1 Proof

This section gives the formal proof of Janus. Before giving the proof we first provide some prerequisites:

Definition 1. *Start timestamp / End timestamp* of a transaction T: the value of T.startTS and T.endTS

Definition 2. *Timestamp* of an object x: the *end timestamp* of the latest transaction who updates *x*.

Definition 3. *Concurrent transactions:* Any two transactions, T_i and T_j, are concurrent transactions, if T_i's start timestamp is smaller than T_j's end timestamp and T_j's start timestamp is smaller than T_i's end timestamp

Definition 4. *Snapshot Write:* Assume transaction T_i writes an object *x*. The write is Snapshot Write if there is no concurrent transaction which also writes object *x*.

Definition 5. *Snapshot Read:* Assume transaction T_i reads an object *x*. The read is Snapshot Read if:

1. T_i's start timestamp is greater than or equal to *x*'s timestamp
2. Do not exist a transaction T_j which also writes object *x* and its end timestamp is greater than *x*'s current timestamp[1], but not greater than T_i's start timestamp.

According to the specification of *Snapshot Isolation* [1], To prove Janusprovides *Snapshot Isolation* we need to prove following theorm:

Theorem: In Janus, each transactional read is a **Snapshot Read** and each transactional write is a **Snapshot Write**.

Proof. Following is the formal proof. We first prove each write in Janusis *Snapshot Write*, then prove each read is *Snapshot Read*. The line number in the proof is related to the pseudocode in Algorithm 1.

1. All writes are ***Snapshot Write***.

Assume two transaction T_i and T_j update the same object *x*, we prove that T_i's end timestamp is not greater than T_j's start timestamp or an inverse.

1.1 T_i and T_j update *x* exclusively.

1.1.1 To write an object, the transaction needs to acquire the lock
by line 26.

1.1.2 A transaction releases the acquired locks at the end of the transaction.
by line 58.

1.1.3 Q.E.D.
by 1.1.1, 1.1.2

1.2 If T_i updates *x* before T_j, then T_i's end timestamp is not greater than T_j's start timestamp.

1.2.1 CASE 1. T_j sets start timestamp (line 4) after T_i updates global timestamp.
by OBVIOUS.

1.2.2 CASE 2. T_j sets start timestamp before T_i updates global timestamp.

1.2.2.1 T_j can not start write back phase if an ative transaction's start timestamp is smaller than its end timestamp
by lines 52 - 54

1.2.2.2 T_j releases all its locks at the end of write back phase
by line 58

1.2.2.3 T_j can not release *x*'s lock until T_i aborts or commits
by 1.2.2, 1.2.2.1, 1.2.2.2

1.2.2.4 T_i aborts itself if it failed to acquire a lock
by lines 26, 27

[1] current timestamp means the timestamp of *x* when T_i performs the read

Q.E.D.
by 1.2.2.3, 1.2.2.4

Q.E.D.
by 1.2.1, 1.2.2

1.3 Q.E.D.
by 1.1, 1.2

2. All reads are ***Snapshot Read***.

Assume a transaction T_i reads an object x and x's timestamp is TS_x when T_i performs its read.

2.1 T_i's start timestamp is greater than or equal to x's timestamp (TS_x)

2.1.1 CASE 1: T_i's Read function returns x's current copy (lines 10, 18)

2.1.1.1 After T_i starts, a transaction T_j whose end timestamp is larger than T_i's start timestamp can not update x's current copy until T_i become inactive (commit or abort).
by 1.2.2.1

2.1.1.2 Q.E.D.
by 2.1.1.1

2.1.2 CASE 2: T_i's Read function returns x's next copy (lines 13, 16)

2.1.2.1 The content of *next* copy is updated by *wtx* transaction read at line 11

2.1.2.1.1 A transaction assign a real number to its end timestamp after it finishes all updates
by OBVIOUS

2.1.2.1.2 T_i is able to read the *next* copy only if the transaction *wtx* is itself or its end timestamp is assigned
by line 12, 15

2.1.2.1.3 *next* can not be reclaimed until T_i commits
by RCU garbage collection

2.1.2.1.4 Q.E.D.
by 2.1.2.1.1, 2.1.2.1.2, 2.1.2.1.3

2.1.2.2 Q.E.D.
by 2.1.2.1, line 15

2.1.3 Q.E.D.
by 2.1.1, 2.1.2

2.2 Assume a transaction T_j which updates x and its end timestamp is not greater than T_i's start timestamp, prove T_j's end timestamp can not be greater than TS_x.

2.2.1 T_i gets its start timestamp (line 4) after T_j calculates its end timestamp with the global timestamp (line 45)
by Assumption

2.2.2 T_i reads x after T_j calculates the end timestamp
by 2.2.1

2.2.3 CASE 1: T_i reads x before T_j writes back its update to x (line 58)

2.2.3.1 the *next* reference read by T_i at line 8 is the copy of x in T_j's log

2.2.3.1.1 T_j linkes its copy to x's next before its commit phase
by OBVIOUS

Q.E.D.
by 2.2.2, 2.2.3.1.1

2.2.3.2 T_i can not read the *next* until T_j's end timestamp is visible by T_i

2.2.3.2.1 If T_i observes *inCritical* is false (line 46), the end timestamp updated at line 45 is visible to T_i.
by Intel specification (Write-write do not reorder).

2.2.3.2.2 T_i needs to wait for *inCritical* to become false before read T_j's end timestamp (line 14-15).

2.2.3.2.3 Q.E.D.
by 2.2.3.2.1, 2.2.3.2.2

2.2.3.2 TS_j's end timestamp is equalto TS_x
by 2.2.3.1, Assumption

Q.E.D.
by 2.2.3.2

2.2.4 CASE 2: T_i reads x after T_j writes back its update to x (line 58)

2.2.4.1 The timestamp of x is monotonically increasing
by 1.1, 1.2

2.2.4.2 The copy of x read by T_i is updated by T_j or later transactions
by *TM_Read* function

2.2.4.3 Q.E.D.
by 2.2.4, 2.2.4.1, 2.2.4.2

Q.E.D.
by 2.2.2, 2.2.3, 2.2.4

□

References

[1] Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. 1999.