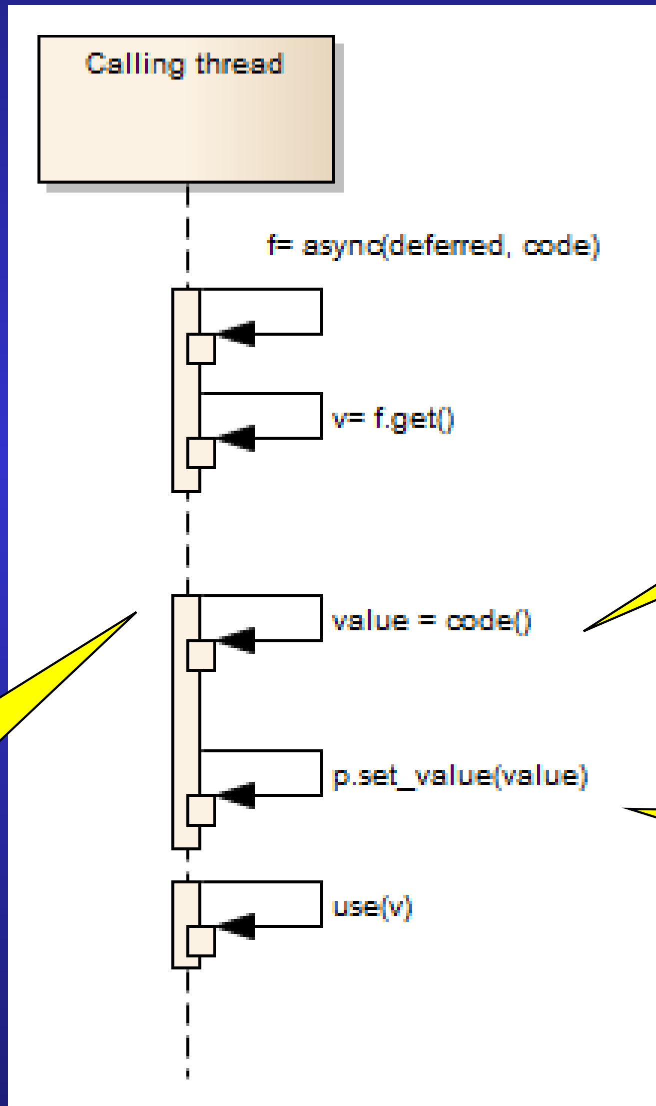
calculate value

set value in promise

std::async and deferred execution

```
void thread1()
{
  auto f
   = std::async(
         std::launch::deferred,
        code);

  auto v = f.get();
  use(v);
}
```

request same thread



calculate value

calculation happens on demand in same thread

set value in promise

Four parts to using futures

1. Some code that calculates a value (can be void)
2. A promise that is set by the code
3. A future that waits for the promise to be set
4. A thread that executes the code

- `std::packaged_task` wraps up code with a promise/future pair and is a "callable object"
  - it still needs a thread to run it
- `std::async` with `launch::async` creates a new thread and wraps up the code as a `packaged_task`
- `std::async` with `launch::deferred` wraps up the code as a `packaged_task` and runs it in the current thread

Futures and exceptions

- Normally the promise of a future/promise channel calls `promise.set_value(x)`
- It can also call `promise.set_exception(copy_exception(obj))` and then `future.get()` will rethrow obj
  - `make_exception_ptr` instead of `copy_exception`
- Deleting a promise without setting it first will store a future_error exception (thrown on future.get())
- `std::exception_ptr` is a reference-counted smart pointer for exceptions

Shared futures

- An ordinary future can be read only once by one thread
  - reading it a second time will throw a future_error with a condition of future_errc::no_state
- A std::shared_future can be shared across threads and can be read multiple times
  - a shared_future can be created from a future using a move c/tr (requires std::move(future))
- It is **not** safe to access one shared_future from multiple threads
- It is safe to access multiple shared_futures that access the same async result

Concurrency – atomics

- std::atomic<> can be applied to built-in types with the usual arithmetic operations (but not *, % and /)
- Also provides load, store, exchange, compare_exchange_strong/weak (CAS – lock-free)
- Must be initialised by c/tr
- Works with user-defined types with trivial copy op= (memcpy) + bitwise == (memcmp)
- std::atomic<T>::is_lock_free()
- Free-function i/f for C compatibility
- Can make use of memory ordering (next section)
  - default is std::memory_order_seq_cst

Concurrency – atomics examples

```
std::atomic<int> ai(2);
ai++;                    // usual ops, now ai == 3
std::cout << ai << std::endl;        // implicit load
std::cout << ai.load() << std::endl;  // explicit load
auto i = ai.load();

int expect = 4;          // CAS loop – used in lock-free algorithms
while (! ai.compare_exchange_weak(expect, 5))
  std::cout << “i = “ << i << “, expect = “ << expect << std::endl;

std::cout << “ai = “ << ai << std::endl;
// prints: i = 3, expect = 3  // expect gets existing value
//         ai = 5             // update succeeds on second loop
```

- Comp/exchg updates the atomic if the current value is the “expected” value (which is pass-by-ref)
- Otherwise the “expected” value is updated instead
- Strong and weak versions to allow for non-atomic hardware implementations

Further concurrency

Threads

```
// C++11
std::thread t1;                    // thread with no associated code
std::thread t2(func);              // run func() with no params
std::thread t3(f, 3, “foo”);       // run f(3, “foo”)
std::thread t4(&X::m, obj, 5);     // run obj.m(5) (decltype(obj) == X)
std::thread t4a(&X::m, &obj, 5);   // run obj->m(5) (obj ptr, not copy)
std::thread t5(g, std::ref(x));    // run g with x passed by reference
std::thread t6(std::move(t4));     // can’t copy a thread, only move
                                   // t4 is now “empty”, like t1

if (t2.joinable()) t2.join();      // threads must be joined or detached
t3.detach();                       // create background thread
```

- Threads must be joined or detached, or an error will occur in the destructor
- std::thread::hardware_concurrency() returns the number of available cores

The current thread

```
// C++11
// delays execution of this thread for a given time (or more)
std::chrono::milliseconds fiveMillis(5);
std::this_thread::sleep_for(fiveMillis);

// prints an implementation-defined thread handle
std::cout << std::this_thread::get_id() << std::endl;

// thread-local variable – useful only for non-local variables
thread_local int counter = 0;

void f()
{
  int counter = 0;         // automatically per thread
}

std::this_thread::yield();    // allow other threads to run
```

- thread_local is a portable version of pthread_getspecific() and TlsGetValue()

Mutexes

```
// C++11
// may block if locked more than once by the same thread
std::mutex m;
m.lock();
...
m.unlock();

if (m.try_lock()) {  // test if mutex can be acquired
  std::cout << “I got the lock!\n”;
  m.unlock();
}

// wait up to a certain amount of time to acquire a lock
auto locked = m.wait_for(std::chrono::seconds(10));

auto timeT = std::chrono::steady_clock::now()
                   + std::chrono::seconds(10);
auto locked2 = m.wait_until(timeT);

// recursive mutex can be locked more than once by the same thread
std::recursive_mutex rm;
```

Mutexes and exception safety

```
// C++11
std::mutex m;

void print(int i)
{
  std::lock_guard<std::mutex> lock(m);
  f(i);      // might throw
  std::cout << “i = “ << i << std::endl;
}
```

- std::lock_guard<> guarantees that m will be unlocked even if an exception is thrown

Condition variables

```
std::condition_variable start;
std::mutex mutex;
std::unique_lock<std::mutex>
          startMutex(mutex);
int dataItems = 0;

void workerCalc()
{
  startMutex.lock();
  start.wait(startMutex,
    []{ return dataItems > 0; });
  claimDataItem();
  startMutex.unlock();
  workOnDataItem();
}
```

blocks until condition is true

```
void setUpWorkers()
{
  startMutex.lock();
  dataItems = N;
  startMutex.unlock();
  start.notify_all();
}
```

signal all threads waiting for condition

- How to wait until some logical condition is true
  - an alternative to polling
- Need to acquire a lock then wait on condition
  - lock is unlocked during the wait and relocked on wakeup
  - internal loop testing predicate to avoid “spurious wakeups”

One-time initialisation

```
std::once_flag onceFlag;
std::ofstream logFile;
std::mutex logMutex;

void initLogFunc()
{
  system(“rm –rf /opt/myapp/output/”);
  logFile.open(“/opt/myapp/myapp.log”, ios::ate | ios::app);
  if (! logFile) /* handle error */
}

void log(const std::string & msg)
{
  std::call_once(onceFlag, initLogFunc);
  std::lock_guard<std::mutex> guard(logMutex);
  using namespace std::chrono;
  log << system_clock::to_time_t(system_clock::now())
      << “\t” << msg << std::endl;
}
```

- std::once_flag provides a means to guarantee that only one thread will run a piece of code only once

Threads and locks

- Up to now we have not used explicit locks so there has been no possibility of deadlock
- With explicit thread-and-lock programming we enter a whole new world of potential pain
  - use parallel libraries, futures and atomics if you can!
  - if you can't, use only a single lock if performance allows
- Handling multiple locks
  - incorrect locking order can lead to deadlock
  - how to move, adopt or defer locking

std::lock

```
std::mutex lock1, lock2;

std::lock(lock1, lock2); // guarantees all locks acquired or none

std::lock_guard<std::mutex> guard1(lock1, std::adopt_lock);
std::lock_guard<std::mutex> guard2(lock2, std::adopt_lock);
```

- Which order should we lock lock1 and lock2?
  - If different threads do it in different orders then deadlock
- std::lock provides a non-deadlocking ordering and releases any locks it doesn’t acquire (or if an exception is thrown)
  - handles multiple locks (variadic function template)
- Hand over already locked lock to std::lock_guard for exception-safe unlocking

std::unique_lock

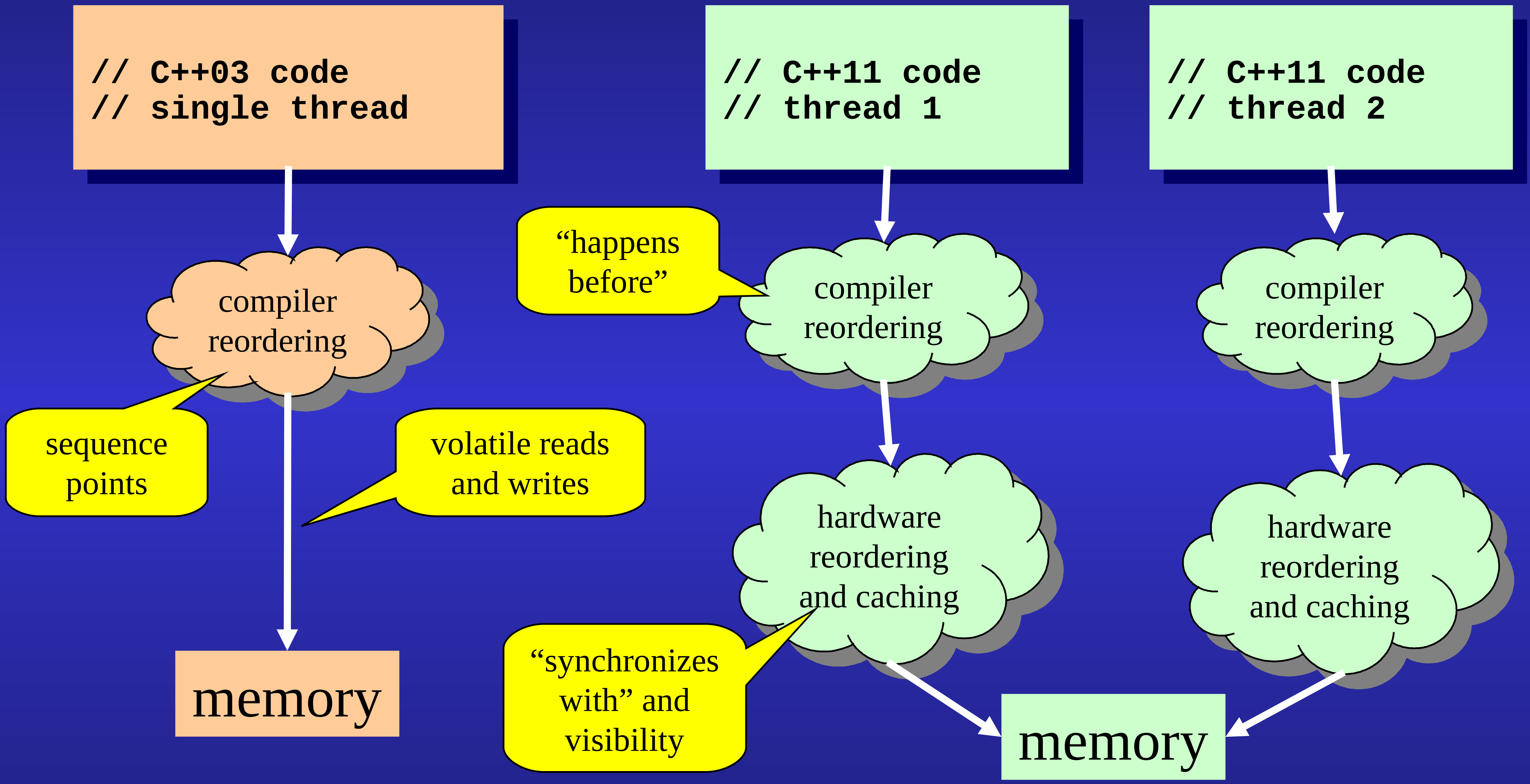
```
std::mutex lock1, lock2;

std::unique_lock<std::mutex> uniq1(lock1, std::defer_lock);
std::unique_lock<std::mutex> uniq2(lock2, std::defer_lock);

std::lock(lock1, lock2); // guarantees all locks acquired or none
```

- std::unique_lock<> allows for moving locks, adopting already acquired locks and referring to locks without acquiring them
  - more flexible version of std::lock_guard
  - small performance overhead
  - can also adopt locks

Memory model

- C++11, unlike C++03, embraces the notion of more than one thread of execution
- In order to support this, C++11 had to create a memory model to define what operations are visible in other threads and when
- Needed to control optimisations by the compiler and the hardware
  - visibility and reordering of operations
- Lack of standards has meant such code is not portable up till now

Need for a memory model



- Correctness now based on memory, not just code
- Need to control caching (register, L1, L2, etc)

Instruction interleaving

```
// thread 1
x = 1;   // 1
r1 = y;  // 2
```

interleave

```
// thread 2
y = 1;   // 3
r2 = x;  // 4
```

```
// 6 possible sequentially consistent
// execution orders

1234// x=1; r1=y; y=1; r2=x;
1324// x=1; y=1; r1=y; r2=x;
1342// x=1; y=1; r2=x; r1=y;
3412// y=1; r2=x; x=1; r1=y;
3142// y=1; x=1; r2=x; r1=y;
3124// y=1; x=1; r1=y; r2=x;
```

- "Sequential consistency" means operations in separate threads are interleaved and that all threads see the same interleaving
  - Sequence is preserved within and across threads
- This is the "natural" mental model for programmers to think of thread execution order and memory
  - It is also the C++11 default memory ordering

Instruction reordering

```
// thread 1
x = 1;   // 1
r1 = y;  // 2
```

reorder

```
// 4 factorial (== 24)
// possible execution orders

1234// x=1; r1=y; y=1; r2=x;
4321// r2=x; y=1; r1=y; x=1;
3124// y=1; x=1; r1=y; r2=x;
// etc...
```

could be executed in reverse order

```
// thread 2
y = 1;   // 3
r2 = x;  // 4
```

- In order to gain performance both the compiler and the hardware may reorder instructions
  - compiler may move loads earlier (to allow for cache misses)
  - hardware may not write back to memory immediately (store buffers)
- x, y, r1 and r2 are all independent so code can be reordered
- Even worse, changes in one thread may not be visible in another thread so results are not defined – ***data race***

Hardware memory reordering

	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

http://en.wikipedia.org/wiki/Memory_ordering

- Hardware can reorder memory operations in different ways
  - Can also depend on operating system
    - Solaris on SPARC uses Total Store Order (TSO)
    - Linux on SPARC uses Relaxed Memory Order (RMO)

Synchronisation with seq. consistency

```
// thread 1
x = 42;
x_init = true;
```

```
// thread 2
while (! x_init) {}
y = x;
```

- This code is correct when sequentially consistent
  - thread 2 doesn’t access x until it has been set by thread 1
- But in the presence of reordering it can fail
- The problem is that we haven’t specified that cross-thread order or visibility is important
- We need to use synchronisation variables – atomics
- Making everything atomic is slow – 30-60 cycles
  - cache synchronisation is slow and has limited bandwidth

Synchronisation with atomics

```
// thread 1
std::atomic<bool> x_init;
int x;

x = 42;
x_init.store(true);

// or x_init = true;
```

```
// thread 2
extern std::atomic<bool> x_init;
extern int x;

while (! x_init.load());
y = x;

// or while (! x_init);
```

- This now works without relying on having sequential consistency everywhere (just atomics)
  - atomics prevent the compiler moving code across accesses
  - atomics also cause memory updates to be visible
- Load and store uses sequential consistency
  - uses default parameter of std::memory_order_seq_cst

Low-level synchronisation detail

```
// thread 1
std::atomic<bool> x_init;

x = 42;
// blue fence
x_init.store(true);
// red fence
```

prevents x and x_init being reordered

makes store to x_init visible

```
// thread 2
extern std::atomic<bool> x_init;

while (/*rfence*/! x_init.load());
// blue fence
y = x;
```

prevents reordering

rfence here in loop forces load of latest value of x_init

- Blue fences prevent the compiler reordering code
  - they don’t generate any run-time code
- Red fences force memory to make changes visible
  - they do generate code: fence, lock prefix, CAS opcodes
  - depends heavily on underlying hardware (c.f. reordering)
  - only need one of the two red fences, usually on store
- One reason that threads can’t just be a library

Generated assembler code

sequential consistent by default

```
// thread 1
x = 42;
// blue fence
x_init.store(true);
// red fence
```

```
// thread 2

while (/*rfence*/ ! x_init.load());
// blue fence
y = x;
```

```
// with atomic bool x_init
 mov     DWORD PTR x, 42
 mov     BYTE PTR x_init, 1
 mfence

// with bool x_init
 mov     DWORD PTR x,
 mov     BYTE PTR x_init, 1
```

fence needed on store for X86

```
// with atomic bool x_init
L25:
 movzx   eax, BYTE PTR x_init
 test    al, al
 je      .L25
 mov     eax, DWORD PTR x
 mov     DWORD PTR y, eax

// with bool x_init
 cmp     BYTE PTR x_init, 0
 jne     .L3
.L5:
 jmp     .L5
.L3:
 mov     eax, DWORD PTR x
 mov     DWORD PTR y, eax
```

no fence needed on load for X86

g++ 4.7 output

infinite loop because visibility not specified

Using memory order flags

acquire means no reads in this thread reordered before here

```
// thread 1
x = 42;
// blue fence
x_init.store(true,
std::memory_order_release);
```

```
// thread 2
while (!
x_init.load(std::memory_order_acquire));
// blue fence
y = x;
```

```
// with bool x_init seq_cst
 mov     DWORD PTR x, 42
 mov     BYTE PTR x_init, 1
 mfence

// with bool x_init release
 mov     DWORD PTR x, 42
 mov     BYTE PTR x_init, 1
```

needed on seq_cst store

no fence for release store

```
// with bool atomic x_init
L25:
 movzx   eax, BYTE PTR x_init
 test    al, al
 je      .L25
 mov     eax, DWORD PTR x
 mov     DWORD PTR y, eax

// same code for x_init acquire
```

no fence needed on load for X86

- Release provides only the blue fence (no writes in this thread reordered after the store)
- Controls compiler reordering but not hardware

Memory model advice

- This is a complex and subtle area and you should avoid using it unless you can prove that you can't get adequate performance without it
  - yes, really, I mean it....
- Even experts get confused by this stuff!
  - did I mention you should avoid it....
- If you do use it, use acquire on load and release on store
  - Anything else will be a source of subtle bugs