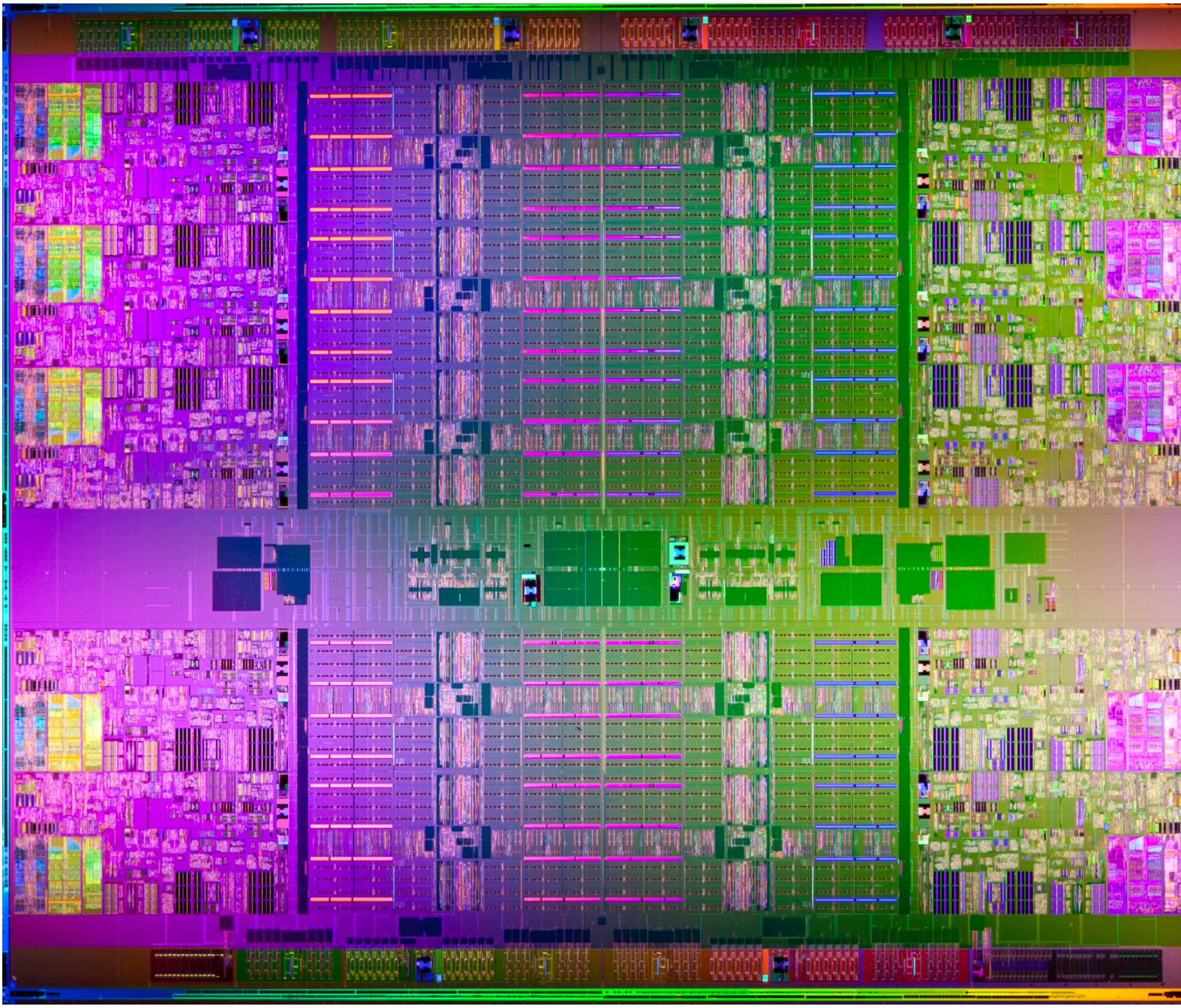


60007

Theory and Practice of Concurrent Programming

Imperial College London

Contents

1 Introduction **3**
1.1 Course Structure & Logistics . . . 3
1.1.1 Structure . . . 3
1.1.2 Extra Materials . . . 3
1.2 Preface for Concurrency . . . 4
1.2.1 Moore's Law . . . 4
1.2.2 Concurrency Difficulties . . . 5
1.2.3 OS Concepts . . . 5

2 Mutexes **6**
2.1 Spinlocks . . . 6
2.1.1 Test-and-Set Spinlock . . . 6
2.1.2 Local Spinning . . . 7
2.1.3 Fixed Backoff . . . 7
2.1.4 Exponential Backoff . . . 8
2.2 Ticket Locks . . . 9
2.3 Futexes . . . 9
2.3.1 Futex Operations . . . 10
2.3.2 Futex Implementation . . . 10
2.4 Hybrid/Adaptive Locking . . . 12

3 Concurrency In C++ **13**
3.1 Threads . . . 13
3.1.1 Vectors of Threads . . . 14
3.1.2 This Thread . . . 15
3.2 Locks . . . 16
3.2.1 Using Mutexes . . . 17
3.2.2 Lock Guards . . . 18
3.3 Race Conditions in C++ . . . 21
3.3.1 Thread Sanitiser . . . 22
3.4 Condition Variables . . . 23
3.4.1 Using Condition Variables . . . 24
3.5 Atomics . . . 25
3.5.1 Atomic Template Class . . . 25
3.5.2 Atomic Integral Types . . . 26
3.6 Memory Order . . . 27
3.7 Message Passing . . . 28
3.7.1 Expensive Approach . . . 28
3.7.2 Incorrect Approach . . . 29
3.7.3 Release-Acquire Consistency . . . 29

4 Concurrency in Haskell **30**
4.1 Managing Threads . . . 30
4.2 MVars . . . 31
4.3 Mutexes . . . 32
4.4 Channels . . . 33
4.5 Mutable References . . . 34

5 Concurrency in Rust **36**
5.1 Learning Rust . . . 37
5.2 Ownership in Rust . . . 37
5.3 Lifetimes . . . 38
5.4 Closures . . . 39
5.5 Threads . . . 39
5.5.1 Reference Counting . . . 40

6 Dynamic Data Race Detection **41**
6.1 Vector Clocks . . . 41
6.1.1 Notation . . . 41
6.1.2 Vector Clock Algorithm State . . . 42
6.1.3 Intercepted Operations . . . 42
6.1.4 Inference Rules . . . 43

7 Operational Semantics **44**
7.1 Formal Properties . . . 44
7.2 Shared-Memory Concurrency . . . 45
7.2.1 Read-Modify-Write . . . 45
7.2.2 Consistency/Memory Models . . . 45
7.3 Sequential Consistency . . . 46
7.3.1 Configurations . . . 46
7.3.2 Transitions . . . 47
7.3.3 Concurrent Program Transitions . . . 47
7.3.4 Storage Transitions . . . 48
7.3.5 Combining Operational Semantics . . . 48
7.3.6 Traces . . . 49
7.3.7 Properties of Sequential Consistency . . . 49
7.4 Total Store Ordering . . . 49
7.4.1 Storage Transitions . . . 51
7.4.2 Traces . . . 52
7.4.3 Properties of Total Store Ordering . . . 52

8 Declarative Semantics **53**
8.1 Label and Event Notation . . . 53
8.2 Consistency Predicates . . . 55
8.3 Sequential Consistency . . . 56
8.4 Total Store Order . . . 59
8.5 Coherent . . . 60
8.5.1 Bad Patterns . . . 61
8.6 Atomicity . . . 62
8.7 Release/Acquire . . . 62
8.8 Ordering Models . . . 63

9 Concurrent Objects **64**
9.1 Concurrent Queues . . . 64
9.1.1 Circular Queue . . . 64
9.1.2 Lock Based Queue . . . 65
9.1.3 Wait Free 2-Thread Queue . . . 65
9.2 Sequential and Concurrent Objects . . . 65
9.2.1 Sequential Specifications . . . 65
9.2.2 Concurrent Specifications . . . 65
9.3 Formal Model of Executions . . . 68
9.4 Linearisability . . . 69
9.5 Sequential Consistency . . . 70

10 Credit **71**

Chapter 1

Introduction

1.1 Course Structure & Logistics

1.1.1 Structure

Dr Azalea Raad

Prof Alastair Donaldson

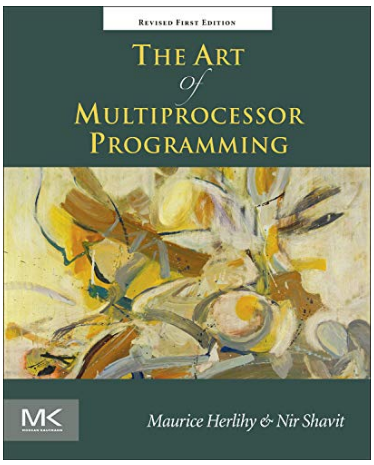
Theory For weeks 2 → 5:

- 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.

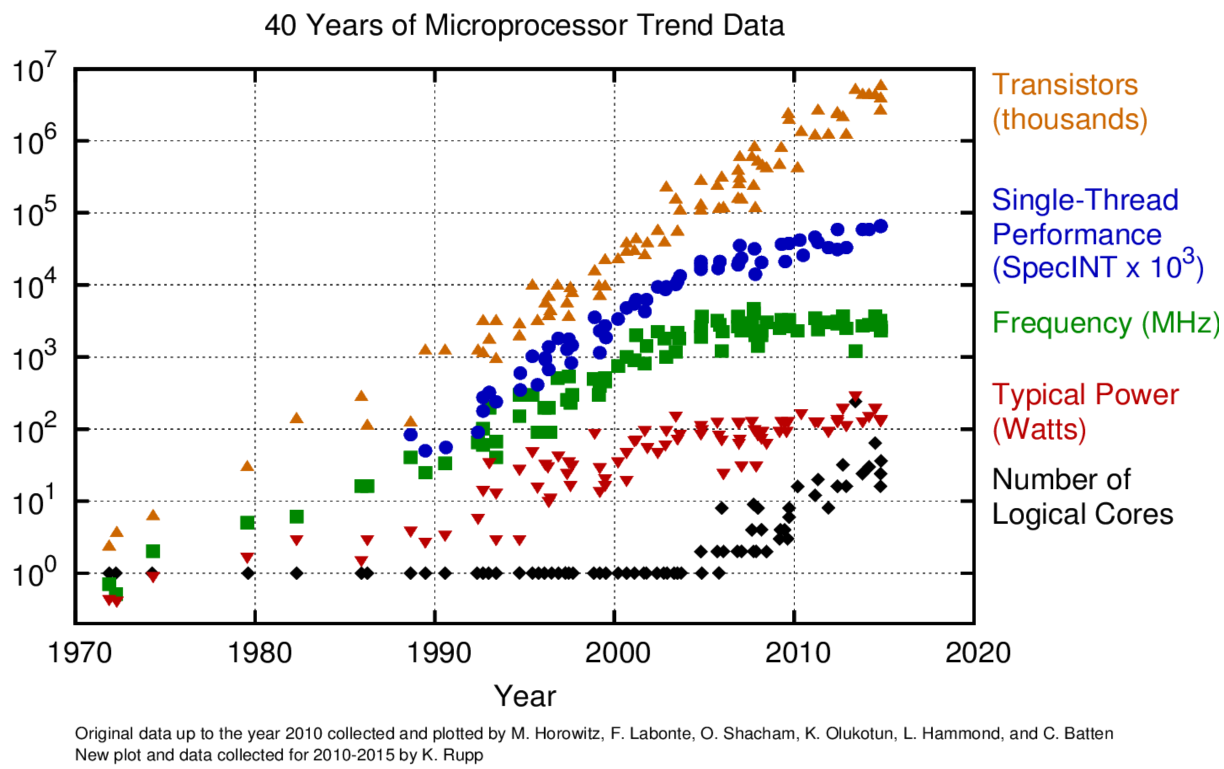
- The observation is named after Gordon Moore (co-founder and later CEO of Intel).
- This law no longer holds, and sequential performance improvements have declined.

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).



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)

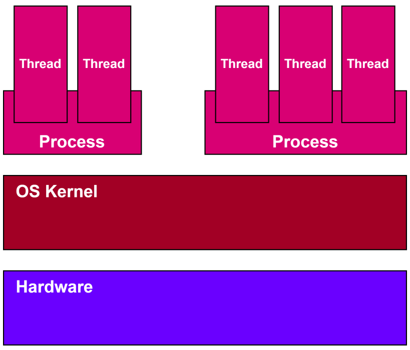
Race conditions can be intentional, where the result of the program is intended to be based off some non-deterministic input.

- 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

Mutexes

2.1 Spinlocks

2.1.1 Test-and-Set Spinlock

Test-and-Set Spinlock **Definition 2.1.1**

A single bit is used to determine if the lock is acquired or not.

- Threads can use an atomic test-and-set operation to attempt to acquire the lock.
- Avoids making system calls (expensive)
- Wastes CPU time while spinning - hence useful only when the expected wait / critical section is short.
- Poor worst case behaviour.

Hybrid locks that spin before using an OS provided lock mechanism exist to overcome the worst-case behaviour.

A simple spinlock can be implemented in C++ as:

```
#include <atomic>

class SpinLock {
  public:
    SpinLock() : _lock_bit(false) {}

    void Lock() {
      while (_lock_bit.exchange(true));
    }

    void Unlock() {
      _lock_bit.store(false)
    }

  private:
    std::atomic<bool> _lock_bit;
}
```

This simple lock has an issue with *cache thrashing*.

- Each core on a multicore CPU has its own caches (L2 (*often*), L1 cache)
- A *cache coherency protocol* is used to ensure different cache lines in different caches on different cores are coherent / if valid contain the correct value (not an outdated one)
- Typically when a location is written to in some cache, invalidation messages are sent to the other caches.

As in every attempt to acquire the lock, the test and set instruction writes to the lock

2.1.2 Local Spinning

To reduce the frequency of invalidations a ticket based spinlock can be used.

- First check if the lock is held (read - does not invalidate)
- Then if the lock is not held, attempt to access with an RMW operation (exchange) which writes & invalidates.

```
class SpinLockLocalSpinning : public SpinLock {
  public:
    SpinLock() : _lock_bit(false) {}

    void Lock() {
      while (_lock_bit.exchange(true)) { // <- threads can still have a 'TAS fight' here
        // could not get the lock, so keep reading until it is free
        while (_lock_bit.load());
        // the lock is now free, so we attempt to acquire it again
      }
    }
}
```

Bus traffic is still an issue - we want to reduce the rate of accesses. This will also help to reduce the contention.

2.1.3 Fixed Backoff

Active Backoff — **Definition 2.1.2**

On each iteration of local spinning, do nothing / wait for some fixed time period. This waiting is *active* (e.g spin in a loop).

We can implement active backoff by adding a fixed, redundant `for` loop.

```
class SpinLockLocalActiveBackoff : public SpinLock {
  public:
    void Lock() {
      while (_lock_bit.exchange(true)) { // <- threads can still have a 'TAS fight' here
        // could not get the lock, so keep reading until it is free
        do {
          // volatile prevents the loop from being eliminated
          // spin for N iterations, we could also use library provided busy-wait functions
          for (volatile size_t i = 0; i < N; i++);
        } while (_lock_bit.load());
        // the lock is now free, so we attempt to acquire it again
      }
    }
}
```

Passive Backoff — **Definition 2.1.3**

On each iteration of local spinning, do nothing / wait for some fixed time period using a processor instruction that informs the CPU to wait/do nothing.

- `pause` on x86, accessed through the `_mm_pause()` SSE2 intrinsic
- `YIELD` on ARM

These instructions inform the CPU to do nothing for some period of time.

- Less energy consumption than actively spinning.
- Does not induce an OS context switch, simply a *do nothing* instruction.

```
#include <emmintrin.h>

class SpinLockLocalPassiveBackoff : public SpinLock {
```

```
  public:
    void Lock() {
      while (_lock_bit.exchange(true)) {
        // could not get the lock, so keep reading until it is free
        do {
          // pause N times
          for (size_t i = 0; i < N; i++) {
            _mm_pause()
          }
        } while (_lock_bit.load());
        // the lock is now free, so we attempt to acquire it again
      }
    }
}
```

Both active and passive backoff use a fixed backoff period.

- For a short backoff while little time is *wasted* (lock free but threads are backed-off), threads are more likely to TAS at similar times (more potential for contention on the lock acquire)
- For a long backoff the timings of TAS diverge for different threads, however more time can be spent backed-off when the lock is acquirable.

2.1.4 Exponential Backoff

Exponential Backoff **Definition 2.1.4**

We can increase the backoff time exponentially for every iteration of local spinning.

- Start from a small value
- Double at each local spin iteration, until some maximum value is hit.

```
// So we can specify backoffs: SpinLockLocalExponentialBackoff<4,4096>()
#template<
  size_t MIN_BACKOFFS = 2,
  size_t MAX_BACKOFFS = 2048
>
class SpinLockLocalExponentialBackoff : public SpinLock {
  public:
    void Lock() {
      size_t backoffs = MIN_BACKOFFS;
      while (_lock_bit.exchange(true)) {
        // could not get the lock, so keep reading until it is free
        do {
          for (size_t i = 0; i < backoffs; i++) {
            _mm_pause()
          }
          // double the backoffs up to a limit
          backoffs = std::min(backoffs * 2, MAX_BACKOFFS)
        } while (_lock_bit.load());
      }
    }
}
```

- Much like with fixed backoffs, there may be *wasted* time between the lock being released and a thread acquiring it (thread is backed-off/doing nothing)
- Higher contention means more failed acquisitions and longer backoff periods

2.2 Ticket Locks

Unfairness

There is a tension between fairness and performance for spinlocks:

- Some thread may be unable to acquire a lock continually, while others may be able to reacquire frequently
- Threads reacquiring is good for temporal locality of cache (i.e a thread traversing a data structure will get better cache performance if it does not wait often - waiting allows opportunities for other threads to evict cached parts of the data structure)
- The lack of a fairness guarantee means thread starvation can occur, this can result in longer waits (e.g thread A is waiting on data from thread B, but thread B is being starved of a lock it needs by thread C to M)

A ticket lock a lock implementation that provides fairness guarantees.

Ticket Lock — **Definition 2.2.1**

A spinlock that uses an integer *ticket* to determine which of some waiting threads can run.

- To acquire, threads request a ticket, then spin until the lock matches their ticket.
- To release a thread sets the lock to serve the next ticket

```
#include <atomic>

class TicketLock {
  public:
    SpinLock() : _next_ticket_(0), _serving_ticket_(0) {}

    void Lock() {
      const auto ticket = _next_ticket_.fetch_add(1);
      while (_serving_ticket_.load() != ticket) {
        _mm_pause();
      }
    }

    void Unlock() {
      _serving_ticket_.store(_serving_ticket_.load() + 1);
    }

  private:
    std::atomic<size_t> _next_ticket_;
    std::atomic<size_t> _serving_ticket_;
}
```

2.3 Futexes

Spinlock

- Thready busy-waits until the lock is free.
- Uses atomic RMW operations
- No system calls required, operates entirely in user-space

Staying awake is expensive.

Useful when contention is low and critical sections are short

High contention results in lots of wasteful *spinning* rather than useful work

Sleeping Lock

- Use of a system call to ask the OS to block the thread until the lock is available.
- While the thread is blocked, other threads can be run.

Going to sleep is expensive.

Useful for long critical sections.

Overhead of system call is problematic when critical section is short - lock will be free imminently.

Confusing Terminology *Extra Fun!* **2.3.1**

note that `futex` is a name for a linux system call.

- Exposes kernel functionality (so is not userspace).
- Works on userspace data, hence can be used to implement synchronisation primitives (not just mutexes) that operate mostly in userspace.

Fast Userspace Mutex (Futex) **Definition 2.3.1**

2.3.1 Futex Operations

```
/* Wait on a futex
 * if *p != v -> Returns immediately
 * else       -> Adds the calling thread to the wait queue associated with p.
 */
void futex_wait(int *p, int v);
```

- The threads can be from different processes

```
/* Wake up a wake_count threads in the thread queue associated with p */
void futex_wake(int *p, int wake_count);
```

- Typically `wake_count` is 1 (wake next/one) or INT_MAX (wake all)

Here `int *p` can be a pointer to any *int-sized* data.

2.3.2 Futex Implementation

```
class MutexFutexNaive {
  public:
    MutexFutexNaive() : state_(kFree) {}

    void Lock() {
      while (state.exchange(kLocked) == kLocked) {
        syscall( SYS_futex, &state_, FUTEX_WAIT, kLocked, ...);
      }
    }

    void Unlock() {
      state_.store(kFree);
      syscall(SYS_futex, &state_, FUTEX_WAKE, 1, ...);
    }
  private:
    const int kFree = 0;
    const int kLocked = 1;
    std::atomic<int> state_;
};
```

- The value stored at `*p` is simply used as a key into a queue that the kernel holds. The queue is allocated on the first thread put to sleep, and freed when no threads are sleeping.
- Hence the kernel only *knows about* (has queue allocated for key `*p`) this mutex if some threads are sleeping.

Barging **Definition 2.3.2**

In the provided `class MutexFutexNaive` a thread waking up, and aquiring the lock is not atomic. Hence we can have the scenario:

1. T_1 is awoken
2. T_2 uses TAS to attempt to acquire the lock → Success! (T_2 is *barging in*)
3. T_1 uses TAS to attempt to acquire the lock → Failure!
4. T_1 is put to sleep.

This implementation requires a syscall to wake, even when no threads are waiting. We can remove this inefficiency by storing more state in the mutex.

```
// note: this code can be greatly simplified - it is verbose for understanding
class MutexFutexSmart {
  public:
    MutexFutexSmart() : state_(kFree) {}

    void Lock() {
      if (state_.compare_exchange_strong(kFree, kLockedNoWaiters)) {
        // Was free, is now locked with a single waiter.
        // The lock is now acquired with no waiters.

      } else {
        // The state_ must be either 1 or 2, hence we compare:
        do {
          // If locked with no waiters, set to locked with waiters
          state_.compare_exchange_strong(kLockedNoWaiters, kLockedWaiters);
          // start waiting
          syscall(SYS_futex, &state_, FUTEX_WAIT, kLockedWaiters, ...);

          // if we can acquire the lock (replace free with locked with waiters, we return, otherwise atte
        } while (!state_.compare_exchange_strong(kFree, kLockedWaiters));
      }
    }

    void Unlock() {
      if (state_.sub(1) == kLockedWaiters) {
        // There are potentially other thread
        state_.store(kFree)
        syscall(SYS_futex, &state_, FUTEX_WAKE, kLockedWaiters, ...);
      } else {
        // subtracted to kfree and no need to wake any threads
      }
    }

  private:

    const int kFree = 0;
    const int kLockedNoWaiters = 1;
    const int kLockedWaiters = 2;

    std::atomic<int> state_;
};
```

- A thread that *barges* in and sets the state to `lockedNoWaiters` (from `free`) does not cause any waiters to sleep forever as the thread unlocking awakens a thread, which will set it to `LockedWaiters`
- No kernel memory is used if no threads are waiting

- As we use a key into the kernel for the lock, we can share this key between processes (can synchronise threads in different processes)

2.4 Hybrid/Adaptive Locking

In order to avoid *microcontention* a lock may spin before sleeping.

- The time spent spinning can be adaptive
- Threads can attempt to lock fast (e.g using test-and-set), but worst case spinlock behaviour is avoided by sleeping.

Hybrid Locks in the Wild ***Extra Fun!* 2.4.1**

WebKit abstracts away spinlocks and OS-provided mutexes under a single primitive (`WTF::Lock`)

Java uses adaptive locks (discussed in this blog post)

Chapter 3

Concurrency In C++

3.1 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 3.1.1**

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 3.1.1**

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();
```

3.1.1 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 3.1.2**

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();
}
```

3.1.2 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 3.1.3**

Create program that prints the thread id, and sleeps.

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

```cpp
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;
}
```

3.2 Locks

RAII **Definition 3.2.1**

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`.

```cpp
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.

```cpp
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);
}
```

3.2.1 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 3.2.1**

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;
```

```cpp
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;
}
```

3.2.2 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

```cpp
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`).

```cpp
#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

```cpp
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 3.2.2**

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()
    }
}
```

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

3.3 Race Conditions in C++

Data Race **Definition 3.3.1**

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-oblivious, 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.

Synchronisation is achieved via:

- Mutexes
- Acquire load from release store
- sequentially consistent load from sequentially consistent store

Undefined Behaviour **Definition 3.3.2**

> *"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.

```
#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.

3.3.1 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`.

3.4 Condition Variables

Condition Variable **Definition 3.4.1**

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 };
}
```

3.4.1 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 {...});
}
```

3.5 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)

3.5.1 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
    // the old value (regardless is expected) is placed in expected
    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!* 3.5.1**

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.

3.5.2 Atomic Integral Types

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

```cpp
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.

```cpp
#include <atomic>

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

```asm
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.

```cpp
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;
```

3.6 Memory Order

The `atomic` header provides several memory orderings:

```cpp
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 3.6.1**

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 be reordered and optimised.

Relaxed Memory Order — **Definition 3.6.2**

Guarantees only sequential consistency per location.

3.7 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)

3.7.1 Expensive Approach

We can use sequential consistency to ensure that the

```cpp
#include <atomic>

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

```cpp
use_data(data);

flag.store(true);
```

```cpp
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 3.7.1**

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.

3.7.2 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 spinlock is intended to provide.

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

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

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

    use_data(data);
```

3.7.3 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);
```

The load is hence synchronised with the store, and reordering is prevented.

- Ensures correctness
- Potentially uses fewer memory barriers than SC (depends on the architecture)

Chapter 4

Concurrency in Haskell

Haskell is an exemplar of the functional programming paradigm

- Rich type system with powerful type inference, higher kinded-types, pattern matching, generalized algebraic datatypes, etc.
- Imperative programming can be performed by binding functions operating on monads (e.g IO), often using `do` notation (syntactic sugar)
- Haskell can be easily made parallel with no changes to code (using runtime options for enabling SMP parallelism)

We are interested in concurrency (specifically threads communicating via shared resources).

```
# threaded allows multiple cores to be used
ghc Program.hs -threaded

# We can also optimise with -O flags
ghc -O2 Program.hs -threaded

# Run, informing the haskell runtime system (RTS) to schedule on 8 threads
./Program args +RTS -N8
```

4.1 Managing Threads

```haskell
import Control.Concurrent

{- A data type for a handle of a thread, it also represents a pointer to the
  thread and the thread cannot be garbage collected until this pointer is
  dropped.
-}
data ThreadId

{- Takes in a computation (here in the IO monad) and returns a ThreadID (wrapped
   with IO).

   Note that the thread is run asynchronously (if main thread terminates, all
   threads end).
-}
forkIO :: IO () -> IO ThreadId
```

The threads created with `forkIO` are user-level threads managed by the *Haskell Runtime System*.

- Many of these lightweight thread can be scheduled across a smaller number of OS provided threads
- There is functionality provided in Haskell for creating OS managed threads (`forkOS :: IO () -> IO ThreadId`)

Thready Set Go! — **Example Question 4.1.1**

Create a basic haskell program to print "hello world" on n different threads.

```haskell
import Control.Concurrent (MVar, takeMVar, newEmptyMVar, putMVar, forkIO)
import Control.Monad (replicateM, zipWithM)

printHello :: MVar () -> Integer -> IO ()
printHello h i = do
  -- print out the message
  putStrLn ("This is thread " ++ show i ++ " saying hello world!")

  -- place a value in the mutable variable
  putMVar h ()

main :: IO ()
main = do
  -- create the mutable variables
  mvars <- replicateM 10 newEmptyMVar

  -- for each mutable variable fork off a process with it & the index
  res <- zipWithM (\ x y -> forkIO (printHello x y)) mvars [1..]

  -- for each mutable variable, attempt to get the value (will wait until a value is put)
  mapM_ takeMVar mvars
```

Map a monad — ***Extra Fun!* 4.1.1**

```haskell
import Control.Monad

mapM  :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
mapM_ :: (Foldable t, Monad m)    => (a -> m b) -> t a -> m ()
```

We can use `mapM` to bind together the application of some function to every item in some traversable structure. There are also functions for `zipWithM`, `foldM` etc.

The `function_` variants discard the final value (equivalent to finally binding `return ()`).

4.2 MVars

An `MVar` is a mutable variable.

```haskell
import Control.Concurrent (MVar, newEmptyMVar, newMVar, takeMVar, putMVar)

-- Creates a new empty MVar
newEmptyMVar :: IO (MVar a)

-- Creates a new full MVar
newMVar :: a -> IO (MVar a)

-- Blocks until the MVar is full, then makes it empty
takeMVar :: MVar a -> IO a

-- Blocks until the mVar is empty, and then places a value (making the MVar full)
putMVar :: MVar a -> a -> IO ()
```

- Can be accessed by multiple threads.
- Contains another type (`Int`, `String`, `Maybe Bool`, anything of kind $*$)
- Is either empty, of full, we can create empty variables, put to empty variables & take from empty variables.

- Taking from an empty variable will wait on the variable being made full, in this way we can wait for threads to create values, or to terminate.
- Internally MVars are backed by mutexes.

Join me! **Example Question 4.2.1**

Send a message from a thread to main, have main join on the result.

```haskell
import Control.Concurrent ( forkIO, newEmptyMVar, putMVar, takeMVar, MVar )

childThread :: MVar String -> IO ()
childThread m = do
  let msg = "Hello notes reader!"
  putStrLn $ "Determining the message as: " ++ msg
  putMVar m msg

main :: IO ()
main = do
  msgVar <- newEmptyMVar
  forkIO $ childThread msgVar
  msg <- takeMVar msgVar
  putStrLn $ "Got the message: " ++ msg
```

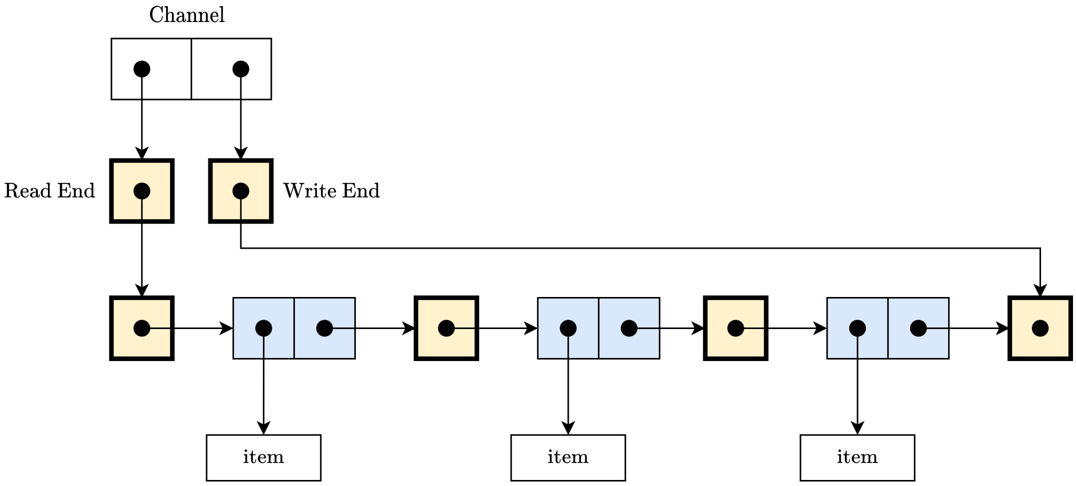
4.3 Mutexes

We can use the `MVar ()` type as a mutex.

```haskell
inlock :: MVar () -> IO a -> IO a
inlock mutex comp = do
    -- Wait until the mutex is empty, and then fill to acquire
    putMVar mutex ()

    -- do some computation
    res <- comp

    -- Release the mutex by emptying it
    takeMVar mutex

    return res
```

We could also do this the other way around (with an empty MVar meaning the mutex is locked)

4.4 Channels



We can represent this using MVars.

```
type Stream a = MVar (item a)
data Item   a = Item a (Stream a)

data Channel a = Channel (MVar (Stream a)) (MVar (Stream a))

{- Create a new Channel

readEnd -> emptyMVar -> writeEnd

-}
newChannel :: IO (Channel a)
newChannel = do
    hole      <- newEmptyMVar
    readEnd   <- newMVar hole
    writeEnd  <- newMVar hole
    return (Channel readEnd writeEnd)

writeChannel :: Channel a -> a -> IO ()
writeChannel (Channel _ writeEnd) val = do
    -- oldHole -> writeEnd
    newHole <- newEmptyMVar
    oldHole <- takeMVar writeEnd

    -- oldHole -> val -> writeEnd
    putMVar oldHole (Item val newHole)
    putMVar writeEnd newHole

readChannel :: Channel a -> IO a
readChannel (Channel readEnd _) = do
    -- readEnd -> item -> tail
    stream         <- takeMVar readEnd
    Item val tail <- takeMVar stream

    -- readEnd -> tail
    putMVar readEnd tail

    return val
```

Control.Concurrent.Chan contains a channel abstraction to use:

```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
```

```haskell
readChan   :: Chan a -> IO a
dupChan    :: Chan a -> IO (Chan a)

-- and for converting to/from lists
getChanContents :: Chan a -> IO [a]
writeList2Chan :: Chan a -> [a] -> IO ()
```

4.5 Mutable References

MVars are not well suited to fine-grained concurrency, hence we can use IORef for atomic mutable access to references.

```haskell
import Data.IORef
-- Create a new reference
newIORef :: a -> IO (IORef a)

-- read a value from an IORef
readIORef :: IORef a -> IO a

-- write a value into an IORef
writeIORef :: IORef a -> a -> IO ()

-- Modify the contents of an IORef with some function,
-- the ' variant is strict
modifyIORef  :: IORef a -> (a -> a) -> IO ()
modifyIORef' :: IORef a -> (a -> a) -> IO ()

-- atomically modify the contents of an IORef (safe for multithreaded programs),
-- the functions can return a second value (here of type b) which is useful for
-- returning the old value, new value etc
-- the ' variant is strict
atomicModifyIORef  :: IORef a -> (a -> (a, b)) -> IO b
atomicModifyIORef' :: IORef a -> (a -> (a, b)) -> IO b
```

Atomic Operations ***Extra Fun!* 4.5.1**

We will use the atomic-primops library here instead of the GHC provided primitives.

```bash
# Assuming you use cabal for haskell packages
cabal install atomic-primops

# If building which ghc (rather than using it as a dependency of a cabal project)
ghc Program.hs -package atomic-primops
```

```haskell
data AtomicCounter
type CTicket = Int

-- we can destructure tickets
peekCTicket :: CTicket -> Int

-- Create a new counter
newCounter :: Int -> IO AtomicCounter

-- atomic operations
casCounter :: AtomicCounter -> CTicket -> Int -> IO (Bool, CTicket)
incrCounter :: Int -> AtomicCounter -> IO Int
incrCounter_ :: Int -> AtomicCounter -> IO ()

-- non atomic
readCounter :: AtomicCounter -> IO Int
readCounterForCAS :: AtomicCounter -> IO CTicket
writeCounter :: AtomicCounter -> Int -> IO ()
```

Thread-Safe Increment **Example Question 4.5.1**

Develop a program to run many threads, each atomically incrementing a counter. Use IORefs but do not use the atomic primitives library.

```haskell
import Control.Concurrent (MVar, takeMVar, newEmptyMVar, putMVar, forkIO)
import Control.Monad (replicateM, zipWithM_)
import Data.IORef ( IORef, atomicModifyIORef, newIORef, readIORef )

increment :: MVar () -> IORef Integer -> Integer -> IO ()
increment h i t = do
  (oldval, newval) <- atomicModifyIORef i (\v -> (v + 1, (v, v+1)))
  putStrLn $ "Thread " ++ show t ++ ": " ++ show oldval ++ " to " ++ show newval
  putMVar h ()

main :: IO ()
main = do
  mvars   <- replicateM 10 newEmptyMVar
  counter <- newIORef 0
  zipWithM_ (\ x y -> forkIO (increment x counter y)) mvars [1..]
  mapM_ takeMVar mvars
  finalValue <- readIORef counter
  putStrLn $ "Final value is: " ++ show finalValue
```

Chapter 5

Concurrency in Rust

Rust is similar to C++ in many ways:

- Designed for high performance (similar performance to C++)
- Extensive support for systems programming (e.g drivers, operating systems) achieved with `unsafe`
- Minimal runtime (i.e no garbage collection)
- Zero cost abstractions

However it has some fundamental differences

- Programs & Libraries packages as crates (rather than includes & headers) managed by a first-party build system (cargo)
- A *proper* type system supporting algebraic data types, traits (see C++ concepts), which also includes mutability (of values and references)
- Type system manages ownership and borrowing, as well as the lifetime of an objects & references (a similar concept to moves, unique pointers and RAII in C++, but strengthened and a first-class part of the language). This ensures that well-typed safe rust code cannot contain data races.
- Powerful type inference (C++ auto on steroids) enabled by the improved type system
- Pattern matching (very similar to & directly inspired by haskell's `data`)
- No undefined behaviour is possible in safe rust code (with the exception of bugs in the compiler, and `unsafe` code)
- Sanitary & powerful macro system (powerful enough to support entire languages as a DSL)

What's all the hype — ***Extra Fun!* 5.0.1**

Rust has gained a significant following despite being a relatively new language (Rust 1.0 release on May 1th, 2015).

- It is being introduced into the Linux kernel as a second language to C.
- As of 2022 it has been the most loved language in the stack-overflow survey for 6 years in a row, and the 5th most wanted language
- It has been embraced by major tech companies (e.g Rust has been used in parts of android since 2019, has become a particular favourite of microsoft and has been introduced to windows & azure)

This is despite the steep leaning curve, new compiler (lacks the decades of experience with gcc, clang) and lack of teaching (few computer science degrees teach rust).

This popularity is mainly a result of deliberate language & compiler design choices to position rust as a better alternative to C++ by systematically removing the most frustrating parts of C++ (undefined behaviour, complex build systems, legacy features, unsanitary macros, memory bugs, poor error message quality in template-heavy code) and to incorporate well-tested & popular concepts from other, more established languages (smart pointers from C++, algebraic data types and pattern matching from functional languages (in particular Haskell), generic traits with associated types (haskell type classes), etc).

5.1 Learning Rust

These notes only cover critical concepts for a basic understanding of what concurrency looks like in Rust.

If you are looking for a more in depth understanding, the following resources are recommended:

- **Rust Book**
- **Rust Reference**
- **Rustlings Problems**
- **Standard Library Documentation**
- **The Rustonomicon**

5.2 Ownership in Rust

In rust ownership rules are enforced by the type system:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.
- You can have many immutable references (readers) to a value, or one mutable reference (writer).

```rust
fn my_function() {
    // a owns the string object
    let a = String::from("This is my string");
    // c owns the string object
    let c = String::from("Keep me!");
    { // start of a new scope
        // a transfers ownership to b
        let b = a;
        // a is no longer valid - it owns no data
        // b's scope ends here, so the string is now dropped
    }
    // end of the function scope, c is dropped
}
```

The transfer of ownership occurs when we assign a value, or use it in a function call. For example:

```rust
fn show(a: String) {
    // a now owns the string
    // a gives a reference of the string to println to allow it to print
    println!("The string is: {}", a)
    // a goes out of scope, and the string is dropped
}

fn print_my_string() {
    let my_string = String::from("This is my string");
    show(a);
    // we cannot use a anymore - the value it owned has been given to show
    // no drop occurs here, as no data is owned by a

}
```

We want to be able to pass values to functions & data structures, without the value being consumed / ownership taken. To do this we use immutable and mutable references.

```rust
fn bar(a: &i32) -> &str {
    match a {
        0 => "none",
        1 => "single",
        2 => "double",
```

```
        _ => "more"
    }
}

fn zig(n: &mut i32) {
    *n += 1
    // n goes out of scope, but nothing is dropped
}

fn foo() {
    let mut a = 3; // we borrow mutably later so must be mut
    {
        let b = &a;         // b borrows a reference to a
        let c = bar(b);     // c takes b (a reference) and
        // b goes out of scope, c goes out of scope and the string is dropped
    }

    zig(&mut a); // a is incremented
    // a goes out of scope
}
```

5.3 Lifetimes

References are associated with a lifetime - the scope within which the reference is valid.

- The compiler can check if the reference outlives the value at compiler time (this would represent a dangling pointer & use after free bug in C/C++)
- Lifetime elison is used to infer the lifetimes of references (so we do not need to explicitly add them for all references), however this is not perfect, and often explicit lifetimes must be added (e.g in data structures)

```
fn bing() { // Scope a starts here
    let mut num_1: Option<&i32> = None;

    { // Scope b starts here
        let num_2 = 7;
        num_1 = Some(&num_2);

        // num_2 is dropped here
    }
    // num_1 no longer valid - it contains a reference to num_2 but outlives num_2
}
```

We can also see this in reference to data structures

```
#[derive(Debug)]
struct PersonID<'a> {
    name: &'a str,
    id_ref: &'a str,
    age: u8
}

fn zapp<'a>() { // we explicitly use lifetime 'a here - elision would otherwise infer the lifetime of the
    let mut bob : PersonID<'a> = PersonID {name: "bob", id_ref: "120dd", age: 99};

    {
        let new_name = String::from("jimmy");
        bob.name = &new_name; // error! the id_ref has a longer lifetime than the new_name
    }
}
```

5.4 Closures

A closure is a anonymous function that can capture variables from their environment.

```
// explicitly specify types
|arg1: SomeType, arg2: OtherType| -> AnotherType { code }

// let parameter types be inferred
|arg1, arg2| { code }

// if the return type is inferred we can potentially remove the braces
|arg1, arg2| expression
```

For example we can capture a mutable reference to a vector in a closure:

```
fn zing() {
    let mut nums = vec![1,2,3,4];

    // add is mutable - each call changes nums, it captures a mutable reference to nums
    let mut add = |n| nums.push(n);

    // we cannot access nums here, a mutable reference is already in add

    for x in 0..=10 {
        add(x)
    }

    // we can now use a mutable reference to nums, add is not used after / can be considered dropped
    nums.push(3);
}
```

- In many languages, e.g lambda expressions in C++
- Closure takes some parameters, returns some value.
- When a variable from the environment (e.g outside the scope of the closure) is used, Rust infers if it is borrowed, mutable borrowed or moved based on the operations used (e.g types of functions the captured variable is passed to)

We can force captured variables to be moved/ownership to be transferred by using a move closure.

```
fn zang() {
    let mut nums = vec![1,2,3,4];

    // add is mutable - each call changes nums, it captures a mutable reference to nums
    let mut add = move |n| nums.push(n);
    // we cannot use nums from here onwards - it has been moved into add

    for x in 0..=10 {
        add(x)
    }
}
```

5.5 Threads

std::thread ***Extra Fun!* 5.5.1**

The standard library's thread documentation covers this section far more extensively.

```
use std::thread;

fn main() {
    let handle = thread::spawn(move || {
```

```
        // some work here
        // return something of type T
    });

    // Result<T, _> we can get the result of the thread back, or an error (e.g thread was killed).
    let result = handle.join();
}
```

As a thread may outlive the scope in which it is spawned, data used must either be `&'static`, heap allocated & shared with an atomic reference counting smart pointer, or be moved into the closure.

5.5.1 Reference Counting

A reference counting smart pointer is available in rust as `Rc`.

- `Rc` is cloned to create new pointers to the same object.
- Can return a reference (immutable) to the contained object. The interior mutability pattern can then be used for shared mutable objects.
- `Rc` is not thread safe, so an `Arc` (atomic reference counter) is used in the context of multithreading.
- We can enclose data in a structure that enforces the ownership rules (such as a read write lock, or mutex).

```
use std::{thread, sync::{Arc, Mutex}};

fn main() {
    let counter = Arc::new(Mutex::new(0));

    (1..10)
        .map(|_| counter.clone())
        .map(|i: Arc<Mutex<i32>>| thread::spawn(move || *i.lock().unwrap() += 1))
        .map(|t| t.join().unwrap())
        .for_each(drop);

    println!("Final value: {}", *counter.lock().unwrap())
}
```

UNFINISHED!!!

Chapter 6

Dynamic Data Race Detection

Thread Sanitizer / TSan — **Definition 6.0.1**

A C++ development tool for detection of data races,

- Compile with `-fsanitize=thread`
- Replaces basic thread, synchronisation and atomic operations into calls to a library that tracks and analyses the program.
- Library updates a dynamic data race detection state.
- All library functions must be mutually exclusive, which is expensive.
- Significant runtime overhead is incurred.
- Proven to always detect a data race if one exists

—Operations considered are:

- Thread launch/join
- Mutex acquire/release
- Read/write from/to potentially-shared address (compiler cannot prove it is only accessed by one thread)
- Atomic operations

6.1 Vector Clocks

6.1.1 Notation

$$Threads \triangleq \{0, 1, \dots, N-1\}$$

N threads tracked by the algorithm, with id starting at 0

$$Locks$$

The set of locks.

$$Locations$$

The set of potentially shared memory locations.

A *logical clock* is a non negative integer.

- A thread's *logical clock* is incremented when a mutex is released (transitioning to the next stage of the program)

$$VC \triangleq (c_0, c_1, \dots, c_{N-1})$$

A *vector clock* is a tuple of N clocks (clock per thread)

- Can also be expressed as a mapping from $ThreadId \to Clock$
- For vector clock V, thread t we have $V(t) =$ logical clock of t.
- VC is the set of all vector clocks

Bottom Vector Clock	$\bot = (0, 0, \ldots, 0)$
Partial Order on Vector Clocks	$V_1 \sqsubseteq V_2 \Leftrightarrow \forall t.\ [V_1(t) \leq V_2(t)]$ (pointwise $\leq$)
Join of vector clocks	$V_1 \sqcup V_2 = (c_0, c_1, \ldots, c_{n-1})$ where $\forall t.\ [V_1 \sqcup V_2(t) = \begin{cases} V_1(t) & V_1(t) > V_2(t) \\ V_2(t) & otherwise \end{cases}]$ (pointwise maximum)
Vector Clock Increment	$inc_t(V) = V[t \mapsto V(t) + 1]$

6.1.2 Vector Clock Algorithm State

$$(\underset{Threads}{C}, \underset{Locks}{L}, \underset{LocationReads}{R}, \underset{LocationWrites}{W})$$

$C : Thread \to VC$

- Each thread has a vector clock
- $C_t \triangleq C(t)$ which represents what thread t knows about the logical clocks of all threads
- $C_t(t)$ is the logical clock of t (always positive, and a source of truth for t's clock)
- For $u \neq t$, $C_t(u) = z$ means thread t knows thread u's logical clock is at least z
- When t acquires a lock, it gets information about the logical clocks of other threads who previously held the lock

$L : Locks \to VC$

- L_m represents the logical clock each thread had the last time it released lock m
- $L_m(t) = 0$ means t has never released m
- $L_m(t) = z \neq 0$ means z was thread t's logical clock the last time it released lock m

$R : Locations \to VC$

- R_x is the logical clock of each thread the last time it read location x
- $R_x(t) = 0$ means t has never read location x
- $R_x(t) = zzneq0$ means z is the was the logical clock of thread t last time it read location x

$W : Locations \to VC$

- W_x is the logical clock of each thread the last time it wrote to location x
- $W_x(t) = 0$ means t has never written to location x
- $W_x(t) = z \neq 0$ means z is the was the logical clock of thread t last time it wrote to location x

Initial State

$$(C, L, R, W) \text{ where}$$

$$\begin{aligned} C &= (inc_0(\bot), inc_1(\bot), \ldots, inc_{N-1}(\bot)) \\ L &= \lambda m.\bot \\ R &= \lambda x.\bot \\ W &= \lambda x.\bot \end{aligned}$$

6.1.3 Intercepted Operations

$rd(t, x)$	Thread t reads location x.
$wr(t, x)$	Thread t writes to location x.
$acq(t, m)$	Thread t acquires lock m
$rel(t, m)$	Thread t releases lock m

Joining threads — ***Extra Fun!* 6.1.1**

Here we do not cover the semantics for creating and joining threads. This is covered in the fastrack paper as $fork(t, u)$ and $join(t, u)$.

Each operation can be considered a transition in a inference system.

6.1.4 Inference Rules

$$\text{(Acquire)}\frac{C' = C[t \mapsto (C_t \sqcup L_m)]}{(C, L, R, W) \overset{acq(t,m)}{\rightarrow} (C', L, R, W)}$$

$$\text{(Release)}\frac{L' = L[m \mapsto C_t] \quad C' = C[t \mapsto inc_t(C_t)]}{(C, L, R, W) \overset{rel(t,m)}{\rightarrow} (C', L', R, W)}$$

$$\text{(Read)}\frac{W_x \sqsubseteq C_t \quad R' = R[x \mapsto R_x[t \mapsto C_t(t)]]}{(C, L, R, W) \overset{rd(t,x)}{\rightarrow} (C, L, R', W)}$$

$$\text{(Write)}\frac{W_x \sqsubseteq C_t \quad R_x \sqsubseteq C_t \quad W' = W[x \mapsto W_x[t \mapsto C_t(t)]]}{(C, L, R, W) \overset{wr(t,x)}{\rightarrow} (C, L, R, W')}$$

We can then include a rule to detect a race condition

$$\text{(Write-Read Race)}\frac{\exists u.\ [W_x(u) > C_t(u)]}{(C, L, R, W) \overset{rd(t,x)}{\rightarrow} \textbf{WriteReadRace}(u, t, x)}$$

$$\text{(Write-Write Race)}\frac{\exists u.\ [W_x(u) > C_t(u)]}{(C, L, R, W) \overset{wr(t,x)}{\rightarrow} \textbf{WriteWriteRace}(u, t, x)}$$

$$\text{(Read-Write Race)}\frac{\exists u.\ [R_x(u) > C_t(u)]}{(C, L, R, W) \overset{wr(t,x)}{\rightarrow} \textbf{ReadWriteRace}(u, t, x)}$$

UNFINISHED!!!

Chapter 7

Operational Semantics

7.1 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 7.1.1**

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 7.1.2**

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 7.1.3**

A model of computation that describes what is computable.

- Efficiency mostly irrelevant
- Only covers sequential computation.

Shared-Memory Computability — **Definition 7.1.4**

A model for concurrent computation.

- Describes what is concurrently computable.
- Efficiency mostly irrelevant

7.2 Shared-Memory Concurrency

7.2.1 Read-Modify-Write

Read-Modify-Write Instructions — **Definition 7.2.1**

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 7.2.2**

Allows for synchronisation between two threads.

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

Strong RMW — **Definition 7.2.3**

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.

7.2.2 Consistency/Memory Models

Sequential Consistency — **Definition 7.2.4**

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

7.3 Sequential Consistency

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

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

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

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

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

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

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

Racey Increment **Example Question 7.3.1**

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

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

We can also express this as:

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

7.3.1 Configurations

$$\text{Shared memory } M \in \mathit{Mem} \triangleq \mathit{Loc} \to \mathit{Val}$$
$$\text{Thread-local Registers } s \in \mathit{Store} \triangleq \mathit{Reg} \to \mathit{Val}$$
$$\text{A store map for threads } S \in \mathit{SMap} \triangleq \mathit{Tid} \to \mathit{Store} \text{ where } S(\tau) = s$$

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

$$(P, S, M)$$

7.3.2 Transitions

The operational semantics are split into two types of transition.

Program Transitions	A step in 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: skip ; $C \rightarrow C$
	$\mid (R, x, v)$	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 $v_0 \rightarrow v_n$ (FFA or successful CAS)
	$\mid (U, x, v_0, \bot)$	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 7.3.1**

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:

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

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

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

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

We must also consider the basic read/write transitions:

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

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.

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

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

$$\frac{\text{eval}(s, E_0) = v_0 \quad v \neq v_0 \quad s' = s[a \mapsto 0]}{a := \text{CAS}(x, E_0, E_n), s \xrightarrow{(U,x,v,\bot)}_c \text{skip}, s'}$$

7.3.3 Concurrent Program Transitions

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

7.3.4 Storage Transitions

A storage transition is of the form $M \xrightarrow{\tau:l}_m M'$ (thread τ updates $M \to M'$ 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).

$$\frac{M(x) = v}{M \xrightarrow{\tau:(R,x,v)}_m M} \text{ Memory Read}$$

$$\frac{M' = M[x \mapsto v]}{M \xrightarrow{\tau:(W x,v)}_m M'} \text{ Memory Write}$$

$$\frac{M(x) = v_0 \quad M' = M[x \mapsto v_n]}{M \xrightarrow{\tau:(U,x,v_0,v_n)}_m M'} \text{ Successful CAS or FFA}$$

$$\frac{M(x) = v}{M \xrightarrow{\tau:(U,x,v,\bot)}_m M} \text{ Failed CAS}$$

7.3.5 Combining Operational Semantics

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

For example with

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

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 7.3.2**

Combine the memory and program transitions for skip.

The program transition can be expressed as:

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

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

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

Read and Assign **Example Question 7.3.3**

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

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

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

Hence we can now include the storage transition:

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

7.3.6 Traces

$\to^*$ for SC — **Definition 7.3.2**

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

The reflexive, transitive closure of $\to$

- *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:

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

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

7.3.7 Properties of Sequential Consistency

Determinism

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

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

It has been determined... — **Example Question 7.3.4**

Provide a counter example to SC being deterministic and confluent.

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

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

Confluence

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

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

7.4 Total Store Ordering

Weak Memory Models (WMM) — **Definition 7.4.1**

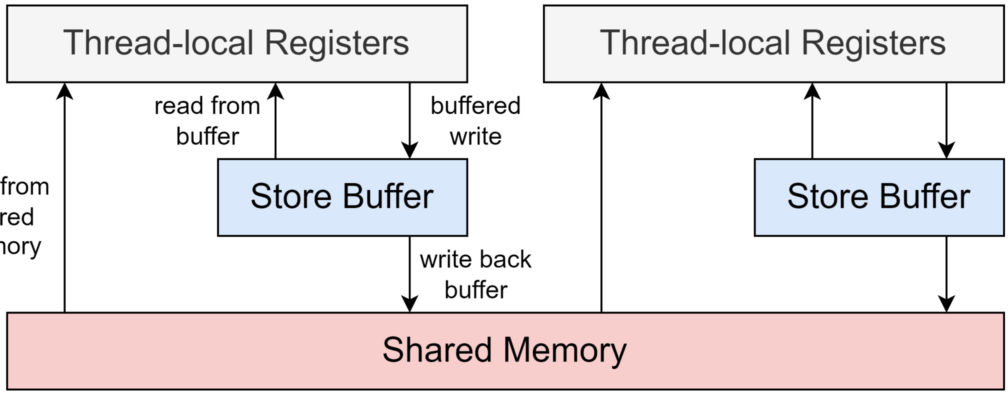
Allow for instructions in a thread to be reordered (e.g dynamically scheduled processors).

- *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 7.4.2**

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.

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

$$\frac{}{\text{mfence}, s \overset{MF}{\rightarrow}_c \text{skip}, s}$$

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.

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

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

Hence a TSO configuration is:

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

7.4.1 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:

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

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

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

Memory Write

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

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.

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

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

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

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

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.

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

sFence — **Example Question 7.4.1**

The x86-64 instruction set includes an sfence instruction. This is similar to mfence, however allows for read reordering.

Original		≈	sfence-read-reordering		≈	TSO allowed reorder	
$x := 1$	$y := 1$		$x := 1$	$y := 1$		$a := y$	$b := x$
sfence	sfence	≈	$a := y$	$b := x$	≈	$x := 1$	$y := 1$
$a := y$	$b := x$		sfence	sfence		sfence	sfence

Adapt the rules of TSO to implement sfence.

---

sfence make no changes to registers/local store, but requires an SF memory transition:

$$\frac{}{\text{sfence}, s \xrightarrow{SF}_c \text{skip}, s}$$

In order to implement, we will add sfences to the store buffer, hence we must redefine the buffer:

$$b \in Buff \triangleq Seq\langle WLab \cup \{SF\}\rangle$$

- sfence adds SF to the store buffer.
- When executing an sfence transition, we do not need to check if the buffer is empty.
- Since sfence and write instructions are still ordered with respect to eachother, the buffers are still a FIFO queue.

Hence:

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

We can consider an sfence as a kind of dummy write, much like normal writes other writes cannot be reordered, but reads can. However unlike a write it has no effect on memory.

7.4.2 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.

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

7.4.3 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 8

Declarative Semantics

Declarative/Axiomatic Semantics — **Definition 8.0.1**

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 8.0.2**

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 8.0.3**

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

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

Non-empty Label — **Definition 8.0.4**

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 $x \in Loc$ and $v_r, v_w \in Val$

8.1 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 $r_x \triangleq r \cap (A_x \times A_x)$
	$r\vert_{loc}$	$\triangleq$	$\{(a,b) \in r \mid \text{loc}(a) = \text{loc}(b)\}$

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	$\triangleq$	$\{(a,b) \in r \mid \text{tid}(a) \neq \text{tid}(b)\}$

$$irreflexive(r) \overset{def}{\Leftrightarrow} \neg\exists a.[(a,a) \in r]$$

$$acyclic(r) \overset{def}{\Leftrightarrow} irreflexive(r^+)$$

Relational Composition **Definition 8.1.1**

Given some relations $R \in X \times Y$ and $S \in Y \times Z$:

$$R; S \triangleq \{(x,z) \in X \times Z \mid \exists y \in Y.[(x,y) \in R \wedge (y,z) \in S]\}$$

$$injective(R;S) \Rightarrow injective(R) \qquad injective(R) \wedge injective(S) \Rightarrow injective(R;S)$$
$$surjective(R;S) \Rightarrow surjective(R) \qquad surjective(R) \wedge surjective(S) \Rightarrow surjective(R;S)$$
$$R;(S;T) = (R;s);T \qquad (R;S)^T = S^T;R^T$$

Partial Orders **Definition 8.1.2**

Given some set A and relation $R \subseteq (A \times A)$:

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 8.1.3**

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

Injective/one-to-one	Every element in input is mapped to a unique element in output.	$\forall x_1, x_2 \in A.[f(x_1) = f(x_2) \Rightarrow x_1 = x_2]$
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 8.1.4**

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

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

po is is such that:

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

Each po_τ is a strict total order on E_τ.

rf is such that $\forall \langle w, r \rangle \in \text{rf}$, (maps a write event, to an event that reads from it).

$$w \neq r$$
$$\text{typ}(w) \in \{W, U\} \wedge \text{typ}(r) \in \{R, U\}$$
$$\text{loc}(w) = \text{loc}(r)$$
$$\text{val}_w(w) = \text{val}_r(r)$$

rf^{-1} is a function (if $\langle w_1, r \rangle, \langle w_2, r \rangle \in \text{rf}$ then $w_1 = w_2$)

The notation for a graph G is:

$$\begin{aligned}
G &= \langle E, \text{po}, \text{rf} \rangle \\
G.E &\triangleq E \\
G.\text{po} &\triangleq \text{po} \\
G.R &\triangleq \{r \in E \mid \text{typ}(r) = R\} \\
G.W &\triangleq \{w \in E \mid \text{typ}(w) = W\} \\
G.U &\triangleq \{u \in E \mid \text{typ}(u) = U\} \\
G.RU &\triangleq G.R \cup G.U \\
G.WU &\triangleq G.W \cup G.U \\
G.R_x &\triangleq G.R \cap \{r \in E \mid \text{loc}(r) = x\} \\
G.W_x &\triangleq G.W \cap \{w \in E \mid \text{loc}(w) = x\}
\end{aligned}$$

8.2 Consistency Predicates

A program is mapped to a set of candidate executions. A consistency predicate filters these candidates.

- The semantics of a program are the consistent executions of a program.
- Consistency predicates include sequential consistency and total store ordering.

Completeness — **Definition 8.2.1**

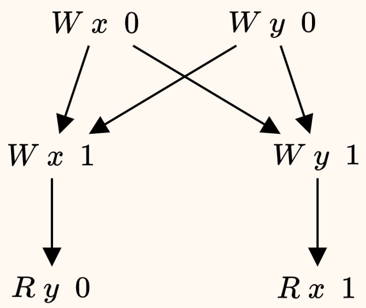
A graph G is complete if:

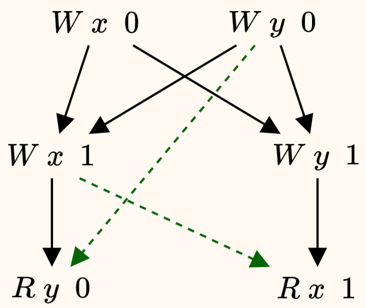
$$rng(G.\text{rf}) = G.RU$$

Every read/update reads from some write/update.

Complete Graphs **Example Question 8.2.1**

Define $G.\mathrm{rf}$ to complete the following graph:





$$G.UW = \{W\ x\ \ 0, W\ y\ \ 0, W\ x\ \ 1, W\ y\ \ 1\}$$
$$G.RU = \{R\ y\ \ 0, R\ x\ \ 1\}$$

The location and value must match.

$$G.\mathrm{rf} = \{\langle W\ y\ \ 0, R\ y\ \ 0\rangle, \langle W\ x\ \ 1, R\ x\ \ 1\rangle\}$$

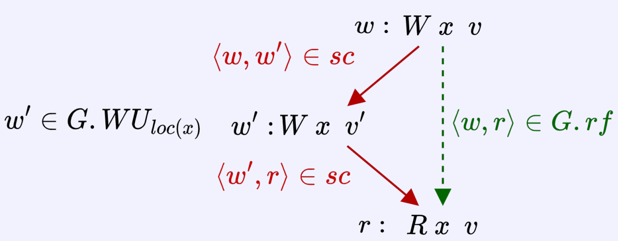
8.3 Sequential Consistency

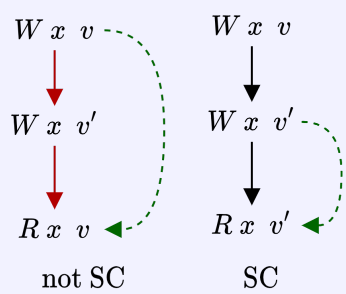
Sequential Consistency (Lamport SC) **Definition 8.3.1**

sc is a strict total order on $G.E$. G is *SC-consistent* if the following hold:

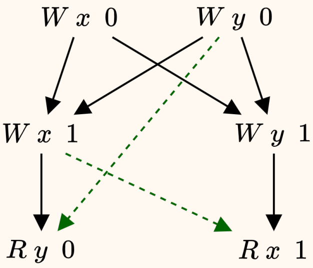
	$\langle a, b\rangle \in G.\mathrm{po} \Rightarrow \langle a, b\rangle \in \mathrm{sc}$	i.e $G.\mathrm{po} \subseteq \mathrm{sc}$
$\wedge$	$\langle w, r\rangle \in G.\mathrm{rf} \Rightarrow \langle w, r\rangle \in \mathrm{sc}$	i.e $G.\mathrm{rf} \subseteq \mathrm{sc}$
$\wedge$	$\langle w, r\rangle \in G.\mathrm{rf} \Rightarrow \neg\exists w' \in G.WU_{\mathrm{loc}(r)}.[\langle w, w'\rangle \in \mathrm{sc} \wedge \langle w', r\rangle \in \mathrm{sc}]$	i.e There is no w' between w and r

- G must be complete
- G is SC-consistent with respect to some strict total order sc on $G.E$

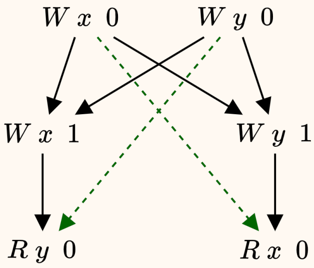




Consistency is key! **Example Question 8.3.1**



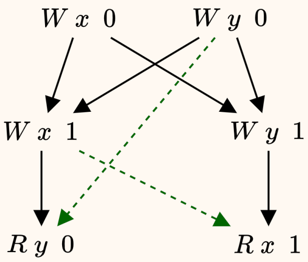
G_1



G_2

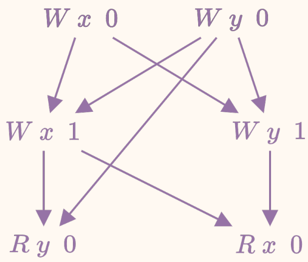
Determine which of these are sequentially consistent using Lamport's definition.

For the graph G_1:



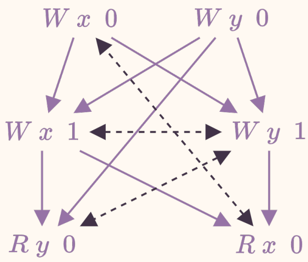
The original graph G

(1)



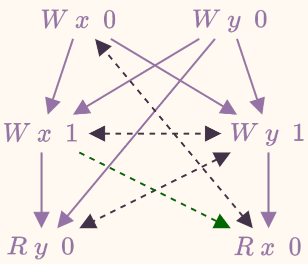
sc must contain all og G.rf and G.po

(2)



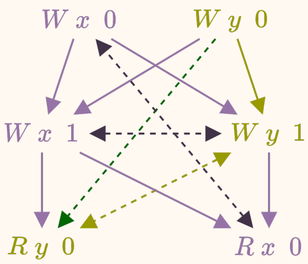
sc is a total order, add in some faux edges to ensure all connected (we will sort direction later)

(3)



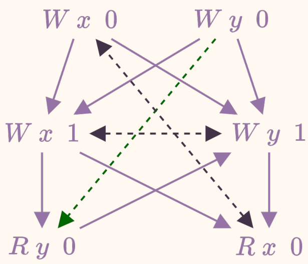
We select an rf edge, we cannot find a write to make this read from non-SC

(4)



We move to the other rf edge, and find a potential issue, we need R y 0 to occur before W y 1

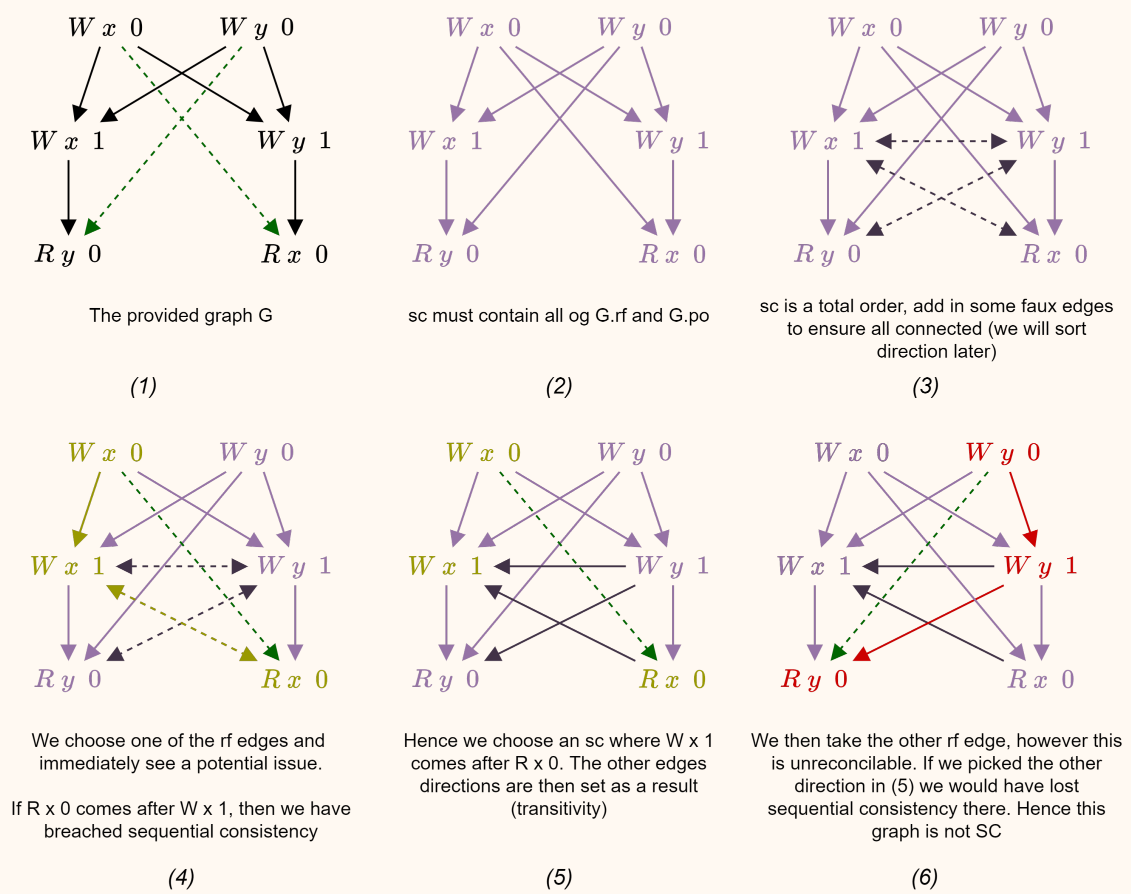
(5)



By setting R y 0 to occur before W y 1 we order the other edges by transitivity (sc is a total order)

(6)

Hence G_1 is SC-consistent. **For the graph G_2:**



Hence G_2 is not SC-consistent.

Modification/Coherence Order (Alternative SC) **Definition 8.3.2**

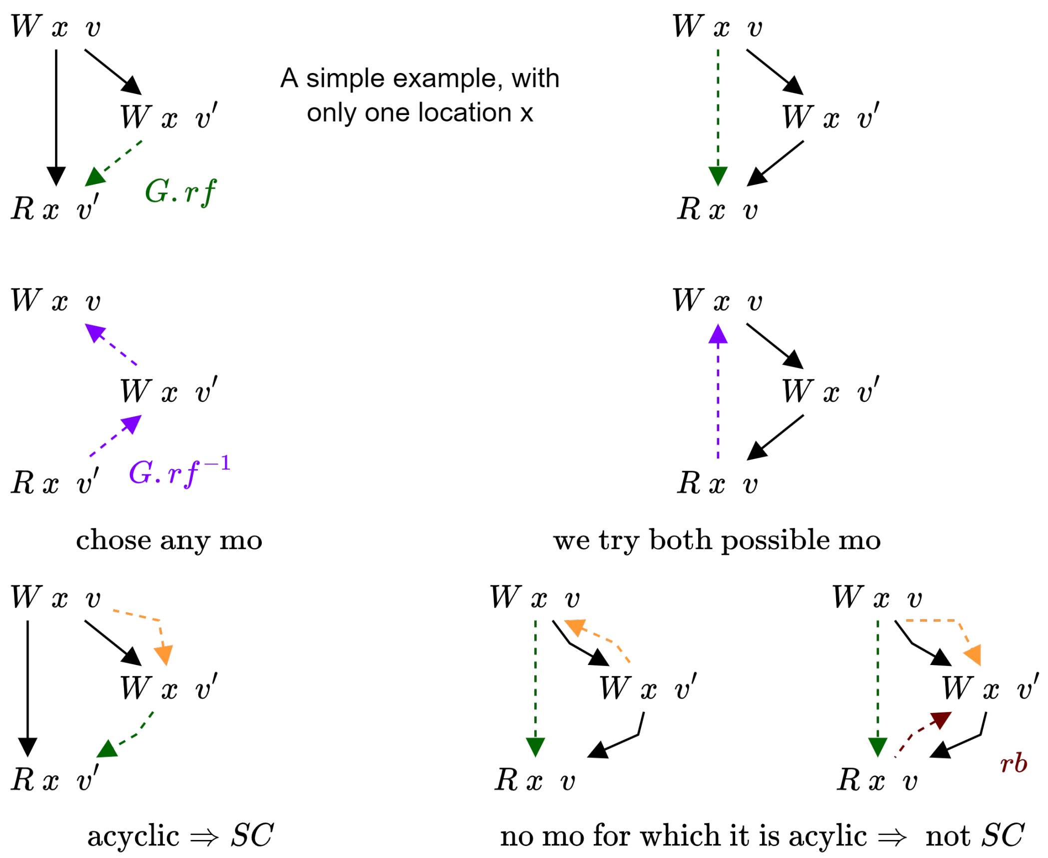
mo is a *modification order* for an execution graph G if:

$$\text{mo} = \bigcup_{x \in Loc} \text{mo}_x \text{ where } \text{mo}_x \text{ is a strict total order on } G.WU_x$$

We can then create an alternative definition for sequential consistency.

$$\begin{aligned} & G \text{ is complete} \\ \land \quad & \exists\ \text{mo for } G.\ [acyclic(G.\text{po} \cup G.\text{rf} \cup \text{mo} \cup \text{rb})] \\ & \text{where rb} \triangleq G.\ \text{rf}^{-1}; \text{mo} \setminus id \end{aligned}$$

- On all architectures there is a strict total order or writes on a given cache line (here we consider location).
- Applies to each memory location.



SC Equivalence — ***Extra Fun!* 8.3.1**

Lamport SC $\Rightarrow$ Alternative SC

1. Take $\mathrm{mo}_x \triangleq [WU_x]; \mathrm{sc}; [WU_x]$
2. The $G.\mathrm{po} \cup G.\mathrm{rf} \cup \mathrm{mo} \cup \mathrm{rb} \subseteq \mathrm{sc}$

Alternative SC $\Rightarrow$ Lamport SC

1. Take sc to be any strict total order extending $G.\mathrm{po} \cup G.\mathrm{rf} \cup \mathrm{mo} \cup \mathrm{rb}$

8.4 Total Store Order

Total Store Order — **Definition 8.4.1**

tso is a strict partial order on $G.E$ where the following holds:

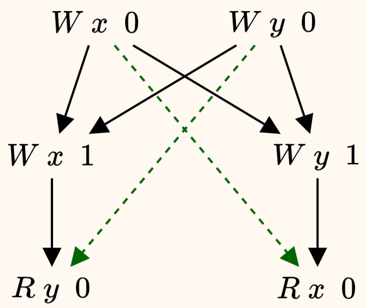
$$\begin{aligned} & \mathrm{tso} \text{ is total on } G.E \setminus G.R \\ \land\ & G.\mathrm{po} \setminus (G.W \times G.R) \subseteq \mathrm{tso} \\ \land\ & G.\mathrm{rf} \subseteq \mathrm{tso} \cup G.\mathrm{po} \\ \land\ & \langle w, r\rangle \in G.\mathrm{rf} \Rightarrow \neg\exists w' \in G.WU_{\mathrm{loc}(r)}.[\langle w, w'\rangle \in \mathrm{tso} \land \langle w', r\rangle \in \mathrm{tso} \cup G.\mathrm{po}] \end{aligned}$$

This implies $G.\mathrm{rfe} \subseteq \mathrm{tso}$

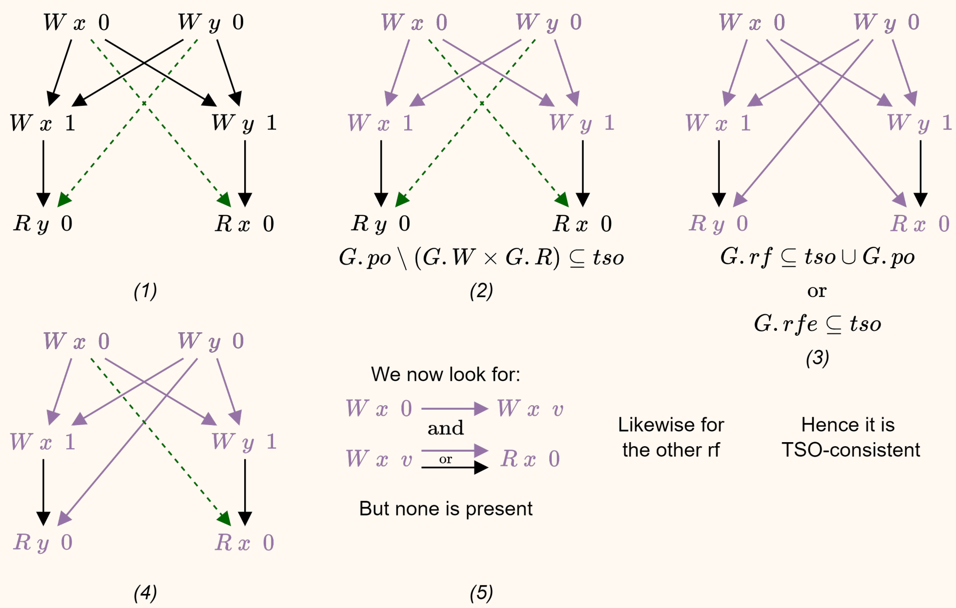
An execution is TSO-consistent if:

- G is complete
- G is TSO consistent with respect to some strict partial order tso on $G.E$
- The writes have a total order, and the program ordering for (write $\to$ write) and (read $\to$ read) are in tso.

TSO Consistency — **Example Question 8.4.1**



Show the graph is TSO-Consistent.



Alternative TSO — **Definition 8.4.2**

An execution graph G is TSO-Consistent if:

$$\begin{array}{ll} & G \text{ is complete} \\ \wedge & \exists \text{ modification order } \mathsf{mo}.[G.\mathit{rfi} \cup G.\mathit{rbi} \subseteq G.\mathsf{po} \wedge \mathit{acyclic}(\mathsf{ppo} \cup G.\mathsf{rfe} \cup \mathsf{mo} \cup \mathsf{rbe})] \end{array}$$

Where $\mathsf{ppo} \triangleq (G.\mathsf{po} \setminus (G.W \times G.R))^+$ (preserved program order) and $\mathsf{rb} \triangleq G.\mathsf{rf}^{-1}; \mathsf{mo} \setminus \mathit{id}$.

8.5 Coherent

Coherence (COH) — **Definition 8.5.1**

Considered *"sc per location"*

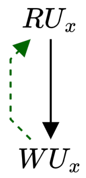
$$\begin{array}{ll} & G \text{ is complete} \\ \wedge & \text{for each location } x \text{ there is a strict total order } \mathsf{sc}_x \text{ such that:} \\ & \quad \langle a, b\rangle \in G.\mathsf{po} \Rightarrow \langle a, b\rangle \in \mathsf{sc}_x \\ \wedge & \quad \langle a, b\rangle \in G.\mathsf{rf}_x \Rightarrow \langle a, b\rangle \in \mathsf{sc}_x \wedge \neg\exists c \in G.WU_x.[\langle a, c\rangle \in \mathsf{sc}_x \wedge \langle c, b\rangle \in \mathsf{sc}_x] \end{array}$$

Coherence (Alternative) — **Definition 8.5.2**

$$\begin{array}{ll} SC: & acyclic(\text{po} \cup \text{rf} \cup \text{mo} \cup \text{rb}) \\ COH: & acyclic(\text{po}|_{loc} \cup \text{rf} \cup \text{mo} \cup \text{rb}) \end{array}$$

8.5.1 Bad Patterns

As coherence is the weakest model, any pattern disallowed under coherence is disallowed under all models.



$r := x$ Read v
$x := v$

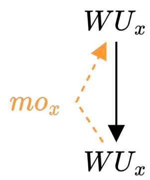
rf; po is irreflexive

No Future Read

$r :=$ CAS(x, v, v) Read v

rf is irreflexive

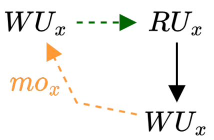
RMW 1 Cannot Read from Self



$x := v$ Write v'
$x := v'$ Write v

mo; po is irreflexive

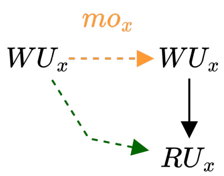
Coherence WW



$x := v$ Write v'	$r := x$ Read v
	$x := v'$ Write v

mo; rf; po is irreflexive

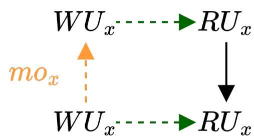
Coherence RW



$x := v$	$x := v'$
	$r := x$ Read v

rf^{-1}; mo; po is irreflexive

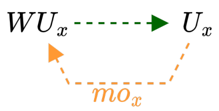
Coherence WR



$x := v$	$r := x$ Read v'
$x := v'$	$r := x$ Read v

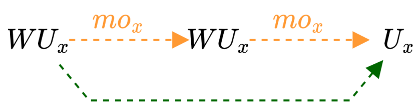
rf^{-1}; mo; rf is irreflexive

Coherence RR



mo; rf is irreflexive

RMW 2



rf^{-1}; mo; mo is irreflexive

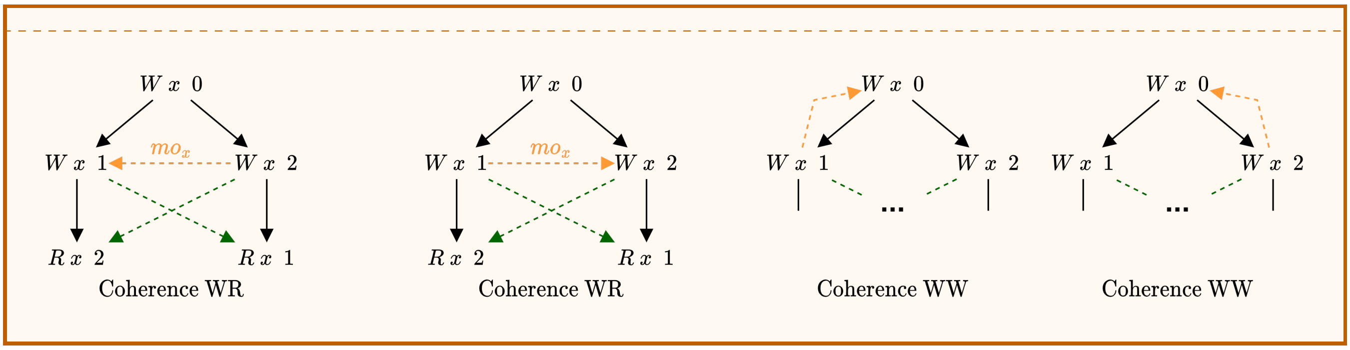
Atomicity

Coherence Test — **Example Question 8.5.1**

Determine if the following is COH-Consistent:

$x := 0$

$x := 1$	$x := 2$
$a := x$ Read 1	$b := x$ Read 2



8.6 Atomicity

COH is often too weak to be useful. As a example if we attempt to implement a lock

$$x := 0$$
$$a := FFA(x, 1) \,\|\, b := FFA(x, 1)$$

Guarantees that $a = 1 \vee b = 1$.

We could attempt to implement spinlocks with a basic CAS

Lock(l)

$a := 0$
while $\neg r$ do $r :=$ CAS$(l, 0, 1)$

Unock(l)

$l := 0$

However this will not provide mutual exclusion as we are relying on the ordering of memory accesses to multiple locations. COH only guarantees sequential consistency per location.

Here we attempt to do basic message passing using the location y. Given initially $x = y = 0$:

$x := 42$	$a := y$
$y := 1$	while $\neg a$ do $a := y$
	$b := x$ Read 0

8.7 Release/Acquire

Release/Acquire Memory Model (RA) **Definition 8.7.1**

Where mo is a modification order on execution graph G, G is RA-Consistent if:

$$G \text{ is complete } \wedge acyclic(\text{hb}_{ra} \cup \text{mo} \cup \text{rb}) \text{ where } \text{hb}_{ra} \triangleq (\text{po} \cup \text{rf})^{+}|_{loc}$$

We can also defined it as if:

$(\text{po} \cup \text{rf})^{+}$	is irreflexive	(No Future Read)
$\text{mo}; (\text{po} \cup \text{rf})^{+}$	is irreflexive	(Coherence WW)
$\text{rf}^{-1}; \text{mo}; (\text{po} \cup \text{rf})^{+}$	is irreflexive	(Coherence WR)
$\text{rf}^{-1}; \text{mo}; \text{mo}$	is irreflexive	(RMW Atomicity)

For comparison:

$$SC \sim acyclic(\text{po} \cup \text{rf} \cup \text{mo} \cup \text{rb})$$
$$RA \sim acyclic((\text{po} \cup \text{rf})|_{loc} \cup \text{mo} \cup \text{rb})$$
$$COH \sim acyclic(\text{po}|_{loc} \cup \text{rf} \cup \text{mo} \cup \text{rb})$$

We can see that the addition of rf to the per location ordering strengthens it over COH.

8.8 Ordering Models

$$COH < RA < TSO < SC$$

We can also express this in terms of the set of consistent executions:

$$execs(SC) \subset execs(TSO) \subset execs(RA) \subset execs(COH)$$

Hence for two memory models $MM_A < MM_B$:

$$G \in execs(MM_B) \Rightarrow G \in execs(MM_A)$$
$$G \notin execs(MM_A) \Rightarrow G \notin execs(MM_B)$$

Chapter 9

Concurrent Objects

9.1 Concurrent Queues

We start with a basic interface defining the behaviour of a queue.

```
interface Queue<T> {
    val count: Int
        get

    val isEmpty: Boolean
        get() = count == 0

    fun enq(item: T): Boolean
    fun deq(): T?
}
```

9.1.1 Circular Queue

```
class CircularQueue<T>(val capacity: Int) : Queue<T> {
    var head: Int = 0  // next pop present here
    var tail: Int = 0  // next item pushed here
    var items = MutableList<T?>(capacity){null}

        override val count: Int
        get() = if  (head >= tail) head - tail else head + (capacity - tail)

    override fun enq(item: T): Boolean {
        if (count < capacity) {
            items[head] = item
            head = (head + 1) % capacity
            return true
        } else {
            return false
        }
    }

    override fun deq(): T? {
        if (isEmpty) {
            return null
        } else {
            val retval: T = items[tail]!!
            tail = (tail + 1) % capacity
            return retval
        }
    }
}
```

9.1.2 Lock Based Queue

Mutual exclusion is used to prevent concurrent modification, and hence make thread safe.

9.1.3 Wait Free 2-Thread Queue

UNFINISHED!!!

9.2 Sequential and Concurrent Objects

Object — **Definition 9.2.1**

Objects are data structures containing internal state (fields/attributes/members) and methods (functions that can be provided input, and access/mutate the object's internal state).

In concurrent programming, the liveness and safety properties of an object must be specified.

- Need to define how to assess the implementation as correct
- Need to define under which conditions progress (e.g no livelock/deadlock) is guaranteed

9.2.1 Sequential Specifications

Precondition	Object state before method call
Postcondition	(Result) Value returned by the method, or exceptions thrown
Postcondition	(State) The state of the object when the method returns

We do not need to consider the state of the object between the rep and post conditions.

- State is meaningful between method calls (postcondition (state) $\rightarrow$ precondition)
- Each method call can be considered a single atomic event
- Methods can be described in isolation
- New methods can be added without changing the descriptions of older methods

9.2.2 Concurrent Specifications

A method call is no longer an atomic event.

- A method call is a sequence of interval events.
- Method calls overlap in time.
- Object may never be *between method calls* as many calls may constantly overlap
- All interactions between concurrent calls must be characterised
- When adding a new method, it must be considered in the context of all existing method definitions
- **everything can interact with everything else**

Linearisable — **Definition 9.2.2**

A set of operations are linearisable if it can be re-expressed as a sequential history.

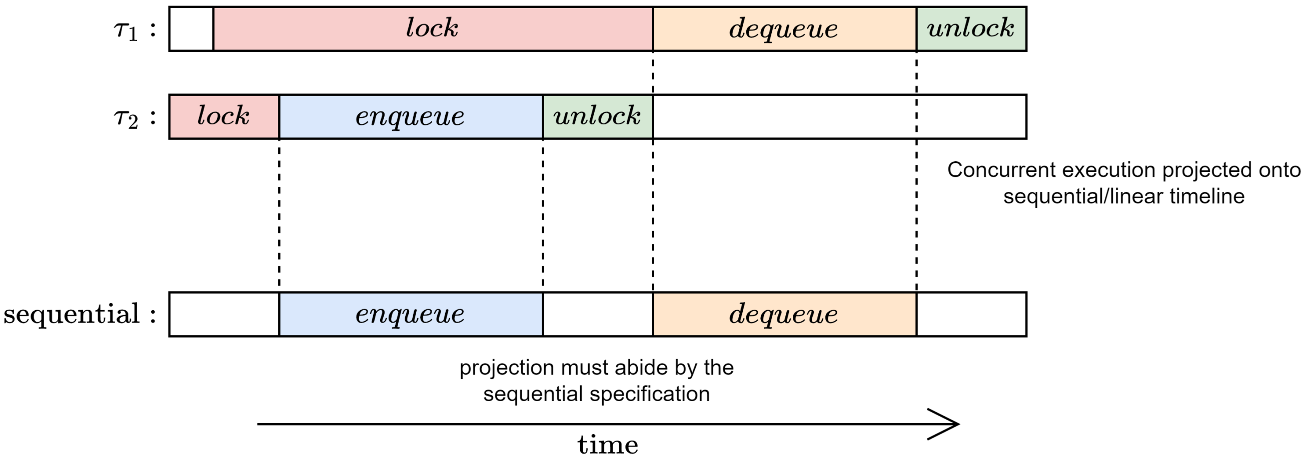
Object Linearisability — **Definition 9.2.3**

An object is linearisable if all its possible executions are linearisable.

Not studied in this course

Execution Linearisability — **Definition 9.2.4**

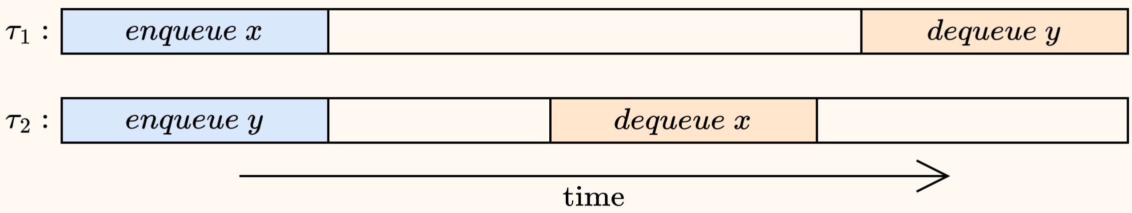
- Method takes effect instantaneously somewhere between the invocation and response events.
- If the *sequential* behaviour is correct then the execution is linearisable.



- To show an execution is linearisable, we find the linearization points (where the method *instantaneously* occurs)
- Typically arrows are used, where a bold arrow shows the time for the execution of some method, and an arrow with a dotted tail is one that never responds/returns (e.g thread killed).
- For invocations that never respond/return, we can decide if the method occurred or not (depending on what *makes sense*/is required to justify the execution).

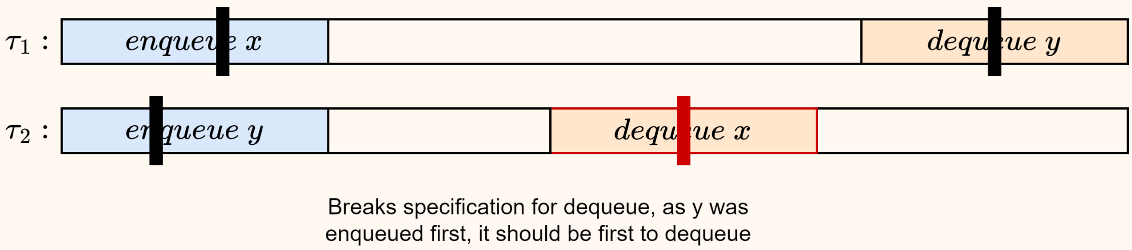
Valid Linearisations — **Example Question 9.2.1**

Determine the valid linearisation for the following execution.

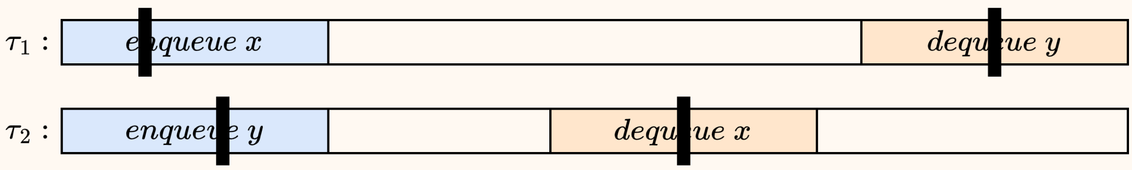


---

invalid linearisation :

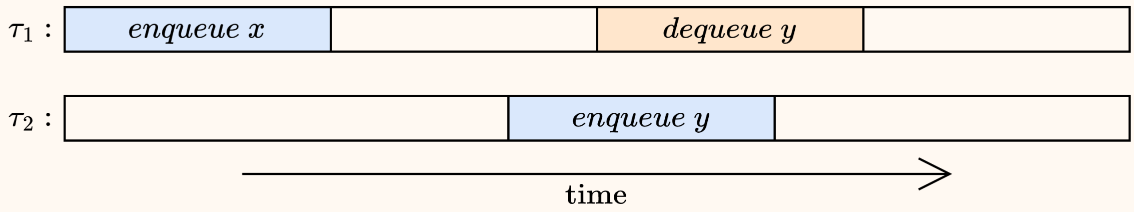


valid linearisation :



Valid Linearisation — **Example Question 9.2.2**

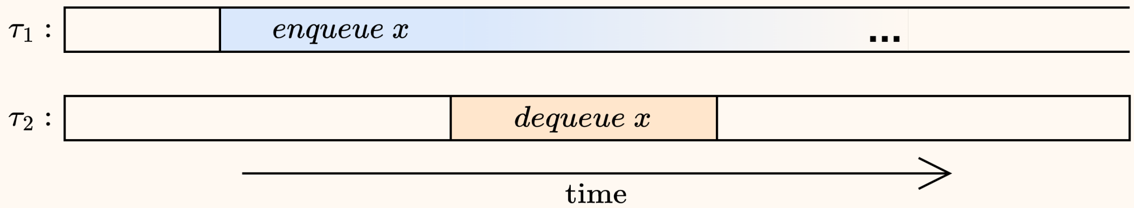
Determine if there is a valid linearisation for the following execution.

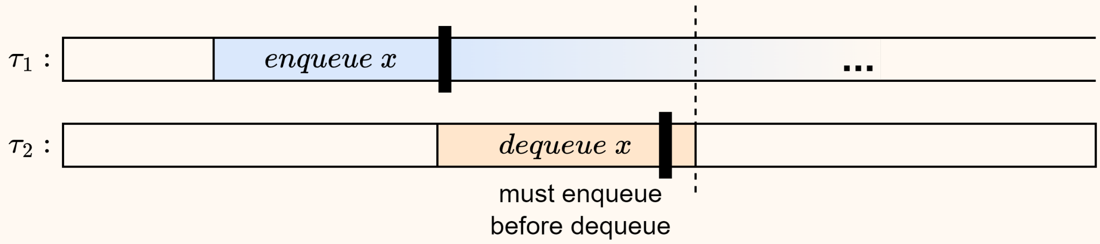


There is none, this is as the entry to *enqueue* x is before the entry to *enqueue* y, hence in any linearisation *enqueue* x comes first. This means that the *dequeue* y breaches the specification of dequeue.

No Return **Example Question 9.2.3**

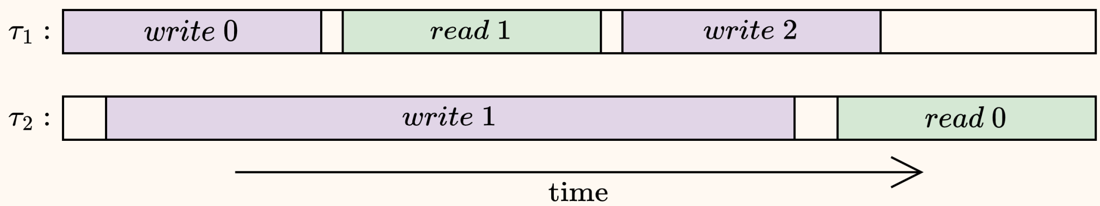
Is the following linearisable?

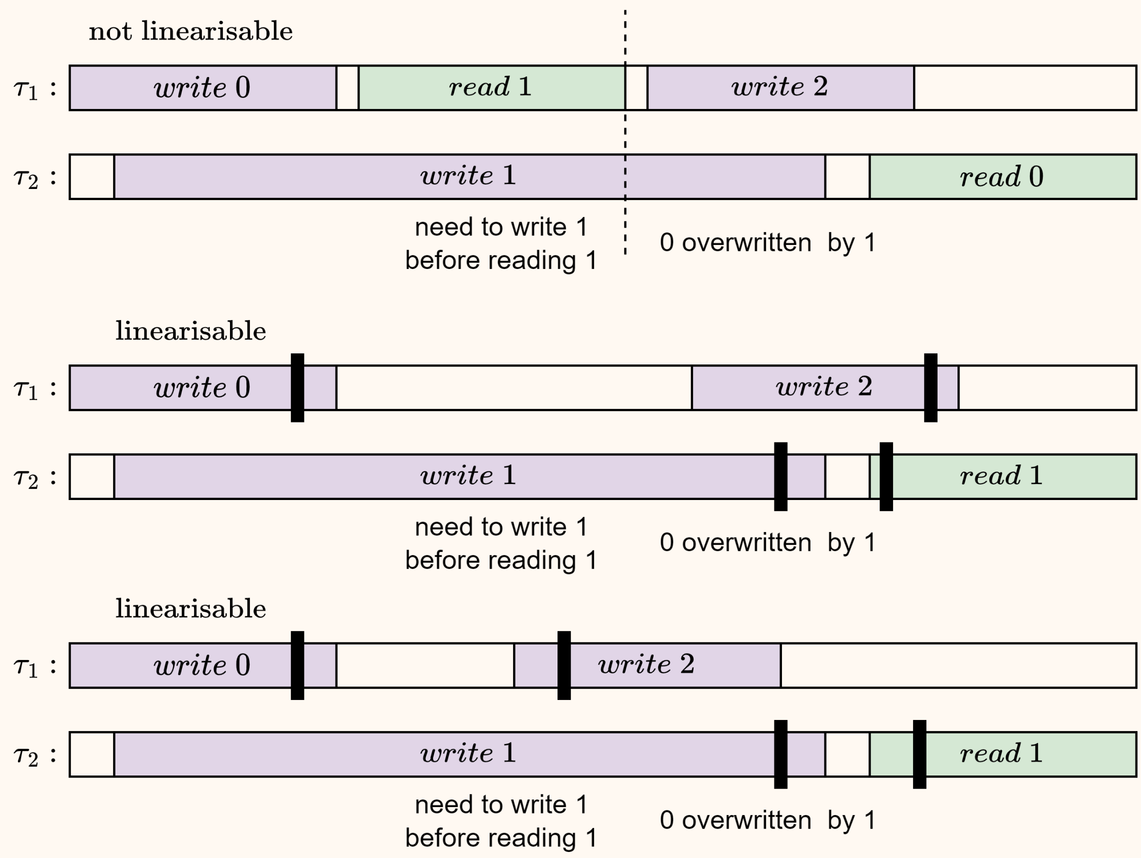




Register Writing **Example Question 9.2.4**

Is the following execution linearisable, and if not then provide modified examples of how it could be linearisable.





9.3 Formal Model of Executions

Invocation and Response Notation **Definition 9.3.1**

Invocation	Response
$Thread \quad object\ .\ method(args\ldots)$	$Thread \quad object\ :\ return$

- Method name is implicit for response (a thread can only execute one method at once, so response directly follow's that thread's last invocation)
- An invocation is pending if there is no matching response

Histories — **Definition 9.3.2**

A history H is a sequence of invocations and responses. For example:

$$H = \begin{array}{ll} A & q.enq(7) \\ B & p.enq(6) \\ B & p : void \\ B & q.deq() \\ A & q : void \\ B & q : 7 \end{array}$$

$$H|A = \begin{array}{ll} A & q.enq(7) \\ A & q : void \end{array} \quad H|B = \begin{array}{ll} B & p.enq(6) \\ B & p : void \\ B & q.deq() \\ B & q : 7 \end{array} \quad H|q = \begin{array}{ll} A & q.enq(7) \\ B & q.deq() \\ A & q : void \\ B & q : 7 \end{array} \quad H|p = \begin{array}{ll} B & p.enq(6) \\ B & p : void \end{array}$$

- A history is *well formed* if the per-thread projections are sequential.
- A history is *sequential* if every invocations is immediately followed by its corresponding response.
- Histories are *equivalent* if the per-thread projections are equivalent.
- A history is *legal* if for every object x, $H|x$ is in the sequential spec for x.

Equivalent Histories — **Example Question 9.3.1**

Are H and G equivalent?

$$H = \begin{array}{ll} A & q.enq(7) \\ B & p.enq(6) \\ B & p : void \\ B & q.deq() \\ A & q : void \\ B & q : 7 \end{array} \text{ and } G = \begin{array}{ll} B & p.enq(6) \\ B & p : void \\ A & q.enq(7) \\ B & q.deq() \\ B & q : 7 \\ A & q : void \end{array}$$

Yes, as $H|A = G|A$ and $H|B = G|B$.

Precedence — **Definition 9.3.3**

A method call x precedes another y if the response of x is before the invocation of y. This is written as:

$$\text{Given } H \quad m_0 \rightarrow_H m_1 \text{ if } m_0 \text{ precedes } m_1$$

- Precedence is a partial order (some methods overlap)
- For a fully sequential history, it is a total order.

9.4 Linearisability

Linearisability Formally — **Definition 9.4.1**

A history H is linearisable if it can be extended to G by:

- Appending zero or more responses to pending invocations.
- Discarding pending invocations.

Where G will be equivalent to the *legal sequential* history S and $(\rightarrow_G \subseteq \rightarrow_S)$

Composability Theorem | **Definition 9.4.2**

$$linearisable(H) \Leftrightarrow \forall x.\ [linearisable(H|x)]$$

A history H is linearisable if and only if for every object x, $H|x$ is linearisable.

- This allows for the linearisability of objects to be considered independently, and then composed.

To show a history H is linearisable:

1. We can split the history using the composability theorem.
2. Make the H history complete (no pending invocations) (history H')
3. Create a sequential history S that is legal (within sequential specification)
4. Show H' is equivalent to S using projections.
5. Get the strict total order $\rightarrow_S$ and show $\rightarrow_H$ is a subset of $\rightarrow_S$

9.5 Sequential Consistency

A history H is Sequentially Consistent if it can be extended to G by:

- Appending zero or more responses to pending invocations.
- Discarding pending invocations.

Where G will be equivalent to the *legal sequential* history S. Unlike with linearisability, G does not have to be a subset of S.

Chapter 10

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.