Computer Science Technical Report

Illustrative test cases for the UPC memory model

by William Kuchera and Charles Wallace

Michigan Technological University
Computer Science Technical Report
CS-TR-03-02
March 14, 2003

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Illustrative test cases for the UPC memory model*

William Kuchera and Charles Wallace
Michigan Technological University

March 14, 2003

1 Introduction

The memory model underlying UPC is an important but subtle aspect of the language, and everyone involved with it must understand its implications. The only resource currently available with detailed coverage of the memory model is the high-level description in the UPC specification [1]. As mentioned in our previous report [3], it is difficult to tie this description to actual behavior of a UPC program. UPC implementors and application developers must be able to readily distinguish program behavior that complies with the UPC specification from behavior that does not comply. For instance, implementors of UPC on a particular platform need to ensure that the optimizations they employ guarantee compliant behavior. Programmers exploiting relaxed consistency must have a grasp of what possible behaviors their programs may induce.

To this end, we have devised a set of test cases for the UPC memory model. These cases fall into two categories:

Compliance tests. These are examples of behavior that falls outside the UPC specification. They illustrate the consistency guarantees that UPC gives the programmer.

"Dark corner" tests. These are examples of acceptable behavior, according to the UPC specification, that may be surprising to the UPC novice. The UPC memory model is designed to be quite lax and allows some cases that programmers may not anticipate. These examples serve to highlight some of these "dark corners".

Our test cases are at the architecture level rather than the program level; that is, each thread's execution is expressed in terms of a sequence of read, write, fence, notify, and wait instructions, rather than UPC statements. Any references to "ordering", "following", "intervening", *etc.* refer to this thread-by-thread instruction ordering. Read and write instructions are assumed to be relaxed unless specified otherwise. For each case, we illustrate why it illustrates compliance or non-compliance, using the operational semantics devised in our previous report [3].

*Financial support for this work has been provided by Hewlett Packard.

2 Memory model overview

The UPC memory model places an order on operations on a per-thread basis. In this regard, it is similar to *processor consistency* [2]. There is no mechanism for ordering operations by different threads. This makes the UPC memory model strictly weaker than sequential consistency [4]. Strict operations by a single thread are linearly ordered; relaxed operations are ordered with respect to strict operations but unordered with respect to one another. The lack of a linear order on all operations by a single thread makes the UPC memory model strictly weaker than processor consistency.

The constraints on local reading (*i.e.,* a thread's reading a value that it wrote) are much stricter than those on remote reading (*i.e.,* a thread's reading a value that another thread wrote). A local read always returns the most recent value written by the thread. A thread remotely reading must follow the ordering of operations established by the writing thread.

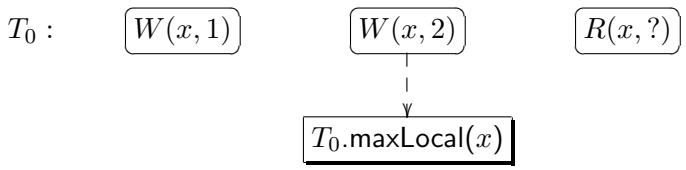
Notify and wait, the so-called "split barrier" operations, bring threads to a consensus on the remotely visible operations for each thread. When each T_i completes its wait, its view of operations by each T_j is updated to include operations all the way up to T_j's notify. Notify and wait do not order operations by different threads, however, so the operations of T_i and those of T_j remain unordered.

The UPC memory model does not have the *coherence* property [2, 5]. Coherence states that there exists a linear order of writes to a single location that is agreed upon by all threads. This condition fails in the UPC memory model for two reasons. First, writes by different threads, even those to a common location, are never ordered. Second, relaxed writes by a single thread, even those to a single location, are never ordered.

3 Compliance test cases

Compliance Test 1 *If ordered writes are followed by a local read, only the latest write may be read.*

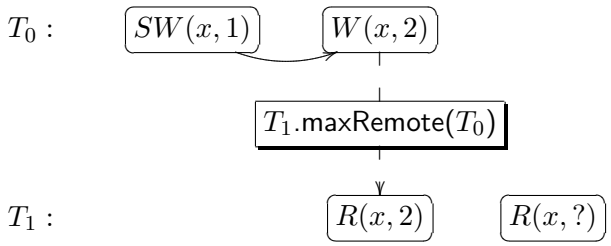
Thread 0: write(x,1); write(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation.
At the time of T_0's read, t_0.maxLocal(x) = W(x,2).
The only write to x locally readable by T_0 is maxLocal(x).
So W(x,1) is not legal for this read.

Compliance Test 2 *A strict write and a following relaxed write, if read by a remote thread, must be read in order.*

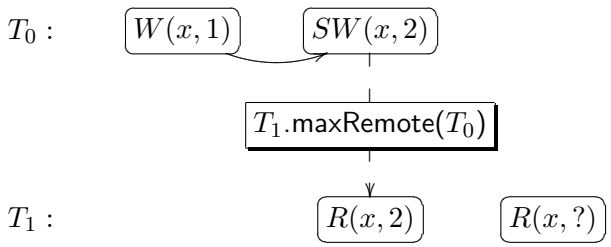
Thread 0: strict-write(x,1); write(x,2)
Thread 1: read(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation.
T_0's write W(x,2) is ordered after its strict write SW(x,1).
T_1's first read updates T_1.maxRemote(T_0) to W(x,2).
Thus at the time of T_1's second read, we have SW(x,1) $\prec$ W(x,2) = T_1.maxRemote(T_0).
So SW(x,1) is not legal for this read.

Compliance Test 3 *A relaxed write and a following strict write, if read by a remote thread, must be read in order.*

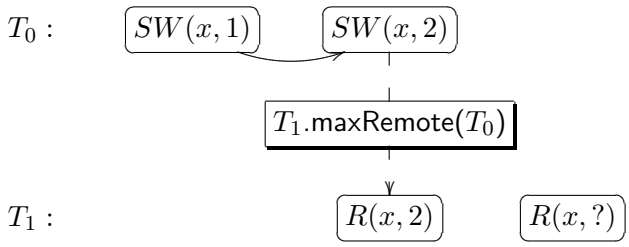
Thread 0: write(x,1); strict-write(x,2)
Thread 1: read(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation.
T_0's strict write SW(x,2) is ordered after its write W(x,1).
T_1's first read updates T_1.maxRemote(T_0) to SW(x,2).
Thus at the time of T_1's second read, we have W(x,1) $\prec$ SW(x,2) = T_1.maxRemote(T_0).
So W(x,1) is not legal for this read.

Compliance Test 4 *Ordered strict writes, if read by a remote thread, must be read in order.*

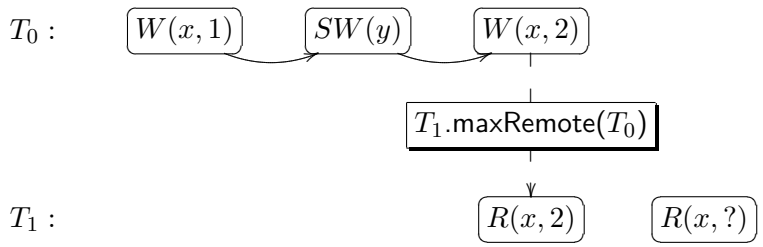
Thread 0: strict-write(x,1); strict-write(x,2)
Thread 1: read(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation.
T_0's strict write SW(x,2) is ordered after its write SW(x,1).
T_1's first read updates T_1.maxRemote(T_0) to SW(x,2).
Thus at the time of T_1's second read, we have SW(x,1) $\prec$ SW(x,2) = T_1.maxRemote(T_0).
So SW(x,1) is not legal for this read.

Compliance Test 5 *Ordered relaxed writes, if read by a remote thread, must be read in order, if a strict write intervenes between the writes.*

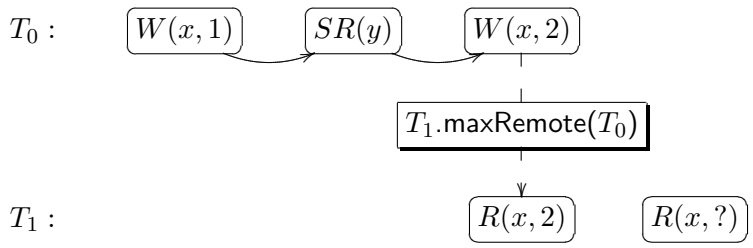
Thread 0: write(x,1); strict-write(y); write(x,2)
Thread 1: read(x,2); **read(x,1) $\Leftarrow$ 1 not a legal value**



Explanation.
T_0's strict write SW(y) is ordered after its write W(x,1).
T_0's write W(x,2) is ordered after SW(y).
T_1's first read updates T_1.maxRemote(T_0) to W(x,2).
Thus at the time of T_1's second read, we have W(x,1) $\prec$ SW(y) $\prec$ W(x,2) = T_1.maxRemote(T_0).
So W(x,1) is not legal for this read.

Compliance Test 6 *Ordered relaxed writes, if read by a remote thread, must be read in order, if a strict read intervenes between the writes.*

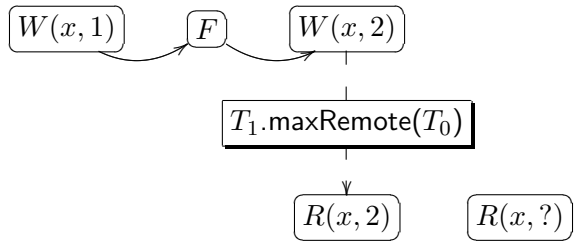
Thread 0: write(x,1); strict-read(y); write(x,2)
Thread 1: read(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation.
T_0's strict read SR(y) is ordered after its write W(x,1).
T_0's write W(x,2) is ordered after SR(y).
T_1's first read updates T_1.maxRemote(T_0) to W(x,2).
Thus at the time of T_1's second read, we have W(x,1) $\prec$ SR(y) $\prec$ W(x,2) = T_1.maxRemote(T_0).
So W(x,1) is not legal for this read.

Compliance Test 7 *Ordered relaxed writes, if read by a remote thread, must be read in order, if a fence intervenes between the writes.*

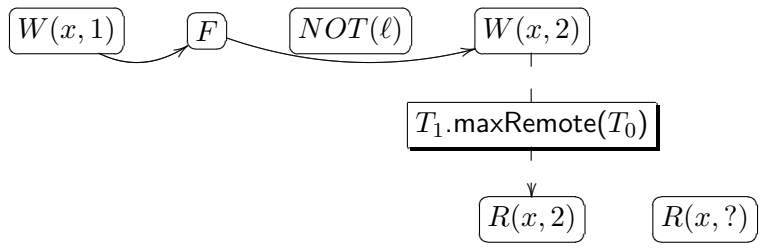
Thread 0: write(x,1); fence; write(x,2)
Thread 1: read(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation.
T_0's fence F is ordered after its write W(x,1).
T_0's write W(x,2) is ordered after F.
T_1's first read updates T_1.maxRemote(T_0) to W(x,2).
Thus at the time of T_1's second read, we have W(x,1) $\prec$ F $\prec$ W(x,2) = T_1.maxRemote(T_0).
So W(x,1) is not legal for this read.

Compliance Test 8 *Ordered relaxed writes, if read by a remote thread, must be read in order, if a notify intervenes between the writes.*

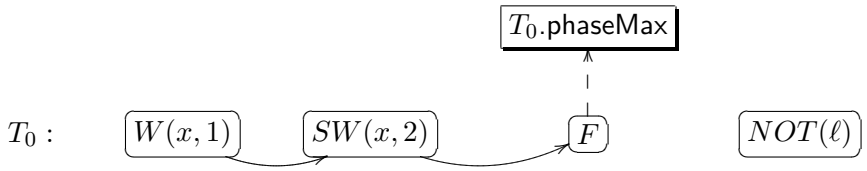
Thread 0: write(x,1); notify(ℓ); write(x,2)
Thread 1: read(x,2); **read(x,?) $\Leftarrow$ 1 not a legal value**



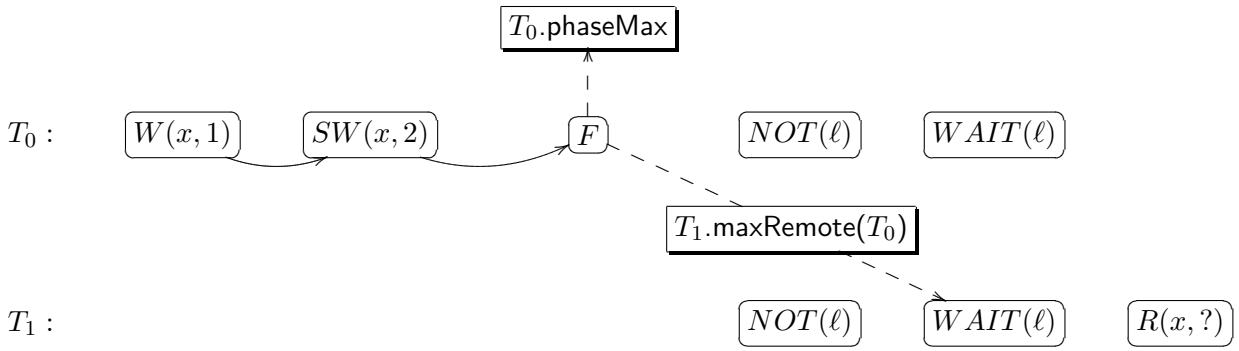
Explanation.
T_0's notify is preceded by an implicit fence F.
F is ordered after T_0's write W(x,1).
T_0's write W(x,2) is ordered after F.
T_1's first read updates T_1.maxRemote(T_0) to W(x,2).
Thus at the time of T_1's second read, we have W(x,1) $\prec$ F $\prec$ W(x,2) = T_1.maxRemote(T_0).
So W(x,1) is not legal for this read.

Compliance Test 9 *If a relaxed write and a following strict write precede a notify and corresponding wait, only the strict write may be read after the wait.*

Thread 0: write(x,1); strict-write(x,2); notify(ℓ); wait(ℓ)
Thread 1: notify(ℓ); wait(ℓ); **read(x,?) $\Leftarrow$ 1 not a legal value**



Explanation (part 1).
T_0's notify is preceded by an implicit fence F.
F is ordered after T_0's strict write SW(x,2).
SW(x,2) is ordered after T_0's write W(x,1).
At the time of T_0's notify, T_0.phaseMax is updated to F.

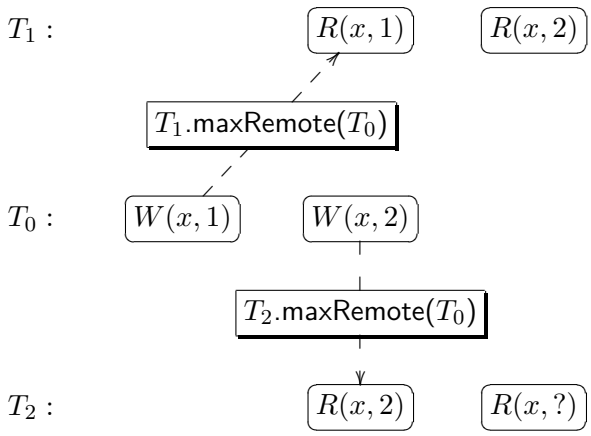


Explanation (part 2).
When T_1's wait completes, T_1.maxRemote(T_0) is updated to T_0.phaseMax, which is F.
Thus at the time of T_1's read, we have W(x,1) $\prec$ SW(x,2) $\prec$ F = T_1.maxRemote(T_0).
So W(x,1) is not legal for this read.

4 "Dark corners" test cases

Dark Corner Test 1 *Ordered relaxed writes may be remotely read in different orders by different threads.*

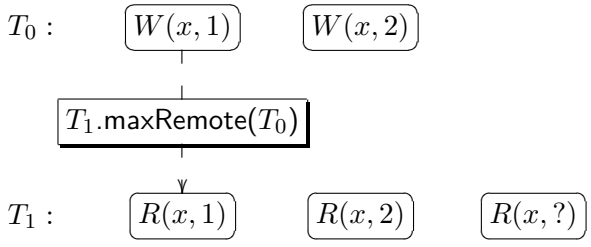
Thread 0: write(x,1); write(x,2)
Thread 1: read(x,1); read(x,2)
Thread 2: read(x,2); **read(x,?) $\Leftarrow$ 1 is a legal value**



Explanation.
W(x,1) and W(x,2) are unordered with respect to each other.
T_1's first read updates T_1.maxRemote(T_0) to W(x,1).
At T_1's second read, since there is no chain W(x,2) $\prec^+$ W(x,*), W(x,2) is readable.
T_2's first read updates T_2.maxRemote(T_0) to W(x,2).
At T_2's second read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is legal for this read.

Dark Corner Test 2 *Ordered relaxed writes may be remotely read in different orders by a single thread.*

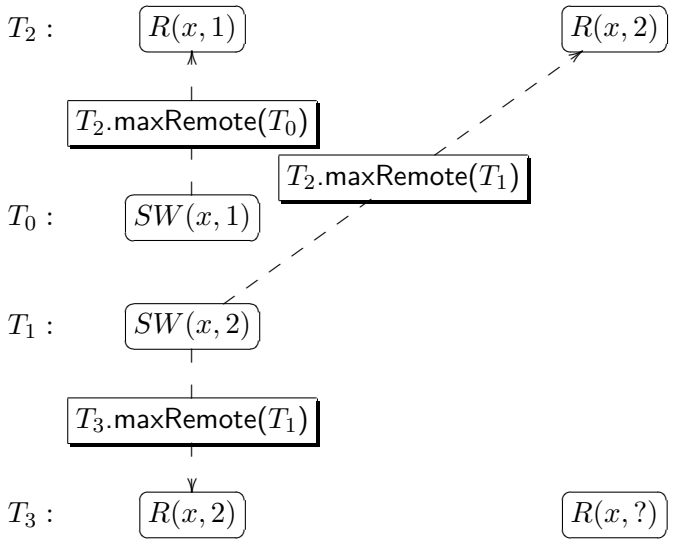
Thread 0: write(x,1); write(x,2)
Thread 1: read(x,1); read(x,2); **read(x,?) $\Leftarrow$ 1 is a legal value**



Explanation.
W(x,1) and W(x,2) are unordered with respect to each other.
T_1's first read updates T_1.maxRemote(T_0) to W(x,1).
At T_1's second read, since there is no chain W(x,2) $\prec^+$ W(x,*), W(x,2) is readable.
Since W(x,2) $\not\prec^+$ W(x,1), T_1.maxRemote(T_0) is unchanged by T_1's second read.
At T_1's third read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.

Dark Corner Test 3 *Unordered strict writes may be read in different orders by different threads.*

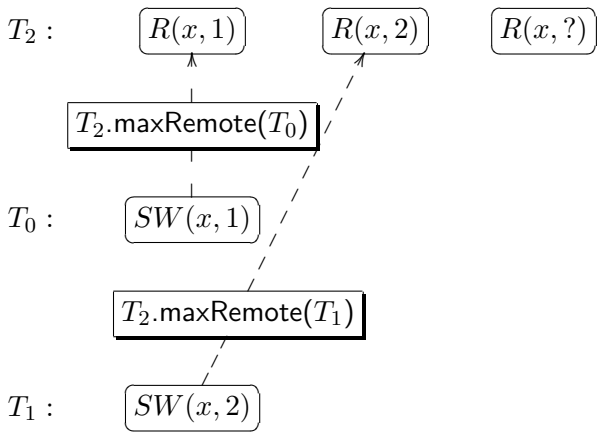
Thread 0: strict-write(x,1)
Thread 1: strict-write(x,2)
Thread 2: read(x,1); read(x,2)
Thread 3: read(x,2); **read(x,?) $\Leftarrow$ 1 is a readable value**



Explanation.
SW(x,1) and SW(x,2) are unordered with respect to each other.
T_2's first read updates T_2.maxRemote(T_0) to SW(x,1).
At T_2's second read, since there is no chain SW(x,2) $\prec^+$ W(x,*), SW(x,2) is readable.
T_2's second read updates T_2.maxRemote(T_1) to SW(x,2).
T_3's first read updates T_3.maxRemote(T_1) to SW(x,2).
At T_3's second read, since there is no chain SW(x,1) $\prec^+$ W(x,*), SW(x,1) is readable.

Dark Corner Test 4 *Unordered strict writes may be read in different orders by a single thread.*

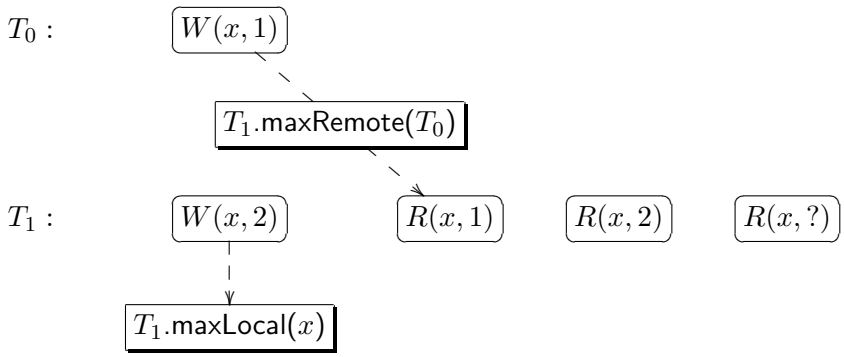
Thread 0: strict-write(x,1)
Thread 1: strict-write(x,2)
Thread 2: read(x,1); read(x,2); **read(x,?) $\Leftarrow$ 1 is a legal value**



Explanation.
SW(x,1) and SW(x,2) are unordered with respect to each other.
T_2's first read updates T_2.maxRemote(T_0) to SW(x,1).
At T_2's second read, since there is no chain SW(x,2) $\prec^+$ W(x,*), SW(x,2) is readable.
T_2's second read updates T_2.maxRemote(T_1) to SW(x,2).
At T_2's third read, since there is no chain SW(x,1) $\prec^+$ W(x,*), SW(x,1) is readable.

Dark Corner Test 5 *Unordered writes, if one is read locally and the other remotely by the same thread, may be read in different orders by that thread.*

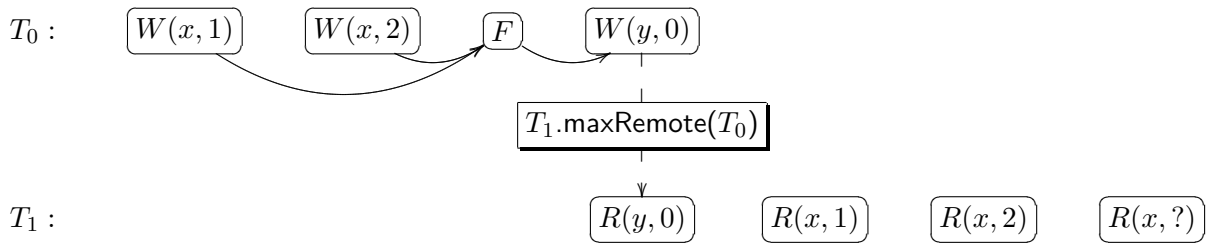
Thread 0: write(x,1)
Thread 1: write(x,2); read(x,1); read(x,2); **read(x,?) $\Leftarrow$ 1 is a legal value**



Explanation.
T_1's write updates T_1.maxLocal(x) to W(x,2).
T_1's first read updates T_1.maxRemote(T_0) to W(x,1), but leaves T_1.maxLocal(x) unchanged.
Since T_1.maxLocal(x) is always a legal readable value, W(x,2) is legal for this read.
T_1's second read has no effect on T_1.maxRemote(T_0) = W(x,1).
So W(x,1) is still a legal readable value.

Dark Corner Test 6 *If ordered relaxed writes precede a fence, the writes may be read in different orders by a single thread.*

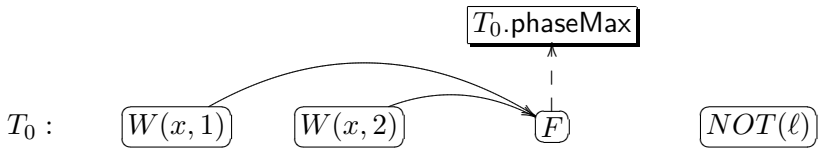
Thread 0: write(x,1); write(x,2); fence; write(y,0);
Thread 1: read(y,0); read(x,1); read(x,2); **read(x,?) $\Leftarrow$ 1 is a legal value**



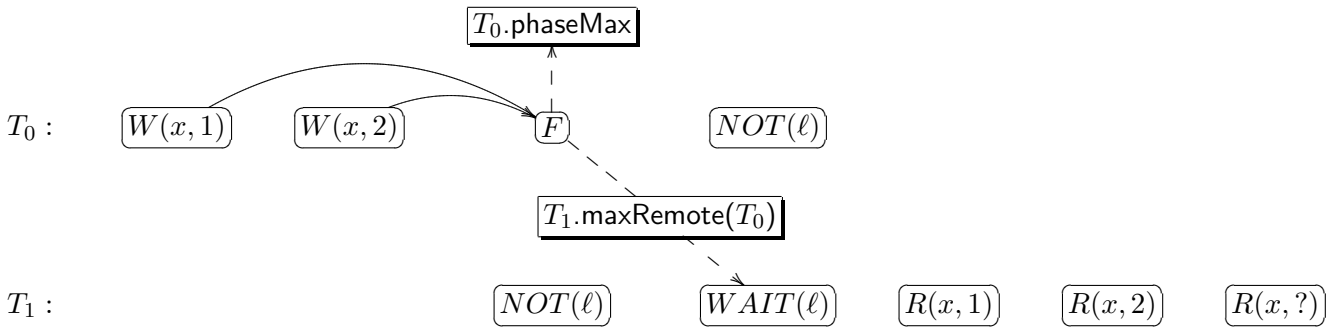
Explanation.
W(x,1) and W(x,2) are unordered with respect to each other.
Both W(x,1) and W(x,2) are ordered before F.
F is ordered before W(y,0).
T_1's first read updates T_1.maxRemote(T_0) to W(y,0).
At T_1's second read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.
Since T_1.maxRemote(T_0) = W(y,0) and W(y,0) $\succ$ W(x,1), T_1.maxRemote(T_0) is unchanged.
At T_1's third read, since there is no chain W(x,2) $\prec^+$ W(x,*), W(x,2) is readable.
Since T_1.maxRemote(T_0) = W(y,0) and W(y,0) $\succ$ W(x,2), T_1.maxRemote(T_0) is unchanged.
At T_1's fourth read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.

Dark Corner Test 7 *If ordered relaxed writes precede a notify and corresponding wait, the writes may be read in different orders by a single thread.*

Thread 0: write(x,1); write(x,2); notify(ℓ); wait(ℓ)
Thread 1: notify(ℓ); wait(ℓ); read(x,1); read(x,2); **read(x,?) $\Leftarrow$ 1 is a legal value**



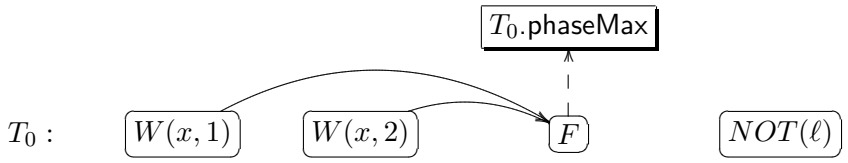
Explanation (Part 1).
W(x,1) and W(x,2) are unordered with respect to each other.
Both W(x,1) and W(x,2) are ordered before F.
T_0's notify updates T_0.phaseMax to F.



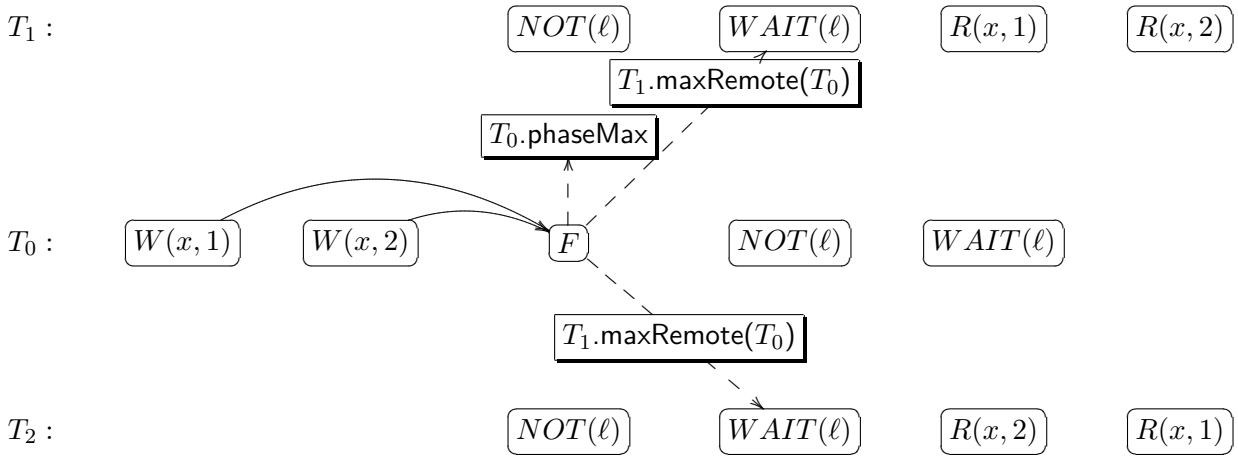
Explanation (Part 2).
T_1's wait updates T_1.maxRemote(T_0) to T_0.phaseMax, which is F.
At T_1's first read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.
Since T_1.maxRemote(T_0) = F and F $\succ$ W(x,1), T_1.maxRemote(T_0) is unchanged.
At T_1's second read, since there is no chain W(x,2) $\prec^+$ W(x,*), W(x,2) is readable.
Since T_1.maxRemote(T_0) = F and F $\succ$ W(x,2), T_1.maxRemote(T_0) is unchanged.
At T_1's third read, since there is still no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.

Dark Corner Test 8 *If ordered relaxed writes precede a notify and corresponding wait, the writes may be read in either order by different threads.*

Thread 0: write(x,1); write(x,2); notify(ℓ); wait(ℓ)
Thread 1: notify(ℓ); wait(ℓ); read(x,1); read(x,2)
Thread 2: notify(ℓ); wait(ℓ); read(x,2); read(x,1)



Explanation (part 1).
W(x,1) and W(x,2) are unordered with respect to each other.
Both W(x,1) and W(x,2) are ordered before F.
T_0's notify updates T_0.phaseMax to F.



Explanation (part 2).
When T_1's wait completes, T_1.maxRemote(T_0) is updated to T_0.phaseMax, which is F.
At T_1's first read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.
Since T_1.maxRemote(T_0) = F and F $\succ$ W(x,1), T_1.maxRemote(T_0) is unchanged.
At T_1's second read, since there is no chain W(x,2) $\prec^+$ W(x,*), W(x,2) is readable.
When T_2's wait completes, T_2.maxRemote(T_0) is updated to T_0.phaseMax, which is F.
At T_2's first read, since there is no chain W(x,2) $\prec^+$ W(x,*), W(x,2) is readable.
Since T_1.maxRemote(T_0) = F and F $\succ$ W(x,2), T_1.maxRemote(T_0) is unchanged.
At T_2's second read, since there is no chain W(x,1) $\prec^+$ W(x,*), W(x,1) is readable.

References

[1] T. El-Ghazawi, W. Carlson, and J. Draper. UPC language specifications, v1.0. Technical report, Center for Computing Sciences, 2001.
Available at `http://www.gwu.edu/~upc/doc/upc_specs.pdf`.

[2] J.R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Science Dept., University of Wisconsin–Madison, 1989.

[3] W. Kuchera and C. Wallace. Toward a programmer-friendly formal specification of the UPC memory model, 2002.

[4] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[5] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A new memory model for architects and compiler writers. In *Proc. ISCA*, pages 150–161, 1999.