1 C++

- Data races occur when distinct threads access a memory location, at least one modifies the location, at least one is a non-atomic access, accesses not ordered by synchronisation
- Compiler assumes data-race freedom when doing optimisations

1.1 Threads

```
void Foo(int a, int& b) {...} std::thread t(Foo, x, std::ref(b));
std::thread t([x, &y]() -> void {...});
t.join();
std::thread::id cur = std::this_thread::get_id(), none = std::thread::id();
```

1.2 Locks and Condition Variables

```
std::mutex mutex_; mutex_.lock(); mutex_.unlock();
{ std::scoped_lock<std::mutex> lock(mutex_); ... };
std::condition_variable condition_; condition_.notify_one(); condition_.notify_all()
{ std::unique_lock<std::mutex> lock(mutex_);
    condition_.wait(lock, [...]() -> bool {return x == 1;}); // returns if predicate is true; ... }
```

1.3 Atomics

```
std::atomic<int> x(1); x.store(2); y = x.load(); z = x.exchange(y);
success = x.compare_exchange_strong(&expected, desired); // store old value to &expected
x.fetch_add(y); x.fetch_sub(y); x.fetch_and(y); x.fetch_or(y); x.fetch_and(y);
std::memory_order_seq_cst,
std::memory_order_acq_rel, std::memory_order_acquire, std::memory_order_release,
// message passing: acquire load reads from release store
std::memory_order_relaxed // sequential consistency per location only, no inter-thread synchronisation
```

1.4 Spinlocks

```
void Lock() {
  while (lock_bit_.exchange(true, std::memory_order_acquire)) {
    do { for (size_t i = 0; i < backoff; i++) {_mm_pause();} backoff = std::min(max_backoff, backoff * 2);
    // Active Backoff: do { for (volatile size_t i = 0; i < 100; i++) {}
  } while (lock_bit_.load(std::memory_order_relaxed));
}}
void Unlock() { lock_bit_.store(false, std::memory_order_release); }
class SpinLockTicket {
  public: SpinLockTicket() : next_ticket_(0); now_serving_(0);
  void Lock() { const auto ticket = next_ticket_.fetch_add(1);
    while(now_serving_.load() != ticket) {_mm_pause();} }
  void Unlock() { now_serving_.store(now_serving_.load() + 1); }
  private: std::atomic<size_t> next_ticket_; std::atomic<size_t> now_serving_;
}
```

1.5 Futex

```
class MutexFutex {
  public: MutexFutex() : state_(kFree) {}
  void Lock() {
    int old = cmpxchg(kFree, kLockedNoWaiters); if (old_value == kFree) return;
    do { if (old == kLockedWaiters || cmpxchg(kLockedNoWaiters,kLockedWaiters) != kFree) {
      syscall(SYS_futex, &state_, FUTEX_WAIT, kLockedWaiters, ...);
    } old = cmpxchg(kFree, kLockedWaiters); } while (old_value != kFree);
  }
  void Unlock() {
    if (state_.fetch_sub(1) == kLockedWaiters) {
    state_.store(kFree); syscall(SYS_futex, &state_, FUTEX_WAKE, 1, ...);}
  }
  private: const int kFree = 0, kLockedNoWaiters = 1, kLockedWaiters = 2; std::atomic<int> state_;
  int cmpxchg(int expected, int desired) {
    state_.compare_exchange_strong(expected, desired); return expected;}
}
```

2 Haskell

```
forkIO :: IO () -> IO ThreadId; newMVar :: a -> IO (MVar a); newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a; putMVar :: MVar a -> a -> IO (); readMVar :: MVar a -> IO a
```

2.1 Unbounded Channel

```
type Stream a = MVar (Item a); data Item a = Item a (Stream a)
data Channel a = Channel (MVar (Stream a)) (MVar (Stream a))
newChannel = do hole <- newEmptyMVar; readVar <- newMVar hole; writeVar <- newMVar hole
    return (Channel readVar writeVar)
readChannel (Channel readVar _) = do stream <- takeMVar readVar
    (Item value tail) <- takeMVar stream; putMVar readVar tail; return value
writeChannel (Channel _ writeVar) value = do newhole <- newEmptyVar; oldHole <- takeMVar writeVar
    putMVar oldHole (Item value newHole); putMVar writeVar newHole
```

3 Rust

```
let result : Arc<Mutex<u32>> = Arc::new(Mutex::new(0)); let result_t1 = result.clone();
let t1 = thread::spawn(move || {let mut data = result_t1.lock().unwrap(); *data += 1}); t1.join().unwrap();
let atomic_result : Arc<AtomicU32> = Arc::new(AtomicU32::new(0)); let atomic_result_t1 = result.clone()
let t1 = thread::spawn(move || {atomic_result.fetch_add(1, Ordering::Relaxed)}); t1.join().unwrap();
```

4 Dynamic Data Race Detection

- $C : Threads \to VC$: C_t is what t knows about all threads. $C_t(t)$ is my clock. t acquires information by acquiring locks.
- $L : Locks \to VC$: L_m is the VC of the last thread that released m.
- $R : Locations \to VC$: R_x is the clock of each thread the last it read from x.
- $W : Locations \to VC$: W_x is the clock of each thread the last it wrote to x.
- Initial analysis state: $C = (inc_0(\bot), \ldots, inc_{N-1}(\bot)), L = \lambda m.\bot, R = W = \lambda x.\bot$ where $\bot = (0, \ldots, 0)$

$$\frac{C' = C[t \mapsto (C_t \sqcup L_m)]}{(C, L, R, W) \xrightarrow{acq(t,m)} (C', L, R, W)} \qquad \frac{W_x \sqsubseteq C_t \qquad R' = R[x \mapsto R_x[t \mapsto C_t(t)]]}{(C, L, R, W) \xrightarrow{rd(t,x)} (C, L, R', W)}$$

$$\frac{L' = L[m \mapsto C_t] \qquad C' = C[t \mapsto inc_t(C_t)]}{(C, L, R, W) \xrightarrow{rel(t,m)} (C', L', R, W)} \qquad \frac{R_x \sqsubseteq C_t \qquad W_x \sqsubseteq C_t \qquad W' = W[x \mapsto W_x[t \mapsto C_t(t)]]}{(C, L, R, W) \xrightarrow{wr(t,x)} (C, L, R, W')}$$

$$\frac{\exists u. W_x(u) > C_t(u)}{(C, L, R, W) \xrightarrow{rd(t,x)} WriteReadRace(u, t, x)}$$

$$\frac{\exists u. R_x(u) > C_t(u)}{(C, L, R, W) \xrightarrow{wr(t,x)} ReadWriteRace(u, t, x)} \qquad \frac{\exists u. W_x(u) > C_t(u)}{(C, L, R, W) \xrightarrow{wr(t,x)} WriteWriteRace(u, t, x)}$$

5 Theory

- Safety: It is a safety property because if XXX then the property is violated in finite time. The "bad thing" of concern is XXX.
- Liveness: It can be read as XXX eventually. It is a liveness property as one cannot violate it in a finite number of steps, even if not XXX, it may happen eventually. The "good thing" is XXX.
- Mutual exclusion is a safety property but no deadlock is a liveness property.
- Amdahl's Law: Speedup = $\frac{\text{1 thread execution time}}{\text{n thread execution time}} = \frac{1}{1-p+\frac{p}{n}}$ where p is the parallel fraction and n is the number of threads

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

$$\frac{}{\text{while B do C}, s \xrightarrow{\epsilon}_c \text{if B then (C; while B do C) else skip, s}} \qquad \frac{eval(s, E) = v \qquad s' = s[a \mapsto v]}{a := E, s \xrightarrow{\epsilon}_c skip, s'}$$

- An execution graph G is complete if every read/update reads from some write/update
- If instruction reordering does not lead to valid outcome, assume outcome is valid and proof by contradiction.
- rf GREEN write-read; mo ORANGE write-write; rb RED read-write; ppo PINK; hb BLUE.
- If declarative semantics is inconsistent, justify G.MO is the only valid option. If not, justify why G is complete.
- $COH < RA < TSO < SC$

5.1 Sequential Consistency

Instructions of each thread executed in order, instructions of different threads can be interleaved arbitrarily.

$$\frac{s(a) = v}{x := a, s \xrightarrow{(W,x,v)}_c skip, s} \qquad \frac{s' = s[a \mapsto v]}{a := x, s \xrightarrow{(R,x,v)}_c skip, s} \qquad \frac{eval(s, E) = v \qquad v_n = v_o + v}{FAA(x, E), s \xrightarrow{(U,x,v_o,x_n)}_c skip, s}$$

$$\frac{eval(s, E_o) = v_o \qquad eval(s, E_n) = v_n \qquad s' = s[a \mapsto 1]}{a := CAS(x, E_o, E_n), s \xrightarrow{(U,x,v_o,x_n)}_c skip, s} \qquad \frac{eval(s, E_o) = v_o \qquad v \neq v_o \qquad s' = s[a \mapsto 0]}{a := CAS(x, E_o, E_n), s \xrightarrow{(U,x,v,\perp)}_c skip, s}$$

$$\frac{M(x) = v}{M \xrightarrow{\tau:(R,x,v)}_m M} \qquad \frac{M' = M[x \mapsto v]}{M \xrightarrow{\tau:(W,x,v)}_m M'} \qquad \frac{M(x) = v_o \qquad M' = M[x \mapsto v_n]}{M \xrightarrow{\tau:(U,x,v_o,v_n)}_m M'} \qquad \frac{M(x) = v}{M \xrightarrow{\tau:(U,x,v,\perp)}_m M'}$$

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

Lamport: An execution graph G is SC-consistent if G is complete and is SC-consistent w.r.t. some strict total order sc on G.E.
(1) If $\langle a, b\rangle \in G.po$ then $\langle a, b\rangle \in sc$.
(2) If $\langle w, r\rangle \in G.\text{rf}$ then $\langle w, r\rangle \in sc$.
(3) If $\langle w, r\rangle \in G.\text{rf}$ then there does not exist $w' \in G.WU_{loc(r)}$ such that $\langle w, w'\rangle \in sc$ and $\langle w', r\rangle \in sc$.
Alternative: An execution graph G is SC-consistent if G is complete and there exists a modification order mo for G such that $acyclic(G.po \cup G.\text{rf} \cup mo \cup rb)$ holds, where $rb \equiv G.\text{rf}^{-1}; mo \backslash id$.

5.2 Total Store Ordering

Following adjacent instructions may be reordered:

- write-read on different locations - x:=1; b:=y; becomes b:=y; x:=1;
- write-read forwarding on the same location - x:=1; b:=x; becomes x:=1; b:=1;
- write-assigment - x:=1; b:=1; becomes b:=1; x:=1;

$$\frac{}{\text{mfence,s} \xrightarrow{MF}_c skip, s} \qquad \frac{B(\tau) = b \qquad b' = b.(W, x, v) \qquad B' = B[\tau \mapsto b]}{M, B \xrightarrow{\tau:(W,x,v)}_m M, B'}$$

$$\frac{B(\tau) = b \qquad \text{get(M,b,x)=v}}{M, B \xrightarrow{\tau:(R,x,v)}_m M, B} \quad get(M, b, x) = \begin{cases} v & \text{if } \exists b_1, b_2. b = b_1.(W, x, v).b_2 \wedge \neg\exists v'.(W, x, v') \in b_2 \\ M(x) & \text{otherwise} \end{cases}$$

$$\frac{B(\tau) = \emptyset \qquad M(x) = v_o \qquad M' = M[x \mapsto v_n]}{M, B \xrightarrow{\tau:(U,x,v_o,v_n)}_m M', B} \qquad \frac{B(\tau) = \emptyset \qquad M(x) = v}{M, B \xrightarrow{\tau:(U,x,v,\perp)}_m M, B}$$

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

$$P, S_0, M_0, B_0 \rightarrow^* P_{skip}, S, M, B_0$$

Original: An execution G is TSO-consistent if G is complete and is TSO-consistent w.r.t. some strict partial order tso in G.E.
(1) tso is total on $G.E \backslash G.R$
(2) $G.po \backslash (G.W \times G.R) \subseteq tso$
(3) $G.\text{rf} \subseteq tso \cup G.po$
(4) If $\langle w, r\rangle \in G.\text{rf}$ then there does not exists $w' \in G.WU_{loc(r)}$ such that $\langle w, w'\rangle \in tso$ and $\langle w', r\rangle \in tso \cup G.po$
Alternative: An execution G is TSO-consistent if G is complete and there exists a modification order mo for G such that $G.\text{rfi} \cup G.\text{rbi} \subseteq G.po$ and $acyclic(ppo \cup G.\text{rfe} \cup mo \cup rbe)$ where $ppo \equiv (G.po \backslash (G.W \times G.R))^+$.

5.3 Partial Store Ordering

Following adjacent instructions may be reordered:

- TSO reorderings: write-read on different locations; write-read forwarding; write-assignment
- sfence-read
- write-write on different locations

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

$$\frac{B(\tau) = b_1.(W, x, v).b_2 \qquad SF \notin b_1 \qquad \neg\exists v'.(W, x, v') \in b_1 \qquad M' = M[x \mapsto v] \qquad B' = B[\tau \mapsto b_1.b_2]}{M, B \xrightarrow{\tau:\epsilon} M', B'}$$

An execution G is PSO-consistent if G is complete and there exists a modification order mo for G such that $G.\text{rfi} \cup G.\text{rbi} \subseteq G.po$ and $acyclic(ppo' \cup G.\text{rfe} \cup mo \cup rbe)$ where $ppo \equiv (G.po \backslash ((G.W \times G.R) \cup (G.\text{sfence} \times G.R) \cup \{(w_1, w_2) \in G | loc(w_1) \neq loc(w_2)\}))^+$.

5.4 Release Acquire

Alternative: An execution graph G is RA-consistent iff G is complete and G is coherent w.r.t some modification order mo for G such that $acyclic(hb_{ra} \cup mo \cup rb)$ holds where $hb_{ra} \equiv (po \cup \text{rf})^+|_{loc}$.

5.5 Coherence

Alternative: An execution graph G is coherent iff G is complete and G is coherent w.r.t some modification order mo for G such that $acyclic(po|_{loc} \cup \text{rf} \cup mo \cup rb)$ holds.

5.6 Concurrent Objects

- Sequential History is when method calls of different threads do not interleave (final pending invocation is ok).
- Two histories are equivalent if the thread projection over every thread is the same.
- A method call precedes ($\rightarrow$) another if the response event precedes invocation event.
- History H is linearalisable if it can be extended to G by appending zero or more responses to pending invocations and discarding other pending invocations, such that G is equivalent to legal sequential history S where
- Composability Theorem: H is linearalisable iff for every object x, $H|x$ is linearalisable.
- History H is sequentially consistent if it can be extended to G by appending zero or more responses to pending invocations and discarding other pending invocations, such that G is equivalent to legal sequential history S.

Steps to prove:

1. Convert diagram to sequence H.
2. Come up with a complete diagram G.
3. Come up with a sequential S such that (1) S is legal, (2) S is equivalent to G, (3) $\rightarrow_G \subseteq \rightarrow_S$ - linearalisable.

Steps to disprove:

1. Take a complete H, then there must be a sequential history S such that (1) S is legal and (2) S is equivalent to H.
2. Since S is legal, come up with some precedence ($\rightarrow_S$).
3. Since S is equivalent to H, come up with some mroe precedence ($\rightarrow_S$) from the definition of H.
4. Obtain a cycle. This is a contradiction as S is a sequential history and thus $\rightarrow_S$ is a strict irreflexive order.