Programming Languages

Dr. Michael Petter, Raphaela Palenta WS 2018/19

Exercise Sheet 1

Assignment 1.1 Quick Quiz.

Answer the following questions:

1. Can a happened-before diagram depict several executions of a distributed system or only one?
2. Can a single happened-before diagram illustrate all the executions (runs) that a synchronization algorithm can perform?

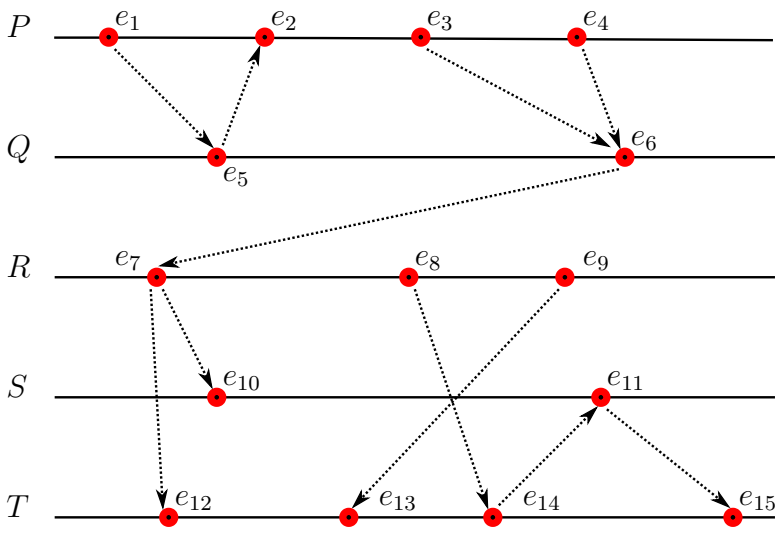
Suggested Solution 1.1

1. It depicts several, namely as many as there are total orderings of the happened before partial order.
2. No, one diagram only depicts one causal dependency between states. Thus, if an event A occurring before B leads to a different outcome than event B happening before A then two diagrams are necessary to depict these different executions.

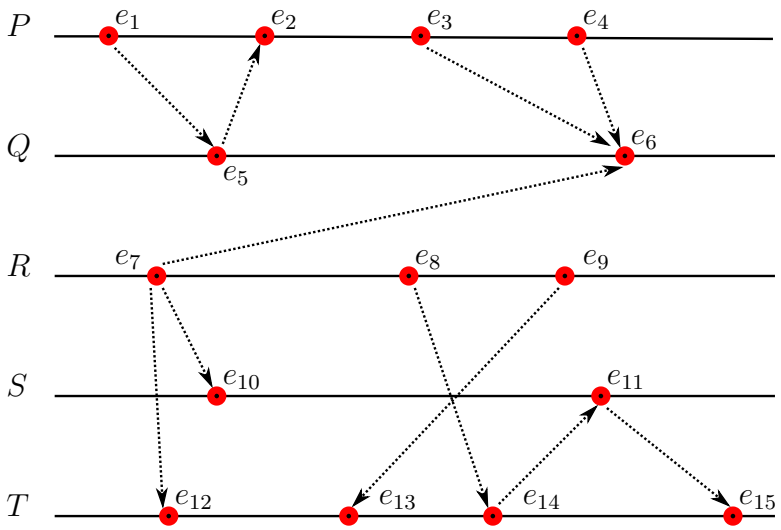
Assignment 1.2 Happened-Before Diagram

For each of the following diagrams, decide if they are valid happened-before diagrams. Prove your answer by defining a mapping $C : E \to \mathbb{N}$ that satisfies the clock condition or by showing that no such mapping exists. Here $E = \{e_1, \dots e_{15}\}$ is the set of events. (The clock condition states that for all $p_i, p_j \in P$, if p_i happens before p_j then $C(p_i) < C(p_j)$.)

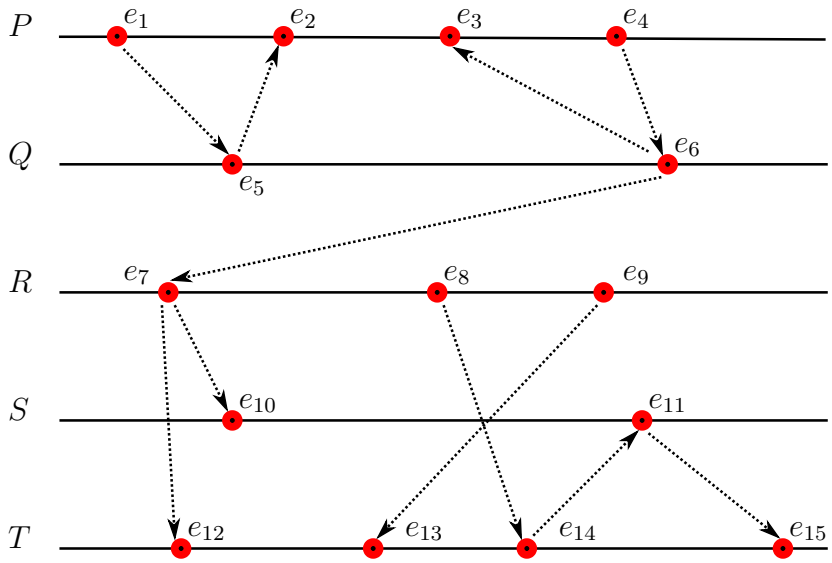
- Diagram one:



- Diagram two: as the diagram of 1., but with the arrow between e_6 and e_7 pointing in the opposite direction



- Diagram three: as the diagram of 1., but with the arrow between e_6 and e_3 pointing in the opposite direction



Suggested Solution 1.2

- Diagram one: valid. There are several mappings that satisfy the clock condition; here are two:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C^{-1}	e_1	e_5	e_2	e_3	e_4	e_6	e_7	e_{10}	e_{12}	e_8	e_9	e_{13}	e_{14}	e_{11}	e_{15}
C^{-1}	e_1	e_5	e_2	e_3	e_4	e_6	e_7	e_{12}	e_{10}	e_8	e_9	e_{13}	e_{14}	e_{11}	e_{15}

- Diagram two: valid. There are many mappings. Any interleaving of C_a and C_b are possible as long as $e_7 \in C_b$ happens before $e_6 \in C_a$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C_a^{-1}	e_1	e_5	e_2	e_3	e_4		e_6								
C_b^{-1}						e_7		e_{12}	e_{10}	e_8	e_9	e_{13}	e_{14}	e_{11}	e_{15}

- Diagram three: invalid. The mapping C would have to obey $C(e_3) < C(e_4) < C(e_6) < C(e_3)$ which means that $<$ would not be a strict partial order.

Assignment 1.3 Concurrent events

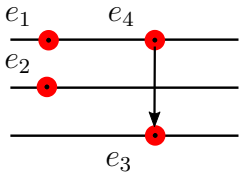
1. List for Diagram one from above three pairs of concurrent events. Use the notion (e_i, e_j) for e_i and e_j are concurrent events.
2. Proof by example that the following statements *do not* hold:
   - If e_1 and e_2 are concurrent and e_2 and e_3 are concurrent then e_1 and e_3 are concurrent. (With other words: If (e_1, e_2) and (e_2, e_3) then (e_1, e_3).)
   - If e_1 and e_2 are concurrent and e_2 happened before e_3 then e_1 and e_3 are concurrent. (With other words: If (e_1, e_2) and $e_2 \rightarrow e_3$ then (e_1, e_3).)

Suggested Solution 1.3

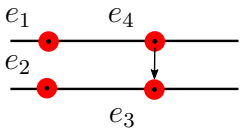
1. This is a (hopefully) full list of all concurrent events in Diagram one:

   (e_{10}, e_8)
   (e_{10}, e_9)
   (e_{10}, e_{12})
   (e_{10}, e_{13})
   (e_{10}, e_{14})
   (e_{12}, e_8)
   (e_{12}, e_9)

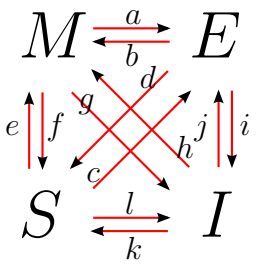
2.
   - e_1 and e_3 are not concurrent as $e_1 \rightarrow e_4 \rightarrow e_3$.

   - e_1 and e_3 are not concurrent as $e_1 \rightarrow e_4 \rightarrow e_3$.

Assignment 1.4 Transitions in the MESI-Protocol.



Consider a distributed system with CPUs A, B, C. For a cache line z in the cache of CPU A explain the transition $b(E \to M), f(M \to S), h(I \to M), i(E \to I)$, from one state $s \in \{M, E, S, I\}$ to another state $s' \in \{M, E, S, I\}$. Which messages (Read (Response), Invalidate (Acknowledge), Read Invalidate, Writeback (Read Response)) are sent between the CPUs?

Suggested Solution 1.4

We give the full list of transitions. The requested ones are marked **bold**.

- Transition a $(M \to E)$: CPU A does a Writeback on cache line z. Therefore the prior made modifications on data in cache line z are written back to the memory and cache line z is know unmodified and exclusive in the cache of CPU A.
- **Transition b $(E \to M)$:** CPU A writes to cache line z. No messages are sent.
- Transition c $(S \to E)$: CPU A sends an Invalidate or Read Invalidate message. Therefore CPUs B and C evicts the cache line in its cache and both send an Invalidate Acknowledge.
- Transition d $(E \to S)$: CPU B or C send a Read for some date in cache line z. CPU A changes the state of cache line z from E to S and then sends a Read Response with the requested data.
- Transition e $(S \to M)$: CPU A sends an Invalidate message for cache line z, CPU B and C evict this cache line in their caches and then send both a Invalidate Acknowledge. After CPU A receives all Invalidate Acknowledge messages it modifies the data item in cache line z and therefore changes the state of this cache line to M.
- **Transition f $(M \to S)$:** CPU B or C send a Read for a data item in cache line z. CPU A has to writeback cacheline z because it is modified. It therefore changes the state of z to S and sends a Writeback Read Response for cache line z.
- Transition g $(M \to I)$: CPU B or C send a Read Invalidate for a data item in cache line z. CPU A needs to writeback the cache line z and responses to the request of CPU B. It therefore sends a Writeback Read Response for cache line z. All CPUs that have not sent the initial Read Invalidate evict the cache line z in their cache and send a Invalidate Acknowledge.
- **Transition h $(I \to M)$:** CPU A sends a Read Invalidate for some data item that is currently not in the cache and will be stored in cache line z. After receiving the Read Response from CPU B or C or the memory *and* after receiving the Invalidate Response of both CPU B and C it can modify the data item. Therefore the state of cache line z changes to M.

- **Transition** i **(**$E \rightarrow I$**)**: CPU B or C send an (Read) Invalidate for a data item in cache line z. In case of a Read Invalidate CPU A sends a Read Response for cache line z. CPU A evicts this cache line and sends an Invalidate Acknowledge. All other CPUs that have not sent the (Read) Invalidate send a Invalidate Acknowledge, too.

- Transition j ($I \rightarrow E$): CPU A sends a Read (Invalidate) for some data item that is currently not in the cache and will be stored in cache line z. In the case of a Read Message the requested data item is sent by the memory and therefore CPU A knows that it holds the cache line z exclusively. In the case of an Read Invalidate the data item is sent by CPU B or C or the memory and CPU B *and* C has to send an Invalidate Acknowledge. After CPU A receives this Invalidate Acknowledge it can change the state of the cache line.

- Transition k ($I \rightarrow S$): CPU A sends a Read for some data item that it currently not in the cache and will be stored in cache line z. CPU B or C send a Read Response for the requested data. As CPU A receives the data from some other CPU it sets the status of the cache line to S.

- Transition l ($S \rightarrow I$): CPU A receives an (Read) Invalidate message for a data item in cacheline z. It therefore changes the state of the cache line to I and sends a Invalidate Acknowledge. All other CPUs that have not sent the inital (Read) Invalidate need to send a Invalidate Acknowledge as well.