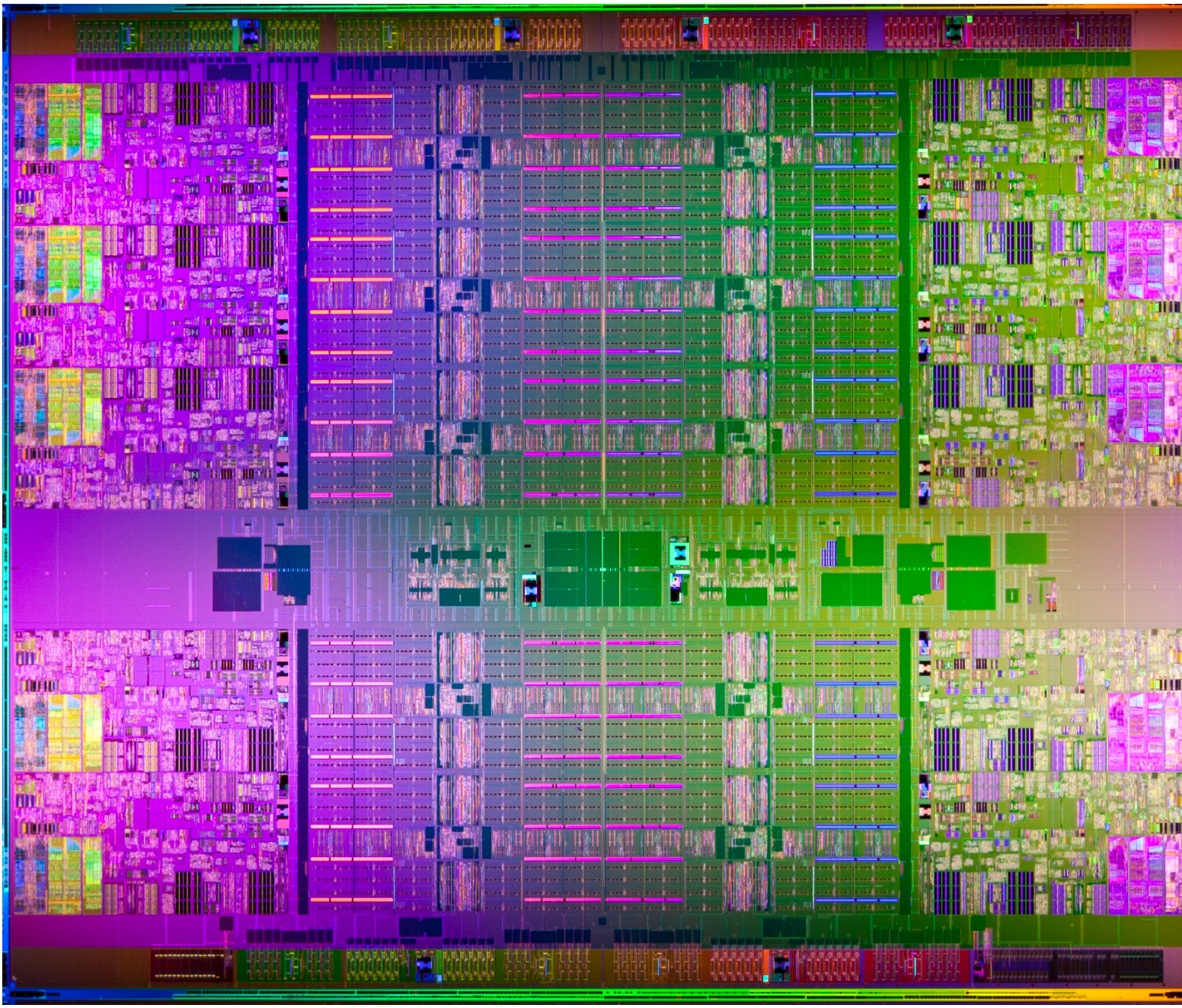


60007

Theory and Practice of Concurrent Programming

Imperial College London

Contents

Chapter 1

Introduction

1.1 Course Structure & Logistics

1.1.1 Structure

Dr Azalea Raad

Prof Alastair Donaldson

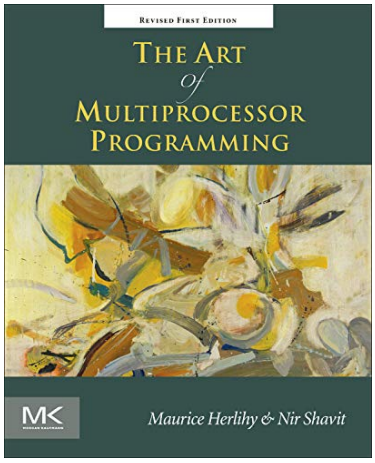
Theory For weeks 2 → 5:

- Intro to synchronisation paradigms (mutual exclusion, readers-writers, producer-consumer)
- Low-level concurrent semantics (sequential consistency, Intel-x86)
- High-level concurrent semantics (concurrent objects, linearisability)
- Transactional memory (serialisability)

Practical For weeks 5 → 8:

- Threads and locks in C++
- Implementing locks
- Concurrency in Haskell
- Race-free concurrency in Rust
- Dynamic data-race detection

1.1.2 Extra Materials



The Art of Multiprocessor Programming
About 65% of the theory course.

1.2 Preface for Concurrency

1.2.1 Moore's Law

Moore's Law — **Definition 1.2.1**

An empirical (supported by observation) law that states the density of transistors in an integrated circuit will double approximately every two years.

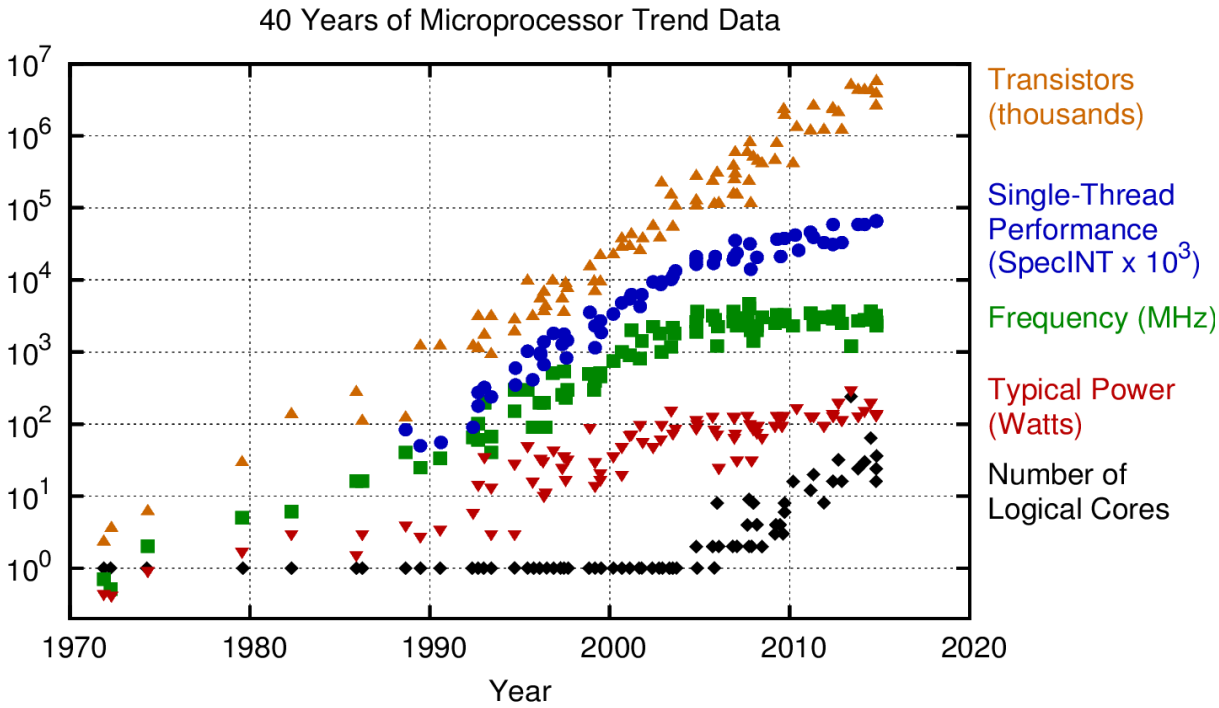
- The observation is named after Gordon Moore (co-founder and later CEO of Intel).
- This law no longer holds, and sequential performance improvements have declined.

Dennard/MOSFET Scaling — **Definition 1.2.2**

$$\text{Power} \propto \text{Transistor Size}$$

A scaling law stating that as transistor density increases, the power requirements stay constant.

- Increasing transistor density results in power staying constant (less power per transistor) and lower circuit delay.
- This allows for higher switching frequency ⇒ higher clocks frequencies ⇒ better sequential performance).



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

The performance improvements typically expected yearly (moore's law and Dennard scaling) no longer apply.

- Sequential (Single-Thread) performance improvements have declined.
- Parallelism is being exploited to improve performance (uniprocessors are virtually extinct).
- Shared-memory multiprocessor systems have lost out to multicore processors.

Amdahl's Law — **Definition 1.2.3**

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}} \text{ where } p = \text{parallel portion and } s = \text{threads}$$

Amdahl's law describes the speedup of a program, associated with the number of threads.

- Can be applied to other resources.
- Versions of the equation exist for different proportions using different numbers of threads.
- As the number of threads increases the sequential part of the program becomes a bottleneck.

1.2.2 Concurrency Difficulties

Writing correct, concurrent code is difficult.

race Condition — **Definition 1.2.4**

A potential for a situation where the result of a program depends on the non-deterministic timing or interleaving of threads.

- Where multiple threads access data (non-atomically) and at least one writes.
- Where a lack of enforced ordering on some events causes differing results (e.g output to user)

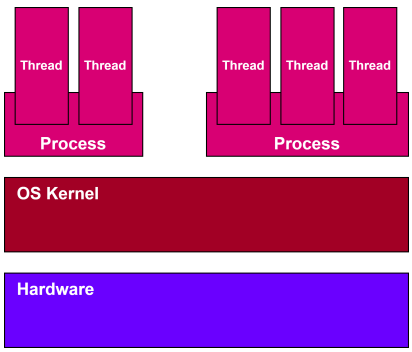
Race conditions can be intentional, where the result of the program is intended to be based off some non-deterministic input.

- Which thread gets to write first?
- Which process is allowed to write to a file?

- A process can have multiple threads executing in parallel.
- Cannot determine at compile time the relative speed of execution of threads (many delays are unpredictable; cache misses, page faults, interrupts).
- Cannot predict how long threads will be blocked (e.g I/O) or when threads will be scheduled (or use up their time quantum).

Hence we must use synchronisation mechanisms to regulate accesses to shared data that can result in a race condition.

1.2.3 OS Concepts



- Operating system provides process and thread abstractions.
- A process contains one or more threads (streams of instructions being executed).
- A process has its own address space, all threads in the process share this address space.
- The OS kernel contains a scheduler which schedules processes & their threads.

Chapter 2

Concurrency In C++

2.1 Threads

To interact with threads the `thread` header must be included.

- It provides a standard, implementation independent, interface for handling threads.
- Provides the `std::thread` class

```
#include <thread>

namespace std {
  class thread {
  public:
    // types
    class id;
    using native_handle_type = /* implementation-defined */;

    // construct/copy/destroy
    thread() noexcept;

    // Constructor takes a function to start from, and its arguments (all type checked)
    template<class F, class... Args> explicit thread(F&& f, Args&&... args);

    // Destructor (terminates current thread if the thread has not been joined)
    ~thread();

    // Attempting to copy a thread is not allowed. Hence delete ensures no compile.
    thread(const thread&) = delete;

    // Can create thread from a thread r-value (copy)
    thread(thread&&) noexcept;

    // Attempting to copy a thread (via an immutable reference).
    // This is not allowed, so if this operator is used it will not compile.
    thread& operator=(const thread&) = delete;

    // Assign a thread from an (r value - e.g expression, literal) reference
    thread& operator=(thread&&) noexcept;

    // members
    void swap(thread&) noexcept;
    bool joinable() const noexcept;

    // Wait for this thread to terminate.
    void join();
```

```
    // Allow the thread to continue executing after the thread handler (this)
    // is destroyed
    void detach();

    // Get the unique id of the thread
    id get_id() const noexcept;

    native_handle_type native_handle();

    // static members
    static unsigned int hardware_concurrency() noexcept;
  };
}
```

Lambda — **Definition 2.1.1**

A lambda is a small function that can be defined in an expression, capture values in its scope (and above), and be passed as a value.

```
// [captures] (arguments) {body}

// a basic lambda with no captures
auto my_lambda = [] (int a, int b) -> int {return a + b;}

// using the lambda
int c = my_lambda(1, 2);

auto another_lambda = [c&] (int d) {return c + d;}
```

We can construct using `std::thread`'s constructors.

```
// idiomatic constructor
std::thread my_thread(StartFunction, arg1, arg2, ...)

// call constructor and assign
std::thread my_thread = std::thread(StartFunction, arg1, arg2, ...)
auto my_thread = std::thread(StartFunction, arg1, arg2, ...)

// pass lambda as function
std::thread my_thread(StartFunction, arg1, arg2, ...)
```

When passing arguments to the thread, if these are by reference, a `std::ref` or `std::cref` must be used.

Reference this! — **Example Question 2.1.1**

Given some function `static void some_func(const int& a)` create a thread to take a reference to the number 42.

```
int a = 42;
std::thread my_thread(some_func, std::cref(42));
my_thread.join();
```

2.1.1 Vectors of Threads

When adding an object to a container (e.g a vector) we want to avoid allocating the object, and then moving it into the container.

- Some objects may not be movable/copyable.
- The object should be allocated within the container.

For this we can use emplacement.

```
template< class... Args >
void emplace_back( Args&&... args );
```

Emplace — **Example Question 2.1.2**

Given some function `static void some_func()` create 10 threads and append to the vector using `std::vector::push_back` and another 10 with `std::vector::emplace`.

```
std::vector<std::thread> threads;

for (int i; i < 10; i++) {
  threads.push_back(std::thread(some_func));
}

for (int i; i < 10; i++) {
  threads.emplace_back(some_func);
}

for (auto& t : threads) {
  t.join();
}
```

2.1.2 This Thread

The threads header also provides functionality for interacting with the current thread.

```
#include <compare>

namespace std {
  class thread;

  void swap(thread& x, thread& y) noexcept;

  // class jthread
  class jthread;

  // methods for interacting with the current thread
  namespace this_thread {
    thread::id get_id() noexcept;

    // indicates another thread should be scheduled (e.g long wait expected)
    void yield() noexcept;

    // Sleeping, generic for
    template<class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

    //
    template<class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

Clock watching — **Example Question 2.1.3**

Create program that prints the thread id, and sleeps.

```
#include <thread>
#include <iostream>
#include <chrono>
```

```
int main() {
  using namespace std::chrono_literals; // to use the ms syntax

  std::cout << std::this_thread::get_id() << " will sleep now!" << std::endl;
  std::this_thread::sleep_for(200ms);

  std::cout << std::this_thread::get_id() << " has awoken!" << std::endl;
}
```

2.2 Locks

RAII **Definition 2.2.1**

Resource Acquisition Is Initialization (also called Scope-Bound Resource Management and Constructor Acquires, Destructor Release) is where a resource's allocation and release is bound to the lifetime of an object.

- a resource may be the memory allocated to an object, or resources such as os provided file handlers.
- When the object goes out of scope (e.g the variable owning the object is destroyed) the resource is released.
- In C++, when a variable goes out of scope, the destructor of the contained object is called, so the destructor must release the resources.
- This concept is heavily embedded in Rust. Lifetimes are a major part of the type system, and ownership rules are enforced by the compiler.
- RAII is used for smart pointers such as `Rc` in rust or `std::shared_ptr`.

```
static void my_scope() {
    MyClass my_object;  // initialised, default constructor called

    // ... do some stuff ...

    return; // destructor my_object.~MyClass() called.
}
```

The `mutex` header contains locks for synchronisation.

```
namespace std {
  class mutex;                  // A regular lock
  class recursive_mutex;        // reentrant/recursive lock
  class timed_mutex;            // A mutex with timeout
  class recursive_timed_mutex;  // A recursive mutex with timeout

  /* used to set the locking strategy when using lock guards
   * e.g create guard (that releases lock on destruction) assuming
   * lock is held.
   */
  struct defer_lock_t { explicit defer_lock_t() = default; };   // do not acquire ownership
  struct try_to_lock_t { explicit try_to_lock_t() = default; }; // try to acquire ownership (no block)
  struct adopt_lock_t { explicit adopt_lock_t() = default; };   // assume calling thread has ownership

  inline constexpr defer_lock_t  defer_lock { };
  inline constexpr try_to_lock_t try_to_lock { };
  inline constexpr adopt_lock_t  adopt_lock { };

  // A RAII like mechanism that releases the lock it guards when destroyed.
  template<class Mutex> class lock_guard;

  // A RAII style lock guard, when taking ownership of multiple locks it attempts
  // deadlock avoidance.
```

```
  template<class... MutexTypes> class scoped_lock;

  // A movable lock guard.
  template<class Mutex> class unique_lock;

  template<class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

  // attempts to acquire locks from references provided, returns index (in args) of lock
  // that could not be acquired.
  template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);

  // Acquire one or more locks (blocking) and use deadlock avoidance.
  template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

  struct once_flag;

  template<class Callable, class... Args>
    void call_once(once_flag& flag, Callable&& func, Args&&... args);
}
```

2.2.1 Using Mutexes

```
namespace std {
  class mutex {
    public:

      // Constructor initialises mutex as unlocked. It is a constexpr
      // as can determine all fields at compile time.
      constexpr mutex() noexcept;

      // Destructor, undefined behaviour if the mutex is held by a thread.
      ~mutex();

      // Cannot create mutex from another, or use assignment to move a mutex.
      mutex(const mutex&) = delete;
      mutex& operator=(const mutex&) = delete;

      void lock();
      bool try_lock();
      void unlock();

      using native_handle_type = /* implementation-defined */;
      native_handle_type native_handle();
  };
}
```

Locked Out — **Example Question 2.2.1**

Create a mutex to protect a counter, and use 100 threads to increment the counter 10 times each. Add a wait of 1ms between each increment and only lock for each increment.

```
#include <thread>
#include <iostream>
#include <mutex>
#include <vector>
#include <chrono>

int cnt;
std::mutex cnt_lock;
```

```
static void increment_cnt() {
  for (int i = 0; i < 10; i++) {
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    cnt_lock.lock();
    cnt++;
    cnt_lock.unlock();
  }
}

int main() {
  std::vector<std::thread> threads;

  for (int i = 0; i < 100; i++) {
    threads.emplace_back(increment_cnt);
  }

  for (auto& t : threads) {
    t.join();
  }

  std::cout << "The counter is: " << cnt << std::endl;
}
```

2.2.2 Lock Guards

We can use `scoped_lock`, `unique_lock` or `lock_guard` to link the time the lock is held to the lifetime of the lock guard object.

- Each has slight differences, separate implementations are provided rather than using complex template magic.
- Deadlock avoidance is used to ensure all threads acquire and release locks in the same order.

Scoped Lock

```
namespace std {
  template<class... MutexTypes>
  class scoped_lock {
  public:
    using mutex_type = Mutex;   // If MutexTypes... consists of the single type Mutex

    explicit scoped_lock(MutexTypes&... m);
    explicit scoped_lock(adopt_lock_t, MutexTypes&... m);
    ~scoped_lock();

    scoped_lock(const scoped_lock&) = delete;
    scoped_lock& operator=(const scoped_lock&) = delete;

  private:
    tuple<MutexTypes&...> pm;   // exposition only
  };
}
```

- Constructed from one or more mutexes.
- Locks all mutexes on construction.
- Unlocks all mutexes on destruction.

> Does not support deferred locking, early unlocking or ownership transfer (with `std::move`).

```
#include <mutex>
#include <iostream>

std::mutex m1, m2;

static void some_fun() {
    std::scoped_lock lock(m1, m2); // acquire lock on m1 and m2 (or any number of locks)
    std::cout << "Critical region here" << std::endl;
}
```

Unique Lock

```
namespace std {
  template<class Mutex>
  class unique_lock {
  public:
    using mutex_type = Mutex;

    // construct/copy/destroy
    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);

    // locking strategies
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t);
    unique_lock(mutex_type& m, adopt_lock_t);

    //
    template<class Clock, class Duration>
      unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
    template<class Rep, class Period>
      unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
    ~unique_lock();

    unique_lock(const unique_lock&) = delete;
    unique_lock& operator=(const unique_lock&) = delete;

    unique_lock(unique_lock&& u) noexcept;
    unique_lock& operator=(unique_lock&& u);

    // locking
    void lock();
    bool try_lock();

    template<class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template<class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

    void unlock();

    // modifiers
    void swap(unique_lock& u) noexcept;
    mutex_type* release() noexcept;

    // observers
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;
```

```
  private:
    mutex_type* pm;             // exposition only
    bool owns;                  // exposition only
  };

  template<class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
}
```

- Constructed from one mutex.
- Locks mutex on construction by default, but can have locking deferred.
- Allows for unlocking and relocking.
- If mutex held on destruction, unlocks.
- Can transfer lock ownership with `std::move`.

> Only works for a single mutex.

Scoped out — **Example Question 2.2.2**

Create a basic implementation of defer that could be used for a scoped lock.

```
#include <mutex>
#include <iostream>
#include <functional>

class Defer {
  private:
    std::function<void(void)> function_;

  public:
    Defer(std::function<void(void)> fun) : function_(fun) {}
    ~Defer() {
      function_();
    }
};


int main() {
  std::mutex m;

  Defer lock([&m] () {
    m.unlock();
    std::cout << "Unlocking" << std::endl;}
  );

  std::cout << "lets do some racey stuff here" << std::endl;
}
```

We could also implement this pattern in rust. As mutexes already work this way in rust, we create a dummy mutex struct to use.

```
#![feature(fn_traits)]
struct Defer<F: FnMut()>(F);

impl<F: FnMut()> Drop for Defer<F> {
    fn drop(&mut self) {
        self.0()
    }
}
```

```rust
fn main() {
    let mut m = Mutex();
    m.lock();
    let _d = Defer(|| m.unlock());
    println!("lets do some racey stuff here")
}
```

2.3 Race Conditions in C++

Data Race — **Definition 2.3.1**

A data race is a race condition on the value of some data shared between threads.

- Distinct threads access a memory location.
- At least one thread modifies the location.
- At least of of the accesses is non-atomic (allows for operations of other threads to be interleaved)
- Accesses are not ordered by synchronisation (e.g for mutual exclusion)

Data races are value-obilvious, meaning a data race is present even if value of some shared data is not affected.

- e.g Two threads write the same value to the same place.
- e.g one thread stores the same value, another thread reads.

Always a bug, considered an unintentional race condition.

Undefined Behaviour — **Definition 2.3.2**

> *"Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended."* - C FAQ

- Programmer must avoid relying on undefined behaviour.
- Different compilers implementing the specification can do anything with undefined behaviour.
- Allows compilers to do more optimisation (fewer guarantees to satisfy).
- Among the most cited language design issues with C++.

The behaviour of a C++ program on some input is undefined if a data race can occur. This means specification is saying a program with a data race can do *anything*, there are no guarantees (even that the result depends on the outcome of the race).

Compilers typically optimise on the assumption there is no undefined behaviour.

- If there is no undefined behaviour, then assuming none is fine.
- If there is undefined behaviour, the language specification says *anything goes* and hence any output is valid.

```cpp
#include <thread>

static void set_x(int& x) {
  x = 1;
}

static void wait_x(int& x) {
  while (x == 0);
}

int main() {
  int x = 0;
```

```
  std::thread t1(set_x, std::ref(x));
  std::thread t2(wait_x, std::ref(x));
  t1.join();
  t2.join();
}
```

Here the loop in `wait_x` can be optimised.

```
static void wait_x(int& x) {
  int temp_register = r;
  while (temp == 0);
}
```

1. x is a non-atomic variable
2. if another thread modified x, then there would be a data race.
3. A data race is undefined behaviour

Place copy of x into a register and compare using this.

1. An infinite loop with no side effects is undefined behaviour.
2. Can assume it is *dead code* and remove.

```
static void wait_x(int& x) {
  // terminate thread
}
```

Dead code can be removed.

2.3.1 Thread Sanitiser

A sanitiser to automatically detect data races.

- Available in clang++ and g++ compilers.
- Enabled with `-fsanitize=thread` and add debug symbols with `-g`.

2.4 Condition Variables

Condition Variable — **Definition 2.4.1**

A condition that can be waited on, and notified.

- Threads can wait on the condition to be signalled.
- Can be used to construct a monitor.
- In languages without a monitor construct, an explicit lock is required.

```
#include <semaphore>
#include <mutex>
#include <deque>
#include <cassert>

class condition_variable {
  public:
    // delete copy constructors
    condition_variable(const condition_variable&) = delete;
    condition_variable& operator=(const condition_variable&) = delete;

    void notify_all(std::unique_lock<std::mutex> monitor_lock&) {
      assert(monitor_lock.owns_lock())
      for (auto& sema : wait_semas_) {
        sema.release();
      }
      wait_semas_.clear();
    }

    void notify_one(std::unique_lock<std::mutex> monitor_lock&) {
      assert(monitor_lock.owns_lock())
      wait_semas_.pop_front().release();
    }

    void wait(std::unique_lock<std::mutex> monitor_lock&) {
      assert(monitor_lock.owns_lock())
      std::counting_semaphore wait_sema(0);
      wait_semas_.push_back(std::ref(wait_sema));
      monitor_lock.release();
      wait_sema.acquire();
      monitor_lock.aquire();
    }

  private:
    std::deque<std::ref<std::counting_semaphore>> wait_semas_;
};

}
```

```
#include <condition_variable>

namespace std {
  class condition_variable;
  class condition_variable_any;

  void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

  enum class cv_status { no_timeout, timeout };
}
```

2.4.1 Using Condition Variables

The `std::condition_variable` class is as follows:

```
namespace std {
  class condition_variable {
  public:
    condition_variable();
    ~condition_variable();

    // delete copy constructors
    condition_variable(const condition_variable&) = delete;
    condition_variable& operator=(const condition_variable&) = delete;

    // signal a condition variable
    void notify_one() noexcept;
    void notify_all() noexcept;

    // The current thread waits on the condition variable (until signalled),
    // using the (acquired) lock to synchronise
    void wait(unique_lock<mutex>& lock);

    // Wait on a predict using the provided mutex using the (acquired) lock
    // to synchronise
    template<class Pred>
      void wait(unique_lock<mutex>& lock, Pred pred);

    // wait until time
    template<class Clock, class Duration>
      cv_status wait_until(unique_lock<mutex>& lock,
                           const chrono::time_point<Clock, Duration>& abs_time);
    template<class Clock, class Duration, class Pred>
      bool wait_until(unique_lock<mutex>& lock,
                      const chrono::time_point<Clock, Duration>& abs_time, Pred pred);

    // wait for time
    template<class Rep, class Period>
      cv_status wait_for(unique_lock<mutex>& lock,
                         const chrono::duration<Rep, Period>& rel_time);
    template<class Rep, class Period, class Pred>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time, Pred pred);

    using native_handle_type = /* implementation-defined */;
    native_handle_type native_handle();
  };
}
```

Wait on Signal

1. Associated a `std::mutex` with the condition variable.
2. Acquire a lock on the mutex with a unique lock.
3. Call wait with the lock.

The thread will block until the condition variable is signalled.

```
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cond;

static void some_func() {
    std::unique_lock<std::mutex> lock(m);
    cond.wait(lock);
}
```

Wait on predicate

1. Associated a `std::mutex` with the condition variable.
2. Acquire a lock on the mutex with a unique lock.
3. Call wait with a predicate.

Immediately returns if the predicate is true. Otherwise blocks, when the condition variable is signalled, the predicate will be checked, if true the thread returns, otherwise the thread is blocked again.

```
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cond;

static void some_func() {
    std::unique_lock<std::mutex> lock(m);
    cond.wait(lock, [...]() -> bool {...});
}
```

2.5 Atomics

Defined in the `atomic` header. Allow for the construction of atomic variables for integral and arbitrary types (that are TriviallyCopyable, CopyConstructible and CopyAssignable).

- For integral types atomic operations offer lower overhead alternatives for protecting small amounts of data than using a mutex.
- Atomic declarations prevent data races. This can be useful in declaring an intentional race condition (to prevent undefined behaviour)

2.5.1 Atomic Template Class

```
namespace std {
  template<class T> struct atomic {
    using value_type = T;

    static constexpr bool is_always_lock_free = /* implementation-defined */;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    // operations on atomic types
    constexpr atomic() noexcept(is_nothrow_default_constructible_v<T>);
    constexpr atomic(T) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    // load the value (make a non-atomic copy of the current value)
    T load(memory_order = memory_order::seq_cst) const volatile noexcept;
    T load(memory_order = memory_order::seq_cst) const noexcept;

    // implicit conversion from an instance of this class (std::atomic<T>) to T (used by static_cast)
    operator T() const volatile noexcept;
    operator T() const noexcept;

    // Store (overwrite) the the atomic
    void store(T, memory_order = memory_order::seq_cst) volatile noexcept;
```

```
    void store(T, memory_order = memory_order::seq_cst) noexcept;

    // Assignment
    T operator=(T) volatile noexcept;
    T operator=(T) noexcept;

    // Store desired and return the old value atomically
    T exchange(T desired, memory_order = memory_order::seq_cst) volatile noexcept;
    T exchange(T desired, memory_order = memory_order::seq_cst) noexcept;

    // if the old value is expected, then replace with desired and return true.
    // if the old value is not expected, then return false
    bool compare_exchange_strong(T& expected, T desired, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T& expected, T desired, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T& expected, T desired,
                                 memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_strong(T& expected, T desired, memory_order = memory_order::seq_cst) noexcept;

    // Same as compare_exchange_strong on x86, but different on ARM. Can fail spuriously.
    bool compare_exchange_weak(T& expected, T desired, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T& expected, T desired, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T& expected, T desired,
                               memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_weak(T& expected, T desired, memory_order = memory_order::seq_cst) noexcept;

    // check value of this against old, if equal => blocks until notify called.
    void wait(T old, memory_order = memory_order::seq_cst) const volatile noexcept;
    void wait(T old, memory_order = memory_order::seq_cst) const noexcept;

    void notify_one() volatile noexcept;
    void notify_one() noexcept;
    void notify_all() volatile noexcept;
    void notify_all() noexcept;
  };
}
```

Hence we can use it to make access to complex objects atomic. Large types (i.e not integral types) use locks for this.

```
#include <atomic>

// an atomically accessed struct
typedef struct {
  int a;
  bool c;
} MyStruct;

std::atomic<MyStruct> p({.a=1, .c=true});
```

Weak vs Strong Exchange on ARM ***Extra Fun!* 2.5.1**

The arm architecture allows exchange to spuriously fail.

- This is documented behaviour for `exchange_weak`
- `exchange_strong` contains a loop that makes use of `exchange_weak`
- On x86 strong and weak are identical and do not spuriously fail.

2.5.2 Atomic Integral Types

For integral types, fast assembly supported instructions for atomic integers, booleans and floats can be used.

-

```
namespace std {
  template<> struct atomic</* integral */> {
    // ... normal operations from std::atomic<T>

    // Atomic operations
    /* integral */ fetch_add(/* integral */, memory_order = memory_order::seq_cst) volatile noexcept;
    /* integral */ fetch_add(/* integral */, memory_order = memory_order::seq_cst) noexcept;
    /* integral */ fetch_sub(/* integral */, memory_order = memory_order::seq_cst) volatile noexcept;
    /* integral */ fetch_sub(/* integral */, memory_order = memory_order::seq_cst) noexcept;
    /* integral */ fetch_and(/* integral */, memory_order = memory_order::seq_cst) volatile noexcept;
    /* integral */ fetch_and(/* integral */, memory_order = memory_order::seq_cst) noexcept;
    /* integral */ fetch_or(/* integral */, memory_order = memory_order::seq_cst) volatile noexcept;
    /* integral */ fetch_or(/* integral */, memory_order = memory_order::seq_cst) noexcept;
    /* integral */ fetch_xor(/* integral */, memory_order = memory_order::seq_cst) volatile noexcept;
    /* integral */ fetch_xor(/* integral */, memory_order = memory_order::seq_cst) noexcept;

    // Operator Overloads
    /* integral */ operator++(int) volatile noexcept;
    /* integral */ operator++(int) noexcept;
    /* integral */ operator--(int) volatile noexcept;
    /* integral */ operator--(int) noexcept;
    /* integral */ operator++() volatile noexcept;
    /* integral */ operator++() noexcept;
    /* integral */ operator--() volatile noexcept;
    /* integral */ operator--() noexcept;
    /* integral */ operator+=(/* integral */) volatile noexcept;
    /* integral */ operator+=(/* integral */) noexcept;
    /* integral */ operator-=(/* integral */) volatile noexcept;
    /* integral */ operator-=(/* integral */) noexcept;
    /* integral */ operator&=(/* integral */) volatile noexcept;
    /* integral */ operator&=(/* integral */) noexcept;
    /* integral */ operator|=(/* integral */) volatile noexcept;
    /* integral */ operator|=(/* integral */) noexcept;
    /* integral */ operator^=(/* integral */) volatile noexcept;
    /* integral */ operator^=(/* integral */) noexcept;

    // ... normal operations from std::atomic<T>
  };
}
```

For example we can see a single add is used for the `fetch_add` here.

```
#include <atomic>

int main() {
    std::atomic<int> x(1);
    x += 3;
}
```

```
main:
  mov       DWORD PTR [rsp-4], 1
  lock add  DWORD PTR [rsp-4], 3
  xor       eax, eax
  ret
```

Operator overloading is available for the integral types, however it can be difficult to determine which operations are atomic.

```
x = 42;         /* equivalent to */  x.store(42);
y = x;          /* equivalent to */  y = x.load();
x++;            /* equivalent to */  x.fetch_add(1);
y = ++x;        /* equivalent to */  y = x.fetch_add(1) + 1;
x += 42;        /* equivalent to */  x.fetch_add(42);
y = (x += 42); /* equivalent to */  y = x.fetch_add(42) + 42;
```

2.6 Memory Order

The `atomic` header provides several memory orderings:

```
namespace std {
  // ...

  enum class memory_order : /* unspecified */ {
    relaxed, consume, acquire, release, acq_rel, seq_cst
  };
  inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
  inline constexpr memory_order memory_order_consume = memory_order::consume;
  inline constexpr memory_order memory_order_acquire = memory_order::acquire;
  inline constexpr memory_order memory_order_release = memory_order::release;
  inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
  inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;

  //...
}
```

Sequential Consistency — **Definition 2.6.1**

The order of operations are executed as in the order of the program text.

- The default memory ordering for atomics (`std::atomics::memory_order_seq_cst`)
-

Simple Easy to reason about the interleaving of threads.

Expensive Compiler uses memory barriers which restrict how instructions can eb reordered and optimised.

Relaxed Memory Order — **Definition 2.6.2**

Guarantees only sequential consistency per location.

2.7 Message Passing

Threads can communicate without blocking through atomics.

- Poll on an atomic variable (potentially doing some other work while waiting)
- Often used for synchronising access to shared resources (e.g spin locks)

2.7.1 Expensive Approach

We can use sequential consistency to ensure that the

```
#include <atomic>

std::atomic<bool> flag(true);
SharedObj data(some_data);
```

```
use_data(data);

flag.store(true);
```

```
while (!flag.load()) {
// do nothing - a pure spinlock
}

use_data(data);
```

Slow On some architectures memory barriers are required for to ensure sequential consistency, are expensive.

Memory Barrier / Fence **Definition 2.7.1**

These prevent the reordering of load and store instructions by dynamically scheduled processors.

- On a single core processor this is not an issue (dynamic scheduling commits instructions effects in order)
- On a multicore system instruction reordering in execution stages can result in non-sequentially consistent accesses.
- Dynamic scheduling of instructions improves performance by filling potential *stalls* with useful computation.

2.7.2 Incorrect Approach

One approach could be to use relaxed memory ordering.

- Allows for other values (e.g the data being protected by a spinlock) to be re-ordered around the atomic.
- This removes the protection the spinloc is intended to provide.

```
// These can be reordered
use_data(data);

flag.store(true);
```

```
while (!flag.load()) {
// do nothing - a pure spinlock
}

    use_data(data);
```

2.7.3 Release-Acquire Consistency

Release acquire consistency

```
// These can be reordered
use_data(data);

flag.store(true, std::memory_order_release);
```

```
while (!flag.load(std::memory_orde
// do nothing - a pure spinlock
}

  use_data(data);
```

UNFINISHED!!!

Chapter 3

Credit

Image Credit

Front Cover Intel Xeon e7 on wikichip here.

Content

Based on the Concurrency course taught by Dr Azalea Raad and Prof Alastair Donaldson.

These notes were written by Oliver Killane.