

# The Theory & Practice of Concurrent Programming

Lectured by Azalea Raad and Alastair Donaldson

Typed by Aris Zhu Yi Qing

December 6, 2021

## Contents

|   |          |   |           |
|---|----------|---|-----------|
| <b>1 Synchronisation Paradigms</b>                                      | <b>2</b> | 4.3 SpinLock  | 9         |
| 1.1 Properties in Asynchronous computation                              | 2        | 4.3.1 Local spinning                                      | 9         |
| 1.2 Problems in Asynchronous computation                                | 2        | 4.3.2 Active backoff                                      | 9         |
| 1.3 Protocols in Asynchronous computation                               | 2        | 4.3.3 Passive backoff                                     | 9         |
| 1.4 Performance Measurement   | 3        | 4.3.4 Exponential backoff                                 | 10        |
| <b>2 Concurrent Semantics</b>   | <b>3</b> | 4.3.5 Ticket lock (Fairness)                              | 10        |
| 2.1 Notation  | 3        | 4.3.6 Comments  | 10        |
| 2.2 ConWhile concurrent programming language                            | 3        | 4.4 Futex and Hybrid Lock                                 | 10        |
| 2.3 Sequential Consistency (SC)   | 3        | 4.4.1 FUTEX_WAIT  | 10        |
| 2.4 Total Store Ordering (TSO)  | 5        | 4.4.2 FUTEX_WAKE  | 10        |
| <b>3 Linearization</b>  | <b>6</b> | 4.4.3 Simple mutex with <code>futex</code>                | 11        |
| 3.1 Notation  | 6        | 4.4.4 Smart <code>futex</code> -based mutex               | 11        |
| 3.2 Definitions   | 6        | <b>5 Concurrency in Haskell</b>                           | <b>12</b> |
| <b>4 Concurrency in C++</b>   | <b>8</b> | 5.1 Thread  | 12        |
| 4.1 Threads and Locks   | 8        | 5.2 Join Thread   | 12        |
| 4.1.1 <code>std::thread</code>  | 8        | 5.3 Mutual Exclusion                                      | 13        |
| 4.1.2 <code>std::mutex</code>   | 8        | 5.4 Message Passing – Producer-Consumer                   | 13        |
| 4.1.3 <code>std::scoped_lock&lt;std::mutex&gt;</code>                   | 8        | 5.5 Linked List   | 14        |
| 4.1.4 <code>std::unique_lock&lt;std::mutex&gt;</code>                   | 8        | <b>6 Concurrency in Rust</b>                              | <b>15</b> |
| 4.1.5 <code>std::condition_variable</code>                              | 8        | 6.1 Basics  | 15        |
| 4.2 Atomics   | 8        | 6.2 Threading with mutex                                  | 15        |
| 4.2.1 Operations on <code>std::atomic&lt;T&gt;</code>                   | 8        | 6.3 Threading with Atomics                                | 15        |
| 4.2.2 (RMW) Operations on <code>std::atomic&lt;integral_type&gt;</code> | 9        | 6.4 Striped Hashset                                       | 16        |
| 4.2.3 Memory ordering   | 9        | <b>7 Dynamic Data Race Detection (Vector Clock-based)</b> | <b>18</b> |
|   |          | 7.1 Definitions and Assumptions                           | 18        |
|   |          | 7.2 Vector Clock Algorithm State                          | 18        |

|     |                                  |    |
|-----|----------------------------------|----|
| 7.3 | Initial Analysis State . . . . . | 19 |
| 7.4 | Intercepted Operations . . . . . | 19 |
| 7.5 | Race Rules . . . . .             | 19 |

## 1 Synchronisation Paradigms

### 1.1 Properties in Asynchronous computation

#### 1. Safety

- Nothing bad happens ever
- If it is violated, it is done by a finite computation

#### 2. Liveness

- Something good happens eventually
- Cannot be violated by a finite computation

### 1.2 Problems in Asynchronous computation

#### 1. Mutual Exclusion (Safety)

- **cannot** be solved by transient communication or interrupts
- **can** be solved by shared variables that can be read or written

#### 2. No Deadlock (Liveness): Some event $A$ eventually happens.

### 1.3 Protocols in Asynchronous computation

#### 1. Flag Protocol (from B's perspective):

- Raise flag
- While A's flag is up
  - Lower flag
  - Wait for A's flag to go down
  - Raise flag
- Do something
- Lower flag

#### 2. Producer/Consumer:

- For A(producer), while flag is up wait. So when flag becomes down, do something, then raise the flag.
- For B(consumer), while flag is down, wait. So when flag becomes up, do something, then put down the flag.

#### 3. Readers/Writers:

- Each thread  $i$  has `size[i]` counter. Only it increments or decrements.
- To get object's size, a thread reads a “snapshot” of all counters.
- This eliminates the bottleneck of “having exclusive access to the common counter”.

## 1.4 Performance Measurement

Amdahl's law:

$$\text{Speedup} = \frac{\text{1-thread execution time}}{\text{$n$-thread execution time}} = \frac{1}{1 - p + \frac{p}{n}},$$

where  $p$  is the fraction of the algorithm having parallel execution, and  $n$  is the number of threads.

## 2 Concurrent Semantics

### 2.1 Notation

- $x, y, z, \dots$  shared memory locations
- $a, b, c, \dots$  private registers
- $E, E_1, \dots$  expressions over values (integers) and registers
- $a := x$  **read** from location  $x$  into register  $a$
- $x := a$  **write** contents of register  $a$  to location  $x$
- $a := E$  **assignment**: compute  $E$  and write it to  $a$

### 2.2 ConWhile concurrent programming language

|                     |     |  |                         |
|---------------------|-----|--|-------------------------|
| $B \in \text{Bool}$ | ::= | <code>true</code>   <code>false</code>   ...                               |                         |
| $E \in \text{Exp}$  | ::= | ...   $E + E$   ...  |                         |
| $C \in \text{Com}$  | ::= | $a := E$   | assignment              |
|                     |     | $a := x$   | (memory) read           |
|                     |     | $x := a$   | (memory) write          |
|                     |     | $a := \text{CAS}(x, E, E) \mid \text{FAA}(x, E)$                           | (memory) RMWs           |
|                     |     | <code>skip</code>   $C$ ; $C$   <code>while</code> $B$ <code>do</code> $C$ |                         |
|                     |     | <code>if</code> $B$ <code>then</code> $C$ <code>else</code> $C$ ,          |                         |
|                     |     | <code>mfence</code>  | memory fence (TSO only) |

where **FAA** (fetchAndAdd) is considered *weak* RMW because it enables synchronisation between two threads only, whereas **CAS** (compareAndSet) is considered *strong* RMW because it enables synchronisation among an arbitrary number of threads.

### 2.3 Sequential Consistency (SC)

Also called Interleaving Semantics. The instructions of each thread are executed in order. Instructions of different threads interleave arbitrarily.

- We model ConWhile concurrent program as a map from thread identifiers ( $\tau \in \text{Tid}$ ) to sequential commands:

$$P \in \text{Prog} \triangleq \text{Tid} \rightarrow \text{Com}.$$

- We use  $\parallel$  notation for concurrent programs and write

$$C_1 \parallel C_2 \parallel \dots \parallel C_n$$

for the  $n$ -threaded program  $P$  with

$$\text{dom}(P) = \{\tau_1, \dots, \tau_n\}$$

and  $P(\tau_i) = C_i$  for  $i \in \{1, \dots, n\}$ .

- For instance, we write  $\text{dom}(P_{\text{sb}}) = \{\tau_1, \tau_2\}$ , with  $P_{\text{sb}}(\tau_1) = x := 1; a := y;$  and  $P_{\text{sb}}(\tau_2) = y := 1; b := x;$ , therefore

$$P_{\text{sb}} \triangleq x := 1; a := y; \parallel y := 1; b := x; .$$

- We model the shared memory as a map from locations to values:

$$M \in \text{Mem} \triangleq \text{Loc} \rightarrow \text{Val},$$

where  $\text{Val}$  denotes the set of all values, including integer and Boolean values.

- We define store as a map from registers to values:

$$s \in \text{Store} \triangleq \text{Reg} \rightarrow \text{Val}.$$

- We define store map associating each thread with its private store:

$$S \in \text{SMap} \triangleq \text{Tid} \rightarrow \text{Store}.$$

- An SC configuration is a triple,  $(P, S, M)$ , comprising a program  $P$  to be executed, the store map  $S$ , and the shared memory  $M$ .
- The program transitions describe the steps in program executions.
- The storage transitions describe how instructions interact with the storage (memory) system.
- An SC transition label,  $l \in \text{Lab}$ , may be:
  - the *empty* label  $\epsilon$  to denote a silent transition
  - a *read* label  $(R, x, v)$  to denote reading value  $v$  from memory location  $x$
  - a *write* label  $(W, x, v)$  to denote writing value  $v$  to memory location  $x$
  - a *successful RMW* label  $(\text{RMW}, x, v_0, v_n)$  to denote updating the value of location  $x$  to  $v_n$  when the old value of  $x$  is  $v_0$

– a *failed RMW* label  $(\text{RMW}, x, v_0, \perp)$  to denote a failed **CAS** instruction where the old value of  $x$  does not match  $v_0$ .

- Assume that store  $s$  has the mapping for all Boolean expressions  $B$  and program expressions  $E$ .
- SC Sequential Transitions (Familiar Cases):

$$\frac{C_1, s \xrightarrow{l}_c C'_1, s'}{C_1; C_2, s \xrightarrow{l}_c C'_1; C_2, s'} \quad \frac{}{\text{skip}; C, s \xrightarrow{\epsilon}_c C, s}$$

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

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

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

- SC Sequential Transitions (New Cases):

$$\begin{array}{l} x := a \quad \frac{s(a) = v}{x := a, s \xrightarrow{(W, x, v)}_c \text{skip}, s} \\ \\ a := x \quad \frac{s' = s[a \mapsto v]}{a := x, s \xrightarrow{(R, x, v)}_c \text{skip}, s'} \\ \\ \text{FAA}(x, E) \quad \frac{s(E) = v \quad v_n = v_0 + v}{\text{FAA}(x, E), s \xrightarrow{(\text{RMW}, x, v_0, v_n)}_c \text{skip}, s} \\ \\ \text{CAS}(x, E_0, E_n) \text{ (success)} \quad \frac{s(E_0) = v_0 \quad s(E_n) = v_n \quad s' = s[a \mapsto 1]}{a := \text{CAS}(x, E_0, E_n), s \xrightarrow{(\text{RMW}, x, v_0, v_n)}_c \text{skip}, s'} \\ \\ \text{CAS}(x, E_0, E_n) \text{ (failure)} \quad \frac{s(E_0) = v_0 \quad v \neq v_0 \quad s' = s[a \mapsto 0]}{a := \text{CAS}(x, E_0, E_n), s \xrightarrow{(\text{RMW}, x, v, \perp)}_c \text{skip}, s'} \end{array}$$

- SC (Concurrent) Program Transitions:

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

- SC Storage Transitions (of the form  $M \xrightarrow{\tau:l}_m M'$ ):

$$\begin{array}{lcl}
\text{Read} & \dfrac{M(x) = v}{M \xrightarrow{\tau:(R,x,v)}_m M} & \\
\text{Write} & \dfrac{M' = M[x \mapsto v]}{M \xrightarrow{\tau:(W,x,v)}_m M'} & \\
\text{RMW, } x, v_0, v_n & \dfrac{M(x) = v_0 \quad M' = M[x \mapsto v_n]}{M \xrightarrow{\tau:(\text{RMW},x,v_0,v_n)}_m M'} & \\
\text{RMW, } x, v, \perp & \dfrac{M(x) = v}{M \xrightarrow{\tau:(\text{RMW},x,v,\perp)}_m M'} & 
\end{array}$$

- SC Operational Semantics:

$$\begin{array}{lcl}
\text{silent transition} & \dfrac{P, S \xrightarrow{\tau:\epsilon}_p P', S'}{P, S, M \rightarrow P', S', M} & \\
\text{both program and storage systems} & \dfrac{P, S \xrightarrow{\tau:l}_p P', S' \quad M \xrightarrow{\tau:l}_m M'}{P, S, M \rightarrow P', S', M'} & \\
\text{take the same transition} & & 
\end{array}$$

- We write  $\rightarrow^*$  for the reflexive, transitive closure of  $\rightarrow$ .
- SC Traces
  - The initial memory,  $M_0 \triangleq \lambda x.0$ .
  - The initial store,  $s_0 \triangleq \lambda a.0$ .
  - The initial store map,  $S_0 \triangleq \lambda \tau.s_0$ .
  - The terminated program,  $P_{\text{skip}} \triangleq \lambda \tau.\text{skip}$ .
  - Given a program  $P$ , an **SC-trace** of  $P$  is an evaluation path s.t.

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

where the pair  $(S, M)$  denotes an **SC-outcome**.

- SC is **neither** deterministic **nor** confluent.

## 2.4 Total Store Ordering (TSO)

TSO = SC + write-read reordering. This allows the weak Store Buffering (SB) behaviour. We can stop the reordering by using memory fences or RMWs, which can impede performance.

- In addition to the concurrent program, shared memory, store, and store map defined in the SC, we have an addition buffer associating each thread, modelled as a FIFO sequence of (delayed) write label:

$$b \in \text{Buff} \triangleq \text{Seq} \langle \text{WLab} \rangle \quad \text{WLab} \triangleq \{ (W, x, v) \mid x \in \text{Loc} \wedge v \in \text{Val} \}.$$

That is, a buffer entry  $(W, x, v)$  denotes a delayed write on  $x$  with value  $v$ .

- We define buffer map associating each thread with its private buffer:

$$B \in \text{BMap} \triangleq \text{Tid} \rightarrow \text{Buff}.$$

- An TSO configuration is a quadruple,  $(P, S, M, B)$ , comprising the program  $P$  to be executed, the store map  $S$ , the shared memory  $M$  and the buffer map  $B$ .
- A TSO transition label,  $l \in \text{Lab}$ , may be:
  - an SC label, namely  $\epsilon$ ,  $(R, x, v)$ ,  $(W, x, v)$ ,  $(\text{RMW}, x, v_0, v_n)$ ,  $(\text{RMW}, x, v_0, \perp)$
  - a *memory fence* label **MF** for executing an **mfence**.
- TSO Sequential Transition (New case):

$$\text{mfence} \quad \dfrac{}{\text{mfence}, s \xrightarrow{\text{MF}}_c \text{skip}, s}$$

- TSO Program Transitions: the same as SC Program Transition.
- TSO Storage Transitions (of the form  $M, B \xrightarrow{\tau:l}_m M', B'$ ):

$$\begin{array}{lcl}
\text{Read} & \dfrac{B(\tau) = b \quad \text{get}(M, b, x) = v}{M, B \xrightarrow{\tau:(R,x,v)}_m M, B}, \text{ where} & \\
& \text{get}(M, b, x) \triangleq \begin{cases} v & \text{if } \exists b_1, b_2 \text{ s.t. } b = b_1.(W, x, v).b_2 \\ & \wedge \neg \exists v' \text{ s.t. } (W, x, v') \in b_2 \\ M(x) & \text{otherwise} \end{cases} & \\
\text{Write} & \dfrac{B(\tau) = b \quad b' = b.(W, x, v) \quad B' = B[\tau \mapsto b']}{M, B \xrightarrow{\tau:(W,x,v)}_m M, B'} & \\
\text{Memory Fence} & \dfrac{B(\tau) = \emptyset}{M, B \xrightarrow{\tau:\text{MF}}_c M, B} & 
\end{array}$$

$$\begin{array}{c}
\text{RMW}, x, v_0, v_n \quad \frac{B(\tau) = \emptyset \quad M(x) = v_0 \quad M' = M[x \mapsto v_n]}{M, B \xrightarrow{\tau: (\text{RMW}, x, v_0, v_n)}_m M', B} \\
\\
\text{RMW}, x, v, \perp \quad \frac{B(\tau) = \emptyset \quad M(x) = v}{M, B \xrightarrow{\tau: (\text{RMW}, x, v, \perp)}_m M, B} \\
\\
\text{unbuffer} \quad \frac{B(\tau) = (W, x, v).b \quad M' = M[x \mapsto v] \quad B' = B[\tau \mapsto b]}{M, B \xrightarrow{\tau: \epsilon}_m M', B'}
\end{array}$$

- TSO Operational Semantics:

$$\begin{array}{c}
\text{silent transition in program} \quad \frac{P, S \xrightarrow{\tau: \epsilon}_p P', S'}{P, S, M, B \rightarrow P', S', M, B} \\
\\
\text{silent transition in storage system} \quad \frac{M, B \xrightarrow{\tau: \epsilon}_m M', B'}{P, S, M, B \rightarrow P, S, M', B'} \\
\\
\text{both program and storage system} \quad \frac{P, S \xrightarrow{\tau: \epsilon}_p P', S' \quad M, B \xrightarrow{\tau: \epsilon}_m M', B'}{P, S, M, B \rightarrow P', S', M', B'} \\
\text{take the same transition}
\end{array}$$

- We write  $\rightarrow^*$  for the reflexive, transitive closure of  $\rightarrow$ , the same as the SC's.
- TSO Traces
  - In addition to the initial memory, initial store, initial store map, and the terminated program defined in SC, we have the initial buffer map,  $B_0 \triangleq \lambda \tau. \emptyset$ .
  - Given a program  $P$ , the initial TSO-configuration of  $P$  is  $(P, S_0, M_0, B_0)$ .
  - Given a program  $P$ , a **TSO-trace** of  $P$  is an evaluation path s.t.

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

where the pair  $(S, M)$  denotes a **TSO-outcome**.

- TSO is also **neither** deterministic **nor** confluent.

## 3 Linearization

### 3.1 Notation

- $A \text{ q.enq}(x)$  Invocation:  $\langle \text{thread} \rangle \langle \text{object} \rangle . \langle \text{method} \rangle (\langle \text{arguments} \rangle)$
- $A \text{ q:void}$  Response:  $\langle \text{thread} \rangle \langle \text{object} \rangle : \langle \text{result} \rangle$
- $H$  Sequence of invocations and responses, which looks like:

$$\begin{array}{l}
A \text{ q.enq}(3) \\
A \text{ q:void} \\
A \text{ q.enq}(5) \\
H = B \text{ p.enq}(4) \\
B \text{ p:void} \\
B \text{ q.deq}() \\
B \text{ q:3}
\end{array}$$

### 3.2 Definitions

- Invocation and response match if thread and object names agree
- Object Projections:

$$\begin{array}{ccc}
A \text{ q.enq}(3) & & A \text{ q.enq}(3) \\
A \text{ q:void} & & A \text{ q:void} \\
A \text{ q.enq}(5) & & A \text{ q.enq}(5) \\
H = B \text{ p.enq}(4) & \implies & H|q = \\
B \text{ p:void} & & \\
B \text{ q.deq}() & & B \text{ q.deq}() \\
B \text{ q:3} & & B \text{ q:3}
\end{array}$$

- Thread Projections:

$$\begin{array}{ccc}
A \text{ q.enq}(3) & & \\
A \text{ q:void} & & \\
A \text{ q.enq}(5) & & \\
H = B \text{ p.enq}(4) & \implies & H|B = B \text{ p.enq}(4) \\
B \text{ p:void} & & B \text{ p:void} \\
B \text{ q.deq}() & & B \text{ q.deq}() \\
B \text{ q:3} & & B \text{ q:3}
\end{array}$$

- An invocation is pending if it has no matching response. It may or may not have taken effect.
- A complete subhistory is a history where pending invocations are discarded.
- A sequential history is one whose invocations are always *immediately* followed by their respective responses.
- A well-formed history is one whose per-thread projections are sequential.
- Equivalent histories are those which have the same threads and their per-thread projections are the same.
- A sequential specification is some way of telling whether a single-thread, single-object history, is legal.
- A sequential history  $H$  is legal if for every object  $x$ ,  $H|x$  is in the sequential specification for  $x$ .
- A method call precedes another if its response event precedes the other's invocation event.

– Given history  $H$ , method executions  $m_0, m_1$  in  $H$ , we say

$$m_0 \rightarrow_H m_1$$

if  $m_0$  precedes  $m_1$ .

– The above relation is a partial order. It is total order if  $H$  is sequential.

- History  $H$  is linearizable if
  - it can be extended to a *complete* history  $G$
  - $G$  is equivalent to a *legal sequential* history  $S$ , where  $\rightarrow_G \subseteq \rightarrow_S$ .
- Remarks on linearizability:
  - For pending invocations which took effect, keep them, and discard the rest.
  - $\rightarrow_H$  stands for the set of all precedence relations in history  $H$ .
  - Focus on total(defined in every state) method.
  - Partial methods are equivalent to thread blocking, and blocking is unrelated to synchronisation.
  - We can identify “linearization points” to help check if executions are linearizable. The point

- \* is between invocation and response events
- \* correspond to the effect of the call
- \* “justify” the whole execution

- **Composability Theorem:**

History  $H$  is linearizable  $\iff \forall$  object  $x$ ,  $H|x$  is linearizable.

- History  $H$  is sequentially consistent (SC) if
  - it can be extended to a *complete* history  $G$
  - $G$  is equivalent to a *legal sequential* history  $S$ , where  $\rightarrow_G \subseteq \rightarrow_S$ .
- Remarks on SC:
  - *Cannot* re-order operations done by the same thread
  - *Can* re-order non-overlapping operations done by different threads
  - SC is too strong for hardware architecture, yet too weak for software *specification*.
  - SC is useful for abstracting software *implementation*.
- (non-examinable) Progress conditions (from least ideal to most ideal):
  - Deadlock-free: *some* thread trying to acquire the lock eventually succeeds.
  - Starvation-free: *every* thread trying to acquire the lock eventually succeeds.
  - Lock-free: *some* thread calling a method eventually returns.
  - Wait-free: *every* thread calling a method eventually returns.

## 4 Concurrency in C++

### 4.1 Threads and Locks

#### 4.1.1 `std::thread`

```
#include <thread>
#include <functional> // Provides std::ref

// approach 1
void Foo(int by_value, std::string& by_reference) { ... }
std::string world = "World";
std::thread t1(Foo, 1, std::ref(world));

// approach 2
int y, x = 3;
std::thread t2([x, &y]() -> void {
    y = x * 42;
});

t1.join();
t2.join();
```

#### 4.1.2 `std::mutex`

```
#include <mutex>

std::mutex mutex_;
mutex_.lock();
mutex_.unlock();
```

#### 4.1.3 `std::scoped_lock<std::mutex>`

```
class ScopedLock {
public:
    explicit ScopedLock(std::mutex &mutex) : mutex_(mutex) {
        mutex_.lock();
    }

    ~ScopedLock() {
        mutex_.unlock();
    }

private:
```

```
    std::mutex &mutex_;
};
```

#### 4.1.4 `std::unique_lock<std::mutex>`

- constructed with one mutex
- locks mutex on construction (default), or deferred locking:  
`std::unique_lock<std::mutex> lock(mutex_, std::defer_lock);`
- allows unlocking and relocking
- unlocks mutex on destruction if still held
- supports transfer of ownership to a distinct *unique* owner via `std::move`.

#### 4.1.5 `std::condition_variable`

```
#include <condition_variable>

std::condition_variable condition_;
condition_.notify_one();
condition_.notify_all();

std::unique_lock<std::mutex> lock(mutex_);
/* Assuming that the thread has locked the mutex:
 * 1. return if predicate holds (continue executing code after wait())
 * 2. release lock
 * 3. block until another thread signals (via notify_one or notify_all)
 * 4. acquire lock, then return to step 1
 */
condition_.wait(lock, [...]() -> bool {
    return ...;
});
```

## 4.2 Atomics

### 4.2.1 Operations on `std::atomic<T>`

- `store(x)`: store value x of type T
- `load()`: yields value of type T
- `exchange(x)`: store x and return old value
- `compare_exchange_strong(expected, desired)`:



- Success: old value is **expected**, store **desired**, return **true**
- Failure: old value not **expected**, store old value to **expected**, return **false**
- `compare_exchange_weak(expected, desired)`: the same as the previous operation, except that it allows to fail spuriously, i.e. fail even if old value is **expected**.

#### 4.2.2 (RMW) Operations on `std::atomic<integral_type>`

- `fetch_add(x)`: replace the value with (value + x), return old value.
- `fetch_sub(x)`: similar
- `fetch_and(x)`: similar
- `fetch_or(x)`: similar
- `fetch_xor(x)`: similar

#### 4.2.3 Memory ordering

- Atomics are sequentially consistent by default.
- **Relaxed memory order** only guarantee sequential consistency *per location*.
- **Acquire semantics** prevents memory reordering of the read-acquire with any read or write operation that *follows* it in program order, i.e. all memory operations after read-acquire happen after read-acquire.
- **Release semantics** prevents memory reordering of the write-release with any read or write operation that *precedes* it in program order. i.e. all memory operations before write-release happen before write-release.
- In fact, acquiring a lock implies acquire semantics, releasing a lock implies release semantics!
- Please refer to this for detailed explanation and discussion on acquire-release semantics.
- various atomics semantics:
  - `std::memory_order_seq_cst`
  - `std::memory_order_relaxed`
  - `std::memory_order_release`

- `std::memory_order_acquire`
- `std::memory_order_acq_rel`

- example code snippet: `x.fetch_add(1, std::memory_order_acq_rel);`

### 4.3 SpinLock

#### 4.3.1 Local spinning

```
void Lock() {
    // lock_bit_ is a boolean, represents if lock is acquired
    while (lock_bit_.exchange(true)) {
        // Did not get the lock -- spin until it's free
        while (lock_bit_.load()) {
            // someone still holds the lock
        }
        // observed the lock being free -- try to grab it
    }
}
```

#### 4.3.2 Active backoff

```
void Lock() {
    while (lock_bit_.exchange(true)) {
        // Did not get the lock -- spin until it's free
        do {
            for (volatile size_t i = 0; i < 100; i++) {
                // Do nothing
            }
        } while (lock_bit_.load());
        // observed the lock being free -- try to grab it
    }
}
```

#### 4.3.3 Passive backoff

```
#include <emmintrin.h>
void Lock() {
    while (lock_bit_.exchange(true)) {
        // Did not get the lock -- spin until it's free
        do {
            for (size_t i = 0; i < 4; i++) {
                // Tell hardware that we are spinning
            }
        } while (lock_bit_.load());
    }
}
```

```

        _mm_pause();
    }
} while (lock_bit_.load());
// observed the lock being free -- try to grab it
}
}

```

#### 4.3.4 Exponential backoff

```

void Lock() {
    const size_t kMinBackoffIterations = 4u;
    const size_t kMaxBackoffIterations = 1u << 10u;
    size_t backoff_iterations = kMinBackoffIterations;
    while (lock_bit_.exchange(true)) {
        // Did not get the lock -- spin until it's free
        do {
            for (size_t i = 0; i < backoff_iterations; i++) {
                // Tell hardware that we are spinning
                _mm_pause();
            }
            backoff_iterations =
                std::min(backoff_iterations << 1, kMaxBackoffIterations);
        } while (lock_bit_.load());
        // observed the lock being free -- try to grab it
    }
}

```

#### 4.3.5 Ticket lock (Fairness)

```

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_;
};

```

#### 4.3.6 Comments

- `lock_bit_.exchange(true)`: performs Test-And-Set(TAS) operation.
  - It enters the memory to set `lock_bit_` to `true`, and return its old value
  - It also thrashes the cached values (**cache thrashing**) for other cores, i.e. the cached value of `lock_bit_` is invalidated for other cores. This leads to memory access when other cores load the value of `lock_bit_` and cache the value thereafter.
- `lock_bit_.load()`: loads the value of `lock_bit_`.
  - If cached value of `lock_bit_` is valid, load the value from cache.
  - Otherwise load the value from memory.
- `volatile`: a hint to the implementation to avoid aggressive optimisation involving the object.
- `_mm_pause()`: calls x86's underlying processor instructions for hinting that program is spinning.
  - Allows the processor to “do nothing” more efficiently, thereby reducing energy consumption.
  - Does *not* include OS context switch.

### 4.4 Futex and Hybrid Lock

#### 4.4.1 FUTEX\_WAIT

- Arguments: `int *p, int v`.
- Returns immediately if `*p != v`.
- Otherwise, adds thread to wait queue associated with `p`.

#### 4.4.2 FUTEX\_WAKE

- Arguments: `int *p, int wake_count`.
- Wake up `wake_count` threads that are on the wait queue for `p`.
- 1 and `INT_MAX`: the only sensible values for `wake_count`.

### 4.4.3 Simple mutex with futex

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

    /* Atomically exchange state_ with 1
     * Result is not kLocked: got the mutex! No need to call futex
     * Otherwise call FUTEX_WAIT
     */
    /* If NOW state_ == kLocked: someone else unlocked the mutex!
     * return immediately to the while loop condition to perform TAS
     * otherwise go to sleep.
     */
    void Lock() {
        while (state_.exchange(kLocked) == kLocked) {
            syscall(SYS_futex, reinterpret_cast<int*>(&state_), FUTEX_WAIT,
                    kLocked, nullptr, nullptr, 0);
        }

        /* Store 0 to state_
         * Call FUTEX_WAKE in case other threads are waiting
         * Ask FUTEX_WAKE to wake up one waiter
         */
        void Unlock() {
            state_.store(kFree);
            syscall(SYS_futex, reinterpret_cast<int*>(&state_), FUTEX_WAKE, 1,
                    nullptr, nullptr, 0);
        }

    private:
        const int kFree = 0;
        const int kLocked = 1;
        std::atomic<int> state_;
};
```

### 4.4.4 Smart futex-based mutex

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

    /* It is NOT possible to have a thread sleeping forever.
     */
```

```
    /* Neutral (N): The thread isn't interested in the lock.
     * Critical (C): The thread holds the lock.
     * Trying (T): In the do-while loop to try to acquire lock again.
     * Sleeping (S): Sleeping due to having called FUTEX_WAIT.
     */
    /* To have a thread sleeping forever, we must have either
     * C, N, N, ..., N, S and state_ == 1
     * OR
     * N, N, N, ..., N, S and state_ == 0.
     */
    /* In order to enter the state S, we must have been through:
     * C, N, N, ..., N, T or N, N, N, ..., N, T
     * and T could only be changed from N if there is already a thread in C.
     * The 1st situation implies state_ == 2 after T changes to S.
     * The 2nd situation doesn't make sense at all.
     */
    void Lock() {
        int old_value = cmpxchg(kFree, kLockedNoWaiters);
        if (old_value == kFree) {
            // We got the lock without contention - good
            return;
        }
        do {
            // Call FUTEX_WAIT if someone else got the lock.
            // Think exhaustively in this approach:
            // when old_value == x,
            // what happens when state_ is one of {0, 1, 2}\{x}?
            if (old_value == kLockedWaiters ||
                cmpxchg(kLockedNoWaiters, kLockedWaiters) != kFree) {
                syscall(SYS_futex, reinterpret_cast<int*>(&state_),
                        FUTEX_WAIT, kLockedWaiters, nullptr, nullptr, 0);
            }
            old_value = cmpxchg(kFree, kLockedWaiters);
            // whoever manages to have old_value == kFree got the lock!
        } while (old_value != kFree);
    }

    /* If state_ == 2, we think there were waiters when we locked,
     * therefore should call FUTEX_WAKE.
     * If state_ == 1, we think there were no waiters when we locked,
     * therefore unnecessary to call FUTEX_WAKE.
     * But either way, ensure that state_ is 0 when Unlock() returns.
     */
    void Unlock() {
        if (state_.fetch_sub(1) == kLockedWaiters) {
```

```

        state_.store(kFree);
        syscall(SYS_futex, reinterpret_cast<int*>(&state_), FUTEX_WAKE,
            1, nullptr, nullptr, 0);
    }
}

private:
    int cmpxchg(int expected, int desired) {
        state_.compare_exchange_strong(expected, desired);
        return expected;
    }

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

    std::atomic<int> state_;
}

```

## 5 Concurrency in Haskell

### 5.1 Thread

```

import Control.Concurrent
-- forkIO :: IO () -> IO ThreadId
import Control.Monad
import System.IO

putChars :: Char -> Int -> IO ()
putChars _ 0 = return ()
putChars c n = do
    putChar c
    putChars c (n - 1)

main = do
    hSetBuffering stdout NoBuffering
    forkIO (putChars 'B' 10000)
    putChars 'A' 10000

```

### 5.2 Join Thread

```

-- newMVar :: a -> IO (MVar a)
-- counter <- newMVar 0
-- counter :: MVar Integer

-- newEmptyMVar :: IO (MVar a)
-- handle <- newEmptyMVar
-- handle <- Any

-- takeMVar :: MVar a -> IO a
-- putMVar :: MVar a -> a -> IO ()
-- readMVar :: MVar a -> IO a
-- () is a Unit type, which has exactly one value - () itself.

import Control.Concurrent
import Control.Monad
import System.IO

thread1 :: MVar () -> IO ()
thread1 handle = do
    print "I am thread 1"
    putMVar handle ()

```

```

thread2 :: MVar () -> IO ()
thread2 handle = do
    print "I am thread 2"
    putMVar handle ()

main = do
    hSetBuffering stdout NoBuffering
    handle1 <- newEmptyMVar
    handle2 <- newEmptyMVar
    forkIO (thread1 handle1)
    forkIO (thread2 handle2)
    takeMVar handle1
    takeMVar handle2

```

### 5.3 Mutual Exclusion

```

import Control.Concurrent
import Control.Monad
import System.IO

printProtected :: MVar () -> String -> IO ()
printProtected mutex message = do
    putMVar mutex ()
    print message
    takeMVar mutex

thread1 :: MVar () -> MVar () -> IO ()
thread1 mutex handle = do
    printProtected mutex "I'm"
    printProtected mutex "thread1"
    putMVar handle ()

thread2 :: MVar () -> MVar () -> IO ()
thread2 mutex handle = do
    printProtected mutex "!I'm"
    printProtected mutex "!thread2"
    putMVar handle ()

main = do
    hSetBuffering stdout NoBuffering
    handle1 <- newEmptyMVar
    handle2 <- newEmptyMVar
    mutex <- newEmptyMVar
    forkIO (thread1 mutex handle1)

```

```

forkIO (thread2 mutex handle2)
takeMVar handle1
takeMVar handle2

```

### 5.4 Message Passing – Producer-Consumer

```

import Control.Concurrent
import Control.Monad
import System.IO

-- producer put elem one-by-one to let consumer consume
sendToConsumers :: MVar Int -> Int -> IO ()
sendToConsumers _ 0 = return ()
sendToConsumers p2c n = do
    putMVar p2c 1
    sendToConsumers p2c (n - 1)

-- get all processed elems from c2p and return the number
getFromConsumers :: MVar Int -> Int -> IO Int
getFromConsumers _ 0 = return 0
getFromConsumers c2p n = do
    val <- takeMVar c2p
    theRest <- getFromConsumers c2p (n - 1)
    return (val + theRest)

-- send elems to consumers via p2c,
-- collect the processed ones back via c2p,
-- and eventually put the total number of consumption into the result
producer :: MVar Int -> MVar Int -> Int -> Int -> MVar Int -> IO ()
producer p2c c2p numElems numConsumers result = do
    sendToConsumers p2c numElems
    combinedConsumerValues <- getFromConsumers c2p numConsumers
    putMVar result combinedConsumerValues

-- consume n data from p2c and return them
consumeData :: MVar Int -> Int -> IO Int
consumeData _ 0 = return 0
consumeData p2c n = do
    val <- takeMVar p2c
    rest <- consumeData p2c (n - 1)
    return (val + rest)

-- put the result of consumption into c2p
consumer :: MVar Int -> MVar Int -> Int -> MVar () -> IO ()

```

```

consumer p2c c2p numElems handle = do
  myResult <- consumeData p2c numElems
  putMVar c2p myResult
  putMVar handle ()

main = do
  args <- getArgs
  let numConsumers = read (args!!0) :: Int
  let elemsPerConsumer = read (args!!1) :: Int

  p2c <- newEmptyMVar
  c2p <- newEmptyMVar
  -- mapM :: Monad m => (a -> m b) -> [a] -> m [b]
  -- mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

  -- Create an MVar per consumer to serve as a handle
  consumerHandles <- mapM (\_ -> newEmptyMVar) [1..numConsumers]

  -- Fork a producer
  producerResult <- newEmptyMVar
  forkIO (producer p2c c2p (numConsumers * elemsPerConsumer)
    numConsumers producerResult)

  -- Fork numConsumers consumers
  mapM_ (\handle -> forkIO (consumer p2c c2p elemsPerConsumer handle))
    consumerHandles

  -- Join the consumers (not strictly needed)
  mapM_ takeMVar consumerHandles

  -- Join the producer
  result <- takeMVar producerResult

  -- Print the final result
  print result

```

## 5.5 Linked List

```

import Control.Concurrent
import Control.Monad
import System.IO
import System.Environment

type Stream a = MVar (Item a)

```

```

data Item a =
  Item a      -- the value
    (Stream a) -- the rest

data Channel a =
  Channel (Mvar (Stream a))
    (Mvar (Stream a))

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

readChannel :: Channel a -> IO a
readChannel (Channel readEndPtr _) = do
  readEnd <- takeMVar readEndPtr
  Item value theRest <- takeMVar readEnd
  putMVar readEndPtr theRest
  return value

writeChannel :: Channel a -> a -> IO ()
writeChannel (Channel _ writeEndptr) value = do
  writeEnd <- takeMVar writeEndptr
  newHole <- newEmptyMVar
  putMVar writeEnd (Item value newHole)
  putMVar writeEndptr newHole

```

## 6 Concurrency in Rust

### 6.1 Basics

```
struct Point {
    x: u32,
    y: u32,
}

fn show(p: &Point) {
    println!("{}", p.x, p.y);
}

fn scale(p: &mut Point, factor: u32) {
    p.x *= factor;
    p.y *= factor;
}

fn main() {
    let mut p = Point{x: 1, y: 2};
    show(&p);
    scale(&mut p, 10);
    show(&p);
}
```

### 6.2 Threading with mutex

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

fn main() {
    let max = 16777216;

    let mut data = Vec::<u32>::new();
    for _ in 0..max {
        data.push(1);
    }
    // maintain atomic reference count (Arc) to data
    let data_arc = Arc::new(data);

    let data_arc_t1 = data_arc.clone(); // increase count by 1
    let data_arc_t2 = data_arc.clone();

    // mutex protects a specific object
```

```
// Interior mutability:
// allow mutating wrapped data only if free of data races
let result_arc = Arc::new(Mutex::new(0));
let result_arc_t1 = result_arc.clone();
let result_arc_t2 = result_arc.clone();

// thread borrows by default
// child thread can outlive parent thread, thus borrowing not safe
// use move instead of borrow
let t1 = thread::spawn(move || {
    let mut result: u32 = 0;
    for i in 0..max / 2 {
        result += data_arc_t1[i];
    }
    let mut guard = result_arc_t1.lock().unwrap();
    *guard += result;
});

let t2 = thread::spawn(move || {
    let mut result: u32 = 0;
    for i in max / 2..max {
        result += data_arc_t2[i];
    }
    let mut guard = result_arc_t2.lock().unwrap();
    *guard += result;
});

t1.join().unwrap();
t2.join().unwrap();

println!("{}", result_arc.lock().unwrap());
}
```

### 6.3 Threading with Atomics

```
use std::thread;

use std::sync::Arc;
use std::sync::atomic::{AtomicU32, Ordering};

fn main() {
    let max = 16777216;

    let mut data = Vec::<u32>::new();
```

```

for _ in 0..max {
    data.push(1);
}
let data_arc = Arc::new(data);

let data_arc_t1 = data_arc.clone(); // increase count by 1
let data_arc_t2 = data_arc.clone();

let result_arc = Arc::new(AtomicU32::new(0));
let result_arc_t1 = result_arc.clone();
let result_arc_t2 = result_arc.clone();

// thread borrows by default
// child thread can outlive parent thread, thus borrowing not safe
// use move instead of borrow
let t1 = thread::spawn(move || {
    let mut result: u32 = 0;
    for i in 0..max / 2 {
        result += data_arc_t1[i];
    }
    result_arc_t1.fetch_add(result, Ordering::SeqCst);
});

let t2 = thread::spawn(move || {
    let mut result: u32 = 0;
    for i in max / 2..max {
        result += data_arc_t2[i];
    }
    result_arc_t2.fetch_add(result, Ordering::SeqCst);
});

t1.join().unwrap();
t2.join().unwrap();

println!("{}", result_arc.load(Ordering::SeqCst));
}

```

## 6.4 Striped Hashset

```

use std::collections::hash_map::DefaultHasher
use std::cmp::Eq;
use std::hash::{Hash, Hasher};
use std::sync::Mutex;
use std::sync::atomic::{AtomicUsize, Ordering};

```

```

pub struct StripedHashSet<T: ToString + Eq + Hash> {
    bucket_groups: Vec<Mutex<Vec<Vec<T>>>>,
    size: AtomicUsize,
    num_buckets: AtomicUsize,
}

impl<T: ToString + Eq + Hash> StripedHashSet<T> {
    pub fn new(capacity: usize) -> Self {
        let mut bucket_groups = Vec::new();
        for _ in 0..capacity {
            let mut bucket_group = Vec::new();
            bucket_group.push(Vec::new());
            bucket_groups.push(Mutex::new(bucket_group));
        }
        return StripedHashSet {
            bucket_groups,
            size: AtomicUsize::new(0),
            num_buckets: AtomicUsize::new(capacity),
        }
    }

    pub fn add (&self, elem: T) -> bool {
        {
            let mut hasher = DefaultHasher::new();
            elem.hash(&mut hasher);
            let hashcode : usize = hasher.finish() as usize;

            let bucket_group_index = hashcode % self.bucket_groups.len();
            let mut bucket_group_guard =
                self.bucket_groups[bucket_group_index].lock().unwrap();
            let bucket_index = hashcode % bucket_group_guard.len();

            if bucket_group_guard[bucket_index].contains(&elem) {
                return false;
            }
            bucket_group_guard[bucket_index].push(elem);
            self.size.fetch_add(1, Ordering::SeqCst);
        }
        if result && self.policy() {
            self.resize();
        }
        return true;
    }
}

```



```

pub fn remove (&self, elem: T) -> bool {
    let mut hasher = DefaultHasher::new();
    elem.hash(&mut hasher);
    let hashcode : usize = hasher.finish() as usize;

    let bucket_group_index = hashcode % self.bucket_groups.len();
    let mut bucket_group_guard =
        self.bucket_groups[bucket_group_index].lock().unwrap();
    let bucket_index = hashcode % bucket_group_guard.len();
    for i in 0..bucket_group_guard[bucket_index].len() {
        if elem == bucket_group_guard[bucket_index][i] {
            bucket_group_guard[bucket_index].swap_remove(i);
            self.size.fetch_sub(1, Ordering::SeqCst);
            return true;
        }
    }
    return false;
}

fn policy (&self) -> bool {
    return self.size.load(Ordering::SeqCst) /
        self.num_buckets.load(Ordering::SeqCst) > 4;
}

fn resize (&self) {
    // get all the mutices - put the resulting guards into a vector
    let mut guards = Vec::new();
    for i in 0..self.bucket_groups.len() {
        guards.push(self.bucket_groups[i].lock().unwrap());
    }

    if !self.policy() {
        return;
    }

    // we will put all the data in the hash set here
    let mut data = Vec::new();
    for i in 0..guards.len() {
        for j in 0..guards[i].len() {
            while !guards[i][j].isEmpty() {
                data.push(guards[i][j].swap_remove(0));
            }
        }
        let buckets_per_group = guards[i].len();
        for _ in 0..buckets_per_group {

```

```

            guards[i].push(Vec::new());
        }
    }

    // now redistribute the data into the hash set
    while !data.is_empty() {
        let elem = data.swap_remove(0);

        let mut hasher = DefaultHasher::new();
        elem.hash(&mut hasher);
        let hashcode : usize = hasher.finish() as usize;

        let bucket_group_index = hashcode % self.bucket_groups.len();
        let bucket_index = hashcode % guards[bucket_group_index].len();

        guards[bucket_group_index][bucket_index].push(elem);
    }

    // update the number of buckets
    self.num_buckets.store(self.num_buckets.load(Ordering::SeqCst) * 2,
        Ordering::SeqCst);
}

impl <T: ToString + Eq + Hash> ToString for StripedHashSet<T> {
    fn to_string(&self) -> String {
        let mut result: String = "[".to_owned();
        let mut first = true;

        for i in 0..self.bucket_groups.len() {
            let guard = self.bucket_groups[i].lock().unwrap();
            for j in 0..guard.len() {
                for k in 0..guard[j].len() {
                    if !first {
                        result.push_str(", ");
                    }
                    first = false;
                    result.push_str(guard[j][k].to_string());
                }
            }
        }
        result.push_str("]");
        return result;
    }
}

```

## 7 Dynamic Data Race Detection (Vector Clock-based)

### 7.1 Definitions and Assumptions

- $N$ : maximum number of threads tracked by algorithm.
- $\{0, 1, \dots, N-1\}$ : set of thread ids, denoting *Threads*.
- Synchronisation is via a set of locks, denoting *Locks*.
- *Locations* is a set of possibly-shared memory locations.
- Logical clock  $\in \mathbb{N}$ : increases each time the thread releases a mutex.
- Vector clock  $V \triangleq (c_0, c_1, \dots, c_{N-1})$ : a tuple of  $N$  logical clocks, where

$$V(t) = c_t.$$

- $VC$  is the set of all vector clocks.
- Bottom vector clock  $\perp \triangleq (0, 0, \dots, 0)$ .
- Partial order on  $V$ :

$$V_1 \sqsubseteq V_2 \iff \forall t. V_1(t) \leq V_2(t).$$

- Join of vector clocks:

$$V_1 \sqcup V_2 = \text{pointwise maximum of } V_1 \text{ and } V_2.$$

- Increment:

$$\text{inc}_t(V) = V[t \mapsto V(t) + 1].$$

### 7.2 Vector Clock Algorithm State

- $C : \text{Threads} \mapsto VC$ 
  - $C_t$ : shorthand for  $C(t)$ , represents what thread  $t$  knows about the  $VC$ .
  - $C_t(t)$  is  $t$ 's logical clock.
  - For  $u \neq t$ ,  $C_t(u) = z$  means “thread  $t$  knows that thread  $u$ 's logical clock is at least  $z$ .”
  - When  $t$  acquires a lock,  $t$  gets information about the logical clocks of threads who previously held the lock.

- $L : \text{Locks} \mapsto VC$ 
  - $L_m$ : shorthand for  $L(m)$ , represents the logical clock each thread has last time it released  $m$ .
  - $L_m(t) = 0$  means that  $t$  has never released  $m$ .
  - Otherwise  $L_m(t)$  was  $t$ 's logical clock last time  $t$  releases  $m$ .
- $R : \text{Location} \mapsto VC$ 
  - $R_x$ : shorthand for  $R(x)$ , represents the logical clock each thread had last time it read from  $x$ .
  - $R_x(t) = 0$  means that  $t$  has never read from  $x$ .
  - Otherwise  $R_x(t)$  was  $t$ 's logical clock last time  $t$  read from  $x$ .
- $W : \text{Location} \mapsto VC$ 
  - $W_x$ : shorthand for  $W(x)$ , represents the logical clock each thread had last time it wrote to  $x$ .
  - $W_x(t) = 0$  means that  $t$  has never written to  $x$ .
  - Otherwise  $W_x(t)$  was  $t$ 's logical clock last time  $t$  wrote to  $x$ .

### 7.3 Initial Analysis State

- $C = (\text{inc}_0(\perp), \text{inc}_1(\perp), \dots, \text{inc}_{N-1}(\perp))$
- $L = \lambda m. \perp$
- $R = \lambda x. \perp$
- $W = \lambda x. \perp$

### 7.4 Intercepted Operations

- **rd**( $\mathbf{t}$ ,  $\mathbf{x}$ ): interception of a read by thread  $t$  from possibly-shared memory location  $x$ .

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

- **wr**( $\mathbf{t}$ ,  $\mathbf{x}$ ): interception of a write by thread  $t$  to possibly-shared memory location  $x$ .

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

- $\text{acq}(\mathbf{t}, \mathbf{m})$ : interception of an acquire of  $m$  by  $t$ .

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

- $\text{rel}(\mathbf{t}, \mathbf{m})$ : interception of a release of  $m$  by  $t$ .

$$\frac{L' = L[m \mapsto C_t] \quad C' = C[t \mapsto \text{inc}_t(C_t)]}{(C, L, R, W) \xrightarrow{\text{rel}(\mathbf{t}, \mathbf{m})} (C', L', R, W)}$$

## 7.5 Race Rules

- Write-Read race rule:

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

- Write-Write race rule:

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

- Read-Write race rule:

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