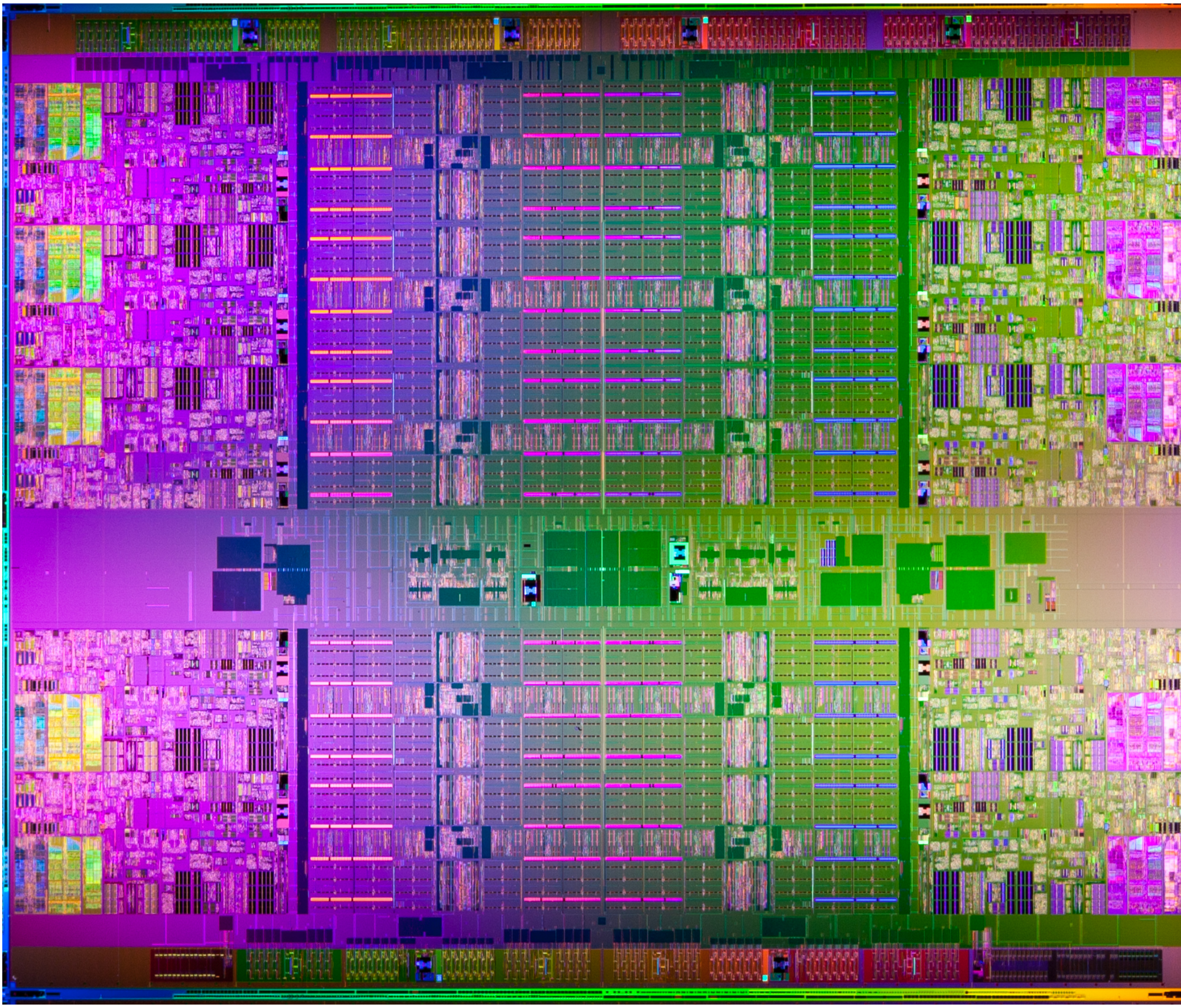


60007

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

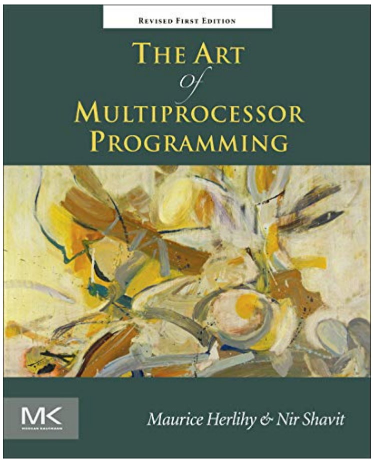
Theory For weeks 2 → 5:

- 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



Revised First Edition
The Art of Multiprocessor Programming
MK
Maurice Herlihy & Nir Shavit

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.

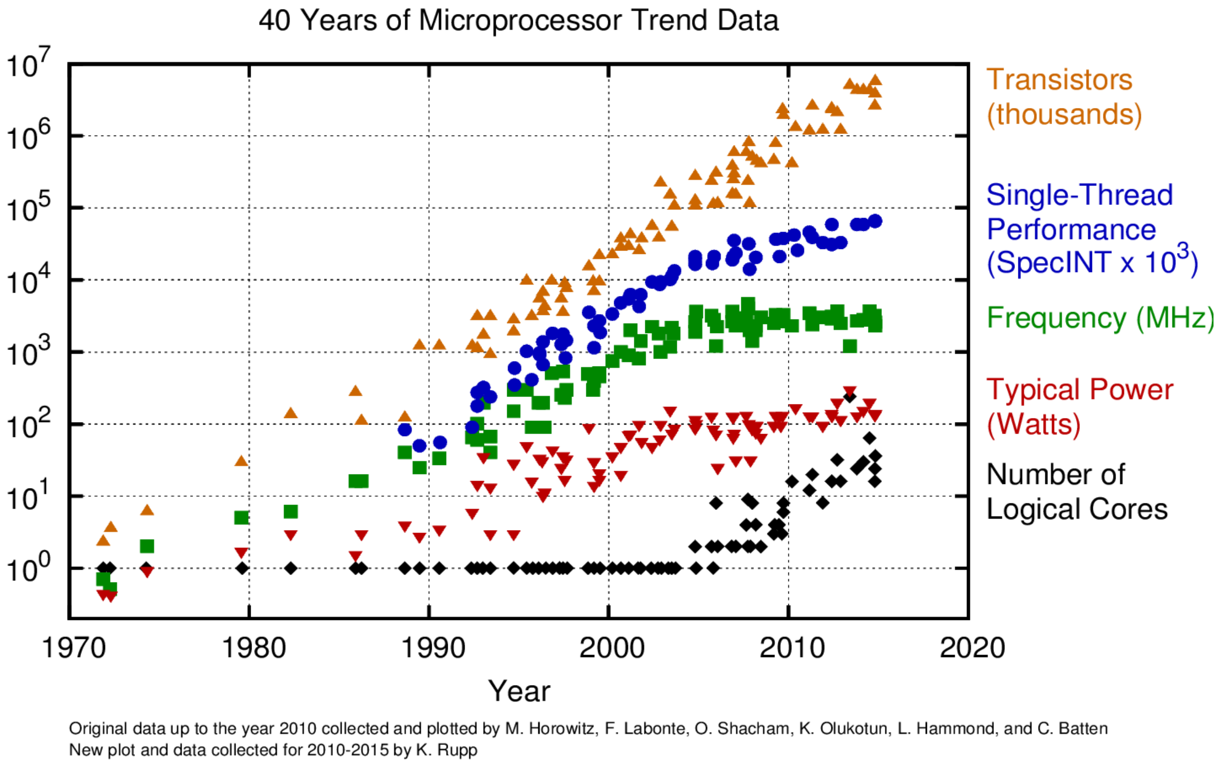
- The observation is named after Gordon Moore (co-founder and later CEO of Intel).
- This law no longer holds, and sequential performance improvements have declined.

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)

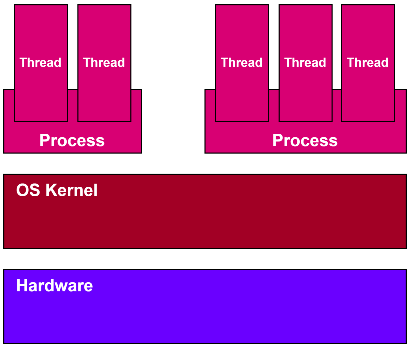
Race conditions can be intentional, where the result of the program is intended to be based off some non-deterministic input.

- 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.
```

```cpp
  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

```cpp
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.

```cpp
#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.

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

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

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_order_acquire)) {
// do nothing - a pure spinlock
}

  use_data(data);
```

UNFINISHED!!!

Chapter 3

Operational Semantics

3.1 Formal Properties

Safety Properties	*Nothing bad happens*	Only violated by finite computations
Liveness Properties	*Something good happens eventually*	Cannot be violated by finite computation

Deadlock is a **liveness** problem, while Mutual exclusion is a **Safety** problem.

Communication Deadlock **Definition 3.1.1**

When using transient communication, messages can be lost. A thread may wait on a reply from another thread, that never received to prompt to reply in the first place, thus causing a deadlock.

- Mutual Exclusion cannot be solved with transient communication
- Interrupts can also not work?

Mutual Exclusion **Definition 3.1.2**

When only one thread can execute in a critical region at a time, there is mutual exclusion.

- Mutual exclusion enforces removes parallelism for the critical section, limiting speedup from parallelism (Amdahl's law)

Turing Computability **Definition 3.1.3**

A model of computation that describes what is computable.

- Efficiency mostly irrelevant
- Only covers sequential computation.

Shared-Memory Computability **Definition 3.1.4**

A model for concurrent computation.

- Describes what is concurrently computable.
- Efficiency mostly irrelevant

3.2 Shared-Memory Concurrency

3.2.1 Read-Modify-Write

Read-Modify-Write Instructions **Definition 3.2.1**

An instruction that reads, modifies (with some function) and writes to a memory location, returning the value prior to the modification.

```
//! Generically we can express this scheme for any data type
struct RMWLocation<A> {data: A}

impl<A: Clone> RMWLocation<A> {
    /// This function is synchronised
    fn read_modify_write(&mut self, apply: fn(&A) -> A) -> A {
        let old_value = self.data.clone();
        self.data = apply(&self.data);
        old_value
    }
}
```

There are many different RMW instructions, a read can be considered an RMW instruction (where modification applies is just identity).

Weak RMW **Definition 3.2.2**

Allows for synchronisation between two threads.

- exchange Write - a new value to the location.
- fetch and add - Atomically add to an integer at a location.

Strong RMW **Definition 3.2.3**

Allows for synchronisation between an arbitrary number of threads.

- compare and set (CAS) - If the value is equal to the expected, set to updated and return true, else return false.

Many early machines provided weak RMW instructions (Test-and-set in IBM 360, Swap in original SPARCs), we now understand the limitations of these.

- All intel x86 architectures support CAS.
- ARM supports CAS through through load-linked and store-conditional instructions.

3.2.2 Consistency/Memory Models

Sequential Consistency **Definition 3.2.4**

Also known as interleaving semantics.

- Instructions for each thread are executed in order.
- Instructions from different threads can be interleaved arbitrarily.

Sequential Consistency Model

- Can work on a uniprocessor system (simple/idealised).
- A good abstraction for concurrency & easier to reason about.
- Not available on any hardware platform by default.
- Inefficient and expensive to implement.

Hardware Consistency Models

- A weak memory model (due to dynamic scheduling on processors)
- Complex for multicore systems.
- Hardware implementation has to deal with complexities such as cache coherence.

Software/Programming Language Consistency Models

- A weak memory model (compiler can reorder instructions, also must accommodate hardware)
- Determined by the language specification, programmer uses this specification, compiler adapts to hardware.
- C/C++ 2011 model (C11 model) (e.g `atomic.h`)
- Java Memory Model

3.3 Sequential Consistency

We can create a basic while-language for sequential consistency.

$$B \in Bool ::= \dots \quad E \in Exp ::= \dots \quad x, y, x \dots \in Loc ::= \text{(Memory Location)} \quad a, b, c \dots \in Reg ::= \text{(Register)}$$

$$\begin{aligned} C \in Com ::= {} & a := E \\ & \mid a := x \\ & \mid x := a \\ & \mid a := \mathsf{CAS}(x, E, E) \\ & \mid \mathit{FFA}(x, E) \\ & \mid \mathsf{skip} \\ & \mid C \;;\; C \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ C \\ & \mid \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C \end{aligned}$$

Concurrent programs are modelled as a map from thread identifiers to sequential commands.

$$\tau \in Tid \quad \text{and} \quad P \in Prog \triangleq Tid \to Com$$

A concurrent program can be expressed using $||$ as:

$$C_1 || C_2 || C_3 || \dots || C_n \text{ for program } P \text{ where } dom(P) = \{\tau_1, \tau_2, \tau_3, \dots, \tau_n\} \text{ and } P(\tau_i) = C_i \text{ for } i \in \{1, 2, 3, \dots, n\}$$

Racey Increment **Example Question 3.3.1**

Write a concurrent program inc that comprises of two threads which increment some shared memory.

$$P_{\text{inc}} \triangleq \begin{array}{l} a1 := cnt \\ a1 := a1 + 1 \\ cnt := a1 \end{array} \;\middle\|\; \begin{array}{l} a2 := cnt \\ a2 := a2 + 1 \\ cnt := a2 \end{array}$$

We can also express this as:

$$dom(P_{\text{inc}}) = \{\tau_1, \tau_2\} \qquad \begin{aligned} P_{\text{inc}}(\tau_1) &= a1 := cnt \;;\; a1 := a1 + 1 \;;\; cnt := a1 \\ P_{\text{inc}}(\tau_1) &= a2 := cnt \;;\; a2 := a2 + 1 \;;\; cnt := a2 \end{aligned}$$

3.3.1 Configurations

$$\text{Shared memory } M \in Mem \triangleq Loc \to Val$$

$$\text{Thread-local Registers } s \in Store \triangleq Reg \to Val$$

$$\text{A store map for threads } S \in SMap \triangleq Tid \to Store \text{ where } S(\tau) = s$$

Hence the configuration is a triple of the concurrent program, shared memory and map to thread local stored.

$$(P, S, M)$$

3.3.2 Transitions

The operational semantics are split into two types of transition.

Program Transitions	A step inb program execution (e.g if condition)
Storage Transitions	Describes behaviour of memory (e.g read/write)

- By splitting operational semantics into two parts we can alter storage transitions later without having to change the program transitions.
- The program and storage transitions are combined through label transitions.

The labels are defined as:

$l \in Lab ::=$	ϵ	empty label such as when transitioning: $\text{skip} \ ; \ C \to C$
	$\mid (R, x, v)$	Read value v from memory location x
	$\mid (W, x, v)$	Write value v to memory location x
	$\mid (U, x, v_0, v_n)$	Successful update of x from $v_0 \to v_n$ (FFA or successful CAS)
	$\mid (U, x, v_0, \perp)$	Failed CAS of x where the old value of x was not v_0

We also have a total function $\text{eval}(s, E)$ or $\text{eval}(s, B)$ to evaluate expressions.

Total Function **Definition 3.3.1**

A function defined for all possible input values.

Hence any transition is:

$$C, s \xrightarrow{l}_c C', s' \text{ where } C, C' \in Com, \quad s, s' \in Store \text{ and } l \in Lab$$

The transitions are:

$$\frac{C_1, s \xrightarrow{l}_c C_1', s'}{C_1 \ ; \ C_2, s \xrightarrow{l}_c C_1' \ ; \ C_2, s'}$$

$$\frac{\text{eval}(s, B) = true}{\text{if } B \text{ then } C_1 \text{ else } C_2, s \xrightarrow{\epsilon}_c C_1, s}$$

$$\frac{\text{eval}(s, B) = false}{\text{if } B \text{ then } C_1 \text{ else } C_2, s \xrightarrow{\epsilon}_c C_2, s}$$

$$\frac{}{\text{skip} \ ; \ C, s \xrightarrow{\epsilon}_c C, s}$$

$$\frac{\text{eval}(s, E) = v \quad s' = s[a \mapsto v]}{a := E, s \xrightarrow{\epsilon}_c \text{skip}, s'}$$

$$\frac{}{\text{while } B \text{ do } C, s \xrightarrow{\epsilon}_c \text{if } B \text{ then } (C \ ; \ \text{while } B \text{ do } C) \text{ else } skip}$$

We must also consider the basic read/write transitions:

$$\frac{s(a) = v}{x := a \xrightarrow{(W, x, v)}_c skip, s}$$

$$\frac{s' = s[a \mapsto v]}{a := x, s \xrightarrow{(R, x, v)}_c skip, s'}$$

Note that program transitions do not consider memory, so no update takes place here on the memory write.

Finally we need to consider FFA and CAS.

$$\frac{\text{eval}(s, E) = v \quad v_n = v_0 + v}{FFA(x, E), s \xrightarrow{(U, x, v_0, v_n)}_c skip, s}$$

$$\frac{\text{eval}(s, E_0) = v_0 \quad \text{eval}(s, E_n) = v_n \quad s' = s[a \mapsto 1]}{a := \text{CAS}(x, E_0, E_n), s \xrightarrow{(U, x, v_0, v_n)}_c \text{skip}, s'}$$

$$\frac{\text{eval}(s, E_0) = v_0 \quad v \to v_0 \quad s' = s[a \mapsto 0]}{a := \text{CAS}(x, E_0, E_n), s}$$

3.3.3 Concurrent Program Transitions

$$\frac{P(\tau) = C \quad S(\tau) = s \quad C, s \xrightarrow{l}_c C', s' \quad P' = P[\tau \mapsto C'] \quad S' = S[\tau \mapsto s']}{P, S \xrightarrow{\tau : l}_p P', S'}$$

3.3.4 Storage Transitions

A storage transition is of the form $M \overset{\tau:l}{\rightarrow}_m M'$ (thread τ updates $M \rightarrow M'$ using label l).

- We will use the thread id τ to combine storage with program transitions later.
- We only consider the labels from program transitions (these affect the shared memory).

$$\frac{M(x) = v}{M \overset{\tau:(R,x,v)}{\rightarrow}_m M}$$

Memory Read

$$\frac{M' = M[x \mapsto v]}{M \overset{\tau:(W x,v)}{\rightarrow}_m M'}$$

Memory Write

$$\frac{M(x) = v_0 \quad M' = M[x \mapsto v_n]}{M \overset{\tau:(U,x,v_0,v_n)}{\rightarrow}_m M'}$$

Successful CAS or FFA

$$\frac{M(x) = v}{M \overset{\tau:(U,x,v,\bot)}{\rightarrow}_m M}$$

Failed CAS

3.3.5 Combining Operational Semantics

The combined semantics are of the form $P, S, M \rightarrow P', S', M'$

For example with

$$\frac{P, S \overset{\tau:\epsilon}{\rightarrow}_p P', S'}{P, S, M \rightarrow P'S', M}$$

Under ϵ label shared memory is unchanged

If the program and storage transitions are the same, then we can combine into a single transition

Skipping it! **Example Question 3.3.2**

Combine the memory and program transitions for skip.

The program transition can be expressed as:

$$\frac{P(\tau) = \text{skip} \ ; C \quad S(\tau) = s \quad \text{skip} \ ; C, s \overset{\epsilon}{\rightarrow}_c C, s \quad P' = P[\tau \mapsto C]}{P, S \overset{\tau:\epsilon}{\rightarrow}_p P', S}$$

As the program transition does not affect memory ($\text{skip} \ ; C, s \overset{\epsilon}{\rightarrow}_c C, s$) we can directly add M to the transition:

$$\frac{P(\tau) = \text{skip} \ ; C \quad S(\tau) = s \quad \text{skip} \ ; C, s \overset{\epsilon}{\rightarrow}_c C, s \quad P' = P[\tau \mapsto C]}{P, S, M \rightarrow P', S, M}$$

Read and Assign **Example Question 3.3.3**

Get the program transition for an assignment (reading memory into a register), where the memory value is 7.

$$\frac{M(x) = 7}{M \overset{\tau:(R,x,7)}{\rightarrow}_m M} \qquad \frac{s' = s[a \mapsto 7]}{a := x, s \overset{(R,x,7)}{\rightarrow}_c \text{skip}, s'}$$

$$\frac{P(\tau) = a := x \quad S(\tau) = s \quad s' = s[a \mapsto 7] \quad a := x, s \overset{(R,x,7)}{\rightarrow}_c \text{skip}, s' \quad P' = P[\tau \mapsto \text{skip}] \quad S' = S[\tau \mapsto s']}{P, S \overset{\tau:(R,x,7)}{\rightarrow}_p P', S'}$$

Hence we can now include the storage transition:

$$\frac{P, S \overset{\tau:(R,x,7)}{\rightarrow}_p P', S' \quad M \overset{\tau:(R,x,7)}{\rightarrow}_m M}{P, S, M \rightarrow P', S', M}$$

3.3.6 Traces

$\rightarrow^*$ for SC **Definition 3.3.2**

$$P, S, M \rightarrow^* P', S', M' \Leftrightarrow \begin{array}{l} (P, S, M) = (P', S', M') \\ \vee \exists (P'', S'', M'').[P, S, M \rightarrow P'', S'', M'' \wedge P'', S'', M'' \rightarrow^* P', S', M'] \end{array}$$

The reflexive, transitive closure of $\rightarrow$

- *Initial memory* is all zeros $M_0 \triangleq \lambda x.0$ (for any x, $M_0(x) = 0$).
- *Initial Store* is also originally all zeros. $s_0 \triangleq \lambda a.0$.
- *Initial store map* is $S_0 \triangleq \lambda \tau . s_0$
- *Terminated Program* is P_{skip} expressed as $P_{\text{skip}} \triangleq \tau.\text{skip}$.
- The *initial configuration* is (P, S_0, M_0).

Given a program P the *SC-trace* is the evaluation path:

$$P, S_0, M_0 \rightarrow^* P_{\text{skip}}, S, M$$

Where (S, M) is the *SC-outcome* of program P.

3.3.7 Properties of Sequential Consistency

Determinism

$$\forall P, P_1, P_2, S, S_1, S_2 M, M_1, M_2.[(P, S, M \rightarrow P_1, S_1, M_1 \wedge P, S, M \rightarrow P_2, S_2, M_2) \Rightarrow ((P_1, S_1, M_1) = (P_2, S_2, M_2))]$$

This does not hold due to the interleavings of the threads of P.

It has been determined... **Example Question 3.3.4**

Provide a counter example to SC being deterministic and confluent.

$$P = \begin{array}{l} a1 := 1 \\ x := a1 \end{array} \middle\| \begin{array}{l} a2 := 0 \\ x := a2 \end{array}$$

Here we can have $P, S_0, M_0 \rightarrow^* P_{\text{skip}}, S, M$ Where $M(x) = 1]$ or $M(x) = 0$.

Confluence

$$\forall P, P_1, P_2, S, S_1, S_2 M, M_1, M_2.[(P, S, M \rightarrow^* P_1, S_1, M_1 \wedge P, S, M \rightarrow^* P_2, S_2, M_2) \\ \Rightarrow \exists P', S', M'.[P_1, S_1, M_1 \rightarrow^* P', S', M' \wedge P_2, S_2, M_2 \rightarrow^* P', S', M']]$$

SC is not confluent for the same reason it is not deterministic (there are many possible *SC-outcomes* for a program)

3.4 Total Store Ordering

Weak Memory Models (WMM) **Definition 3.4.1**

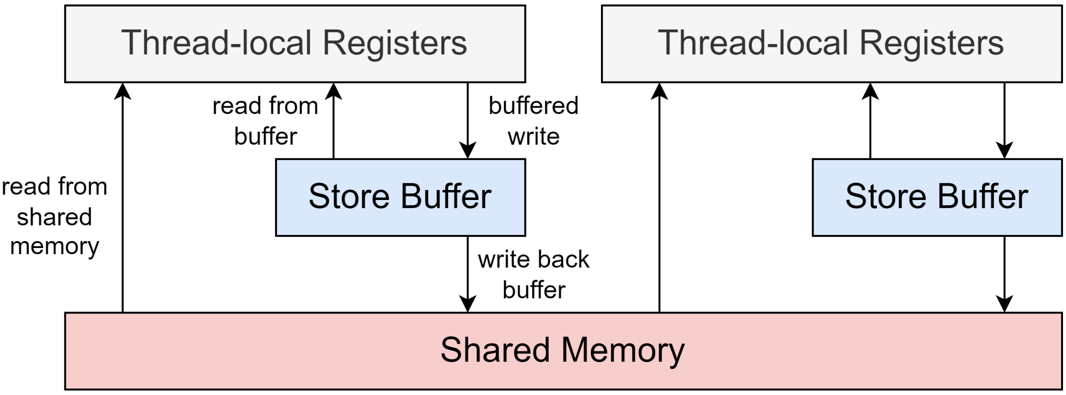
Allow for instructions in a thread to be reordered (e.g dynamically scheduled processors).

- *Weak behaviours* are states not observable under *sequential consistency*.
- Used by virtually all computer architectures (e.g TSO used by x86).

Total Store Ordering (TSO) **Definition 3.4.2**

A weak memory model that allows for write-read reordering between different memory locations.

- A later read on y can be reordered before an earlier write on x when $x \neq y$
- Includes the interleaving semantics from sequential consistency.
- Allows for weak store buffering (where)



$x := 1$	Add $x := 1$ to the store buffer.
$a := x$	If x is in the buffer, read latest entry, else read from shared memory.
unbuffer	Flush buffer to memory (FIFO order)
mfence	Barrier instruction, ensures no delayed writes are in the buffer (hence any subsequent reads happen after the current buffered writes).
RWMs	Act as barriers and ensure no delayed writes are in the buffer, they write directly to memory without delay.

The previous language used for sequential consistency can have fences added.

$$\begin{aligned} C \in Com ::= \ & a := E \\ & \mid a := x \\ & \mid x := a \\ & \mid a := \mathrm{CAS}(x, E, E) \\ & \mid FFA(x, E) \\ & \mid \mathrm{skip} \\ & \mid C \ ; \ C \\ & \mid \mathrm{while} \ B \ \mathrm{do} \ C \\ & \mid \mathrm{if} \ B \ \mathrm{then} \ C \ \mathrm{else} \ C \\ & \mid \mathrm{mfence} \end{aligned}$$

$$\frac{}{\mathrm{mfence}, s \xrightarrow{MF}_c \mathrm{skip}, s}$$

We model memory similarly to before, but now with a local buffer.

$$M \in Mem \triangleq Loc \rightarrow Val \qquad S \in SMap \triangleq Tid \rightarrow Store \qquad s \in Store \triangleq Reg \rightarrow Val$$

A buffer is a FIFO queue of delayed write labels.

$$b \in Buff \triangleq Seq\langle WLab \rangle \qquad Wlab \triangleq \{(W, x, v) | x \in Loc \wedge v \in Val\}$$

$$B \in BMap \triangleq Tid \rightarrow Buff \text{ where } b = B(\tau)$$

Hence a TSO configuration is:

$$(P, S, M, B)$$

3.4.1 Storage Transitions

TSO adds the mfence transition label MF

$$
\begin{aligned}
l \in Lab ::= {} & \epsilon \\
& \mid (R, x, v) \\
& \mid (W, x, v) \\
& \mid (U, x, v_0, v_n) \\
& \mid (U, x, v_0, \bot) \\
& \mid MF
\end{aligned}
$$

Storage transitions are of the form:

$$M, B \xrightarrow{\tau:l}_m M', B'$$

$$\frac{B(\tau) = b \quad b' = b.(W, x, v) \quad B' = B[\tau \mapsto b']}{M, B \xrightarrow{\tau:(W,x,v)}_m M, B'}$$

Memory Write

$$\frac{B(\tau) = b \quad get(M, b, x) = v}{M, B \xrightarrow{\tau:(R,x,v)}_m M, B}$$

$$get(M, b, x) \triangleq \begin{cases} v & \text{if } \exists b_1, b_2.[b = b_1.(W, x, v).b_2] \\ & \wedge \neg\exists v'.[(W, x, v') \in b_2] \\ M(x) & \text{otherwise} \end{cases}$$

Memory Read

$$\frac{B(\tau) = \emptyset}{M, B \xrightarrow{\tau:MF}_m M, B}$$

Memory Fence ensures no buffering

$$\frac{B(\tau) = \emptyset \quad M(x) = v_0 \quad M' = M[x \mapsto v_n]}{M, B \xrightarrow{\tau:(U,x,v_0,v_n)}_m M'B}$$

Successful RMW

$$\frac{B(\tau) = \emptyset \quad M(x) = v}{M, B \xrightarrow{\tau:(U,x,v,\bot)}_m MB}$$

Failed RMW

The buffered writes may be propagated at any time through a silent step, this is done using an ϵ storage transition.

$$\frac{B(\tau) = (W, x, v).b \quad M' = M[x \mapsto v] \quad B' = B[\tau \mapsto b]}{M, b \xrightarrow{\tau:\epsilon}_m M', B'}$$

$$\frac{P, S \xrightarrow{\tau:\epsilon}_p P', S'}{P, S, M, B \rightarrow P', S', M, B}$$

If the program takes a silent step, the storage system is unchanged.

$$\frac{M, B \xrightarrow{\tau:\epsilon}_m M', B'}{P, S, M, B \rightarrow P, S, M', B'}$$

If the storage system takes a silent step, the program & program's register store remains the same.

If both the program and storage systems make the same transition l then we can combine this into a transition over the TSO configuration.

$$\frac{P, S \xrightarrow{\tau:l}_p P', S' \qquad M, B \xrightarrow{\tau:l}_m M', B'}{P, S, M, B \rightarrow P', S', M', B'}$$

3.4.2 Traces

TSO inherits much of the initial state from SC:

$$M_0 \triangleq \lambda x.0 \qquad S_0 \triangleq \lambda \tau.s_0 \text{ with } s_0 \triangleq \lambda a.0 \qquad P_{\text{skip}} \triangleq \lambda \tau.\text{skip}$$

However we add the *initial buffer map*:

$$B_0 \triangleq \lambda \tau.\emptyset$$

The *initial TSO-configuration* is hence:

$$(P, S_0, M_0, B_0)$$

Given some program P the *TSO-trace* is an evaluation path that starts from the *initial TSO-configuration* of P and terminates with P_{skip} and empty buffers.

$$P, S_0, M_0, B_0 \rightarrow^* P_{\text{skip}}, S, M, B_0 \text{ where } (S, M) \text{ is the } \textit{TSO-outcome}$$

3.4.3 Properties of Total Store Ordering

Determinism

Much like *sequential consistency* the interleaving of different threads makes TSO non-deterministic.

Confluence

Likewise, TSO is not confluent.

Chapter 4

Declarative Semantics

Declarative/Axiomatic Semantics **Definition 4.0.1**

An alternative to operational semantics.

- Defines the notion of program execution (generalisation of execution trace)
- Maps a program to a set of candidate executions
- Define a consistency predicate on executions

Semantics are defined as the set of consistent executions of a program.

Catch Fire Semantics **Definition 4.0.2**

The existence of one *bad* consistent execution implies undefined behaviour.

Executions are expressed as graphs.

Events	Graph Nodes	Reads, Writes, Updates & Fences
Relations	Graph Edges	Program order (po) and reads-from (rf).

Event **Definition 4.0.3**

$$\langle n, \tau, l \rangle$$

$n \in \mathbb{N}$ Unique Event Identifier $\tau \in \mathit{Tid} \cup \{0\}$ Thread Identifier l Non-empty label

Non-empty Label **Definition 4.0.4**

As labels are only associated with events, and events interact with memory, there is no concept of a ϵ empty label as in operational semantics.

Labels are model specific, so for sequential consistency they are:

$$(R, x, v_r) \qquad (W, x, v_w) \qquad (U, x, v_r, v_w)$$

where $x \in \mathit{Loc}$ and $v_r, v_w \in \mathit{Val}$

4.0.1 Label and Event Notation

$$\begin{aligned} \text{typ}((R, x, v_r)) &\triangleq R \\ \text{typ}((W, x, v_w)) &\triangleq W \\ \text{typ}((U, x, v_r, v_w)) &\triangleq U \end{aligned}$$

Get Label Type

$$\begin{aligned} \text{val}_r((R, x, v_r)) &\triangleq v_r \\ \text{val}_r((U, x, v_r, v_w)) &\triangleq v_r \\ \\ \text{val}_w((W, x, v_w)) &\triangleq v_w \\ \text{val}_w((U, x, v_r, v_w)) &\triangleq v_w \end{aligned}$$

Get read & write values.

$$\begin{aligned} \text{loc}((R, x, v_r)) &\triangleq x \\ \text{loc}((W, x, v_w)) &\triangleq x \\ \text{loc}((U, x, v_r, v_w)) &\triangleq x \end{aligned}$$

Get read & write values.

Given a set of events A, the relations $r, r' \subseteq A \times A$ and memory location x we have:

Identity on A	$[A]$	$\triangleq$	$\{(a,a) \mid a \in A\}$
Domain of r	$dom(r)$	$\triangleq$	$\{a \mid (a,-) \in r\}$
Range of r	$rng(r)$	$\triangleq$	$\{a \mid (-,a) \in r\}$
Inverse of r	r^{-1}	$\triangleq$	$\{(b,a) \mid (a,b) \in r\}$
Composition of r and r'	$r;r'$	$\triangleq$	$\{(a,c) \mid (a,b) \in r \wedge (b,c) \in r'\}$
Reflexive Closure of r	$r^?$	$\triangleq$	$r \cup [dom(r) \cup rng(r)]$
Transitive Closure of r	r^+	$\triangleq$	$\bigcup_{i=0}^{\infty} r^i$ where $r^0 \triangleq r$ and $r^{i+1} \triangleq r; r^i$ for $i > 0$
Reflexive & Transitive closure of r	r^*	$\triangleq$	$(r^+)^?$
	A_x	$\triangleq$	$\{e \in A \mid \text{loc}(e) = x\}$ and $r_x \triangleq r \cap (A_x \times A_x)$
	$r\|_{loc}$	$\triangleq$	$\{(a,b) \in r \mid \text{loc}(()a) = \text{loc}(b)\}$

Given an event set A, a relation $r \in A \times A$ and a thread τ we have:

Initialisation of events in A	A_0	$\triangleq$	$\{e \in A \mid \text{tid}(()a) = 0\}$
	A_τ	$\triangleq$	$\{e \in A \mid \text{tid}(a) = \tau\}$ and $r_\tau \triangleq \cap(A_\tau \times A_\tau)$
For internal edges (of the same thread)	ri	$\triangleq$	$\{(a,b) \in r \mid \text{tid}(a) = \text{tid}(b)\}$
For external edges	re	$\triangleq$	$\{(a,b) \in r \mid \text{tid}(a) \neq \text{tid}(b)\}$
$irreflexive(r)$		$\overset{def}{\Leftrightarrow}$	$\neg\exists a.[(a,a) \in r]$
$acyclic(r)$		$\overset{def}{\Leftrightarrow}$	$irreflexive(r^+)$

Partial Orders — Definition 4.0.5

Given some set A and relation R:

R is reflexive	$\forall x, \in A.$	$[x \ R \ x]$
R is irreflexive	$\forall x \in A.$	$[\neg(x \ R \ x)]$
R is symmetric	$\forall x, y \in A.$	$[x \ R \ y \Rightarrow y \ R \ x]$
R is anti-symmetric	$\forall x, y \in A.$	$[(x \ R \ y \wedge y \ R \ x) \rightarrow x = y]$
R is transitive	$\forall x, y, z \in A.$	$[(x \ R \ y \wedge y \ R \ z) \Rightarrow x \ R \ z]$

Pre-order	Reflexive and Transitive
Partial order	Anti-symmetric & pre-order. Hence is reflexive, transitive and anti-symmetric.
Strict partial order	Irreflexive and transitive
Total order	Partial order with $\forall x, y \in A.[x \ R \ y \vee y \ R \ x]$
Strict total order	Strict partial order with $\forall x, y \in A.[x \neq y \Rightarrow (x \ R \ y \vee y \ R \ x)]$

Function Types — Definition 4.0.6

$$f : A \to B \qquad dom(f) = A \qquad codom(f) = B$$

Injective/one-to-one	Each output is mapped to by at most one input.	$\forall x_1, x_2 \in A.[f(x_1) = f(x_2) \Rightarrow x_1 = x_2]$
Surjective/onto	Each output is mapped to by at least one input.	$\forall y \in B.\exists x \in A.[f(x) = y]$
Bijective	Each output is mapped to by one input.	$bijective(f) = injective(f) \wedge surjective(f)$

Execution Graph **Definition 4.0.7**

$$\langle E, \mathrm{po}, \mathrm{rf} \rangle$$

E	Finite set of events.
po	The Program order binary relation.
rf	Reads-from binary relation.

po is is such that:

$$\mathrm{po} \triangleq \left(\bigcup_{\tau \in tid} \mathrm{po}_\tau \right) \cup (E_0 \times (E \setminus E_0))$$

Each po_τ is a strict total order on E_τ.

rf is such that $\forall \langle w, r \rangle \in \mathrm{rf}$

$$\begin{aligned} & w \neq r \\ & \mathrm{typ}(w) \in \{W, U\} \land \mathrm{typ}(r) \in \{R, U\} \\ & \mathrm{loc}(w) = \mathrm{loc}(r) \\ & \mathrm{val}_w(w) = \mathrm{val}_r(r) \end{aligned}$$

rf^{-1} is a function:

Chapter 5

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.