# Net Models for Concurrent Object Behaviour

Tony Hoare

Carl Adam Petri Lecture

Hamburg June 2012

# Object behaviour

- is modelled by an occurrence net
- with boxes representing event occurrences
- and arrows recording dependency.

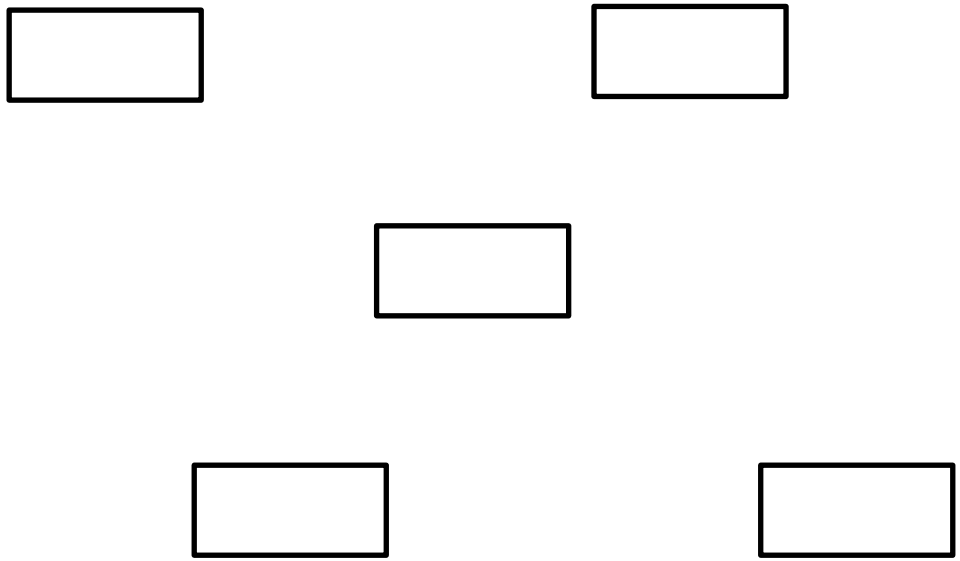
- An object is modelled by the set of all its possible behaviours,
- in all possible circumstances of use.

# Concurrent objects

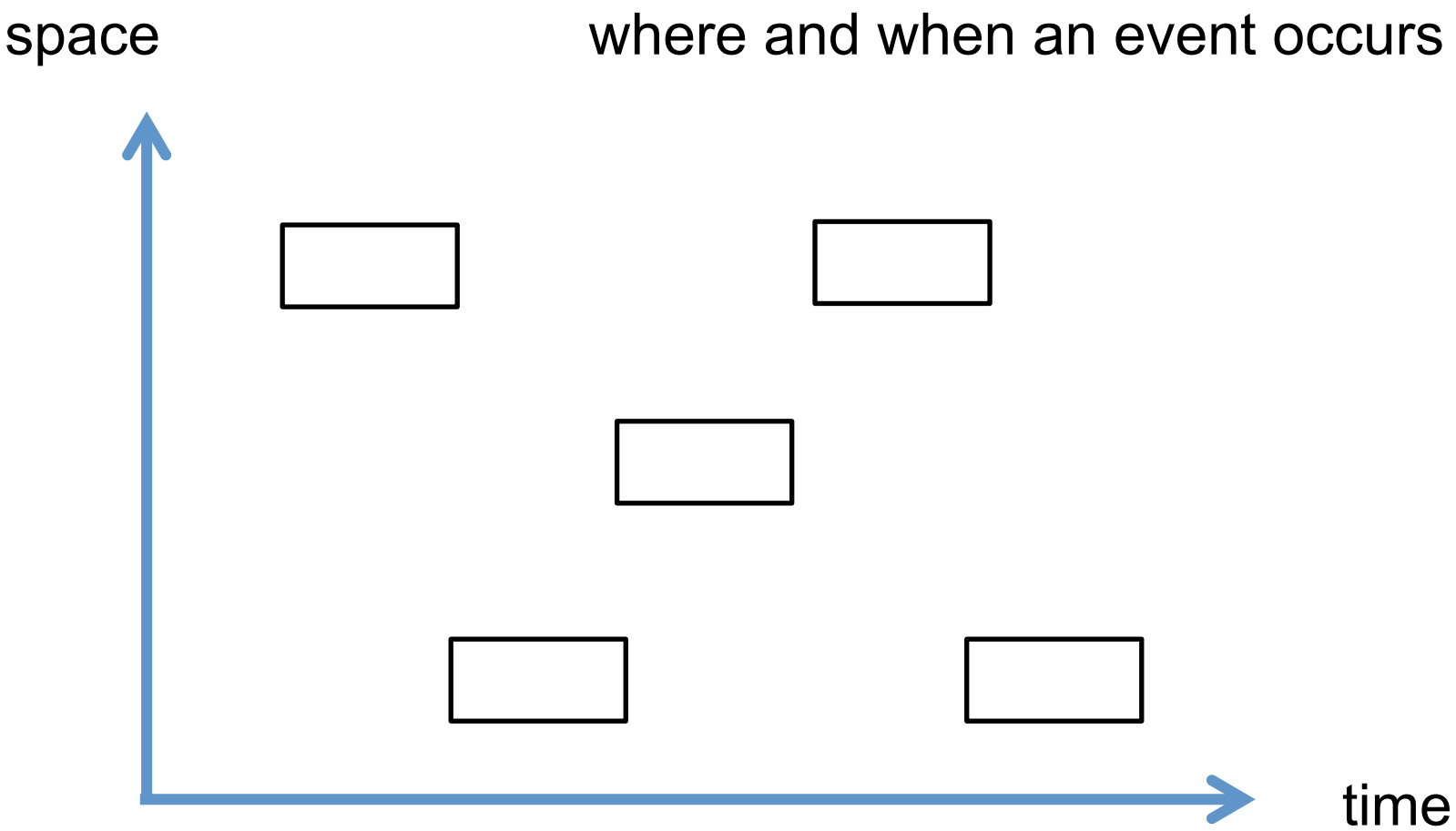
- Objects: allocation, ownership, disposal
- Semaphores: exclusion, signalling
- Channels: buffering, synchrony, overtaking.
- Variables: locality, sharing, weak memory.

# Events

that occur in and around a computer
during execution of a program

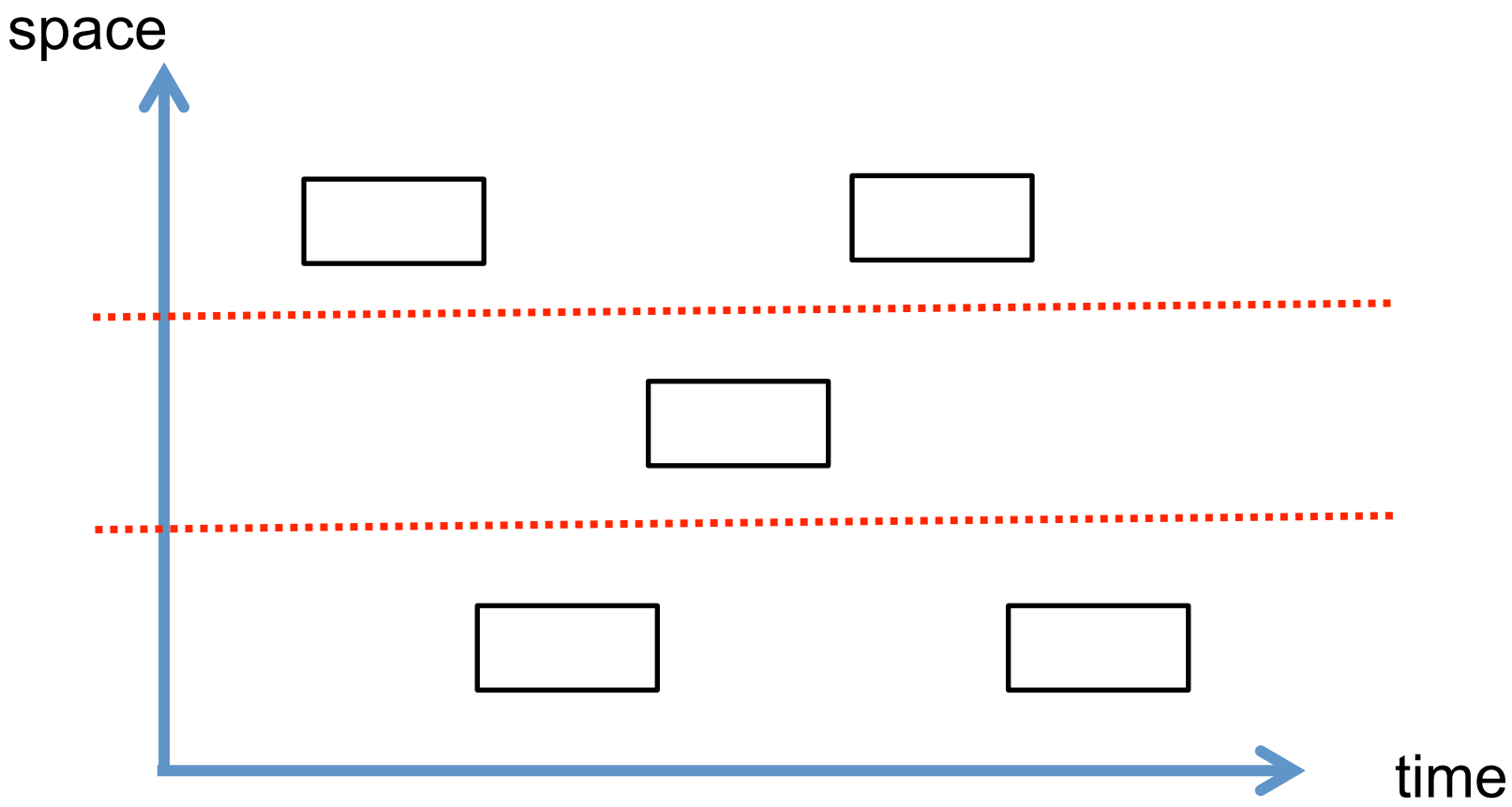


# space and time



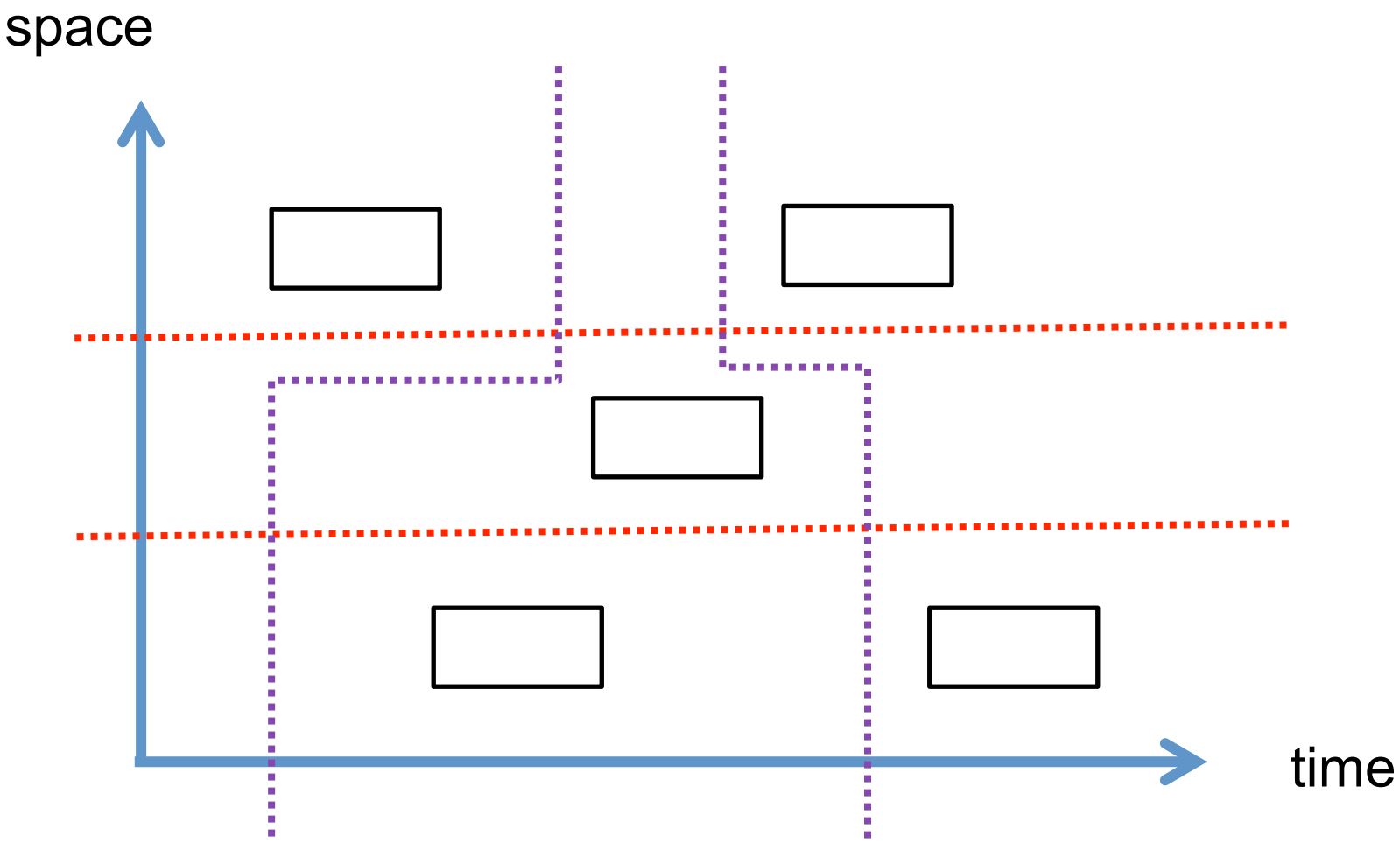
# space boundaries

horizontal, separating concurrent activities



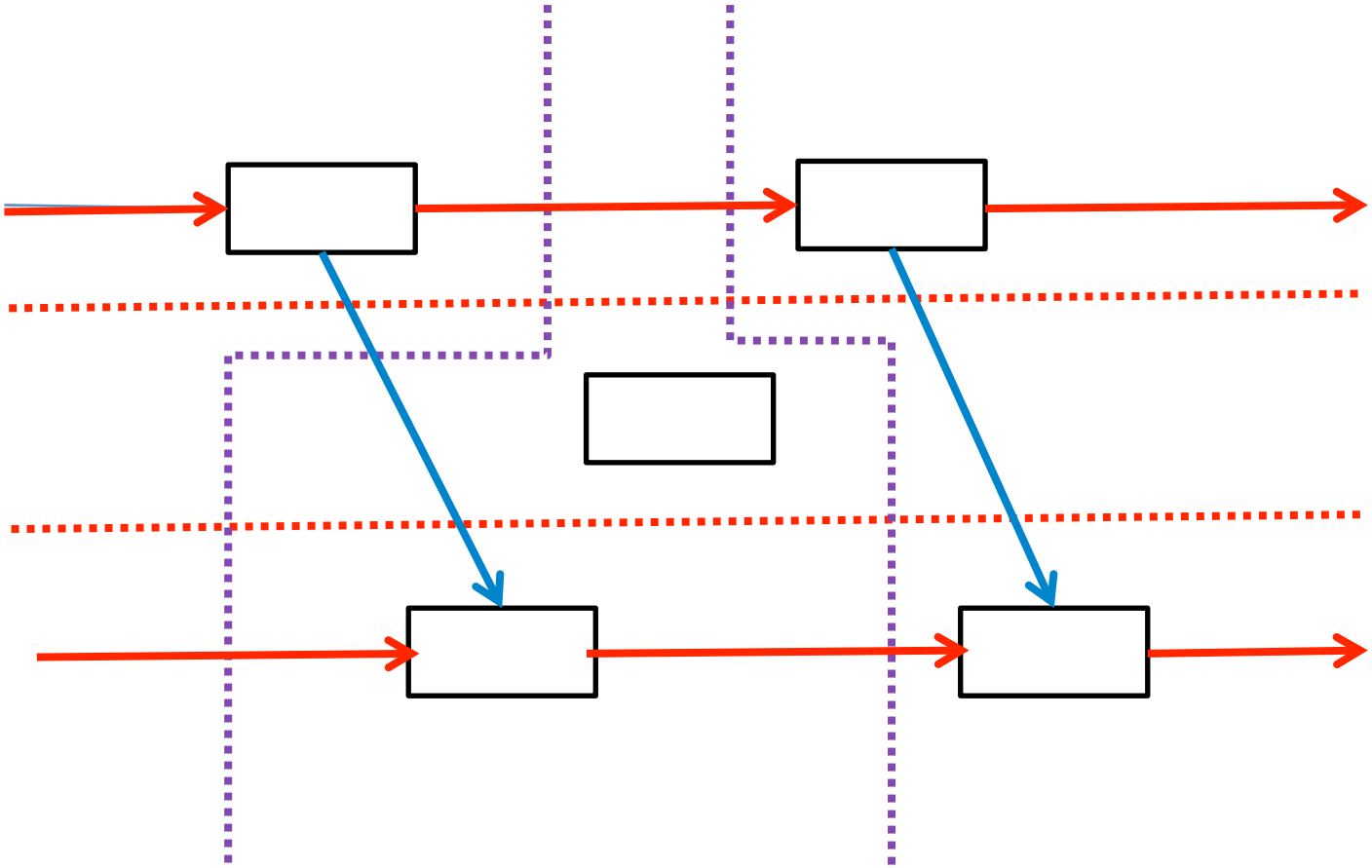
# time boundaries

vertical, separating sequential activities



# Dependencies

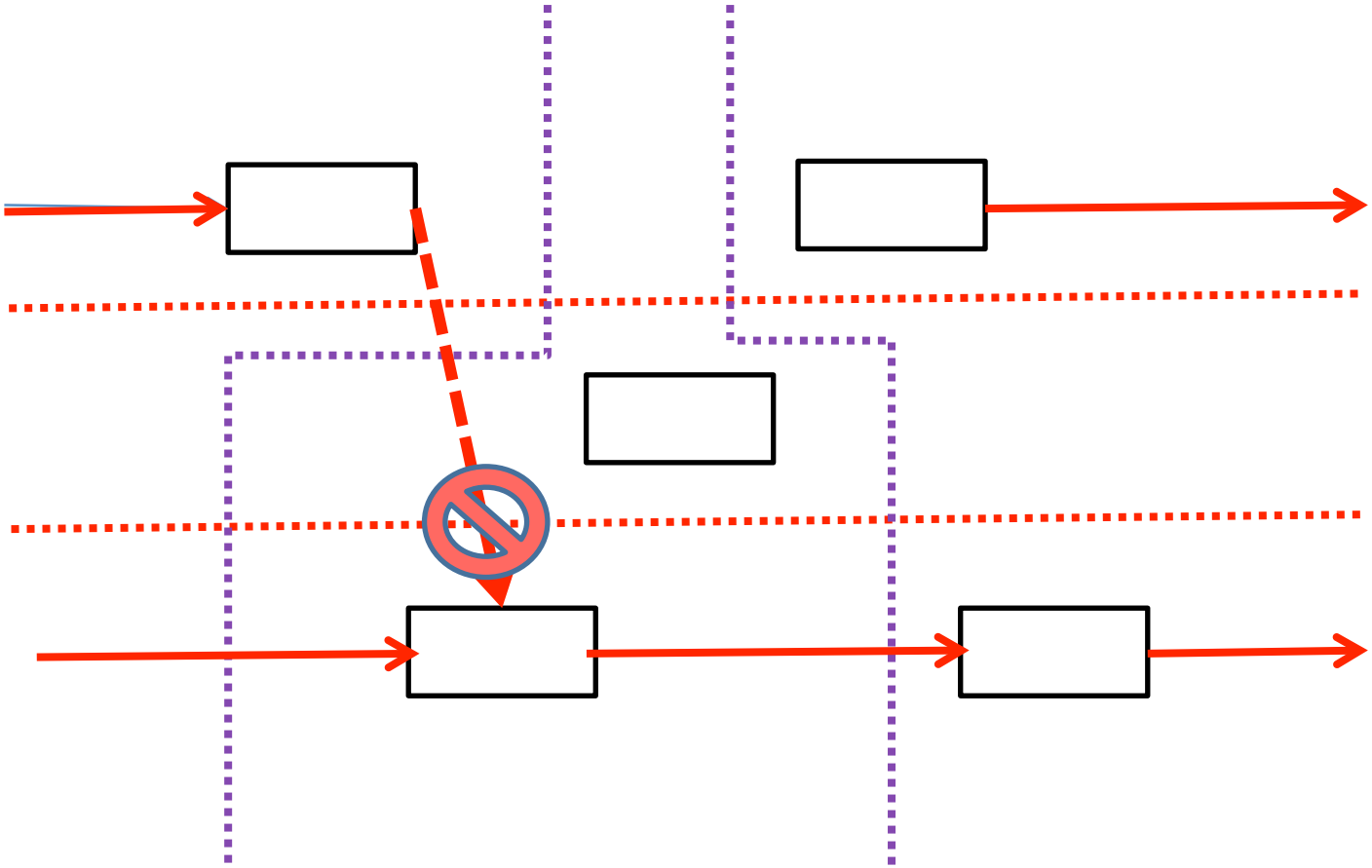
an event at the head of an arrow
depends on the event at the tail



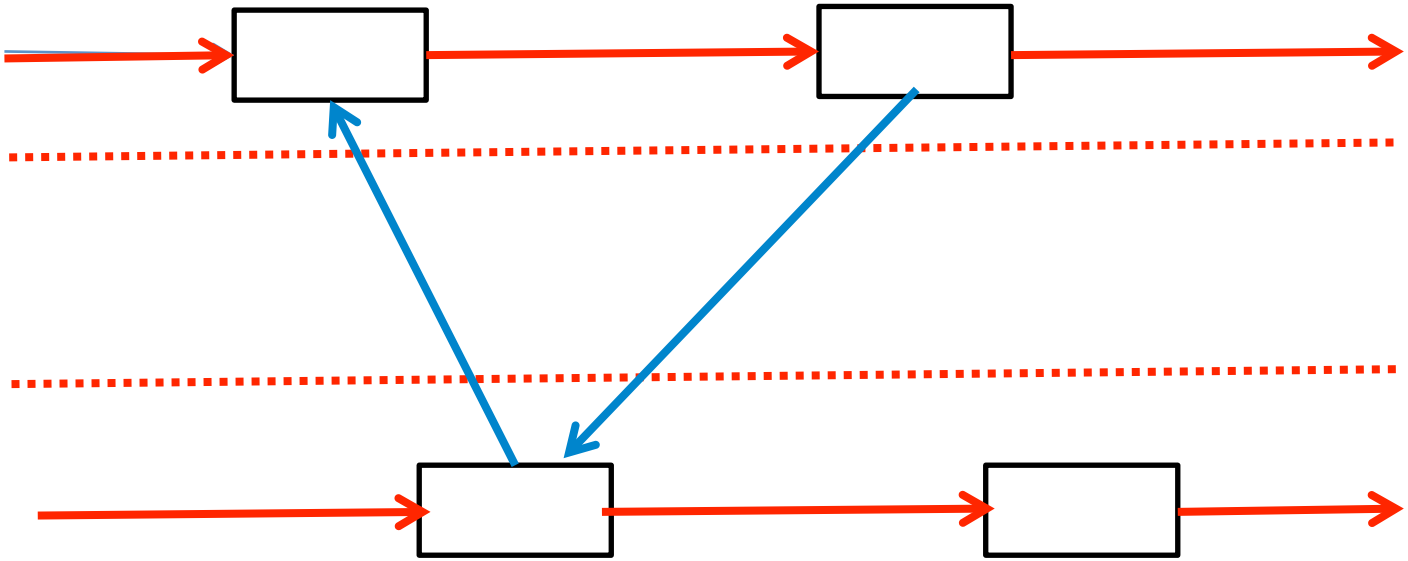
# Rules for arrows

- do not cross a time-boundary backwards
- Local arrows are drawn horizontal
- They must not cross space boundaries
- Non-local arrows are not horizontal
- They must not close a cycle.

# the program must avoid this



# and this cycle

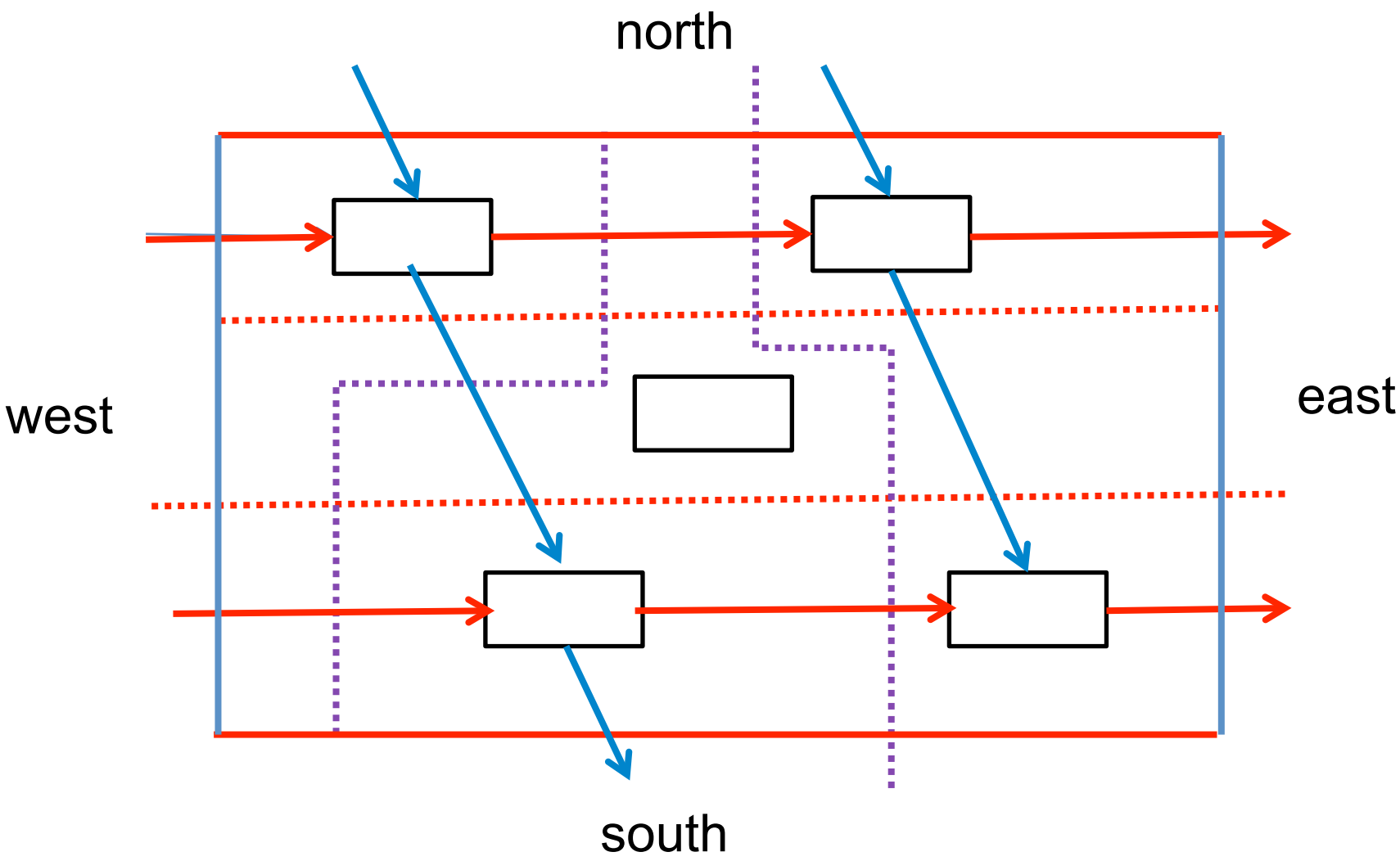


# A box

- models execution of a component sub-command of the program
- e.g., a boxed Petri net.

Eike Best, Raymond Devillers, Maciej Koutny, The Box Algebra = Petri nets + Process Expressions, Information and Computing 178: 44-100 (2002).

# Picture of a box

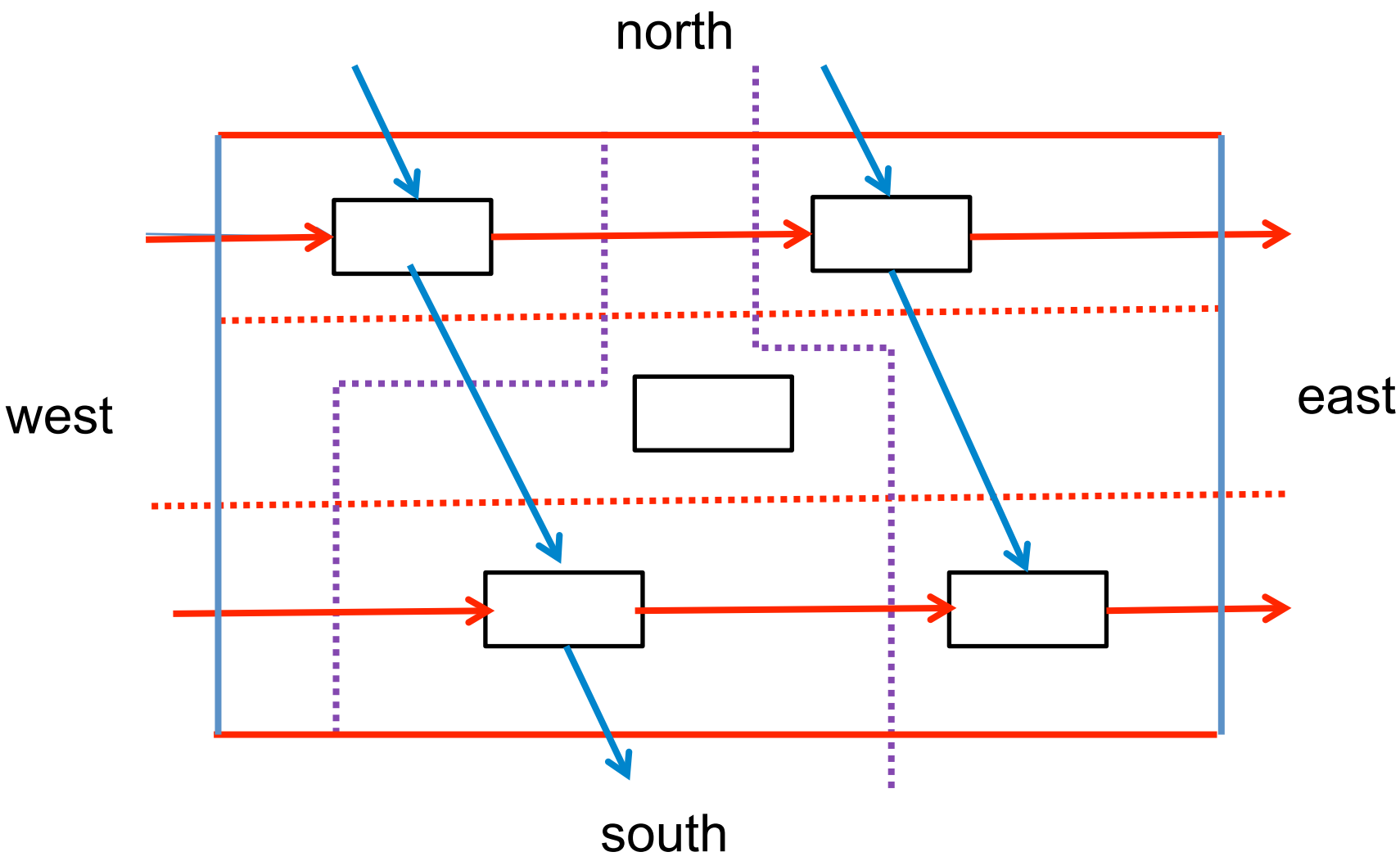


# Box boundaries

- on west and east are time-boundaries
- on north and south are space-boundaries
- on north and west they admit input arrows
- on south and east they emit output arrows.

- A box may contains internal arrows and boxes
- and internal boundaries

# Picture of a box



# Concurrent object behaviour.

- The behaviour of an object is modelled by an occurrence net, containing all the events in which it has engaged.

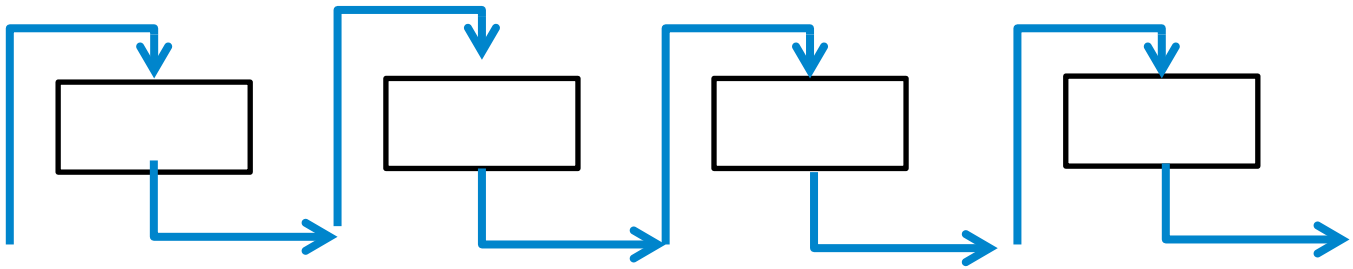
- Every program behaviour is the sum of the behaviours of all the objects that it allocates.

- Choice of object behaviour made by program

# A local object

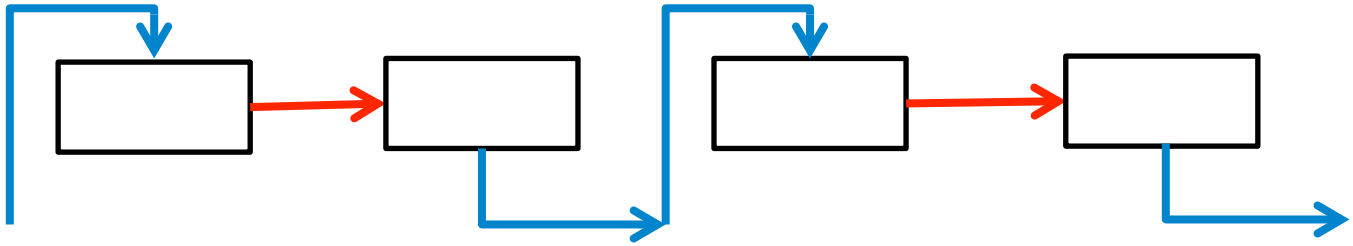
All its actions occur in the same thread.
No interference is possible from other threads

# A shared object



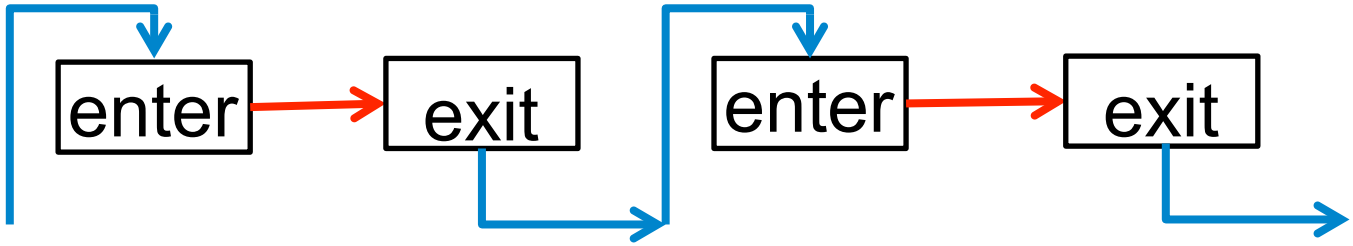
each action may occur in a different thread
from its predecessor
or it may occur in the same thread.

# An exclusion semaphore



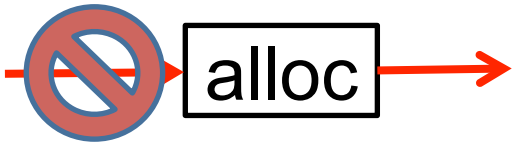
is released by the same thread that most recently acquired it. A released semaphore may be acquired by any thread.

# with labels



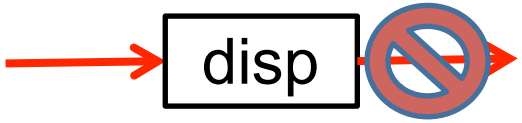

The boxes are labelled to indicate
the nature of the actions, enter and exit

# Object allocation




There is no local input arrow to an alloc.
So there is no action of an object
before it has been allocated.

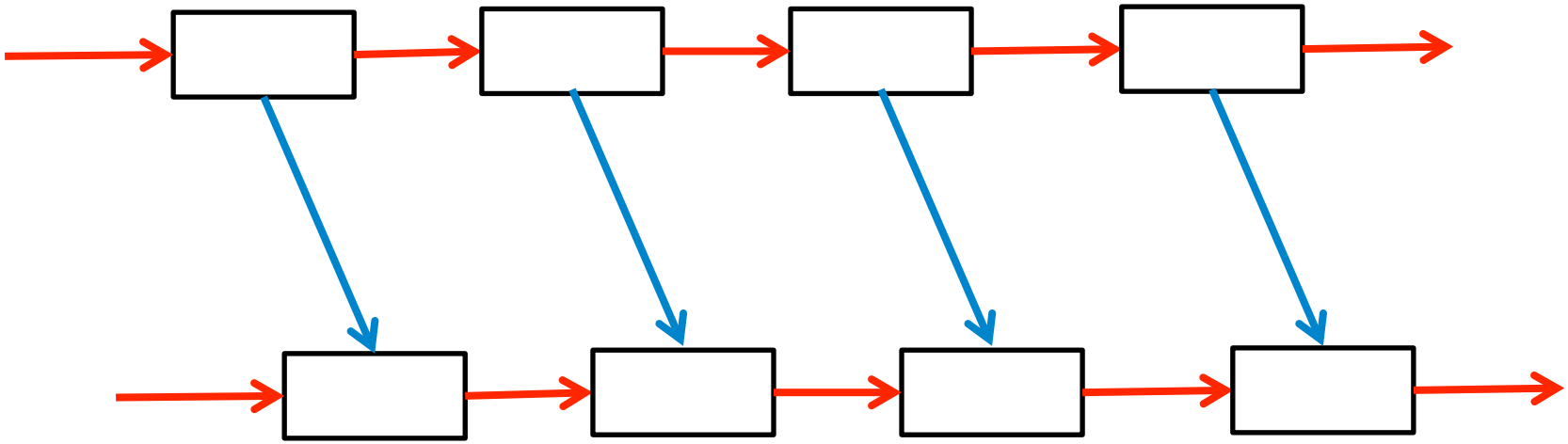
# Object disposal




There is no local arrow from a dispose.
So there is no action on an object
after its disposal.

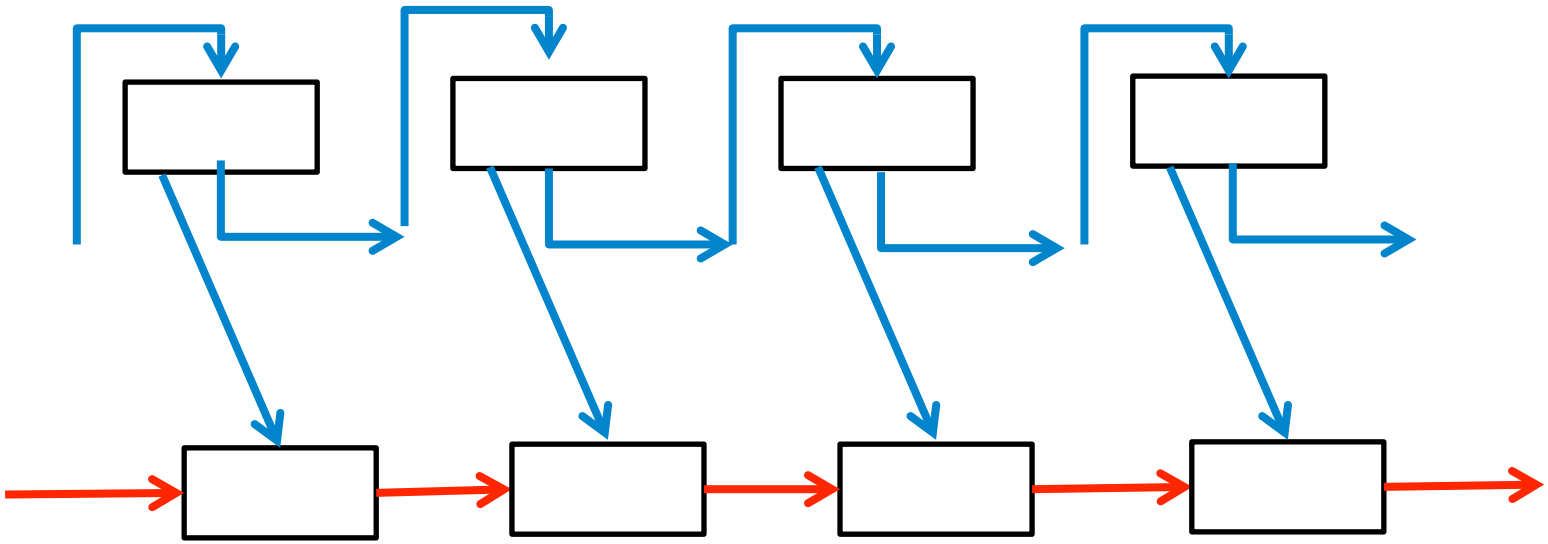
# A Communication channel

single inputter, single outputter



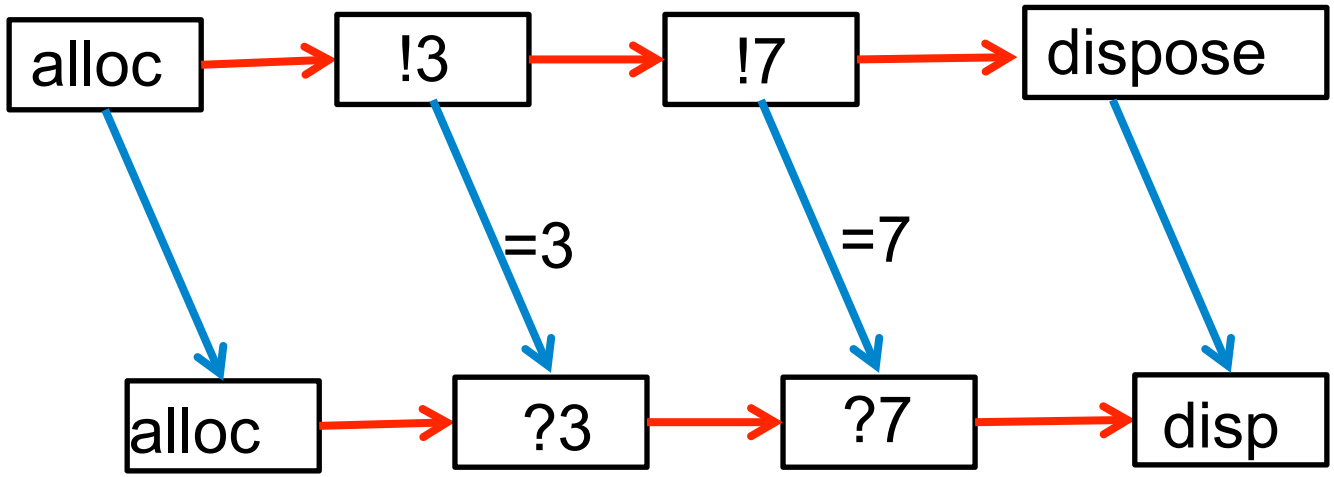
# A shared channel

with multiple outputters



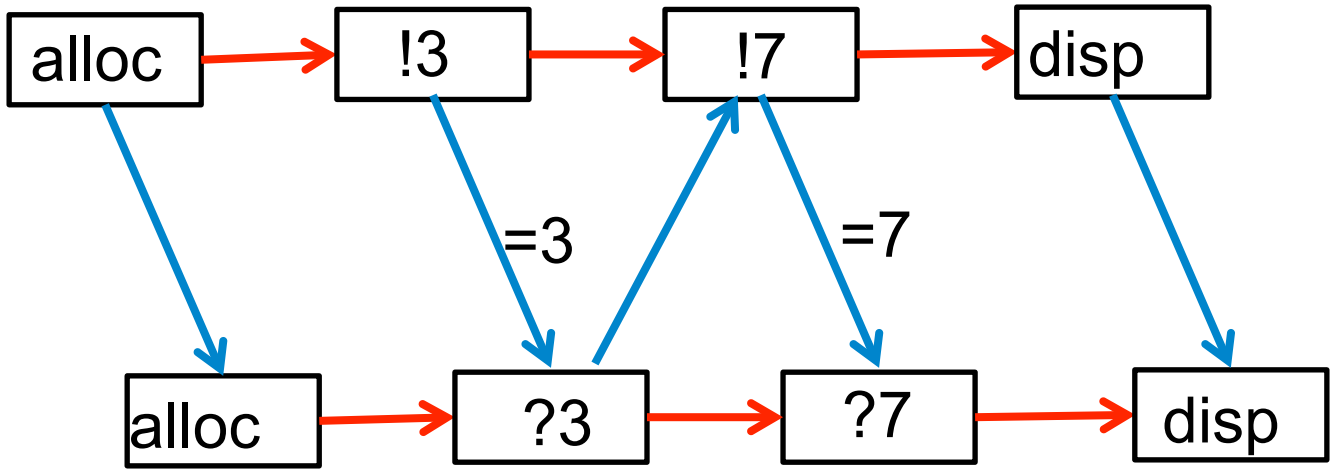
# with labels

to indicate that the value communicated is the same at both ends of the channel

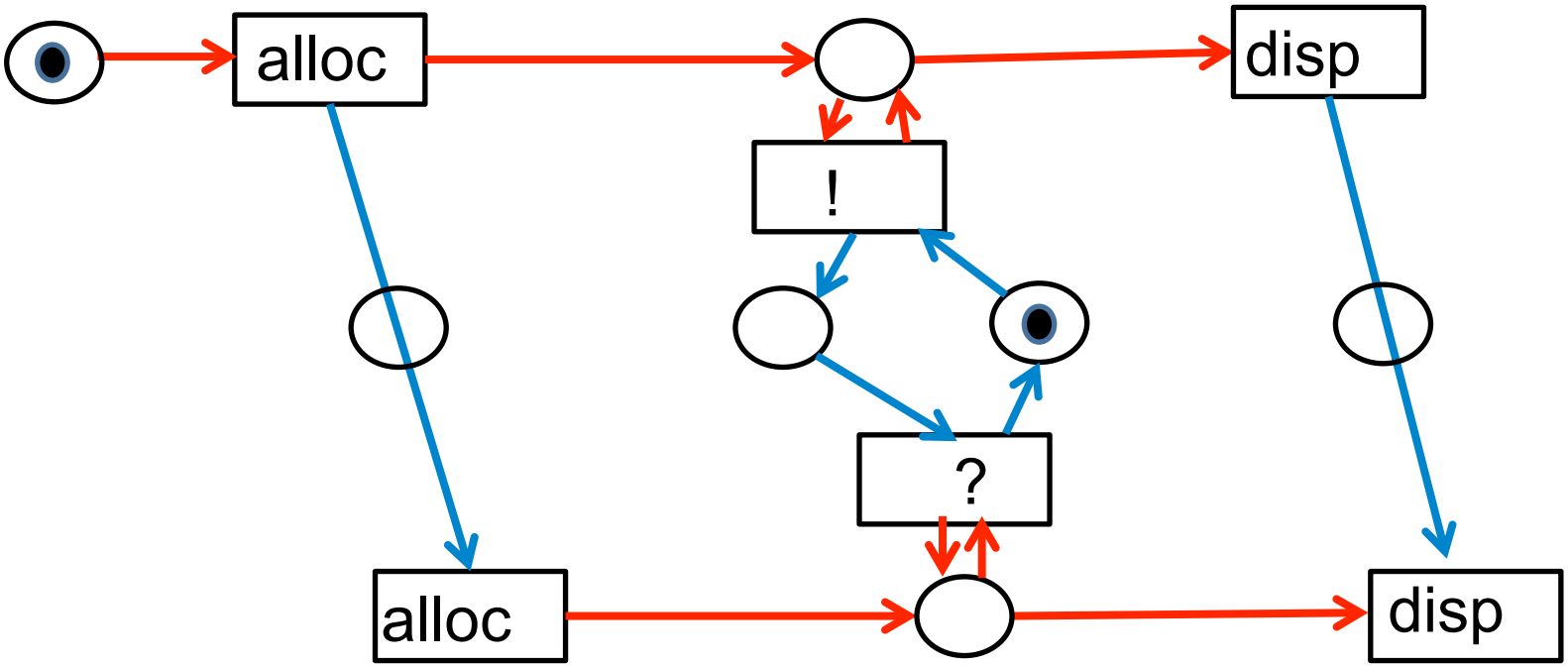


# with a single buffer

which is passed back to outputter for refilling



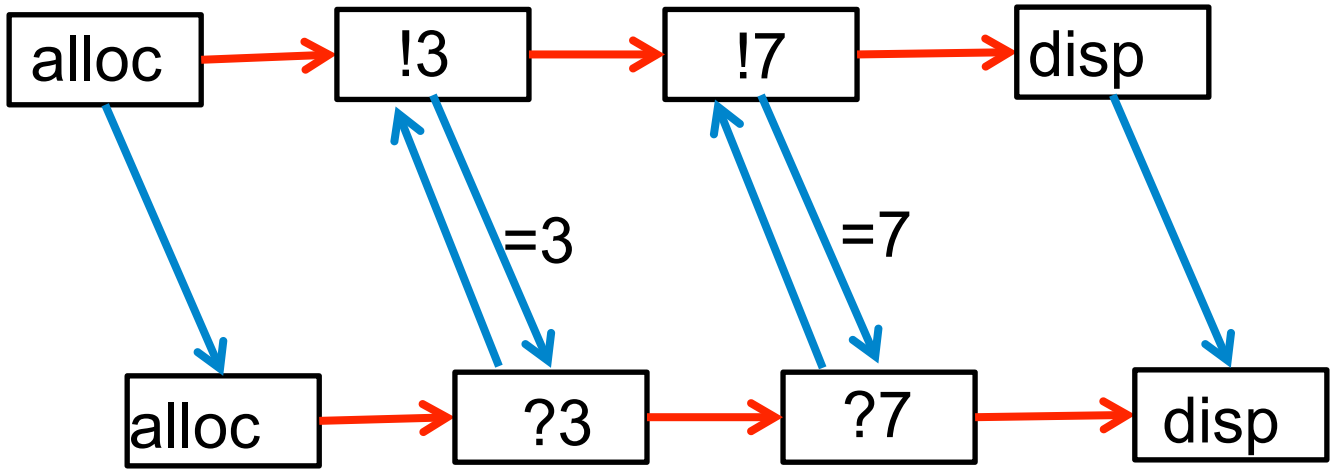
# A generating Petri net




with a single-token place on every arrow.

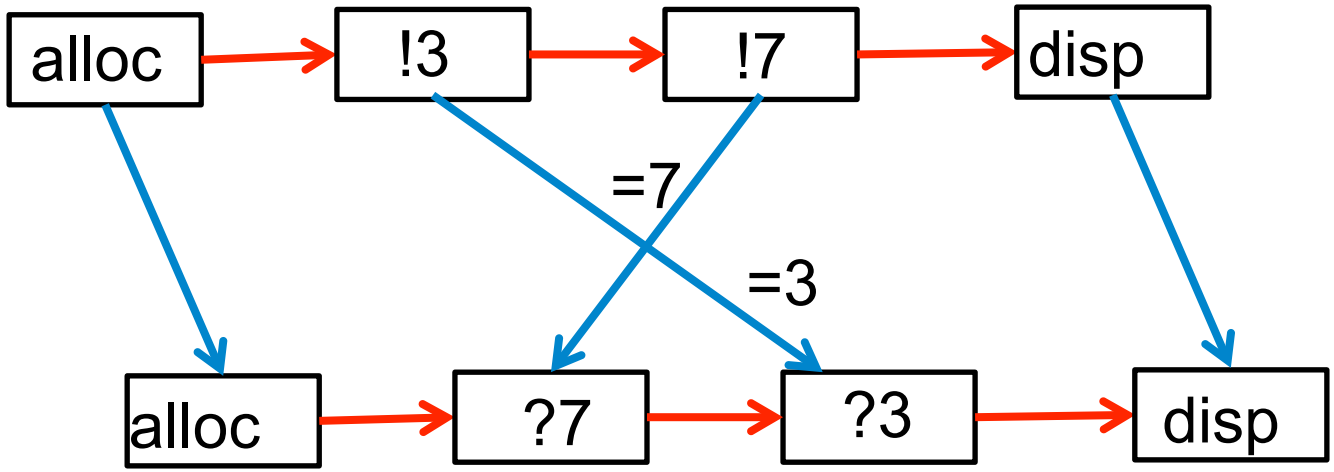
# a synchronous channel

no buffering at all.
input and output are simultaneous



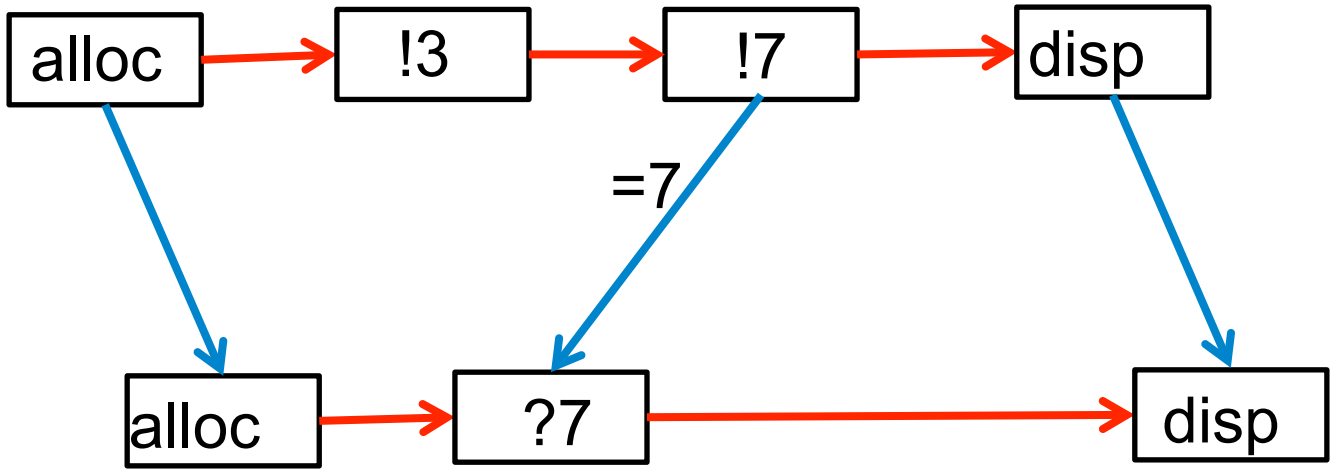
# a re-ordering channel…

allows crossing of arrows

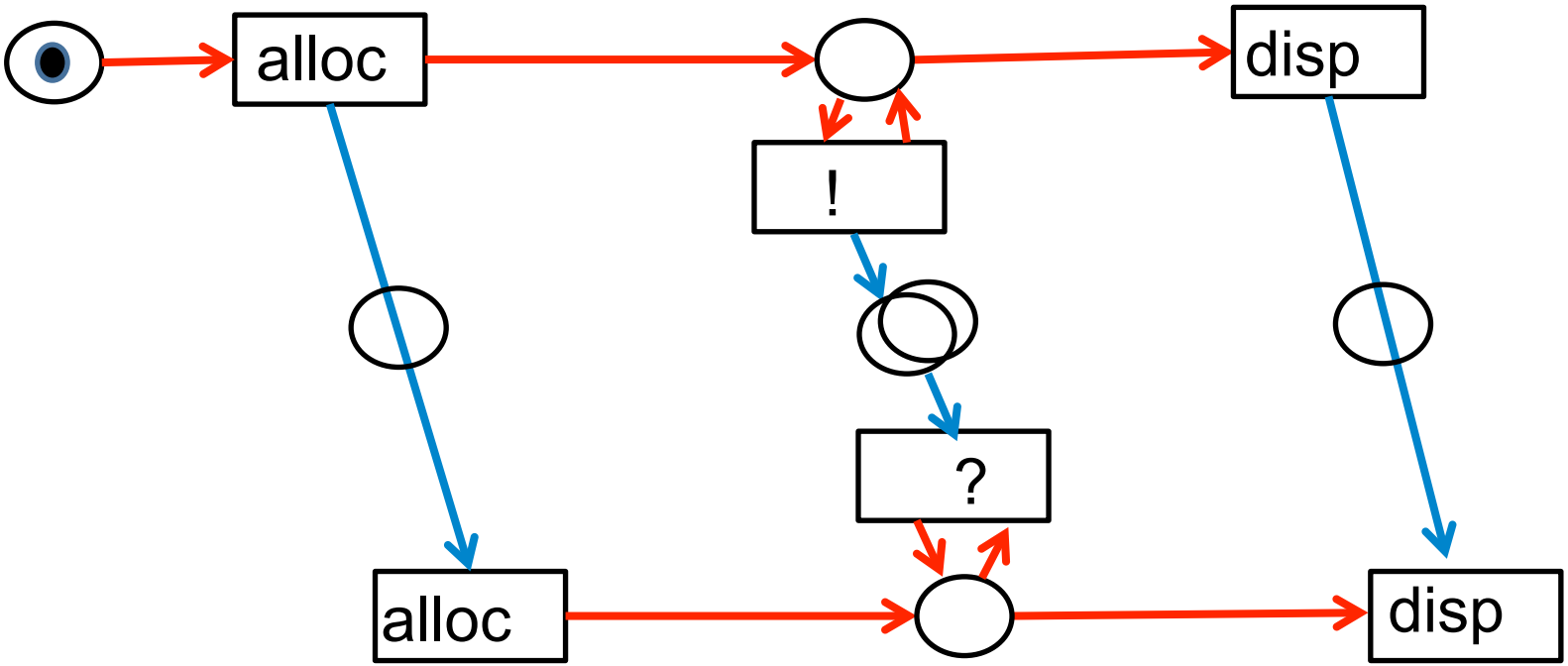


# ...with loss of message

if the inputter does not want any more

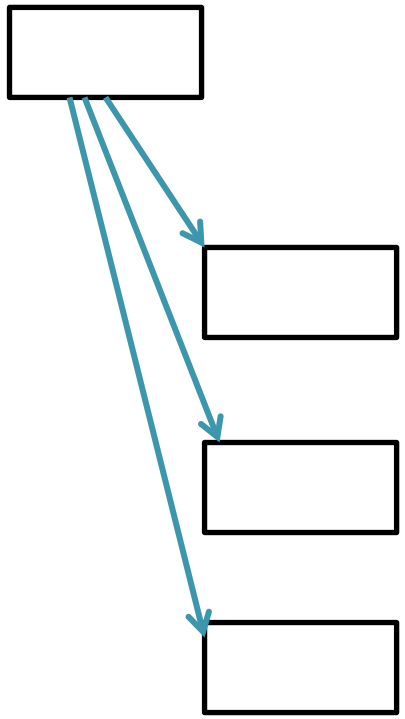


# its generating Petri net




with a multi-token place: 

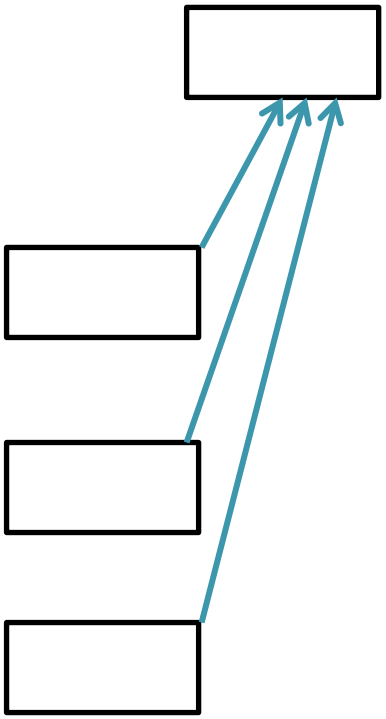
# Fork



all arrows have
the same source
and the same label.

All target boxes have
the same label.

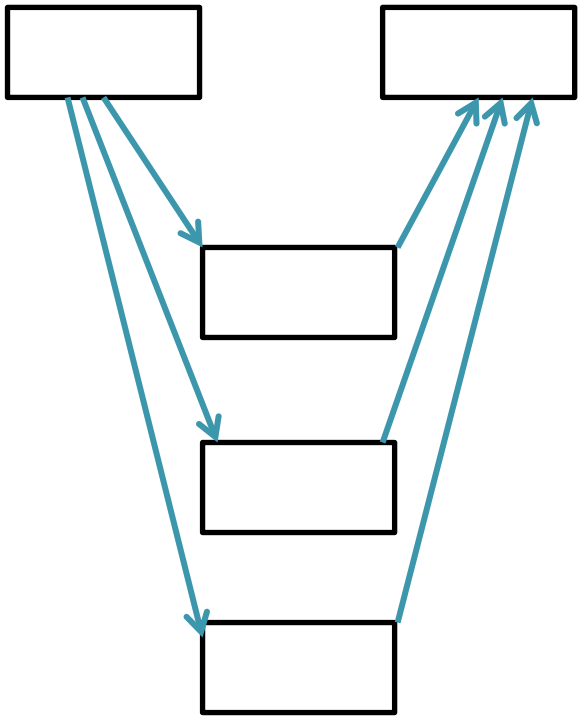
# Join



All arrows have
the same target
and the same label.

All source boxes
have the same label

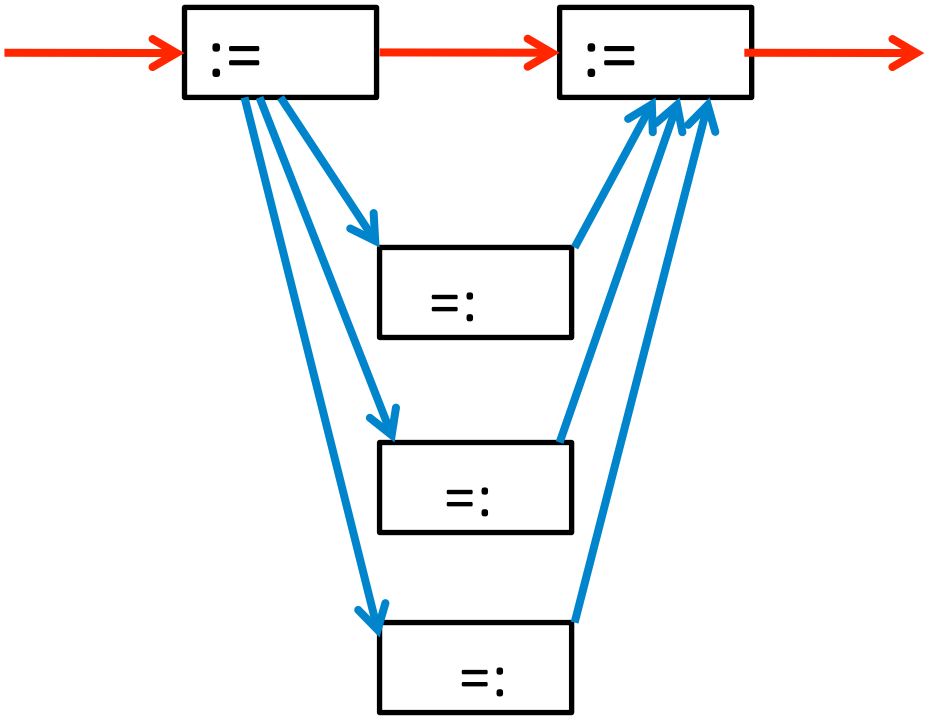
# Concurrency



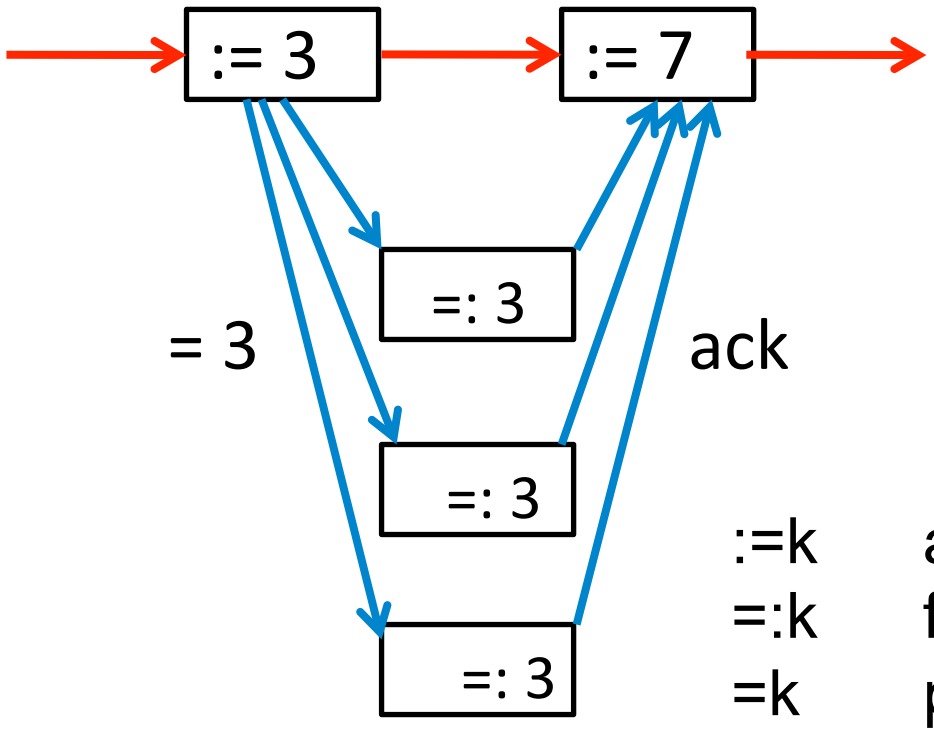
two top events
occur sequentially

bottom three events
occur concurrently

# Assignments and fetches

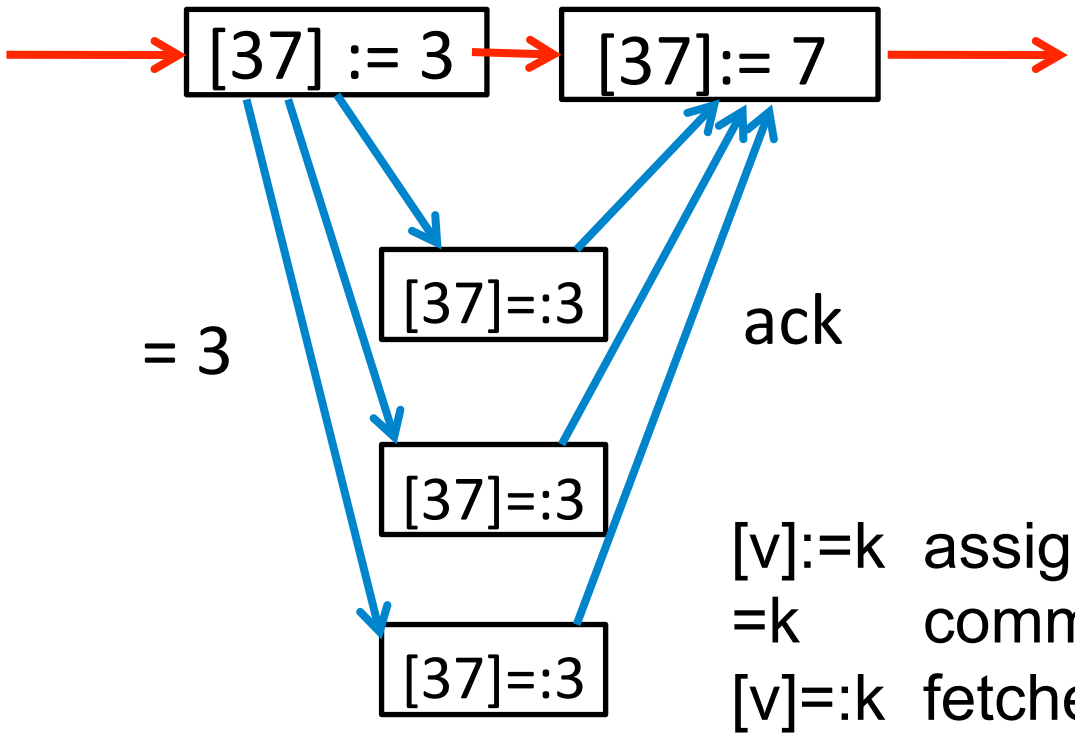


# Assignment with labels



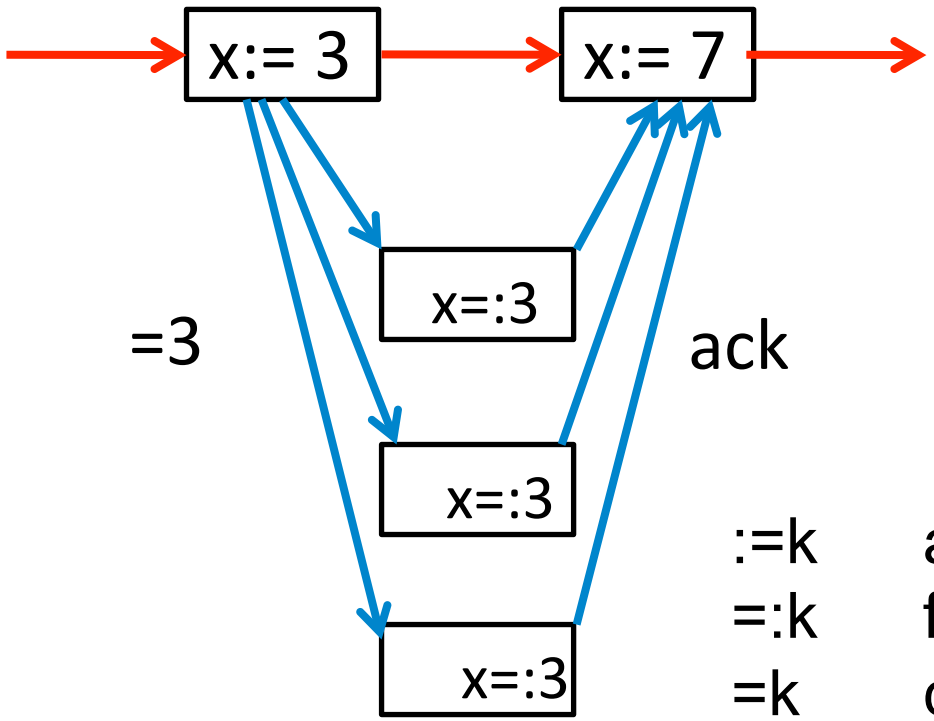

| | |
|---|---|
| :=k | assigns a constant k |
| =:k | fetches a value k |
| =k | points to a value k |
| ack | sync signal |

# to location 37



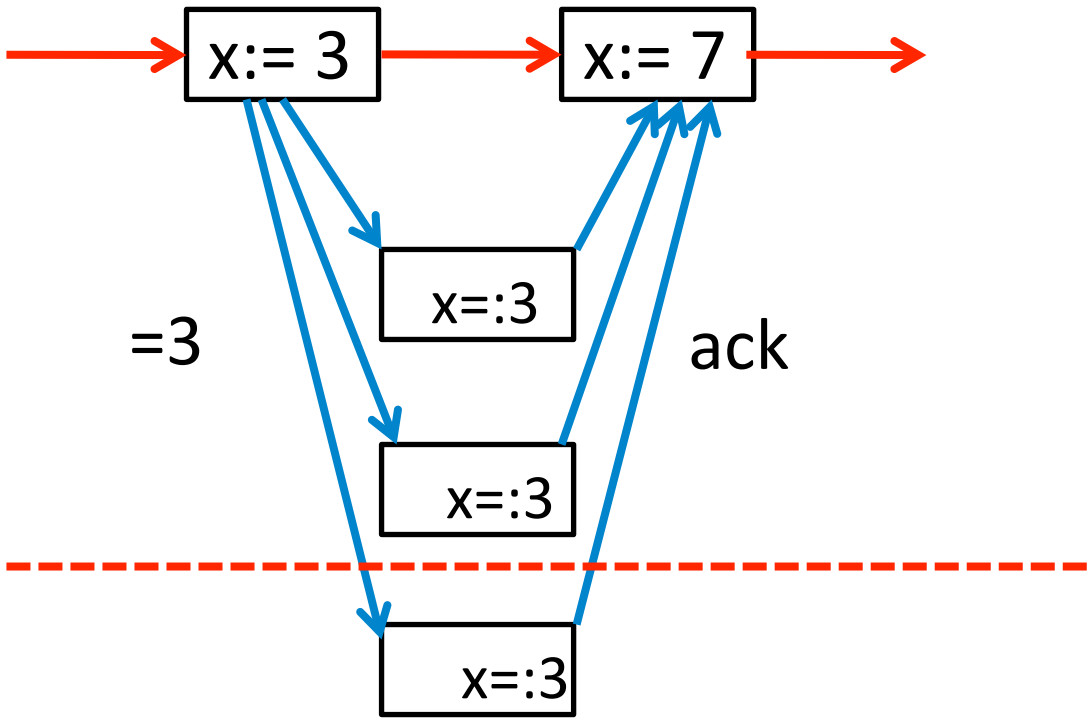

[v]:=k assigns to location v
=k communicatio of k
[v]=:k fetches value from v
ack prevents further reads
of previous assignment

# or to a variable named x

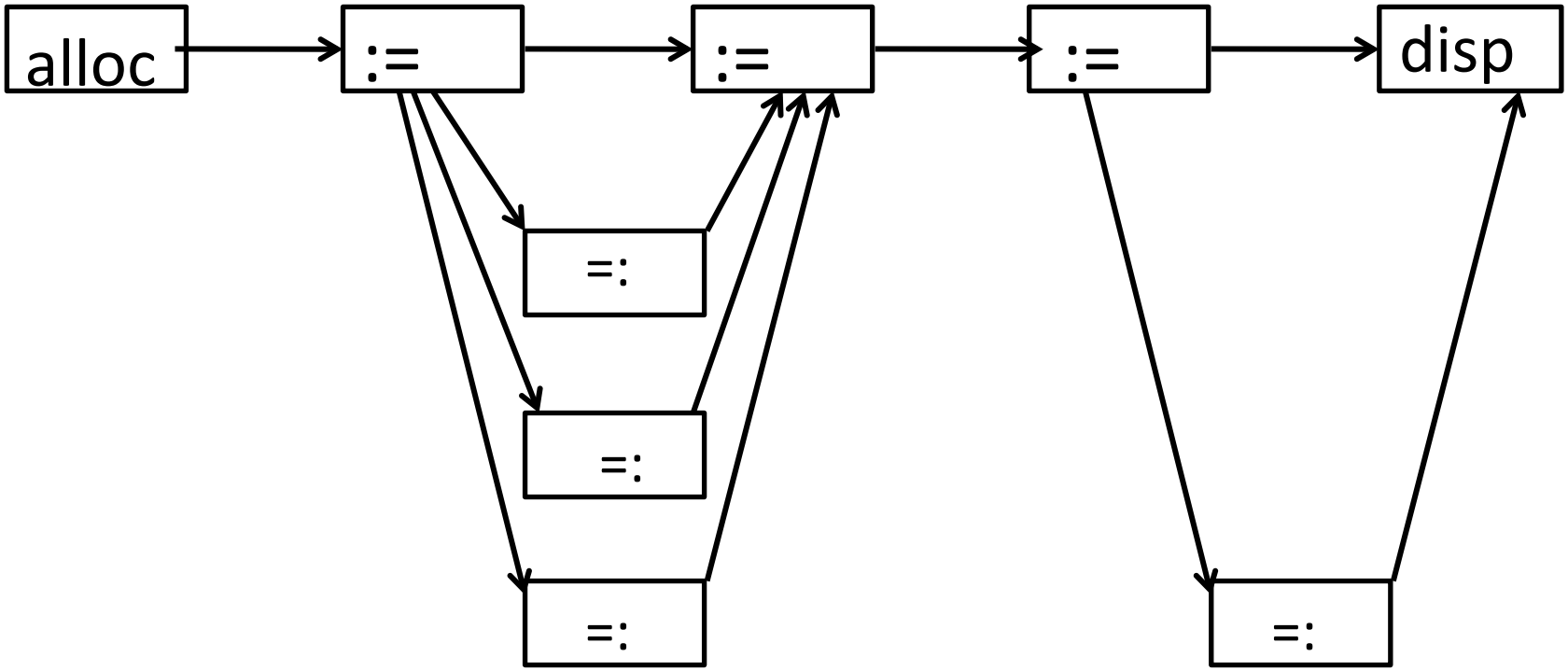



| | |
|---|---|
| :=k | assigns a constant k |
| =:k | fetches a value k |
| =k | communicates k |
| ack | sync signal |

# fetching by another thread

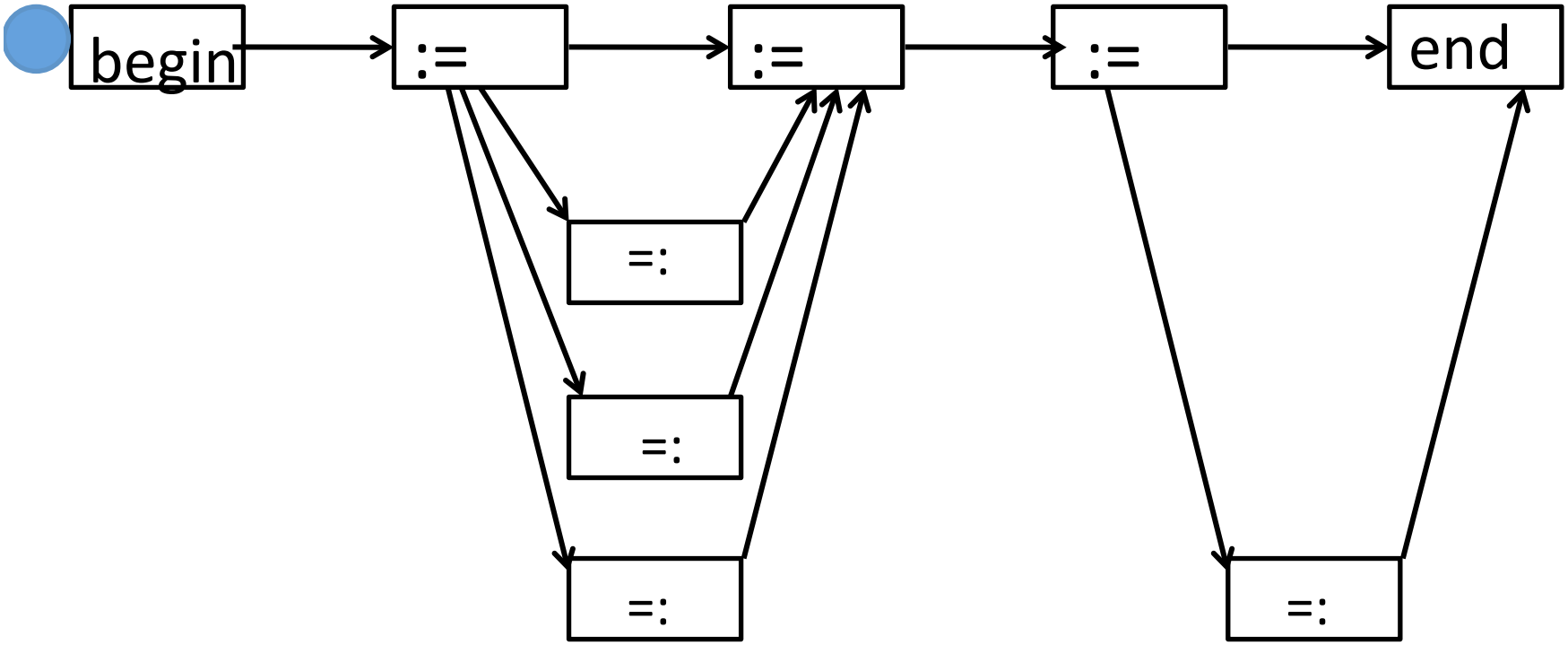


# A variable

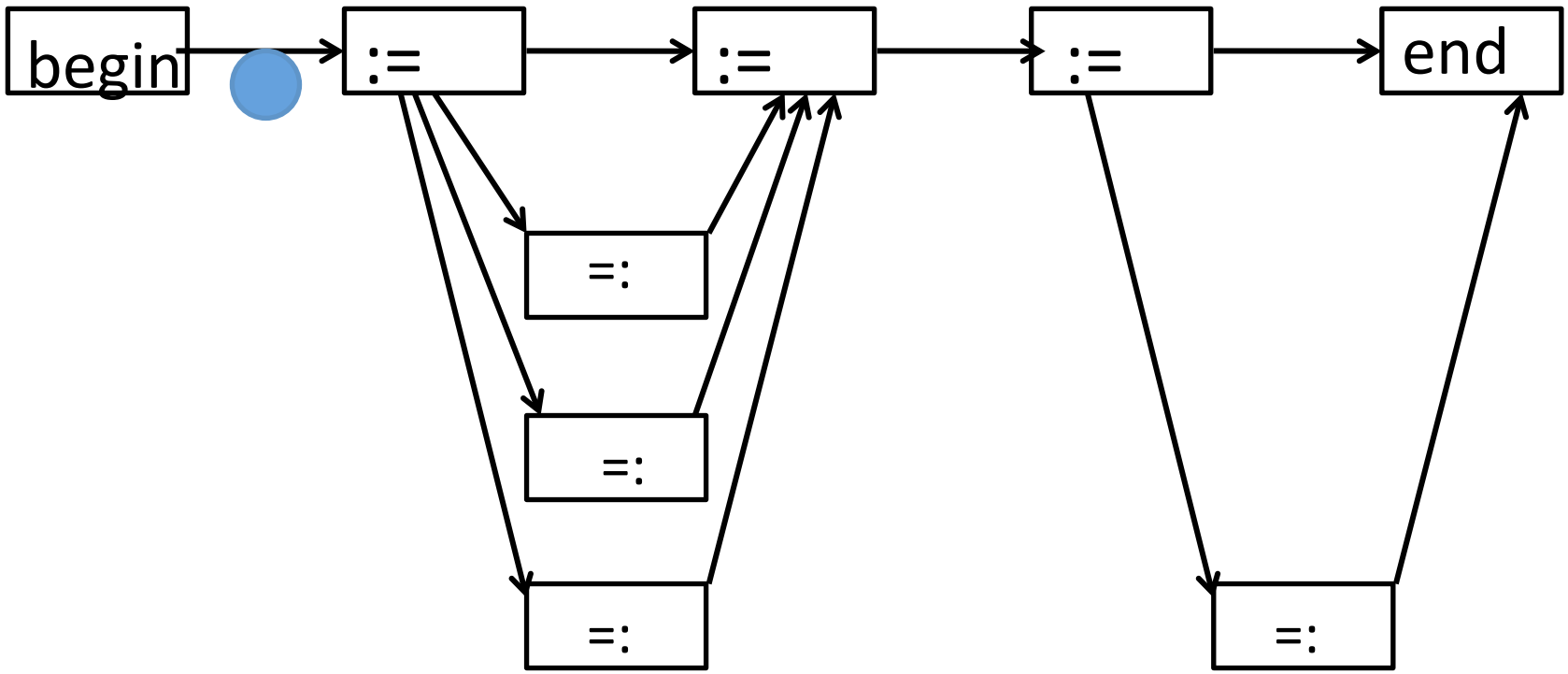



leaving out colours & some labels

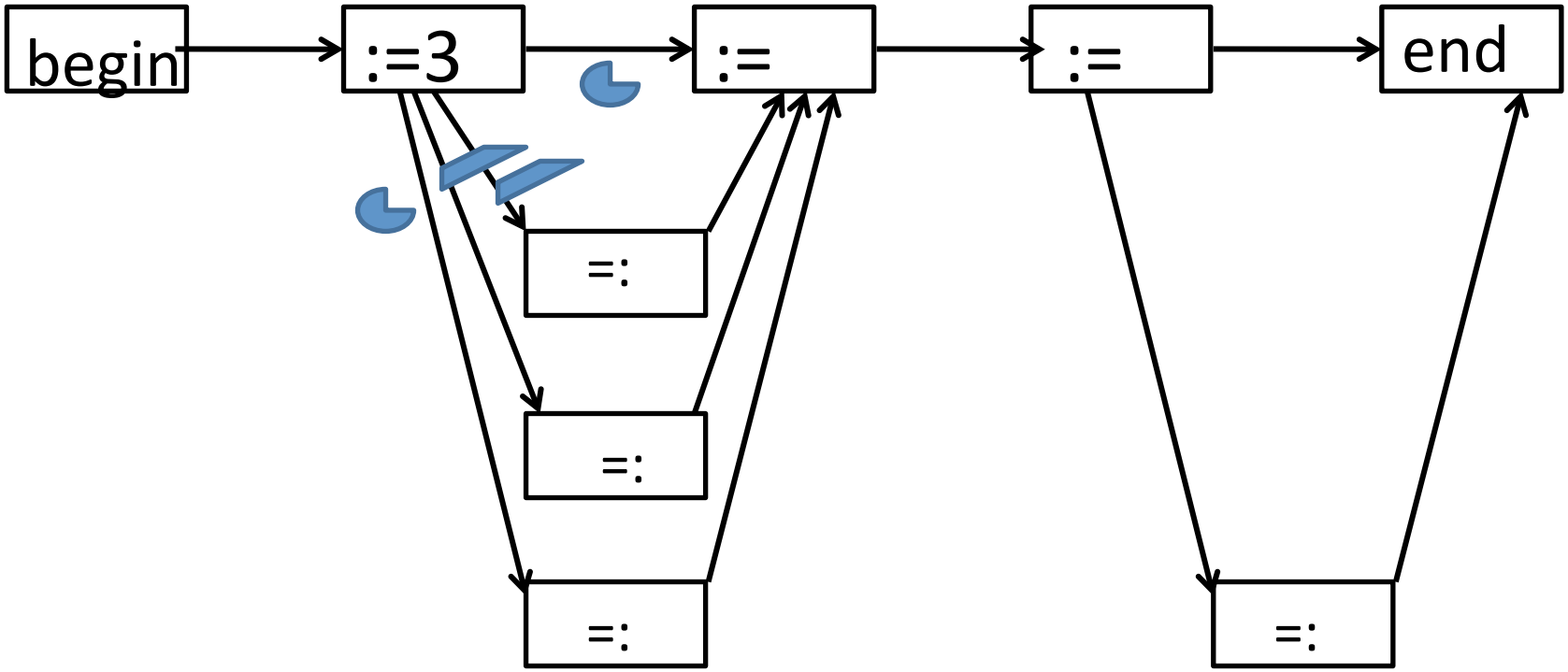
# Execution



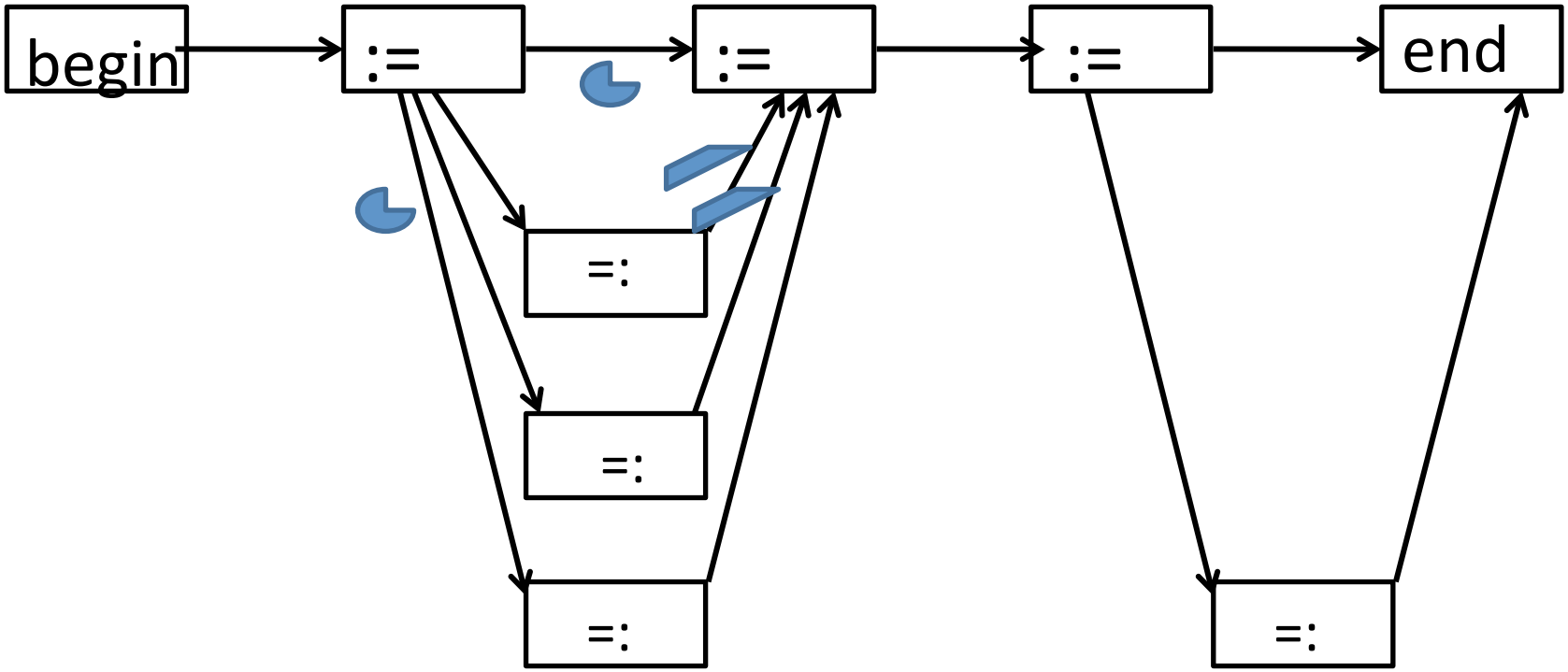
# Token move



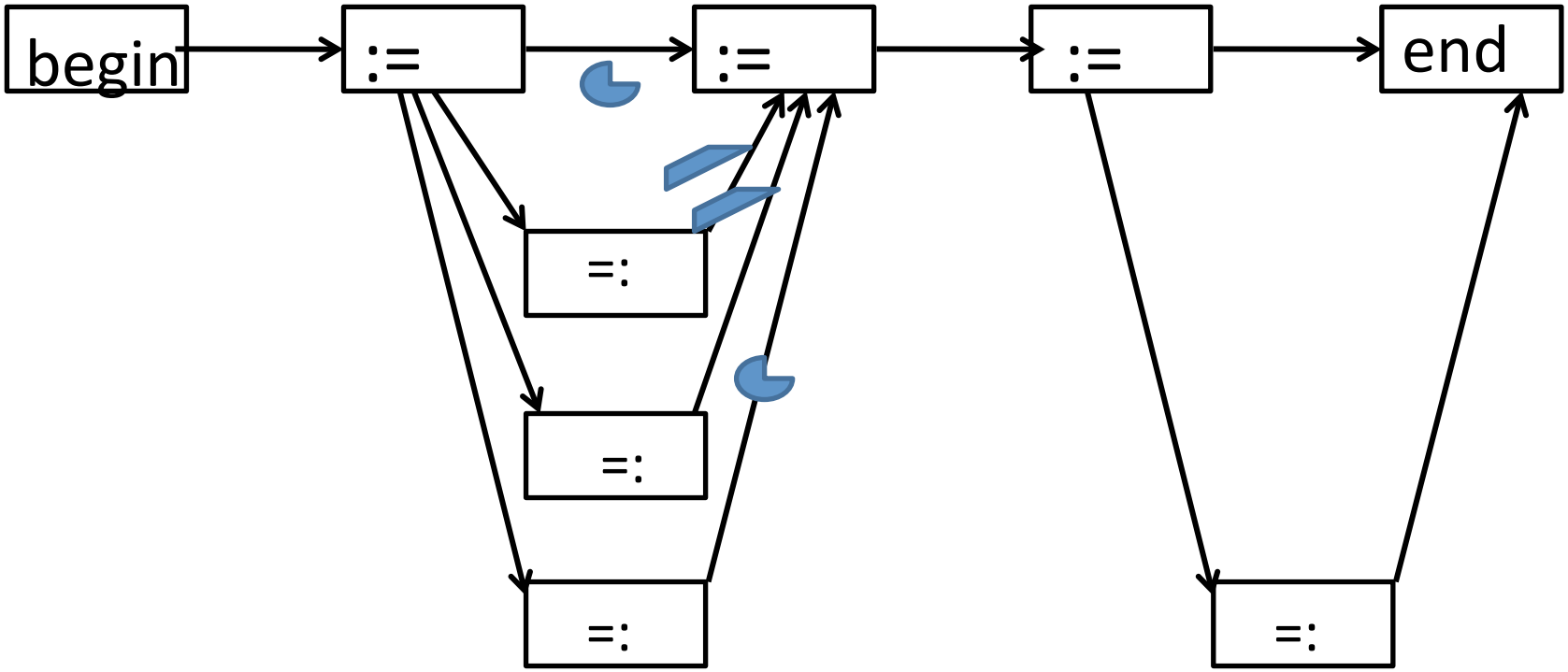
# Token split



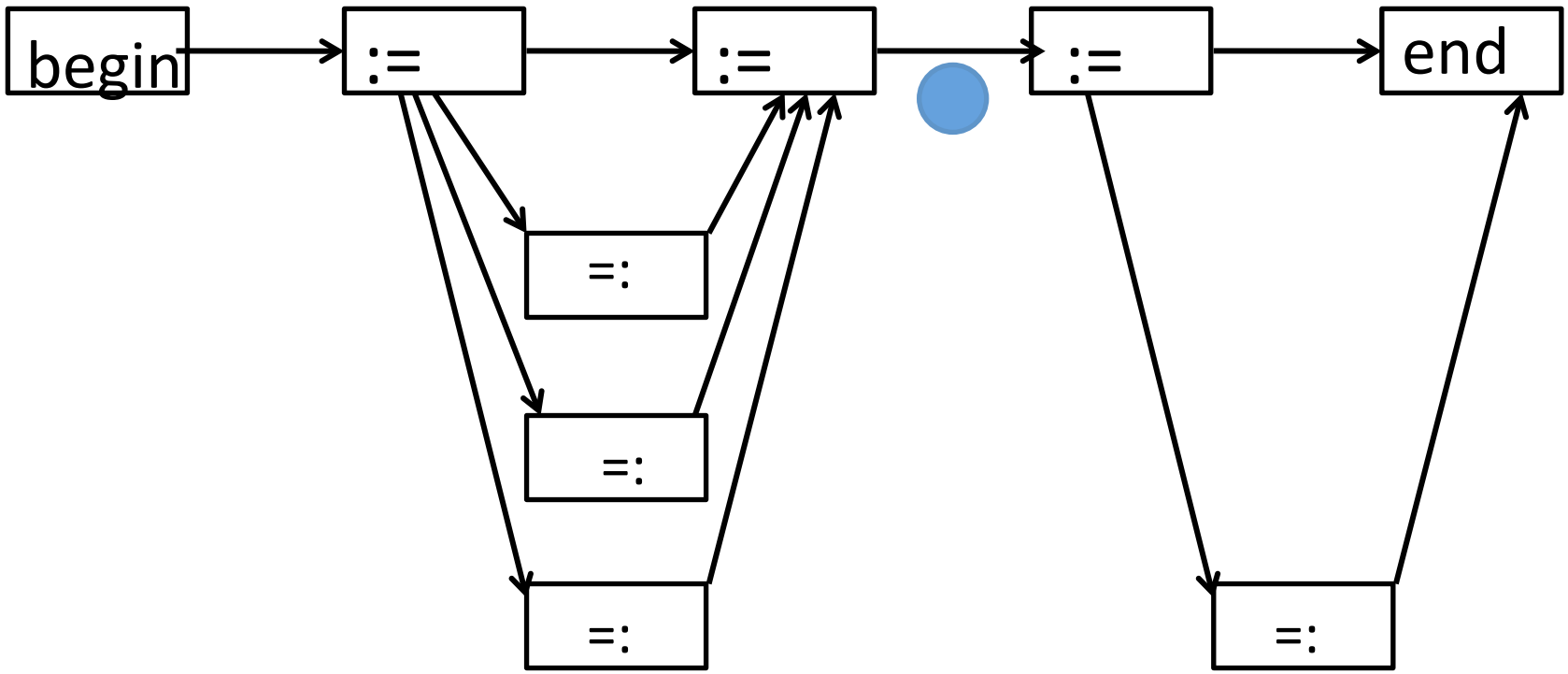
# Concurrency



# Synchronisation



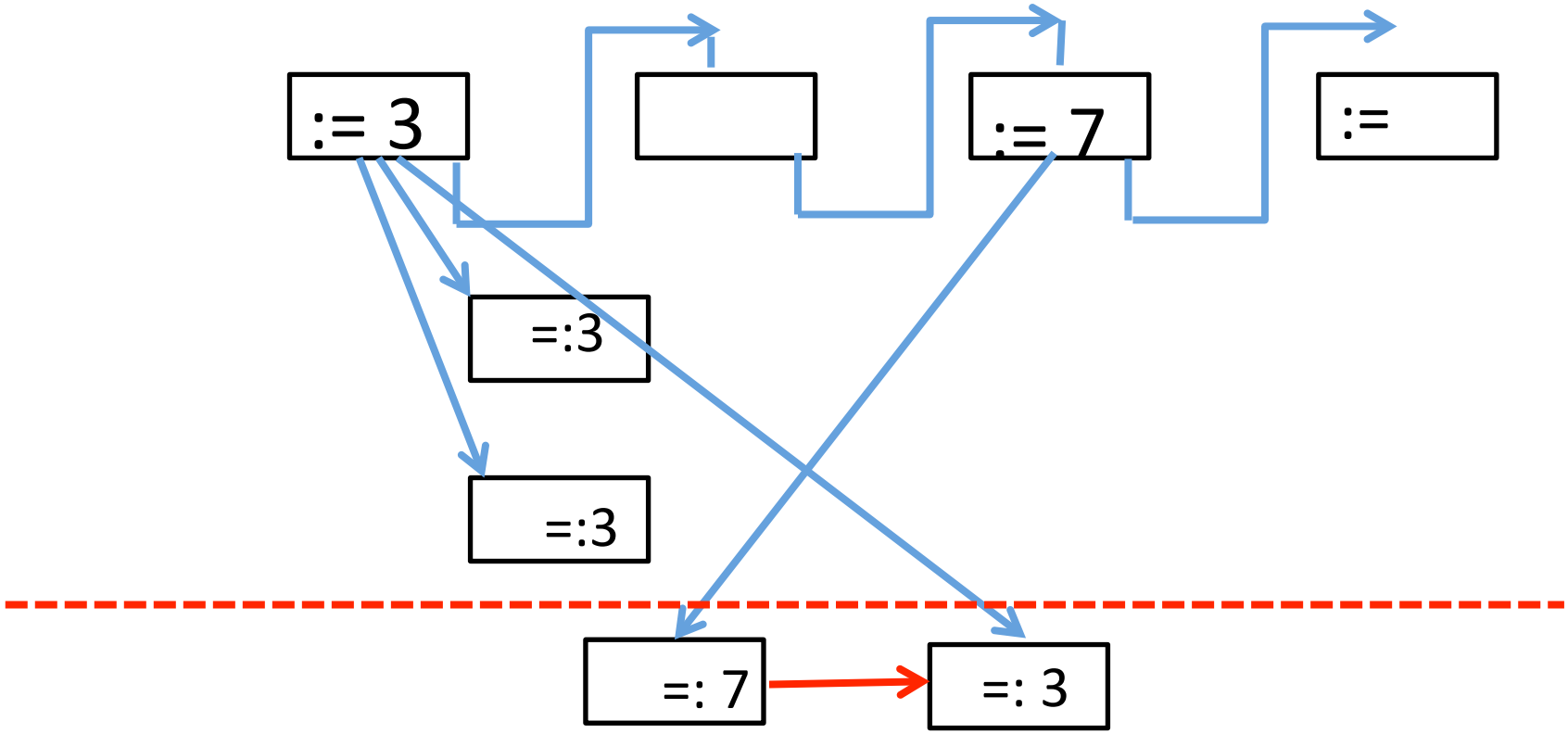
# Token reconstitution



# Weakly consistent memory

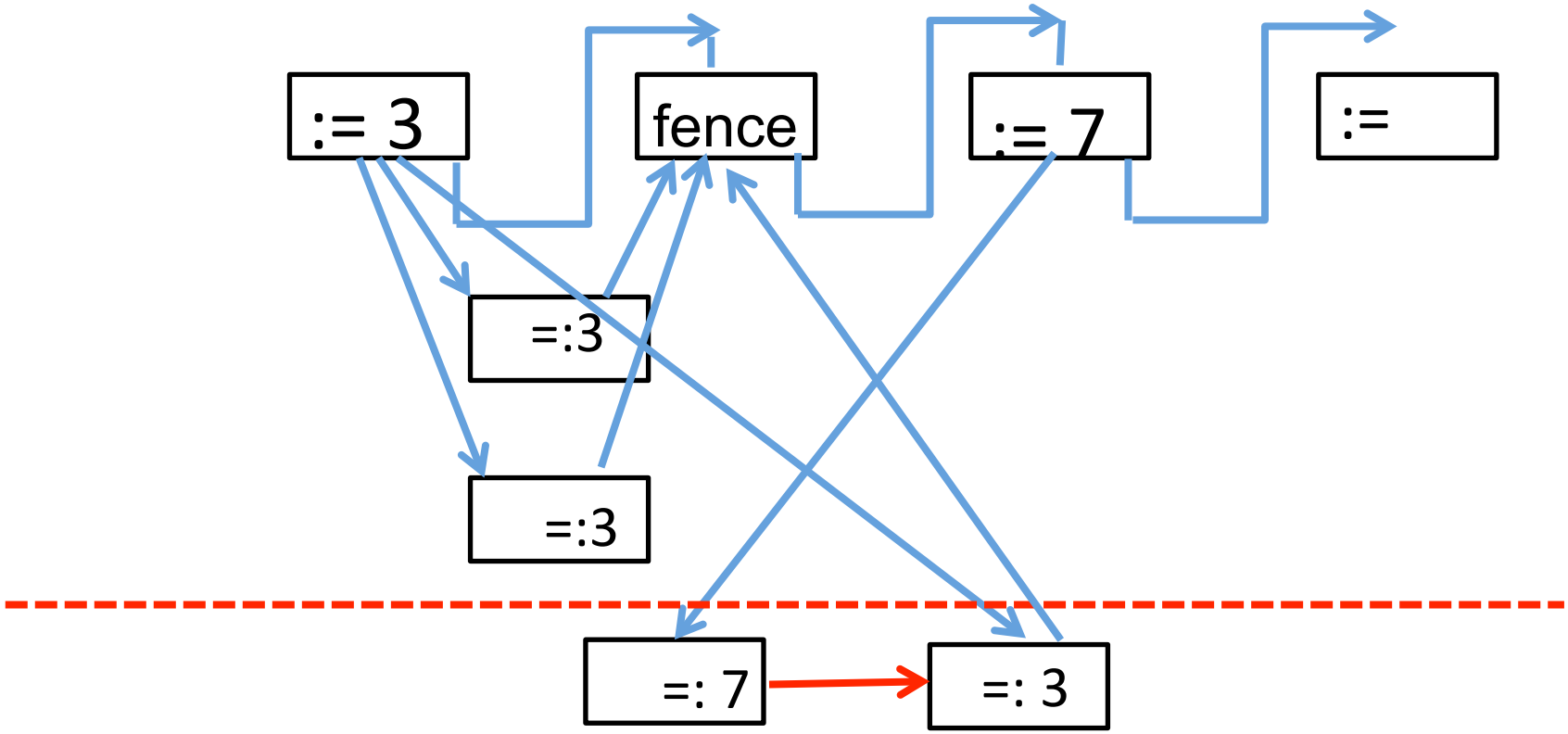
as implemented in multi-core architecture,
is more complicated to define
… and even more complicated to use!

# Weak memory (no ack)



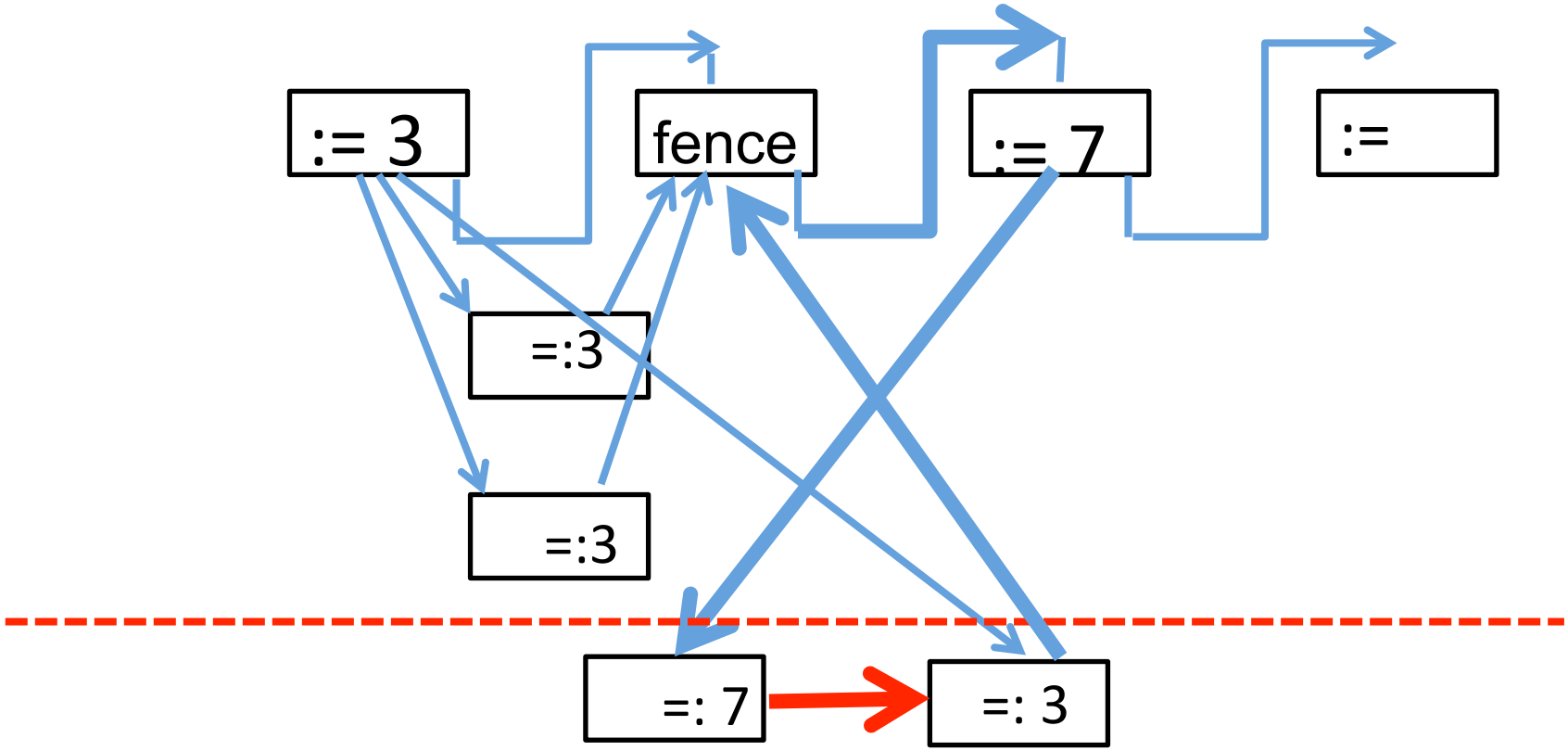

Any past value may be delivered at any later time

# Weak memory (no ack)



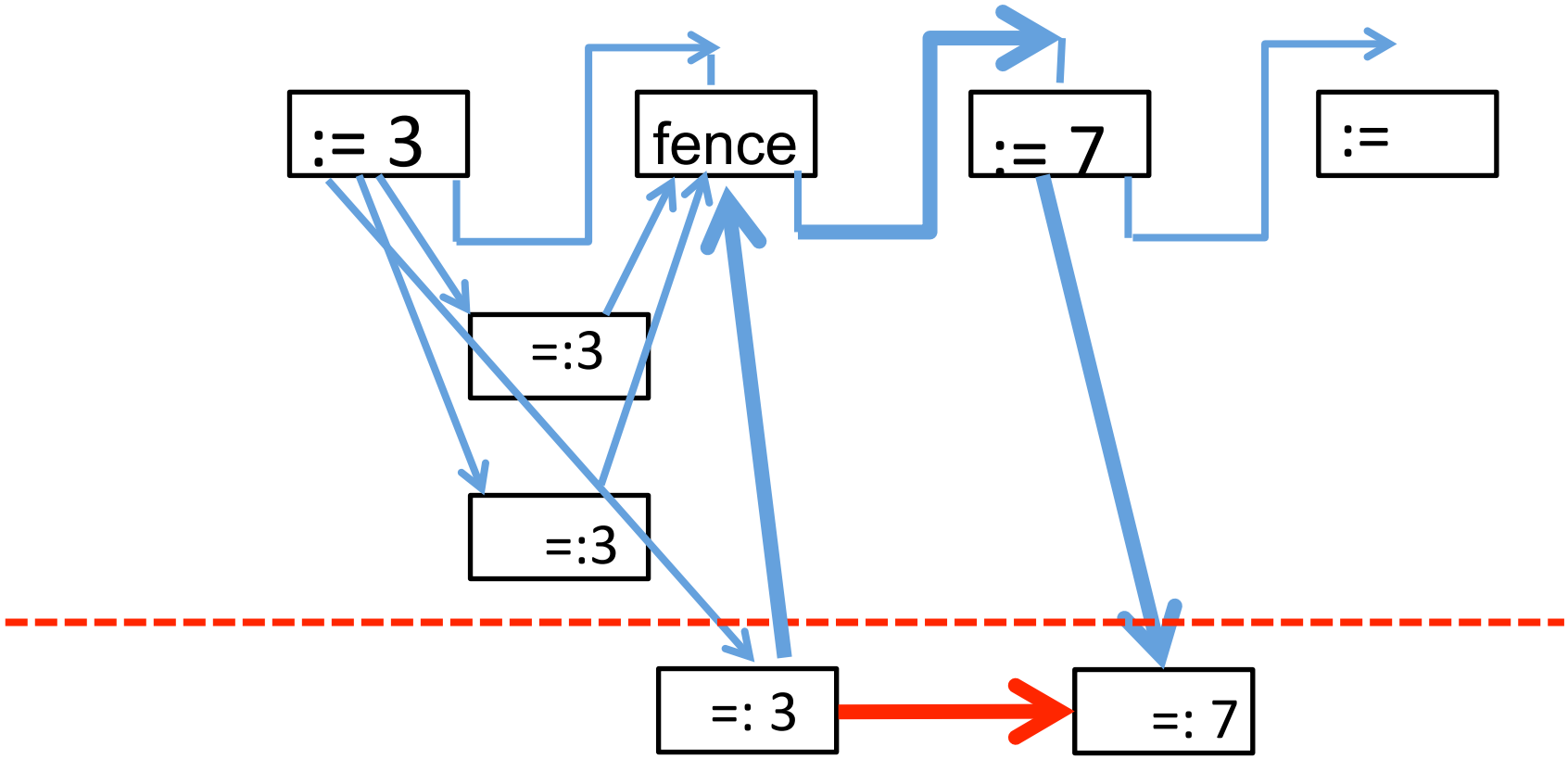

a fence needs ack signals from earlier assignments

# Weak memory (with fence)



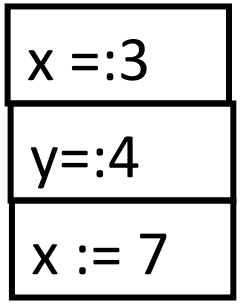

this cycle is impossible,

# Weak memory (with fence)

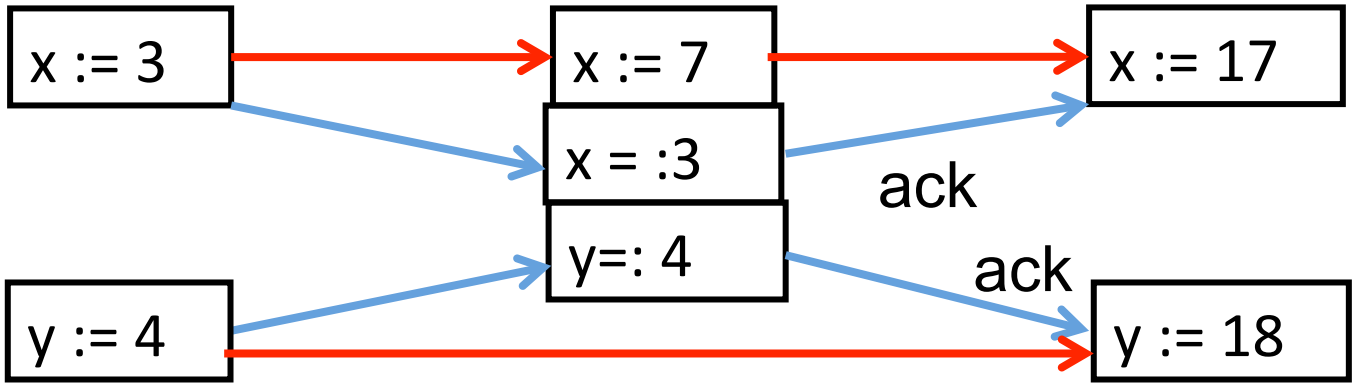



so the 3 and the 7 must be fetched in the right order

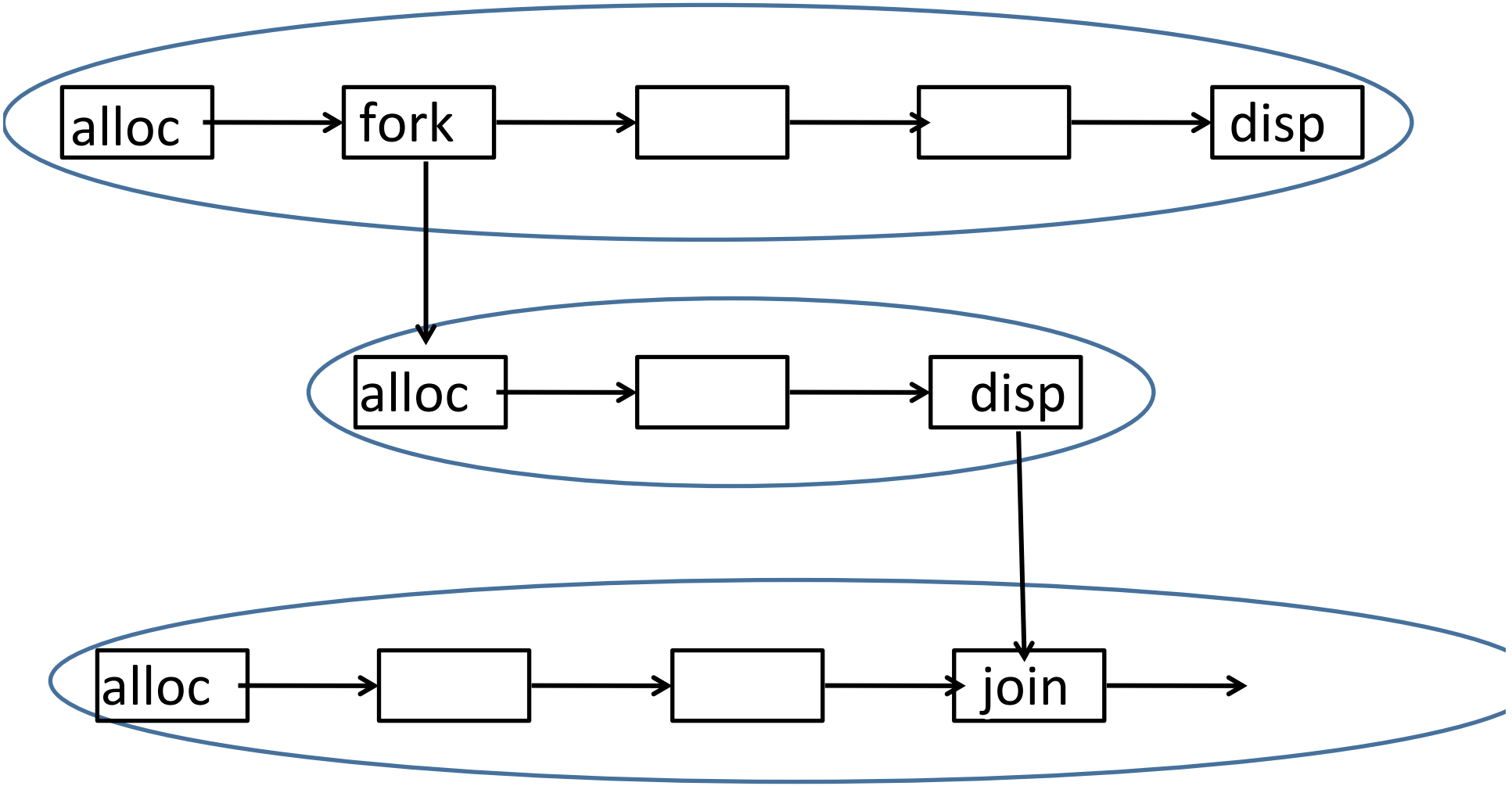
# An atomic event: <x := x + y>



# <x := x + y>  (atomic)



# Threads



# Events and atomic actions

- Each occurrence of an event in the trace of program execution belongs to the trace of exactly one resource (thread, variable, channel,…)
- Atomic actions are groups of synchronised events, including exactly one from the thread which invoked the action, and one (or more) from every resource used by it.

# Summary

- occurrence nets are adequate
  to describe the dynamic behaviour
  of many kinds of concurrent object

# Fundamental Theorem

- Boxed Petri nets are a model of

  Concurrent Kleene Algebra

Tony Hoare, Bernhard Moeller, Georg Struth, Ian Wehrman, Concurrent Kleene Algebra and its Foundations, J. Log. Algebr. Program. 80(6): 266-296 (2011).

# Hoare Logic

- Let P{Q}R = (P:Q) => R

- Theorem: The structural rules of Hoare logic are valid in the net model
  - Ian Wehrman, C.A.R.Hoare, Peter O'Hearn: Graphical Models of Separation Logic. Inf Process. Lett. (IPL) 109(17):1001-1004 (2009)
- Proof: by a short algebraic calculation

# Process Algebra

- Let P –Q-> R = R => Q;P

  = P{Q}R !

- Theorem: The transition rules of operational semantics are valid in the net model.
  - C.A.R.Hoare, A.Hussain, B.Moeller, P.W. O'Hearn, R.L. Petersen, G. Struth. On Locality and the Exchange Law for Concurrent Processes. CONCUR 2011:250-264.
- Proof: by a short algebraic calculation.

# Acknowledgements

Lucia Pomello, Matthew Parkinson,
Philippa Gardiner, Hongseok Yang,
John Wickerson,  Thomas Dinsdale-Young
Georg Struth, Bernhard Moeller,
Rasmus Petersen, Peter O'Hearn