ACQUIRE rule

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

Update the thread's vector clock with the mutex's vector clock, in order to know what other threads have release the mutex (using the pointwise maximum join operator).

RELEASE rule

$$\frac{\begin{array}{c} L' = L[m \mapsto C_t] \\ C' = C[t \mapsto inc_t(C_t)] \end{array}}{(C, L, R, W) \rightarrow^{rel(t,m)} (C', L', R, W)}$$

The releasing thread t makes the world aware of its logical clock value by updating the L_m vector clock and then take a step forward by incrementing the logical clock of t.

READ rule

$$\frac{\begin{array}{c} W_x \sqsubseteq C_t \\ R' = R[x \mapsto R_x[t \mapsto C_t(t)]] \end{array}}{(C, L, R, W) \rightarrow^{rd(t,x)} (C, L, R', W)}$$

First, check that there are no read-write races (make sure that we know that another thread has already written to x and are therefore ordered), and then update t's component of the R_x vector clock with the thread's logical value, letting the world know the time at which thread t last read from x.

WRITE rule

$$\frac{\begin{array}{c} W_x \sqsubseteq C_t \qquad R_x \sqsubseteq C_t \\ W' = W[x \mapsto W_x[t \mapsto C_t(t)]] \end{array}}{(C, L, R, W) \rightarrow^{wr(t,x)} (C, L, R, W')}$$

Check that there are no write-write or read-write races and then update t's component of the W_x vector clock with the thread's logical value, letting the world know the time at which thread t last wrote to x.

When processing a rd() or wr() instruction, if the pre-condition does not apply, then we have a data race.

Write-read data race rule

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

Happens when t reads from x and we find that another thread u has a greater value in its W_x vector clock compared to the current thread t’s knowledge of u.
Summary: u has written to x before t reads from x, but t is not aware of this earlier write.

Read-write data race rule

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

Happens when t writes to x and we find that another thread u has a greater value in its R_x vector clock compared to the current thread t’s knowledge of u.
Summary: u has read from x before t writes to x, but t is not aware of this earlier read.

Write-Write data race rule

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

Happens when t writes to x and we find that another thread u has a greater value in its W_x vector clock compared to the current thread t’s knowledge of u.
Summary: u has written to x before t writes to x, but t is not aware of this earlier write.

Aside: extending support for atomic RMWs (from tutorial sheet)

$$\frac{D = C_t \sqcup L_o \qquad L' = L[o \mapsto D] \qquad C' = C[t \mapsto inc_t(D)]}{(C, L, R, W) \rightarrow^{RMW(t,o)} (C', L', R, W)}$$

The idea here is to combine the effects of acquire, followed by release. This means first joining the object’s vector clock into the execution thread’s vector clock, then updating the atomic object to match the thread’s new vector clock (because we just used it) and then increment the thread’s clock (like we do when releasing a mutex).

For more on atomics, see tutorial sheet 7 but in summary:

We can think of a load from atomic object o as acquiring from o, and a store to o as releasing from o.

Thus the events *acq* and *rel*, which we previously considered to apply to locks, can also be considered to apply to atomic objects. For a thread t and atomic object o:

- event *acq*(t, o) corresponds to t loading from o
- event *rel*(t, o) corresponds to t storing to o

We also extend the mapping L so that it includes atomic locations rather than just mutexes.