Dynamic Race Detection for C++11

PB15111671　李嘉豪

Basics： Atomics

4 Operations: Read(**R**), Write(**W**), Read-Modify-Write(**RMW**), Fence(**F**)

6 Memory Ordering:

- Acquire (acq)
- Release (rel)
- Relaxed (rlx)
- ~~Acquire-Release~~
- Sequential-Consistent(SC)
- ~~Consume~~

Basics： Vector Clock

$$\bot_V = \lambda t.0 \qquad V_1 \cup V_2 \triangleq \lambda t.\max(V_1(t), V_2(t))$$

$$V_1 \leq V_2 \triangleq \forall t.V_1(t) \leq V_2(t)$$

$$inc_t(V) = \lambda u.\ \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u)$$

Each thread holds a vector clock that represents the logical time of the last instruction by corresponding thread that happens before any instruction current thread will perform in the future

Basics： Vector Clock

write-write: $c \leq \mathbb{C}_u(t)$ **write-read:** $c \leq \mathbb{C}_u(t)$

read-write: $c \leq \mathbb{C}_u(t) \wedge \mathbb{R}_x \leq \mathbb{C}_u$

```
void T1() {
    nax = 1;
    x.store(1, std::memory_order_relaxed);
}

void T2() {
    if (x.load(std::memory_order_acquire) == 1) {
        std::cout << nax;
    }
}
```

Basics： Vector Clock

write-write: $c \leq \mathbb{C}_u(t)$ **write-read:** $c \leq \mathbb{C}_u(t)$

read-write: $c \leq \mathbb{C}_u(t) \wedge \mathbb{R}_x \leq \mathbb{C}_u$

```
void T1() {
  nax = 1;                                                    // A
  x.store(1, std::memory_order_release);                      // B
}
void T2() {
  if (x.load(std::memory_order_acquire) == 1)                 // C
    x.store(2, std::memory_order_relaxed);                    // D
}
void T3() {
  if (x.load(std::memory_order_acquire) == 2)                 // E
    nax; // read from 'nax'                                   // F
}
```

Memory Model In C++11

Pre-exectutions:

1. Sequenced-Before (sb): intra-threads relation that orders events by the order they appear in the program
2. Additional-Synchronized-With (asw): relations on thread launch and join.

Witness-Relations:

1. Read-From (rf): a read has a source
2. Modification-Order (mo): Stores to a location have a total order
3. Sequential-Consistent (sc): all atomic operations marked with sequential-consistent have a total order

Memory Model In C++11

Derived-relations:

1. Release-Sequence (rs): Headed by a release store, continues along all stores to the same location. Blocked when another thread performs a store to the location (RMW will continue it).
2. Hypothetical-Release-Sequence (hrs): Headed by all stores, others are same as release-sequence.
3. Synchronises-With (sw): When a thread performs an acquire load, and reads from a store that is part of a release sequence, the head of the release sequence synchronises with the acquire load.
4. Happens-Before (hb): transitive closure of $(sb \cup sw)$

Memory Model In C++11

Data-Race:

A data race occurs between two memory accesses when at least one is non-atomic, at least one is a store, and neither happens before the other according to the *hb* relation

Extended Vector Clock

STATE:

$$\mathbb{C} : Tid \rightarrow VC$$
$$\mathbb{L} : Var \rightarrow VC$$
$$\mathbb{V} : Var \rightarrow (Tid \rightarrow VC)$$

$$\mathbb{F}^{rel} : Tid \rightarrow VC$$
$$\mathbb{F}^{acq} : Tid \rightarrow VC$$

Extended Vector Clock

[ACQUIRE LOAD]

$$\frac{\mathbb{C}' = \mathbb{C}[t := \mathbb{C}_t \cup \mathbb{L}_x]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{load_{acq}(x,t)} (\mathbb{C}', \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELAXED LOAD]

$$\frac{\mathbb{F}^{acq\prime} = \mathbb{F}^{acq}[t := \mathbb{F}^{acq}_t \cup \mathbb{L}_x]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{load_{rlx}(x,t)} (\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq\prime})}$$

Extended Vector Clock

[RELEASE STORE]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{C}_t] \qquad \mathbb{V}' = \mathbb{V}[x := \emptyset[t := \mathbb{C}_t]]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{store_{rel}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELAXED STORE]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{V}_x(t) \cup \mathbb{F}^{rel}_t] \qquad \mathbb{V}' = \mathbb{V}[x := \emptyset[t := \mathbb{V}_x(t)]]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{store_{rlx}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

Extended Vector Clock

[RELEASE RMW]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{L}_x \cup \mathbb{C}_t] \qquad \mathbb{V}' = \mathbb{V}[x := \mathbb{V}_x[t := \mathbb{C}_t]]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{rmw_{rel}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

[RELAXED RMW]

$$\frac{\mathbb{L}' = \mathbb{L}[x := \mathbb{L}_x \cup \mathbb{F}_t^{rel}]}{(\mathbb{C}, \mathbb{L}, \mathbb{V}, \mathbb{F}^{rel}, \mathbb{F}^{acq}) \Rightarrow^{rmw_{rlx}(x,t)} (\mathbb{C}, \mathbb{L}', \mathbb{V}', \mathbb{F}^{rel}, \mathbb{F}^{acq})}$$

Extended Vector Clock

```
void T1() {
  nax = 1;                                                    // A
  x.store(1, std::memory_order_release);                      // B
}
void T2() {
  if (x.load(std::memory_order_acquire) == 1)                 // C
    x.store(2, std::memory_order_relaxed);                    // D
}
void T3() {
  if (x.load(std::memory_order_acquire) == 2)                 // E
    nax; // read from 'nax'                                   // F
}
```

Thank You!