Visibility and Happens-before ordering

Michael Hicks

September 23, 2013

1 Happens before

"Happens before" is a partial order describing program events, invented by Leslie Lamport in 1978. The notion of happens before is critical for understanding what writes are visible to what reads in a multi-threaded execution, according to the Java memory model, and for understanding precisely the notion of a data race.

Definition 1 (Execution trace). Consider multithreaded executions as traces \( R \) of events \( E \), as defined below.\(^1\) (A trace is just a sequence.)

\[
\text{Events } E \ ::= \quad \text{start}(T) \\
\quad \text{end}(T) \\
\quad \text{read}(T, x, v) \\
\quad \text{write}(T, x, v) \\
\quad \text{spawn}(T_1, T_2) \\
\quad \text{join}(T_1, T_2) \\
\quad \text{lock}(T, x) \\
\quad \text{unlock}(T, x)
\]

Here \( T \) is a thread identifier, \( x \) is a variable, and \( v \) is a value. For example, the event \( \text{read}(T, x, v) \) indicates that thread \( T \) read value \( v \) from (global/shared) variable \( x \). We assume that traces \( R \) are well-formed by requiring the first event by a thread \( T \) in \( R \) must be \( \text{start}(T) \). No events by \( T \) may follow \( \text{end}(T) \) in the trace.

In Java, the \( \text{lock}(T, x) \) and \( \text{unlock}(T, x) \) events arise when a thread \( T \) executes the start and end, respectively, of a synchronized block on monitor \( x \). The \( \text{join}(T_1, T_2) \) event occurs after a thread \( T_1 \) completes execution of statement \( T_2, \text{join}() \). The \( \text{spawn}(T_1, T_2) \) event occurs when thread \( T_1 \) executes the statement \( T_2, \text{start}() \). We refer to all these events (i.e., those other than read and write events) as synchronization events since they enforce an ordering of events between threads.\(^2\)

Note that we have not included many interesting program events in traces (like additions, jumps, calls/returns, etc.) because they have no bearing on happens-before—think of them as no-ops for our purposes.

Definition 2 (Event ordering). Let \( E_1 \prec E_2 \) be the ordering of events as they appear in the trace, which is transitive, irreflexive, and antisymmetric, as usual. (We refer to \( \prec \) as the "ordering relation").

For example, suppose trace \( R \) is the following sequence of events: \( \text{write}(T_1, x, 1); \text{read}(T_1, x, 1); \text{read}(T_2, x, 1) \). Then in \( R \), \( \text{write}(T_1, x, 1) \prec \text{read}(T_1, x, 1) \) and \( \text{write}(T_1, x, 1) \prec \text{read}(T_2, x, 1) \), but \( \text{read}(T_2, x, 1) \not\prec \text{read}(T_1, x, 1) \).

Definition 3 (Happens-before ordering). The happens-before ordering \( \preceq \) in a trace \( R \) as follows: \( E_1 \preceq E_2 \text{ iff} E_1 \prec E_2 \text{ and one of the following holds:} \)

1. \( \text{thread}(E_1) = \text{thread}(E_2) \)
2. \( E_1 \) is \( \text{spawn}(T_1, T_2) \), and \( E_2 \) is \( \text{start}(T_2) \)
3. \( E_2 \) is \( \text{join}(T_1, T_2) \), and \( E_1 \) is \( \text{end}(T_2) \)
4. \( E_1 \) is \( \text{unlock}(T_1, x) \) and \( E_2 \) is \( \text{lock}(T_2, x) \)
5. there exists \( E_3 \) with \( E_1 \prec E_3 \) and \( E_3 \prec E_2 \) (i.e., the happens-before ordering is transitive)

---

\(^1\)Note that the lecture slides define a slightly different syntax for events, but the concepts are the same.

\(^2\)Note that a read/write from a volatile variable \( x \) is here modeled as a lock on \( x \), followed by the read/write of \( x \), followed by an unlock on \( x \).
Here we write thread($E$) to mean the thread that caused event $E$ to happen; e.g., thread(read($T_2, x, 5$)) = $T_2$.

There are two things to note about this definition:

- Just because event $E_1$ occurs before $E_2$ in a trace (the < relation), it is not necessarily the case that $E_1$ "happens before" $E_2$ (according to the $\leq$ relation): event ordering is necessary for happens-before, but not sufficient. What is required is that events occurring in different threads must have a synchronization event between them, to link them up.

- Happens-before is defined on program traces (i.e., an execution of a program), not on programs, which we can think of as denoting a (potentially-infinite) set of executions, as determined by the program’s semantics. We can, by looking at a program, determine what traces are possible for that program, and thus whether there exists a trace, due to happens-before ordering, that could result in a surprising outcome.

## 2 Visibility

**Definition 4** (Non-visibility). Given $E_W$ is some event write($T_1, x, v_1$) and $E_R$ is some event read($T_2, x, v_2$) in trace $R$, we have that $E_W$ is not visible to $E_R$ (i.e., $v_1 \neq v_2$) if one of the following two visibility rules applies:

ordered read rule $E_R \leq E_W$ (i.e., the read happens before the write)

intervening write rule there exists some intervening event $E_{W_2}$ of the form write($T, x, v_3$) such that $E_W \leq E_{W_2} \leq R$ (i.e., the first write is overwritten by the second)

**Definition 5** (Visibility). Given $E_W$ is some event write($T_1, x, v_1$) and $E_R$ is some event read($T_2, x, v_2$) in trace $R$, if $E_W$ is not invisible (according to the above definition) to $E_R$, then it is visible to $E_R$, and thus it could be that $v_2 = v_1$.

To understand the consequences of this definition we provide some examples.

**Example 6.** Consider the program on the left in Figure 1, supposing that $x$ and $y$ are global/shared variables. To the right of the figure are five possible execution traces, labeled $R_1$ to $R_5$.

In all of these traces, none of the events by $T_2$ happen before $T_1$, and vice versa. This is because while a $T_1$ event may be ordered before a $T_2$ event in the trace, no synchronization events occur that order them according to happens before. As a result, the write of $x$ in $T_1$ does not happen before the read of $x$ in $T_2$. This means that both 0 and 1 are visible to this read, since neither of the two visibility rules applies, and thus either value is legal.

In trace $R_1$, the trace is as you might have expected: $T_1$ writes 1 into $x$ in the first event, and $T_2$ reads 1 from $x$ in the second event. In contrast to trace $R_1$, trace $R_2$ reads 0 for $x$; this is because 0 is still visible at the position of the read. In trace $R_3$, the read by $T_2$ occurs first, so the only possible value it can read is the initial value for $x$, which is 0. Traces $R_4$ and $R_5$ are the same as $R_2$ and $R_3$, resp., but with the last two events swapped. Notice that these traces show that the final result of $y$, when running this program, can be 0, 1, or 2.

**Example 7.** Consider the program and pair of traces in Figure 2. In the first trace, notice that the read by $T_2$ reads 1 from $x$. This is because the the write($T_1, x, 1$) happens before read($T_2, x, 1$)—these two events are ordered thanks to the fact that unlock($T_1, y$) $\leq$ lock($T_2, y$) and transitivity. It is not possible for $T_2$ to have read 0 at this point. The other trace has thread $T_2$ acquire the lock first.

## 3 Data races

With the happens-before ordering, we can precisely define a data race.

**Definition 8** (Data race). A data race takes place between two events in trace $R$ when both access the same memory location, at least one is a write, and they are unordered according to happens-before.

Here are some examples.

**Example 9.** Consider once again the program and traces in Figure 1. For any possible execution of these two threads, the writes to $x$ and $y$ in Thread 1 are not ordered with the write and read of $y$ and $x$, resp., in Thread 2. Thus they constitute data races. As an exercise, check that the read event of $x$ in thread $T_2$ and the write event
Here is a program that sometimes exhibits races:

Example 11. Here is a program that sometimes exhibits races:

In this case, the following trace does not have a data race:

In this case, the write and the read are ordered by happens before because \( \text{unlock}(T_1, y) \preceq \text{lock}(T_2, y) \) according to the rules above, all other events are ordered by the trace order \(<\) and so with transitivity we have that \( \text{write}(T_1, x, 1) \preceq \text{read}(T_2, x, 1) \) so there is no race.

Here is a trace that does exhibit a race:

Here, the write and read events are not ordered by happens-before, and thus constitute a race.