Motivation

A data race occurs when there are at least two concurrent accesses to the same memory location where

- at least one of them is a write and
- at least two of them are unsynchronized [1]

Example

Consider the following program written in C++:

```
#include <pthread.h>
#include <iostream>

const int iter_count = 100000;

long int Global;
void *Thread1(void *x)
{
  Global = 42;
  return x;
}

int main()
{
  pthread_t t;
  pthread_create(&t, NULL, Thread1, NULL);
  Global = 43;
  std::cout << Global << std::endl;
  pthread_join(t, NULL);
  return 0;
}
```

Here, in the main thread a second thread is spawned. Both threads have access to the global variable `Global`. After the spawning of the second thread, the main thread sets the global variable to the value of 43, while the spawned thread attempts to set the same variable to the value 42. As they are trying to do this multiple times (e. g. in a loop), there is a high chance, that the execution times of these threads overlap. Therefore, both threads may attempt to write to the same memory location at the same time. This may lead to unintended or undefined behaviour; it is not determined whether the output statement

```
std::cout << Global << std::endl;
```

will print 42 or 43, as this depends on the order of execution of the two write operations setting the variable to 42 and 43 respectively and the read operation required for printing the value of the variable.

Data Race Detection Techniques

We want to detect race conditions as the one in the example above in order to debug multithreaded programs and ensure their correct and predictable behaviour. There are several approaches to achieve this.[1]

- **Static**: Analyze the source code of a program at compile time
- **Dynamic**: Work with traces of actual program executions
  - **On-the-fly**: Buffers and analyzes partial trace information, detects races during program execution
  - **Postmortem**: Saves the trace of the program and analyzes it after execution

Static

Static data race detection techinques analyze the source code of the program at compile-time.

Advantages

- Checks the program globally for possible race data races
- Granular analysis of the data involved in possible data races, as they can analyze at the level of variables, structures and classes of the programming language
- As these analysis happens at compile-time, the performance of the program at run-time is not negatively effected at all

Disadvantages

- Static analysis is generally slow, as finding data races is a NP-hard problem in the general case (source).
- Static analysis generates excessive false alarms, masking the important real data races.
- Static analysis may not handle some language features well, such as dynamic class loading in Java.

Dynamic

Dynamic data race detection is based on actual executions of the program. During the execution, a *trace* of a particular is created (see below), stored and the execution is evaluated based on that trace.

Advantages

- Only data races that actually occured during real executions are detected (no false alarms).
- Computationally less complex than static race detection.

Disadvantages

- As the behaviour multithreaded programs may change between several executions, dynamic analysis might miss data races based on only one execution. Therefore, multiple runs of the program may be required in order to detect data races.
- Incurs a big performance hit while running the program.

Trace

A *trace* α is a sequence of operations performed by variaous threads during the execution of a multithreaded program. In this context, the set of operations a thread t can perform are: [3]

- $r(t, x)$: read value from a variable x
- $w(t, x)$: write value to a variable x

- $acq(t, m)$: acquire a lock m
- $rel(t, m)$: release a lock m
- $fork(t, u)$: fork a new thread u
- $join(t, u)$: block thread t until thread u terminates

We will discuss in later sections how traces can be used to determine (data) race conditions.

FastTrack and ThreadSanitizer

$FastTrack$ is dynamic data race detection technique presented in [3]. It is the theoretical basis of the ThreadSanitizer implemented in the LLVM project.

In order to understand $FastTrack$, we will first look at Lamports happens-before relation, Lamport clocks, vector clocks and the $DJIT+$ algorithm FastTrack is based on.

Lamport's Happens-Before Relation

Lamport was originally interested in a consistent (partial) order of all events happening in a distributed system. A distributed system is a system with multiple spatially separated processes running simultaniously. These processes communicate with each other via messages. Due to their spatial separation, the message transmission delay is not negligible compared to the time between events in a single process. This concept of a distributed system can be generalised and statements regarding distributed systems also apply to the case of a multithreaded program.

In such systems, a notion of an event *happening before* another is important for a multitude of scenarios. For example when two processes want to access an exclusive resource, we need a way to grant access to only one of the processes at a time (mutual exclusion). This however is not possible wihtout a strict ordering on the requests for this resource.

Happens-before relation

Consider a distributed system with mutliple processes sending between and receiving messages from each other. Sending and receiving a message is an event in a process. Lamport defines the happens-before relation, denoted by $\leq$, as follows [4]

Definition *happens-before*

The relation $\leq$ on the set of events of a system is the smallest relation satisfying the three conditions:

1. If a and b are events in the same process and a comes before b, then $a \leq b$
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \leq b$
3. If $a \leq b$ and $b \leq c$, then $a \leq c$
4. $a \leq a$ for any event a

Two events are said to be **concurrent** if $a \leq b$ and $b \leq a$.

This definition of a happens-before relation allows us to make statements about the causality of two events; If $a \leq b$, it is possible that a casually affected b, whereas when $a \leq b$ a it is not possible that a casually affected b.

Application in data race detection

We can also make use of the happens-before relation above to detect data races in traces α of program executions. In order to do that, the release and acquisition of the lock now take the role of sending and receiving a message, while threads of the program are now equivalent to processes in distributed systems.

Releasing the lock m by thread t_1

$$rel(t_1, m)$$

is the equivalent of a process sending a message, while the acquisition the same lock m by thread t_2

$$acq(t_2, m)$$

after it has been released by t_1 is equivalent to the reception of the same message by another process. Therefore if $rel(t_1, m)$ appears before $acq(t_2, m)$ in the program trac, according to the definition we have $rel(t_1, m) \leq acq(t_2, m)$.

Sources

- Lamports happens-before relation: https://lamport.azurewebsites.net/pubs/time-clocks.pdf

Lamport Clocks and Vector Clocks

Lamport clocks

A Lamport clock is a function

$$C : E \rightarrow T$$

that assigns timestamps $C(t) \in T$ to any event $e \in E$ while fulfilling the *weak clock-condition*, whose definition is based on the *happens-before* relation from above:

$$e \leq e^{'} \rightarrow C(e) < C(e^{'})$$

In his paper, Lamport proposes a mechanism that fulfills this condition. Each process is assigned a clock, with a counter value representing the current local "time". In the beginning they are all set to the same value t_0.

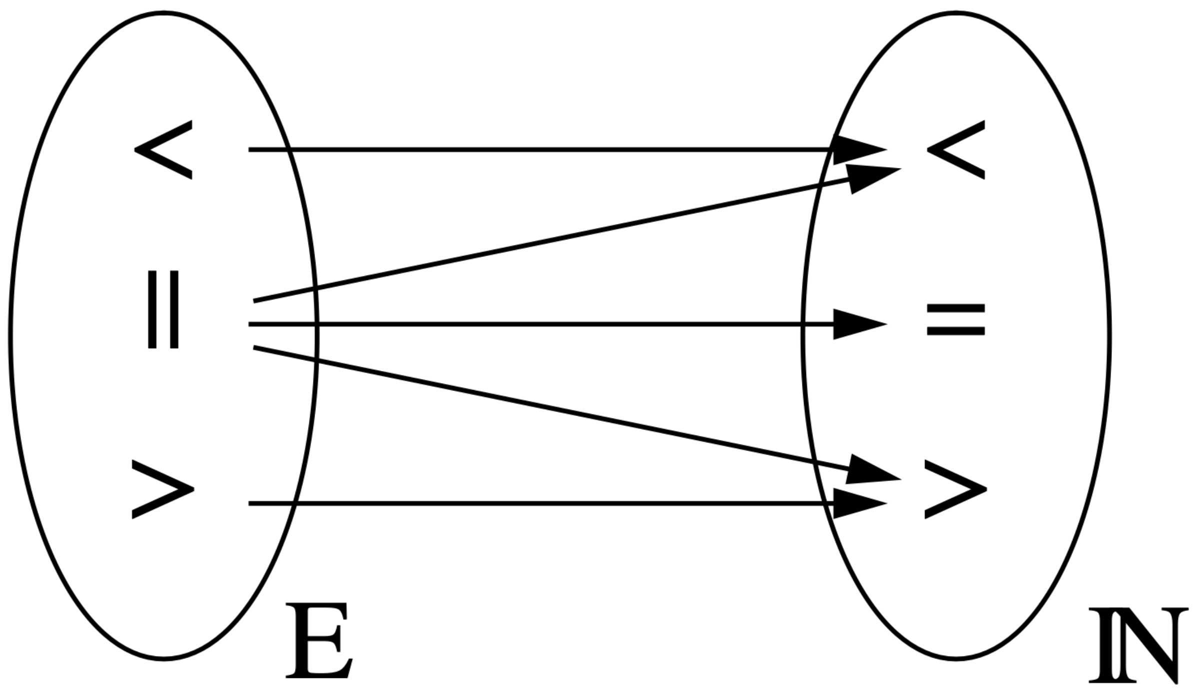
1. A process increments its counter before each local event, e. g. when sending a message
2. When a process sends a message, it includes its counter value with the message after executing step 1
3. On receiving a message, the counter of the recipient is set to the greater of its current counter value and the counter value received in the timestamp, and subsequently incremented by 1

Shortcoming of the Lamport clock

Consider two events e, $e^{'}$, that are not casually related ($e \| e^{'}$). As a result they might get assigned the same or different timestamps which can lead to three different outcomes:

- $C(e) < C(e^{'}) \rightarrow e \leq e^{'}$
- $C(e^{'}) > C(e) \rightarrow e^{'} \leq e$
- $C(e) = C(e) \rightarrow (e \leq e^{'}) \wedge (e^{'} \leq e)$

The last statement however destroys the information about the two events *not* casually influencing each other, as it states that e *might have* casually influenced $e^{'}$ and $e^{'}$ *might have* casually influenced e. This is the exact opposite of the definition of two events *not* casually influencing each other. We can see from the figure below that the Lamport clock is a homomorphism, not an isomorphism:



In order to preserve information about two events *not* casually influencing each other, we would need a way to store the negation of the *happens-before* relation $\neg(e \leq e^{'})$. With this, the information of two events *not* casually influencing each other could be represented by the expression

$\neg(e \leq e^{'}) \wedge \neg(e^{'} \leq e)$

Vector clocks achieve this by replacing the single logical clock per thread with a vector of logical clocks (*vector clock*) per thread, with one vector entry for each thread of the program.

Vector clocks

A vector clock is an extension of Lamport clocks. They are both logical clocks that provide a total ordering of events consistent with causality. However, vector clocks can also determine if any two events are causally dependent or concurrent. To achieve this, each process has a vector clock that stores one value for each process's logical clock.

Suppose we have a system of three processes: P0, P1 and P2. Each process has a vector clock that stores three values: one for each process's logical clock. Initially, all clocks are zero: [0, 0, 0]. Whenever a process experiences an internal event (such as executing a statement), it increments its own logical clock by one. For example, if P0 executes an internal event, its vector clock becomes [1, 0, 0]. Whenever a process sends a message to another process, it attaches its current vector clock to the message. When a process receives a message from another process, it updates its vector clock by taking the maximum of each element in its own vector clock and the vector clock in the message. Then it increments its own logical clock by one. For example, if P1 sends a message with vector clock [0, 2, 0] to P2 and P2 has vector clock [1, 1 ,3], then after receiving the message P2 updates its vector clock to [1, 2, 3] (the maximum of each element) and then increments its own logical clock by one: [1, 2, 4].

A vector clock C_1 is less than C_2, $C_1 < C_2$, if all its elements are less or equal to the respective elements in C_2 ($C_1(t) \leq C_2(t) \forall t$) and at least one element is strictly smaller ($\exists t : C_1(t) < C_2(t)$).

The happens-before relation for vector clocks is defined based on this definition: An event a *happened-before* an event b, if the vector clock of event a C_a is less then that of event b C_b: a happened before b ⇔ $C_a < C_b$.

Notably the relation $<$ is antisymmetric: If $C_1 < C_2$, then $C_2 < C_1$. This is the crucial property of vector clocks that allows them to preserve information about two events *not* casually influencing each other.

Example

Using this algorithm, we can determine the partial ordering of events in the system and detect causality violations. For example, suppose we have these events:

- A1: P0 executes an internal event ([1, 0, 0])
- A2: P1 executes an internal event ([0, 1, 0])
- A3: P2 executes an internal event ([0, 0, 1])
- B1: P0 sends a message to P1 ([1, 0, 0])
- B2: P1 receives the message from P0 ([1, 1, 0])
- B3: P2 sends a message to P1 ([0, 0, 2])
- B4: P1 receives the message from P2 ([1, 2, 2])

We can say that $A1 < B4$ because A1 is causally before B4 (P0 sent a message to P1 that influenced B4). We can also say that $A3 < B4$ because A3 is causally before B4 (P2 sent a message to P1 that influenced B4). However, we cannot say that $A2 < B2$ or $B2 > A2$ because A2 is concurrent with B2 (there is no causal relationship between them).

Sources

- Lamport clocks: https://lamport.azurewebsites.net/pubs/time-clocks.pdf
- Vector clocks: https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

DJIT+

The DJIT+ algorithm is a method for detecting data races in multithreaded program, that is based on the vector clocks (VC) discussed before. A

To identify data races, i.e. concurrent conflicting accesses, a *happens-before* relationship has to be established between the actions of different threads. To that end, the `DJIT+` algorithm uses vector clocks.

The algorithms maintains vector clocks for three types of objects:

- **threads** with vector clocks C_t
- **variables** with vector clocks W_{x_i} and R_{x_i} for read and write accesses respectively
- **locks** with vector clocks L_m

The number of elements in these vectors equals the number of threads, each element representing one thread.

Initial values

- The vector clocks of the threads are initialized with 1 for all entries
- The vector clocks of the variables are initialized with 0 for all entries, indicating no previous access by any thread
- The vector clocks for locks start with an initial value of 0 for all entries, indicating no previous access by any thread

Time frames

DJIT+ splits the program execution of each thread t in *time frames*.

A new time frame starts each time the corresponding thread performs a release operation $rel(t, m)$ on a lock m.

For a given time frame it is sufficient to

- log only the first read and write access to any given location, as any subsequent accesses *by the same thread* are ordered by program order
- check for races only between accesses to the same shared location, which are the first in their respective time frames, as they may violate the happens-before relation and therefore are potential candidates for concurrent accesses

This is the basic idea of the DJIT+ algorithm and allows to reduce the number of checks performed to detect data races.

Updating the vector clocks

1. New time frame (release of a lock)

Action: Thread t releases a lock m: $rel(t, m)$ (i.e. a new time frame begins for thread t)

Result: Two updates are applied to the vector clocks:

1. The t'th element of the vector clock C_t of thread t is incremented: $C_t[t] \leftarrow C_t[t] + 1$.

2. The vector clock L_m is set to the supremum of itself and the vector clock C_t of thread t (each entry of the vector clock L_m of lock m is updated to hold the maximum between the current value and that of t's vector):

   $L_m \leftarrow sup(L_m, C_t)$

   $\Leftrightarrow$

   $\forall i : L_m[i] \leftarrow max(L_m[i], C_t[i])$

2. Acquiring a lock

Action: Thread t acquires a lock m: $acq(t, m)$

Result: Vector clock C_t of thread t is updated so that each entry in its vector holds the maximum between its current value and that of the vector clock L_m:

$C_t \leftarrow sup(L_m, C_t)$

$\Leftrightarrow$

$\forall i : L_m[i] \leftarrow max(L_m[i], C_t[i])$

3. First read/write of a variable in a time frame

Action: Thread t accesses variable x_i the first time in the current time frame

- Read: $r(t, x_i)$
- Write: $w(t, x_i)$

Result:

- Read
  - $R_{x_i}[t] \leftarrow C_t[t]$
  - A read of x_i is race-free if it happens after the last write of each thread. For this to be guaranteed, the write has to be protected by a lock. Therefore, if another thread u released a lock before writing to a variable x_i, it's a possible data race.

    As a thread t releasing a lock m causes the the vector clocks of the lock to be updated with C_t, and thread u acquiring a lock m causes C_u to be joined with the vector clock of the lock, this establishes a happens-before relation between the read/write operation between different threads. Therefore the algorithm checks the condition

    $W_{x_i}[u] \geq C_u[u]$

    If this condition is true, it reports a possible data race, as it means that thread u wrote to x_i *after* releasing its last lock.
- Write
  - $W_{x_i}[t] \leftarrow C_t[t]$
  - The algorithm checks whether there was a thread u that released a lock before reading or writing to variable x_i. Per the same argument as above, it checks the whether the conditions

    $W_{x_i}[u] \geq C_u[u]$

or

$R_{x_i}[u] \geq C_u[u]$

are true for another thread u.

If so, report a data race.

Example

Let’s consider a program with two threads, T1 and T2, that access two shared memory locations, x and y. The program is as follows:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
int x = 0;
int y = 0;
int z = 0;

void T1() {
    m.lock();
    x = 1;
    m.unlock();
    y = 2;
}

void T2() {
    z = x + y;
    m.lock();
    m.unlock();
}

int main() {
    std::thread t1(T1);
    std::thread t2(T2);

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

    std::cout << "z: " << z << std::endl;

    return 0;
}
```

- Each thread has its own vector clock, C_{T_1} and C_{T_2}, which are both initialized to [1, 1].
- The variables/memory locations x and y also have vector clocks R_x, R_y, R_z, W_x, W_y, W_z which are initialized to [0, 0].

The algorithm will perform the following steps:

1. T_1 acquires lock m. As the locks vector clock has the value $[0,0]$, the vector clock of T_1 stays at the value $[0,0]$.

2. T_1 writes to x. This is the first write access to x in this time frame. The algorithm logs this access and updates W_x to $[1, 0]$, indicating a write access of T_1 in its first time frame.
3. T_1 releases lock m. This is an event that updates its vector clock to $[2, 1]$. A new time frame for thread T_1 begins.
4. T_1 writes to y. This is the first write access to y in this time frame. The algorithm logs this access and updates W_x to $[2, 0]$, indicating a write of T_2 in its second time frame.
5. T_2 reads from x. This is the first read access to x in the current time frame of T_2. Two things happen:
   - The algorithm logs this access by updating $R_x[T_2] \leftarrow C_{T_2}[T_2] = 1$.
   - The algorithm checks whether thread T_1 released a lock before writing to x. As $W_x[T_1] = 1 < C_{T_1}[T_1] = 2$, this is not the case. Hence, no data race is reported.
6. T_2 reads from y. This is the first read access to y in the current time frame of T_2. Two things happen:
   - The algorithm logs this access by updating $R_y[T_2] \leftarrow C_{T_2}[T_2] = 1$.
   - The algorithm checks whether thread T_1 released a lock before writing to $<$. As $W_<[T_1] = 2 \geq C_{T_1}[T_1] = 2$, this is the case.

     Therefore a data race is reported.
7. T_2 acquires lock m. This is an event that updates its vector clock to [2, 1].
8. T_2 releases lock m. This is an event that updates its vector clock to [2, 2].
9. ...

Sources

- Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs

- FastTrack paper: https://users.soe.ucsc.edu/~cormac/papers/pldi09.pdf
- Vector Clocks paper: https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

Fast Track

FastTrack builds on DJIT+, but improves on it. The key observation made in the paper is that using vector clocks is not necessary to establish happens-beforerelations for most of the operations executed in the investigated program. It achieves better performance by tracking less information and dynamically adapting its representation of the happens-before relation based on memory access patterns.

To achieve that, FastTrack uses a hybrid representation of vector clocks that combines an epoch (a pair of a thread identifier and a logical time) with a full vector clock. An epoch represents a single thread's view of time, while a full vector clock represents multiple threads' views of time. FastTrack uses epochs whenever possible to reduce space and time overheads, and switches to full vector clocks only when necessary to maintain precision.

As DJIT+ and other vector clock based race detectors require O(n) storage and O(n) time for comparing, copying and joining vector clocks for n threads, using only an epoch instead of a full vector clock significantly improves performance.

Epochs and vector clocks

An epoch in FastTrack is defined as the logical clock c of a given thread t, denoted as $c@t$. It therefore is equivalent to the t'th component of the vector clock for thread t $C_t[t]$ in the DJIT+ algorithm. It can therefore be stored as a single value.

Comparing epochs and vector clocks

The definition of a happens-before relation between an epoch $c@t$ and a vector clock C can be deduced from the definition of an epoch above:

$$c@t \leq C \Leftrightarrow c \leq C[t]$$

In contrast to comparing two vector clocks, which requires $O(n)$ time, comparing an epoch and a vector clock only requires $O(1)$ time.

Usage of epochs and vector clocks in FastTrack

As with DJIT+, full *vector clocks* are used for

- every thread t: C_t
- every lock m: L_m

For variables however, FastTrack mainly uses *epochs*.

- When a thread t writes to a variable x, W_x just records the epoch $c@t$ of the write:. Hence, the variable x stores only the information about the logical clock of the last thread that has written to it.
- When a thread t reads from a variable x, R_x may record the epoch of this read or update its full vector clock, dependent on the circumstances:
  - If all reads are totally ordered, i.e. the reads happen thread-local or lock-protected, R_x is just the epoch of the last read operation

- If the reads are not totally ordered, i.e. the reads happen from different threads and without locking, R_x is a vector clock that is the join of all reads of x

Detecting race conditions

Acquiring locks

Acquiring and releasing locks is handled exactly as in the DJIT+ algorithm, as both the threads and locks have vector clocks.

Read operations

When a thread t reads from a variable x, FastTrack distinguishes between four cases.

- **Subsequent reads from the same epoch**: This happens, when the reads happen within the same and epoch. The reads might be from the same thread or from different threads, as long as their logical clock values are the same. In this case, R_x is an epoch and R_x is just updated to the current read epoch. No race conditions may occur in this case.

  These cases make up 78% of all read operations and require nearly no computing resources at all. If this case is true, no further computations or checks need to be done and the handling of the read operation by FastTrack is finished.

After this simple check, FastTrack checks for a write-read race. Let t be the thread reading variable x at epoch $s@t$. If $W_x \leq C_t \Leftrightarrow W_x \leq s$, meaning the last write was in the epoch before or the same as this read, no race condition is detected. Otherwise, a write-read race is raised.

How the read operation is processed from now on depends on whether the variable is shared or not. Whether a variable is shared or not is determined by the algorithm itself.

Let's first look at the two cases where the variable is not shared. We assume, that $R_x = q@u$, i.e. that the last read of variable x was by thread u in epoch $q@u$ and that the current read is done by thread t in epoch $s@t$.

- **Last read happened in epoch before current read epoch**: This is the case when $R_x \leq C_t \Leftrightarrow u \leq C_t[u]$, i.e. if the last read on the variable happened in an epoch *before* the epoch of the current read (the same epoch is handled in the first case above), either by the same thread or another thread. In this case, FastTrack simply updates the read epoch of x to the epoch of the currently reading thread: $R_x \leftarrow s@t$.
- **Last read happened in epoch after current read epoch**: This is the case if the last read on the variable happened in an epoch *after* the epoch of the current read. This means that the current read may be concurrent to the previous read in another epoch. In this case, the variable x is marked as *shared* and the R_x becomes a full vector clock instead of an epoch. This new vector clock has value 0 for all elements except the clock elements for the thread u of the previous read, which is set to q, and for the thread t, which is set to s. This is importang as either thread u or t could subsequently participate in a read-write race, and we need to track the epochs of their last reads on x in order to check for those races when writing to variable (see below). This incurs a higher memory usage in order to store the vector clock for the variable.

When a variable is marked as read shared by the third rule above, i.e. when its R_x is a vector clock, the following case comes into play:

- **Read on a shared variable**: The vector clock element $R_x[t]$ corresponding to the reading thread t is set to t's current logical clock value s: $R_x[t] = s$. This incurs a higher memory usage in order to store the vector clock for the variable.

Write operations

We assume that the previous write on variable x is $W_x = q@u$, i.e. was done by thread u in epoch q, and that the current write operation is done by thread t in epoch $s@t$.

FastTrack distinguishes between three cases and checks for several possible race conditions.

- **Writing in the same epoch**: When $W_x = s@t$, i.e. when the same threads subsequently writes to the same variable in the same epoch, no race condition is possible and nothing has to be done.

  This case makes up roughly 71% of write operations and require nearly no computing resources at all. If this case is true, no further computations or checks need to be done and the handling of the read operation by FastTrack is finished.

When this simple case is not true, FastTrack checks for a write-write race: If $W_x > C_t \Leftrightarrow q > C_t[u]$, i.e. the last write on the variable happened in an epoch *after* the epoch of the current write, a **write-write race condition is raised**, as this means that two subsequent writes happened without synchronization.

After that, FastTrack distinguishes between read-exclusive and read-shared variables (see above).

- **Write to a read-exclusive variable**: When writing to a read-exclusive variable , R_x is an epoch. FastTrack first checks for a read-write race: If $R_x > C_t$, i.e. if the last read of variable x happened by another thread in an epoch *after* the current write epoch $s@t$, this means that the read may return unexpected values, as the variable may have been changed before the read by another thread without proper synchronization and **read-write race is raised**.
- **Write to a read-shared variable**: When writing to a read-shared variable x, R_x is a full vector clock. FastTrack first checks for a shared-write race: If either $R_x > C_t \Leftrightarrow q > C_t[u]$ or $W_x > C_t$, i.e. if the last read or write of the variable x happened in an epoch *after* the current write epoch $s@t$, the current write is not properly synchronized and may interfere with writes and/or reads from other threads, altering values that are not expected to be altered by the other thread, therefore leading to an overwrite of those values by the other thread or to unexpected values for the other thread when reading the variable. Therefore a **shared-write data race is raised**.

If in neither of those two cases a data race is detected, W_x is simply updated to the current epoch of the write operation $W_x = s@t$.

Example

Let's consider the same program as in the DJIT+ example, T1 and T2, that access two shared memory locations, x and y. The program is as follows:

```
#include <iostream>
#include <thread>
#include <mutex>
```

```
std::mutex m;
int x = 0;
int y = 0;
int z = 0;

void T1() {
    m.lock();
    x = 1;
    m.unlock();
    y = 2;
}

void T2() {
    z = x + y;
    m.lock();
    m.unlock();
}

int main() {
    std::thread t1(T1);
    std::thread t2(T2);

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

    std::cout << "z: " << z << std::endl;

    return 0;
}
```

- The threads 1 and 2 have their own vector clock, C_{T_1} and C_{T_2}, which are both initialized with [0, 0].
- The variables/memory locations x and y have epochs R_x, R_y, W_x, W_y for reading and writing initialized with 0.
- The lock has a vector clock that is initialized with [0, 0].

The algorithm will perform the following steps:

1. T_1 acquires lock m. As the locks vector clock L_m has the value $[0, 0]$, the vector clock of T_1 stays at the value $[0, 0]$.

   New state:

   - $C_1 = [0, 0]$
   - $C_2 = [0, 0]$
   - $L_m = [0, 0]$
   - $R_x = 0@0$, $W_x = 0@0$
   - $R_y = 0@0$, $W_y = 0@0$
   - $R_z = 0@0$, $W_z = 0@0$
2. T_1 writes to x. The algorithm logs this access and updates W_x to $0@1$, indicating a write access of T_1 in its first epoch.

New state:

- $C_1 = [0,0]$
- $C_2 = [0,0]$
- $L_m = [0,0]$
- $R_x = 0@0$, $W_x = 0@1$
- $R_y = 0@0$, $W_y = 0@0$
- $R_z = 0@0$, $W_z = 0@0$

3. T_1 releases lock m. Releasing the lock sets its vector clock to $C_{T_1}, indicating that thread 1 release the lock in epoch $1@1$$: $L_m \leftarrow C_{T_1} = [1,0]$. Additionally, the epoch of thread t is incremented and its vector clock is updated to $[1,0]$, as a new time frame for thread T_1 begins.

   New state:

   - $C_1 = [1,0]$
   - $C_2 = [0,0]$
   - $L_m = [0,0]$
   - $R_x = 0@0$, $W_x = 0@1$
   - $R_y = 0@0$, $W_y = 0@0$
   - $R_z = 0@0$, $W_z = 0@0$
4. T_1 writes to y. The algorithm checks for a read-write race, but as $R_y = 0@0 \leq 2@T_1$, no race is detected. It then logs this access by updating W_x to $2@1$, indicating a write of T_1 in time frame 2.

   New state:

   - $C_1 = [2,1]$
   - $C_2 = [1,1]$
   - $L_m = [1,0]$
   - $R_x = 0@0$, $W_x = 2@1$
   - $R_y = 0@0$, $W_y = 0@0$
   - $R_z = 0@0$, $W_z = 0@0$
5. T_2 reads from x (as part of the operation `int z = x + y`). As this happens in epoch $0@2$ and $R_x = 0@0$ (no previous read), the algorithm just updates to reflect this read access: $R_x \leftarrow 0@2$. It then checks for a write-read race. As $W_x = 2@1 > 0@2 = C_{T_2}[T_2]$, **a write-read race is detected in reported**.

   New state:

   - $C_1 = [2,1]$
   - $C_2 = [1,1]$
   - $L_m = [1,0]$
   - $R_x = 0@2$, $W_x = 2@1$
   - $R_y = 0@2$, $W_y = 0@0$
   - $R_z = 0@0$, $W_z = 0@0$
6. ...

Sources:

- FastTrack: efficient and precise dynamic race detection

- The FastTrack 2 Race Detector - Technical Report