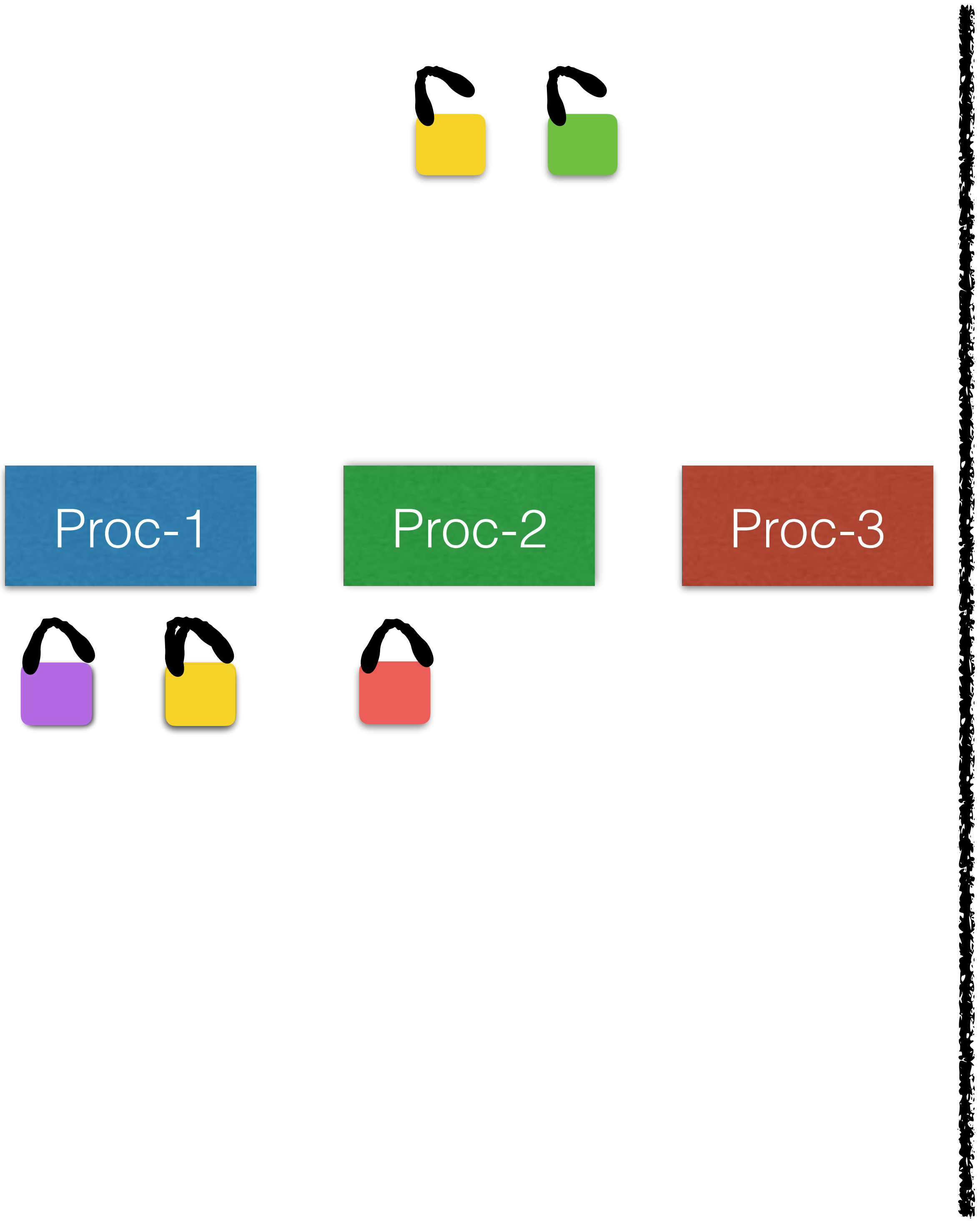
# Verifying Asynchronous programs with nested locks

K Narayan Kumar
CMI, Chennai

Joint work with

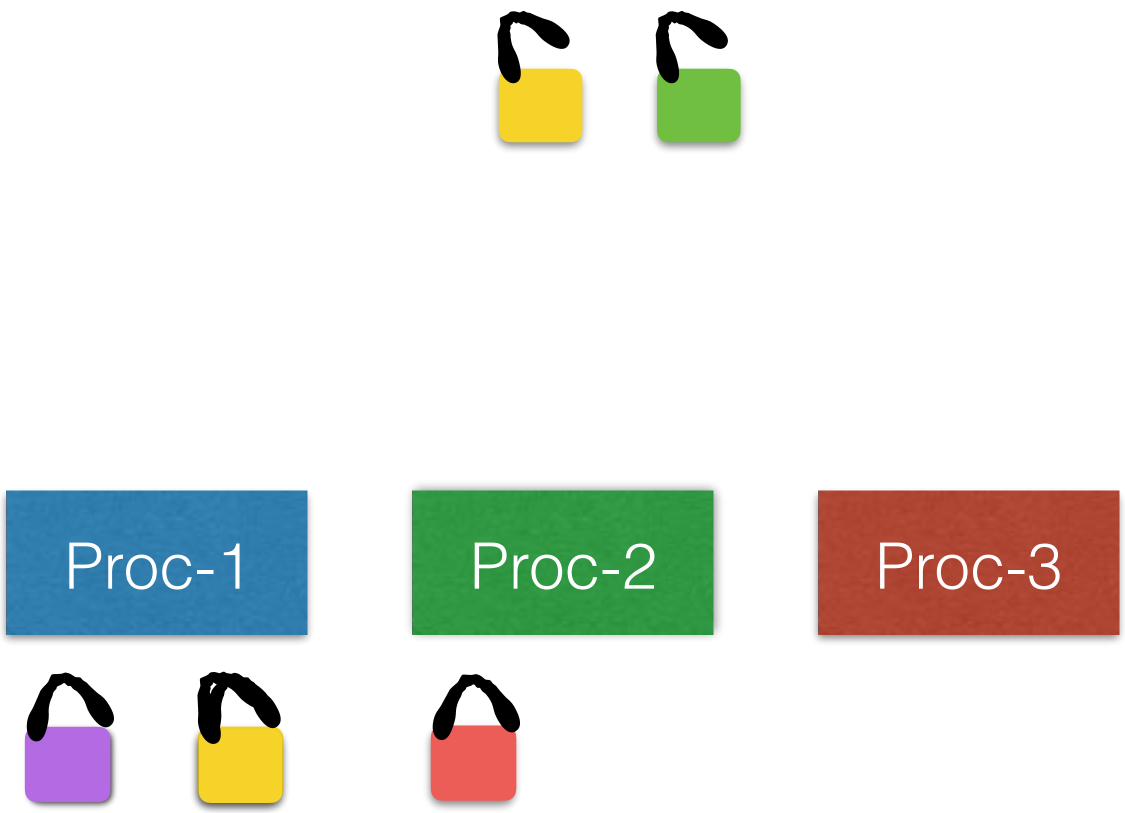
- M.F. Atig
- A. Bouajjani
- Prakash Saivasan

# Programs with Locks:



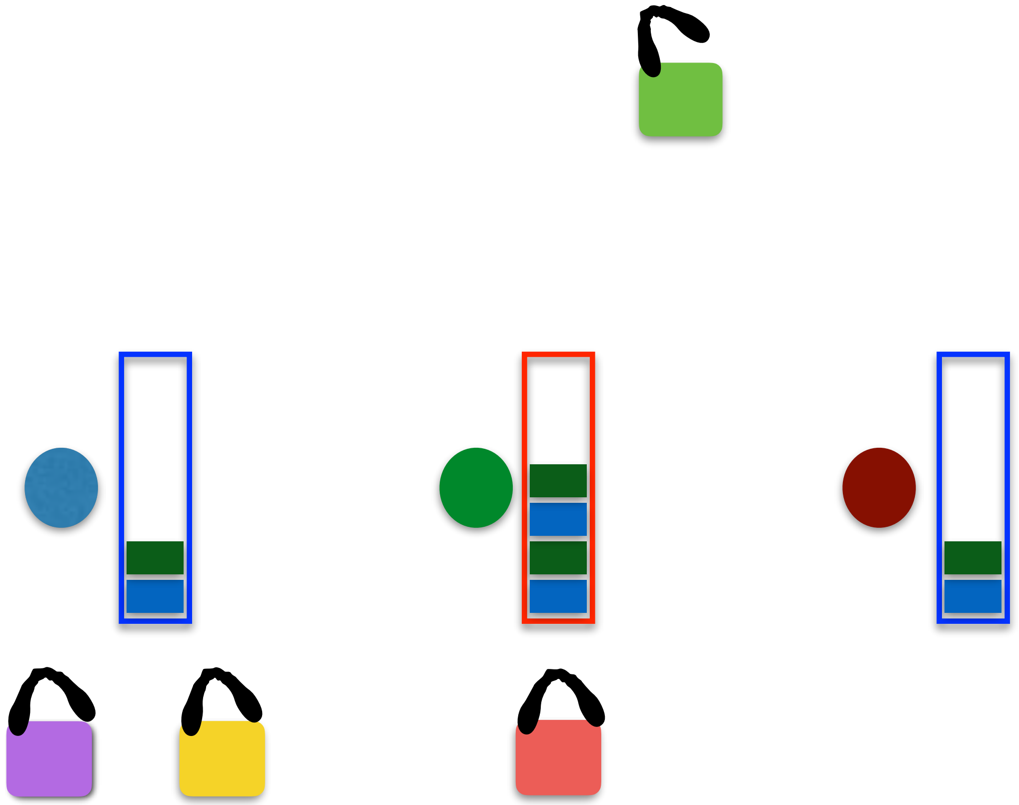
- A collection of processes executing concurrently.
- A finite set of Locks

# Programs with Locks:



- A collection of processes executing concurrently.
- A finite set of Locks
- Processes may take locks that are available and release locks that they hold.

# Programs with Locks:



- Our processes will be recursive processes (over finite data domains)
- Modelled as Pushdown Systems

# Why Locks

- Useful coordination mechanism.

- Can be built with protocols over shared memory. Usually supported by hardware.

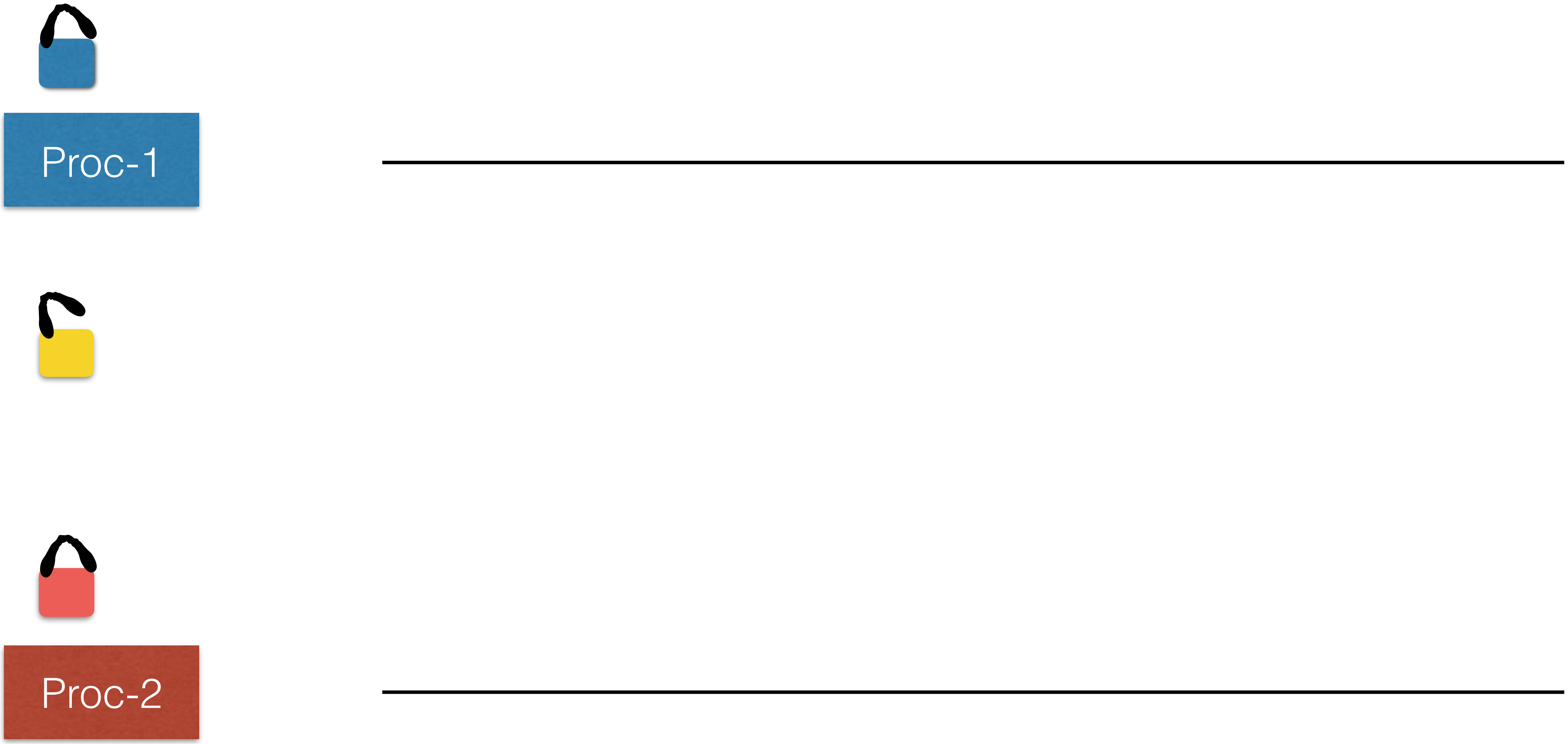
- Available in many programming languages …

# Why Locks

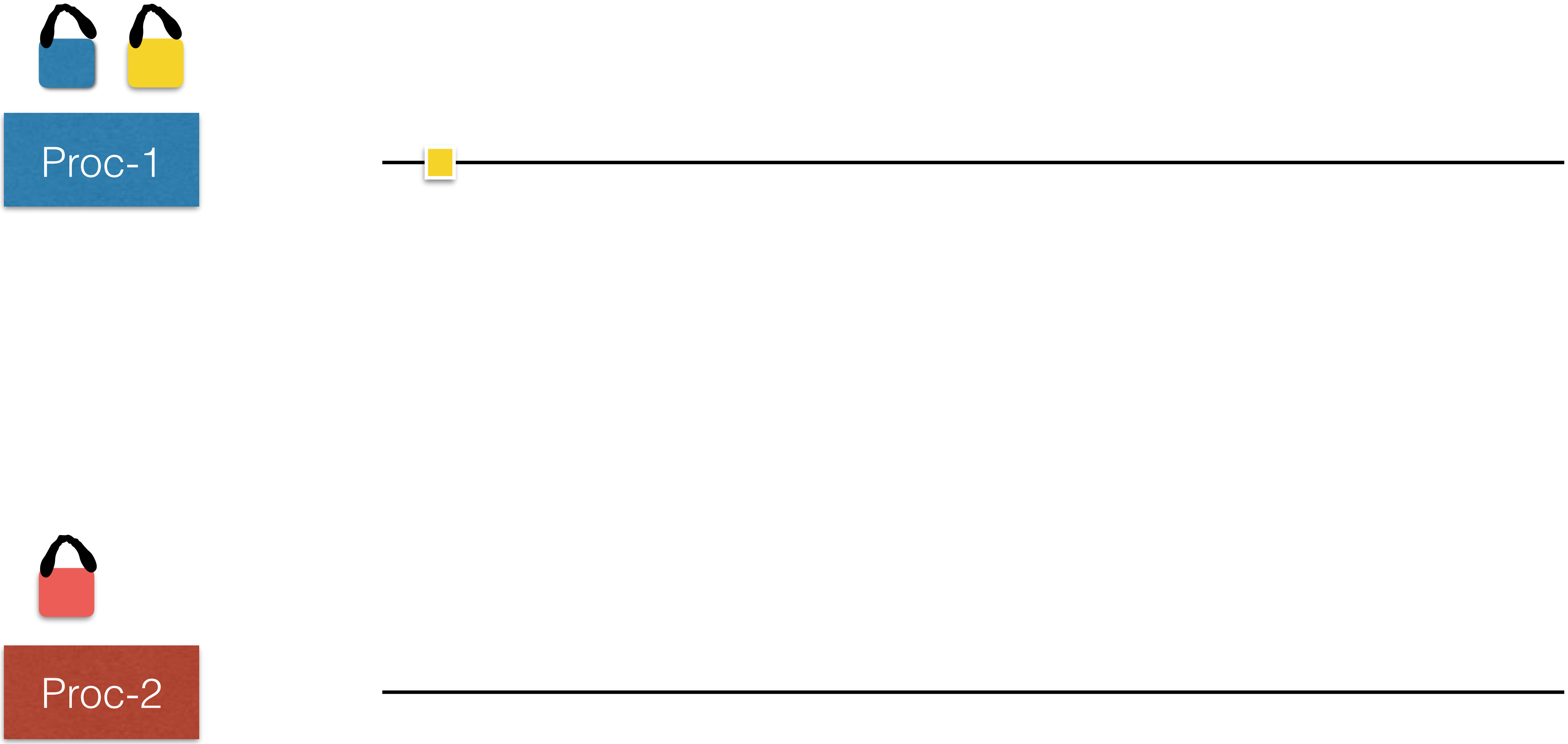
- Useful coordination mechanism.
- Can be built with protocols over shared memory. Usually supported by hardware.
- Available in many programming languages …

How good are they?
Can processes “synchronize” using just locks?

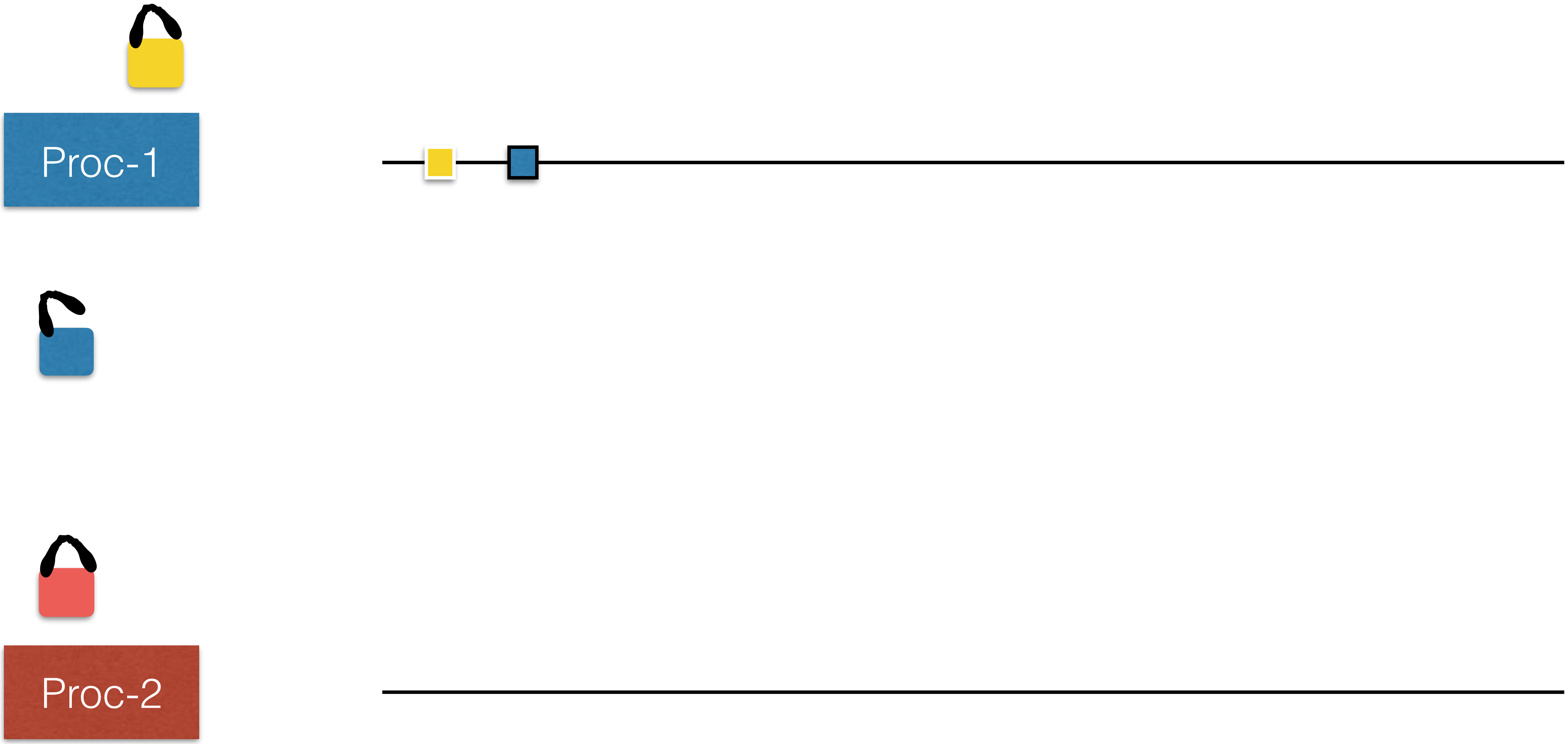
# Synchronizing via Locks



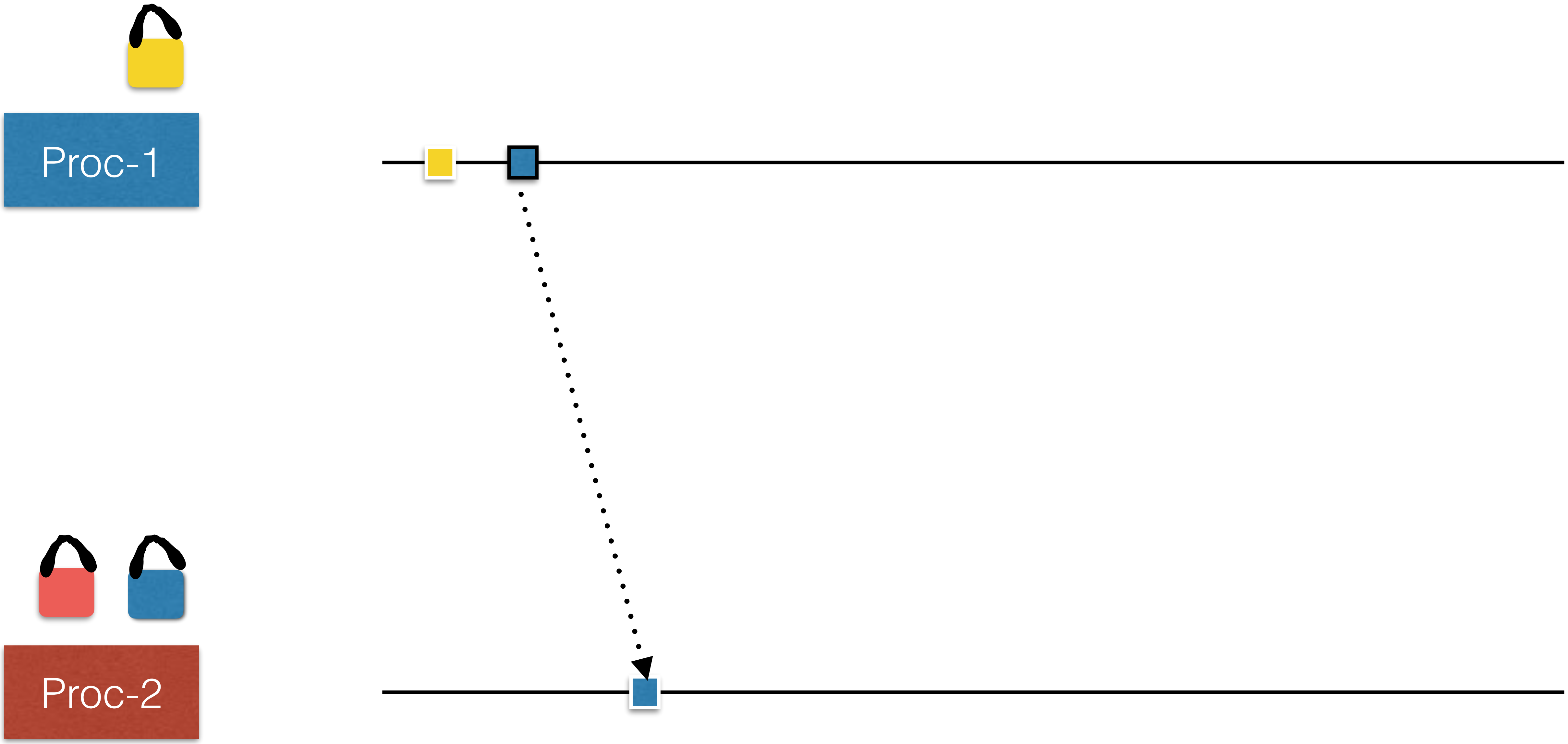
# Synchronizing via Locks



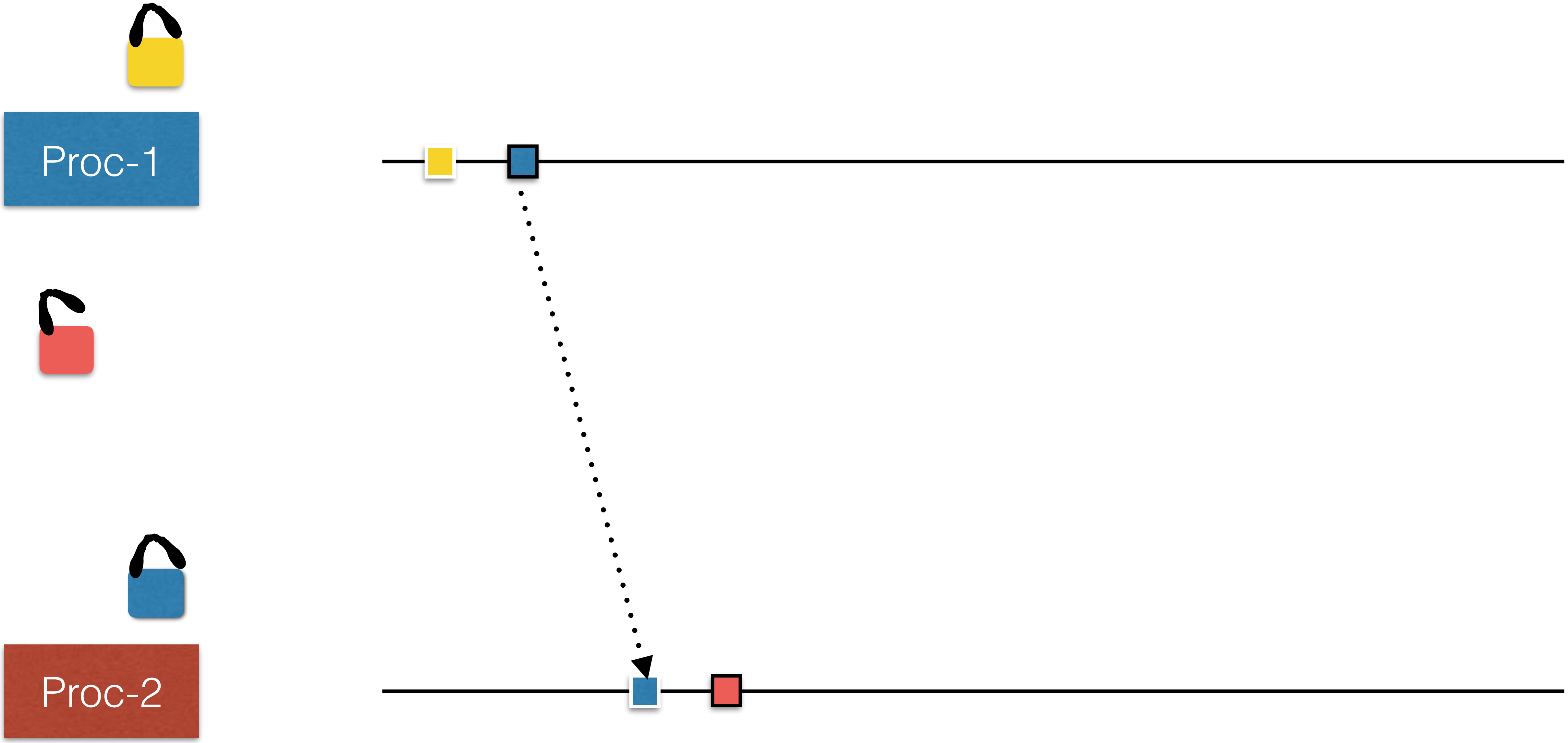
# Synchronizing via Locks



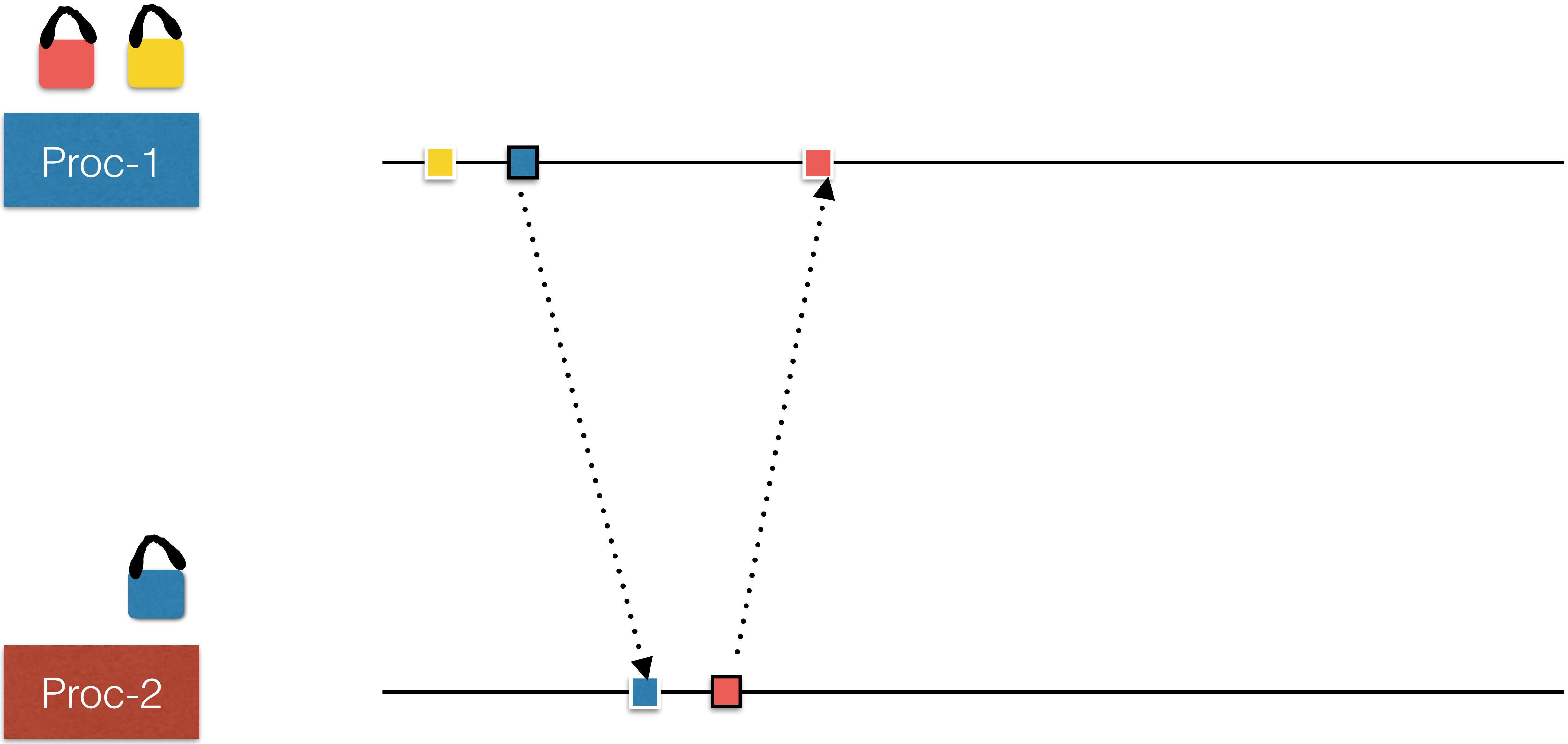
# Synchronizing via Locks



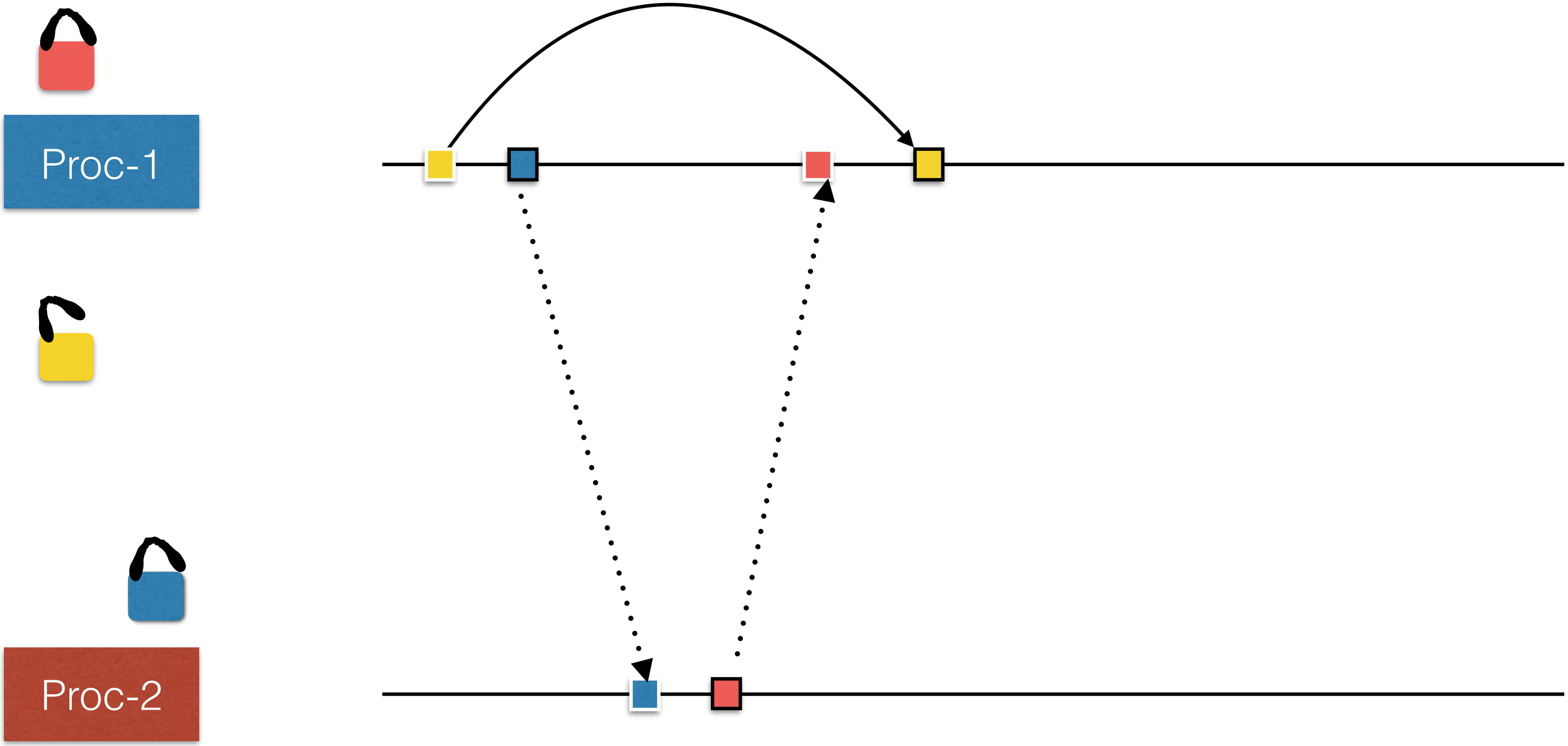
# Synchronizing via Locks



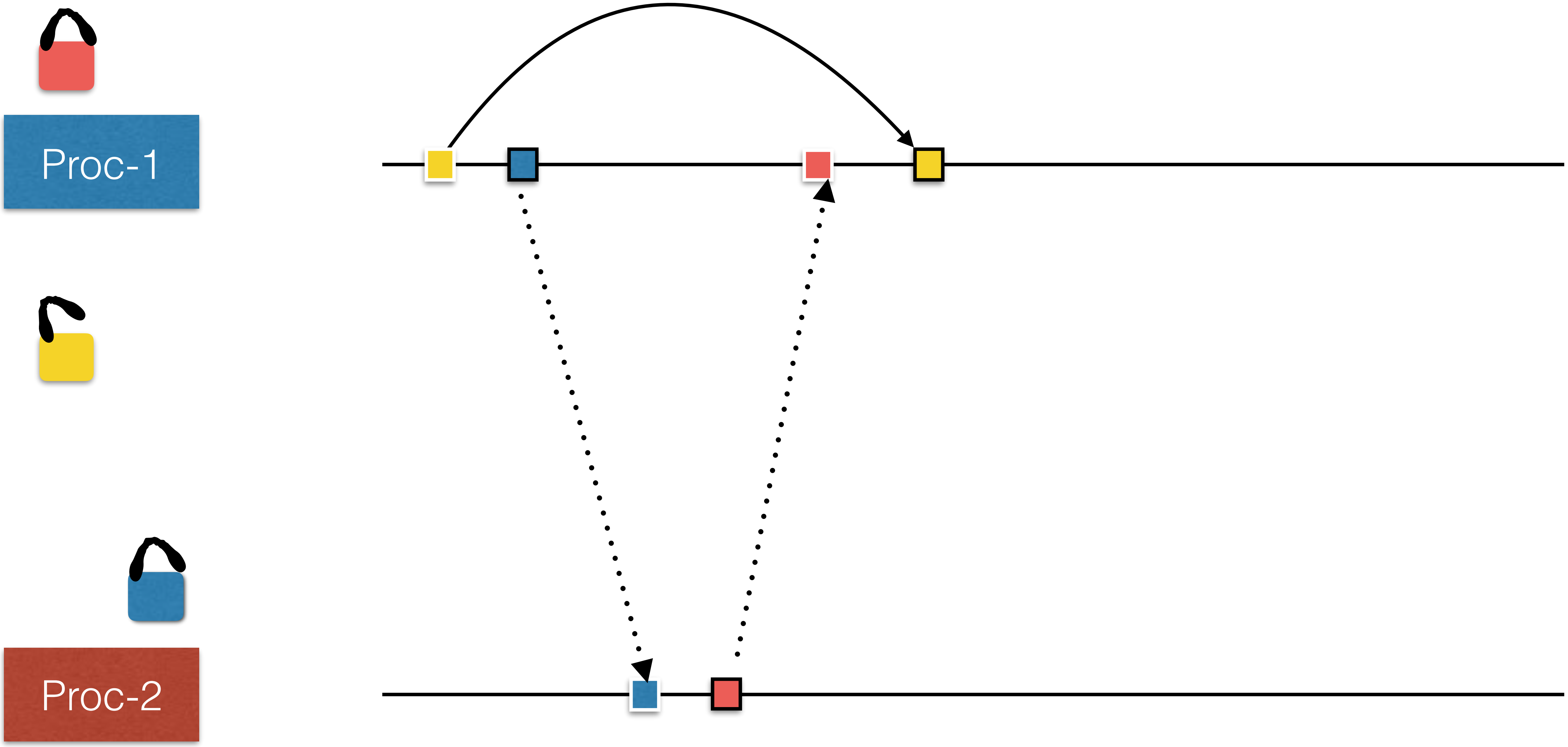
# Synchronizing via Locks



# Synchronizing via Locks

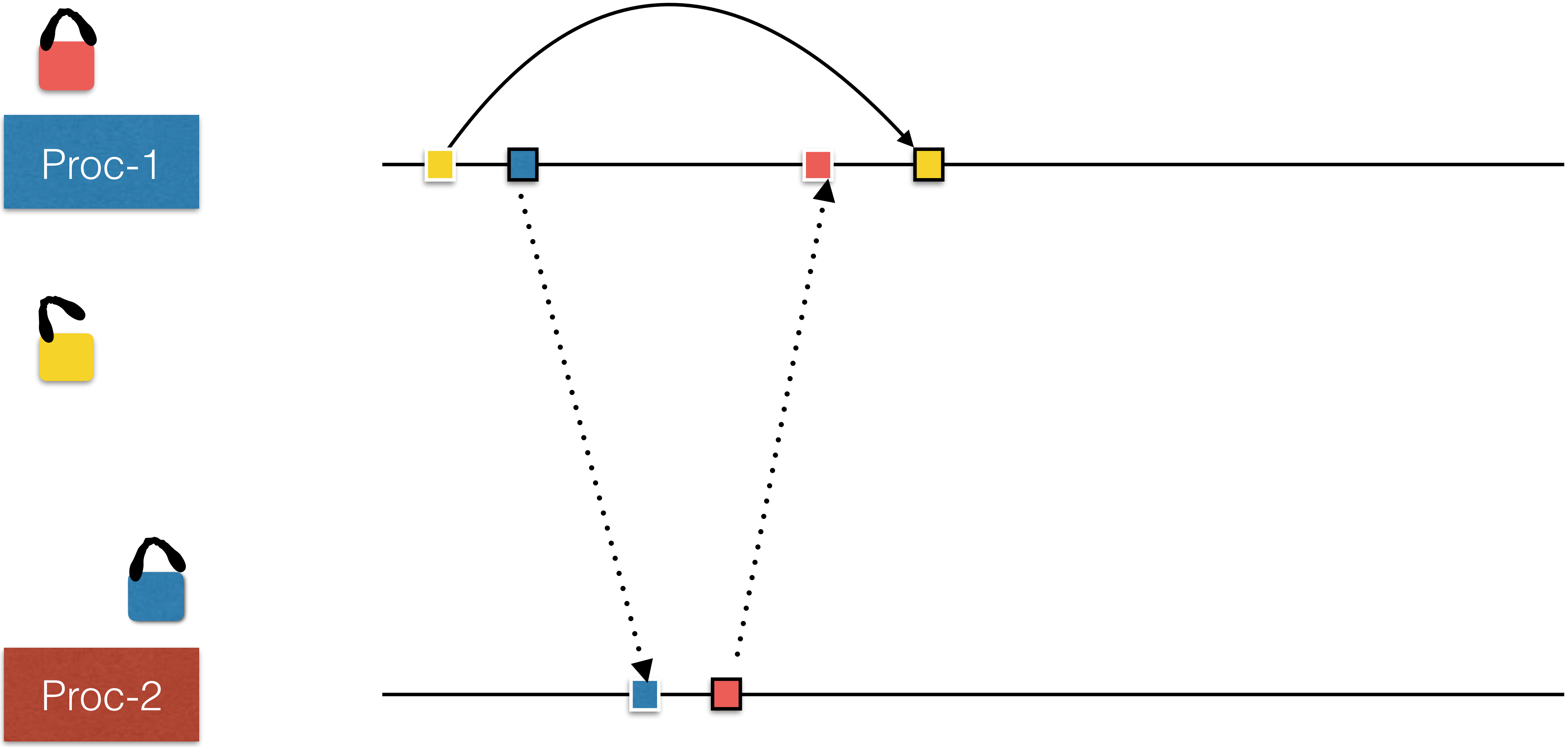


# Synchronizing via Locks

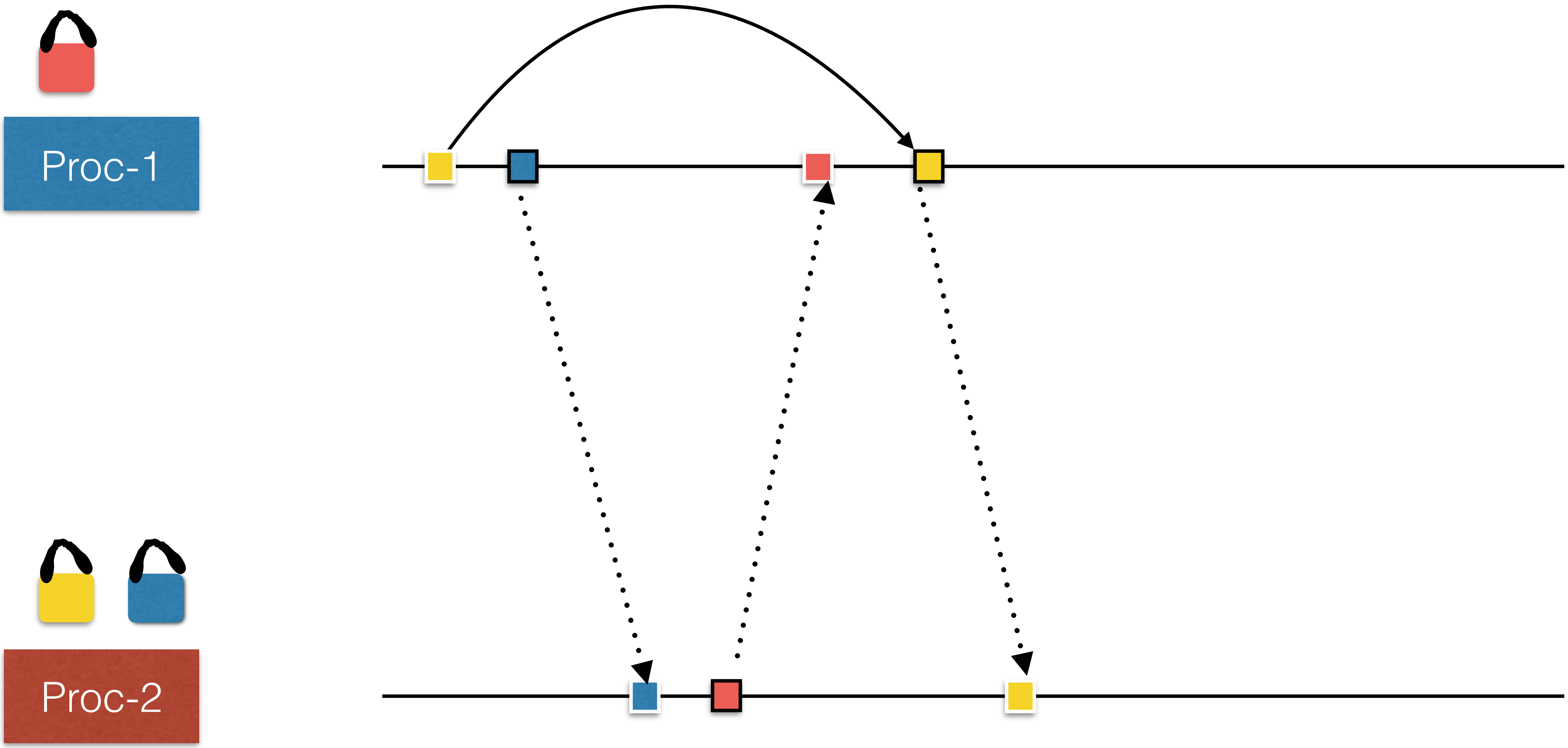


Locks Exchanged.

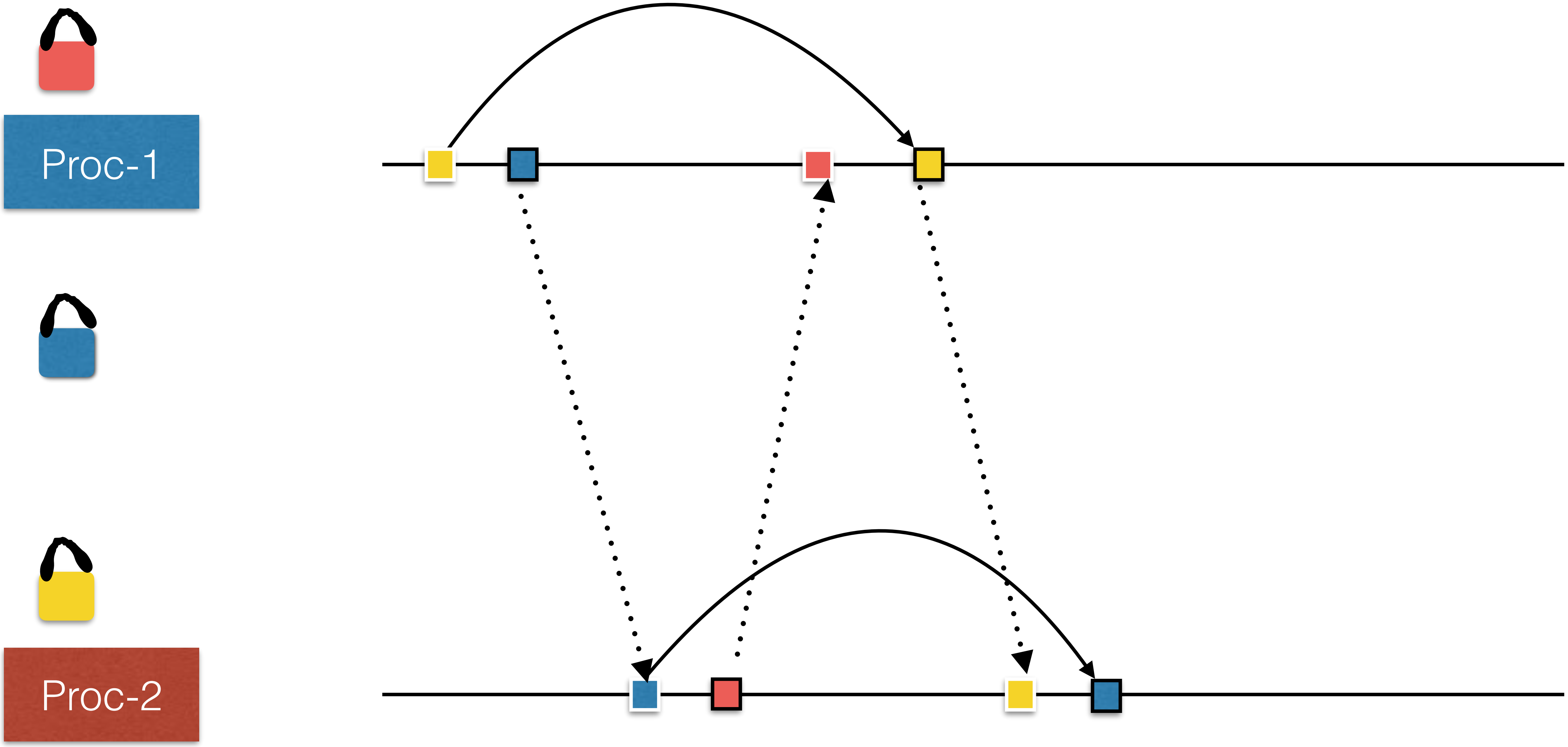
# Synchronizing via Locks



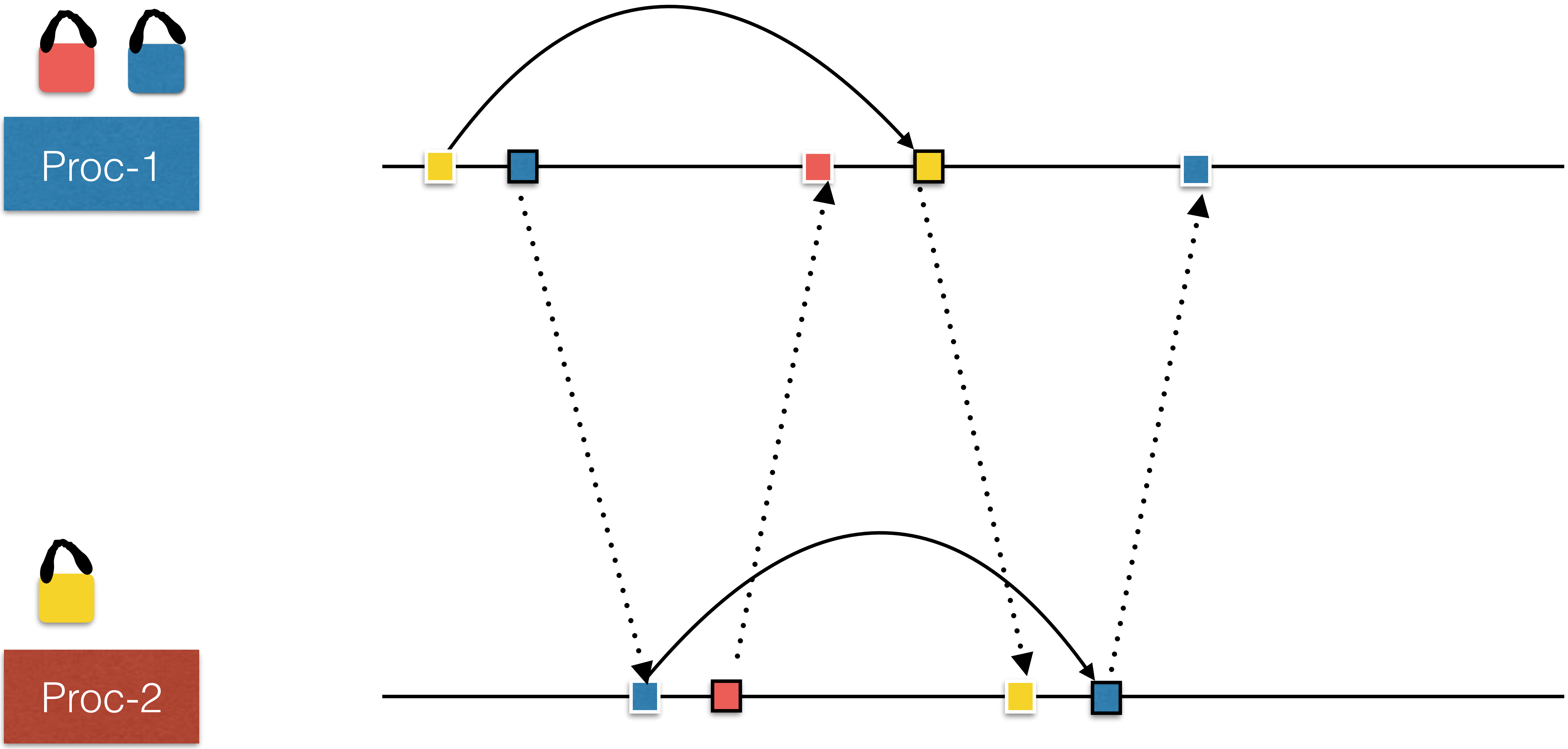
# Synchronizing via Locks



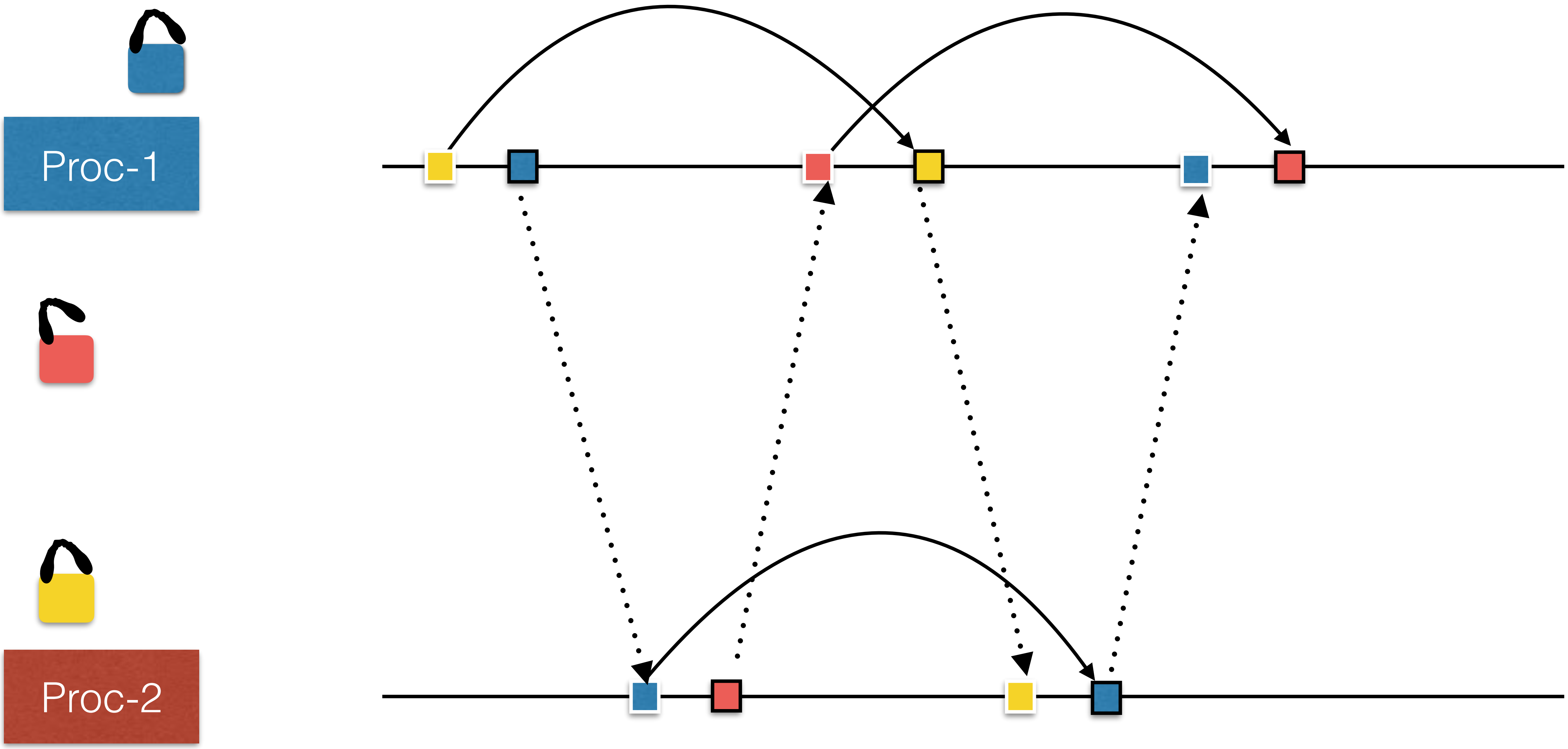
# Synchronizing via Locks



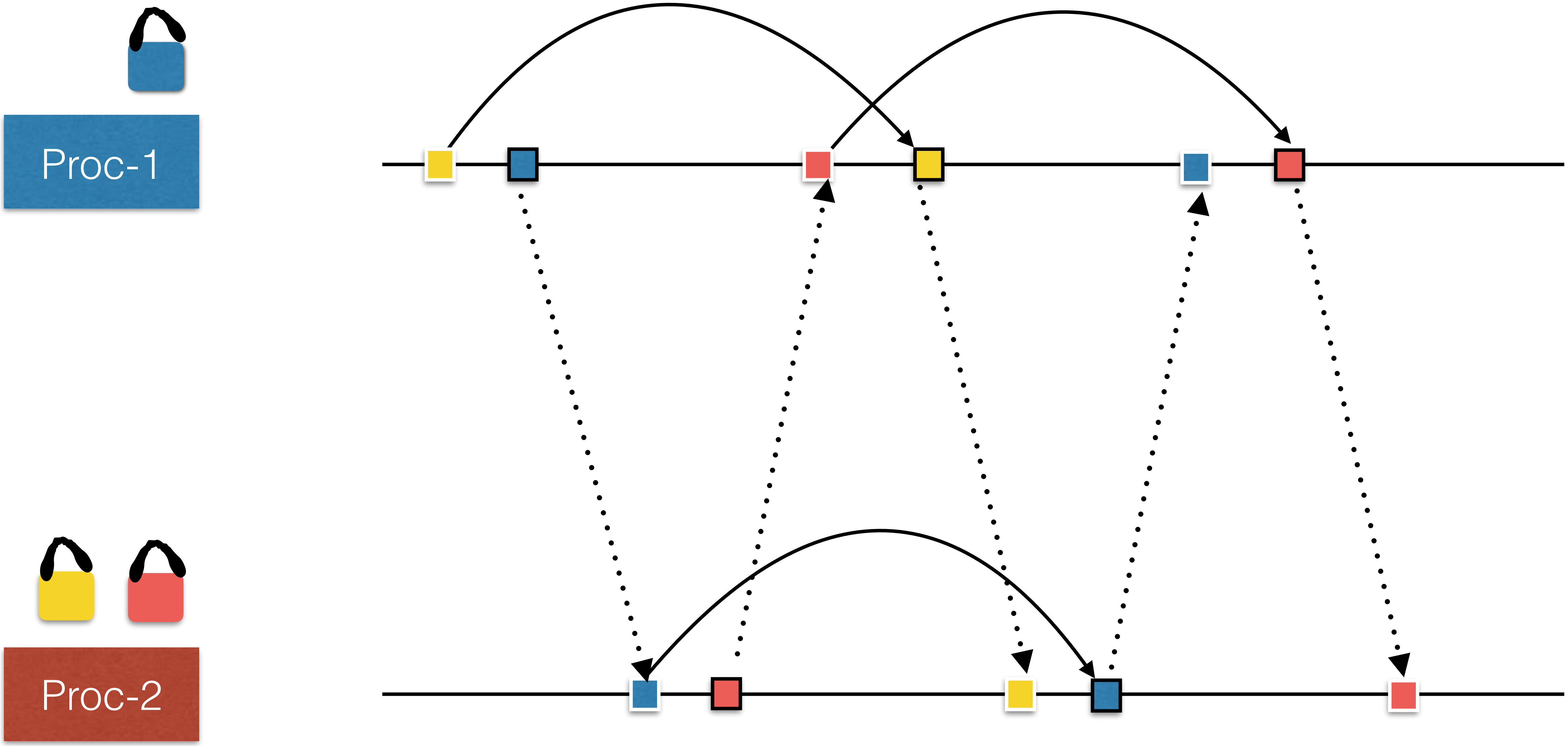
# Synchronizing via Locks



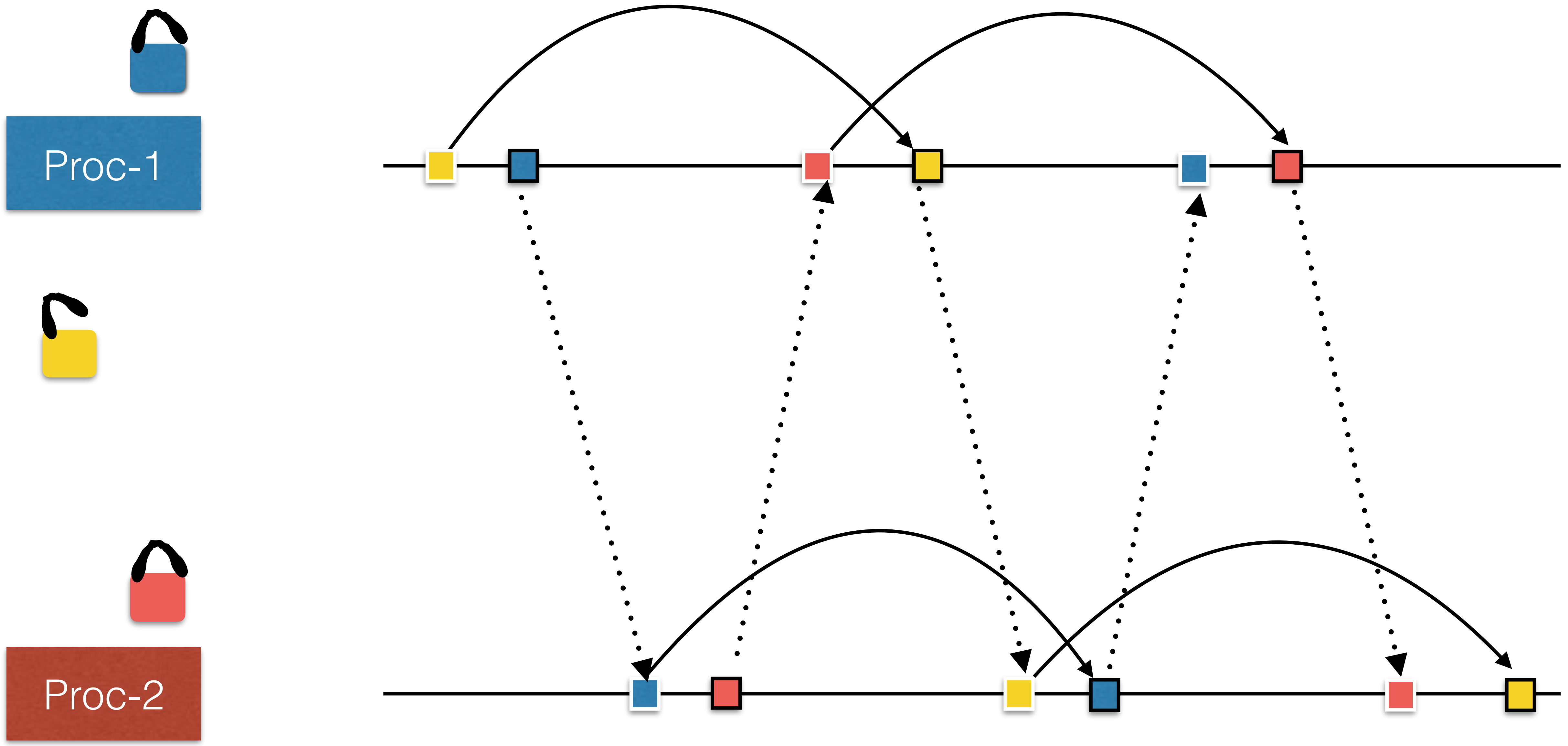
# Synchronizing via Locks



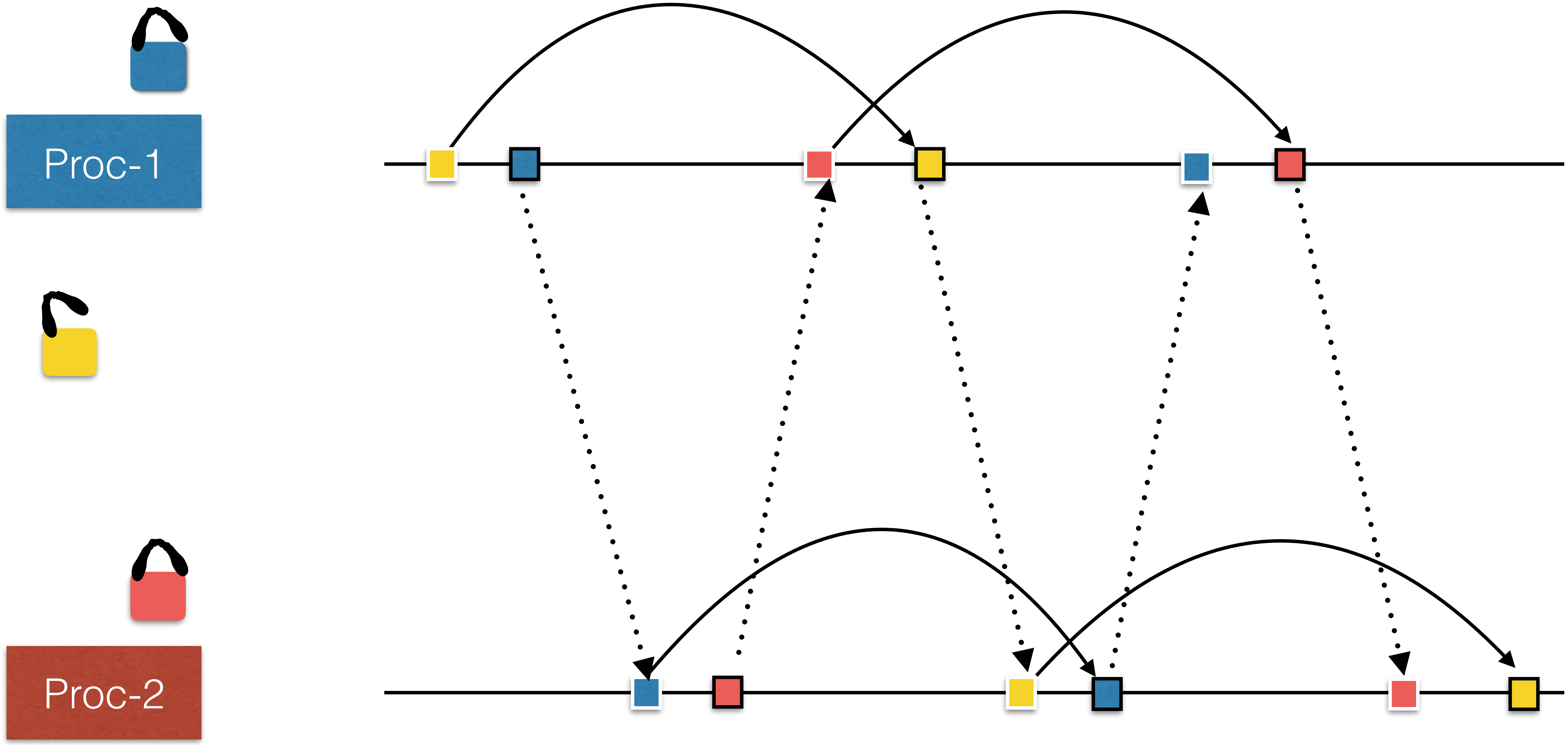
# Synchronizing via Locks



# Synchronizing via Locks

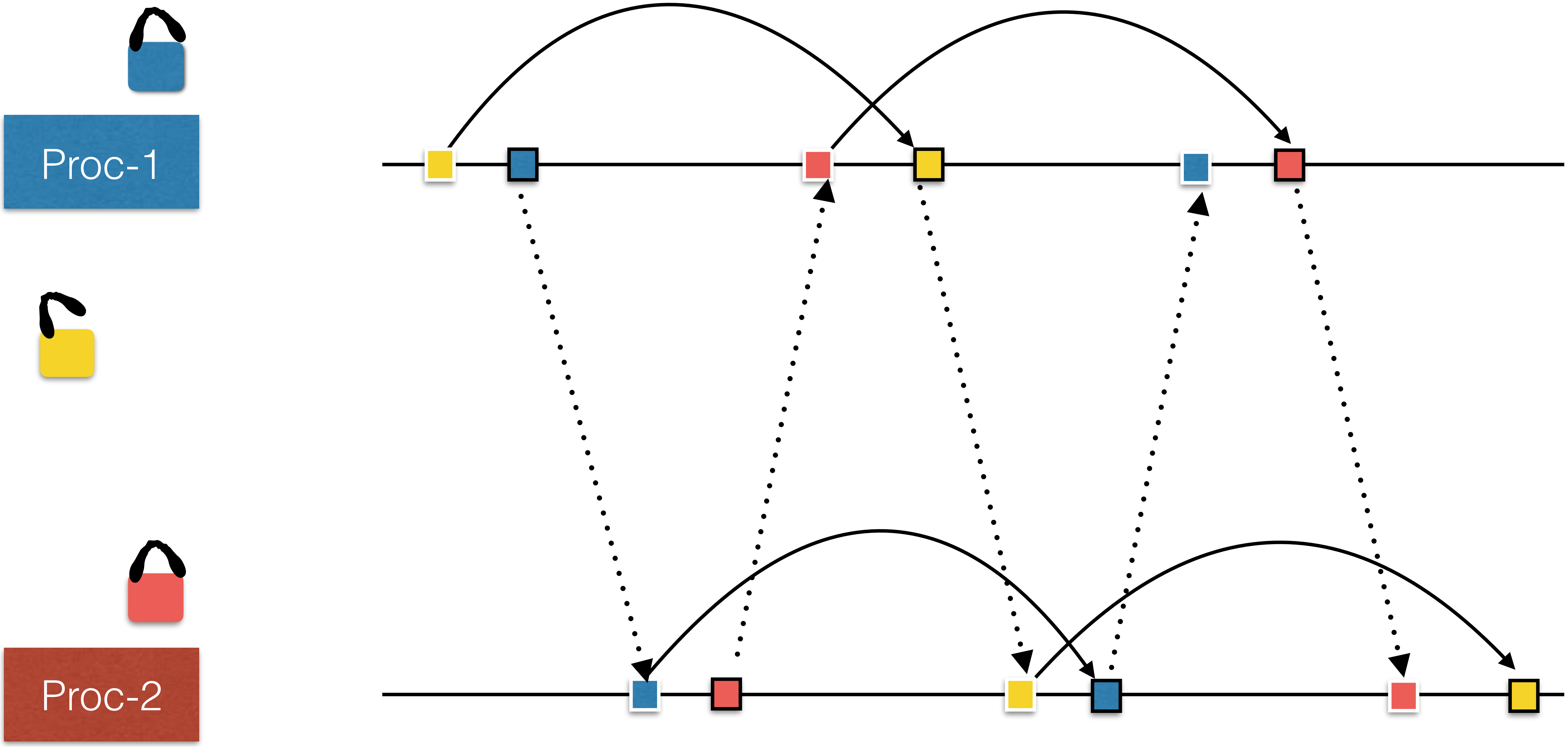


# Synchronizing via Locks

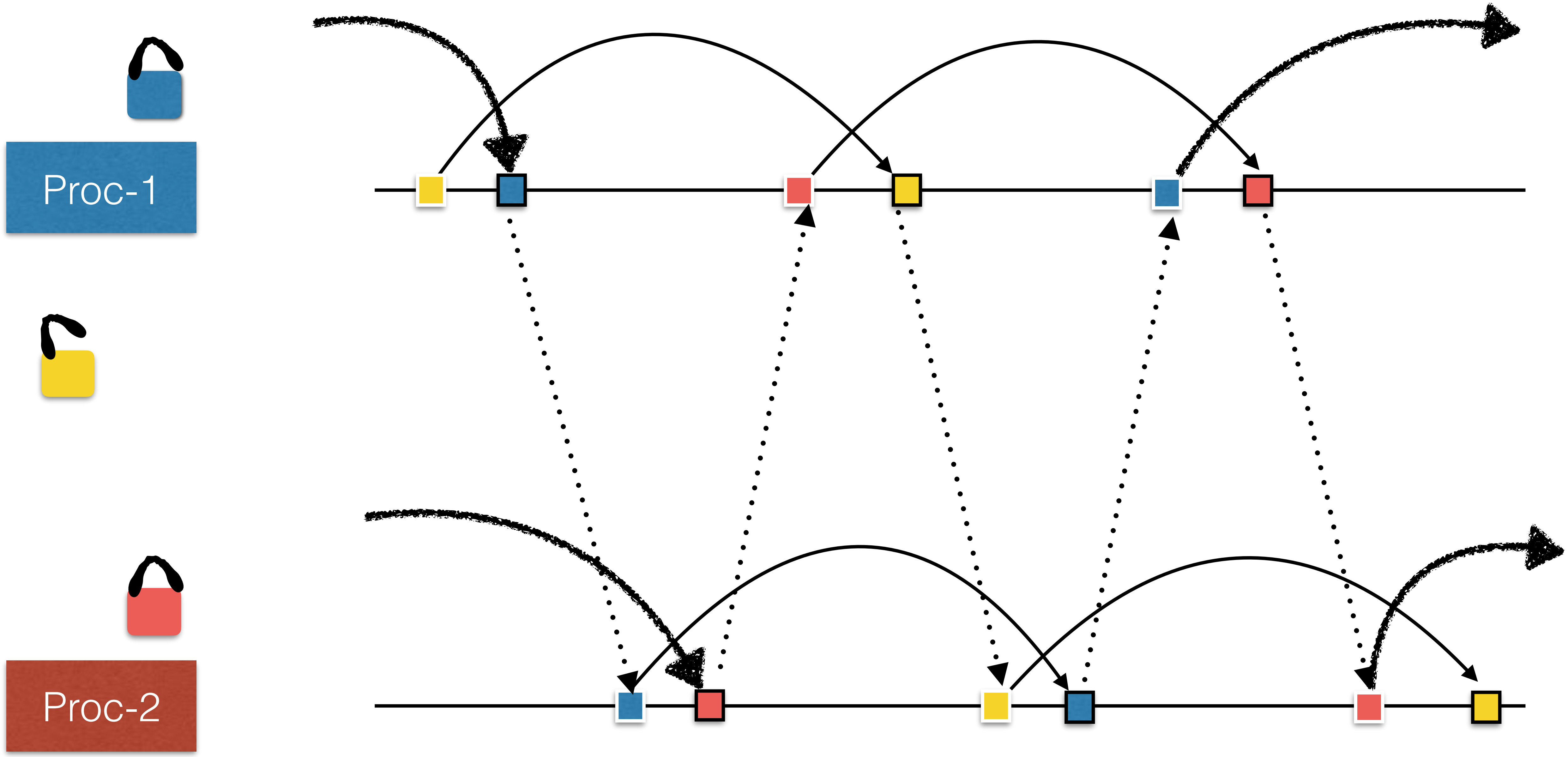


Locking not well-nested

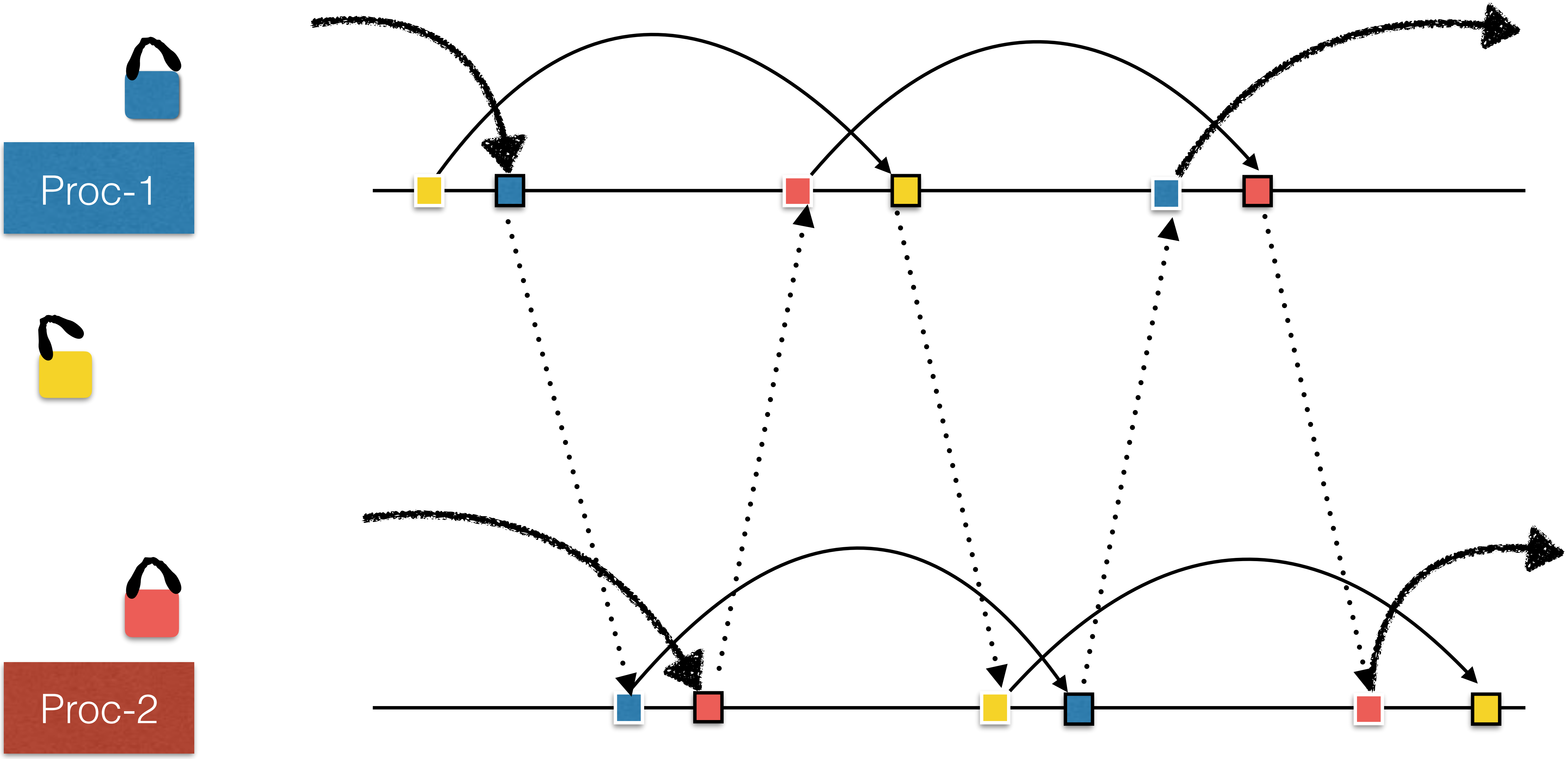
# Synchronizing via Locks



# Synchronizing via Locks



# Synchronizing via Locks



Chaining of locks. Unboundedly long chains.

# Reachability:

The control state reachability problem asks if a given global state can be reached from the initial configuration

Reachability problem for a (even two) recursive programs (PDS) with locks is undecidable.

Ramalingam TOPLAS 2000,
Kahlon,Ivancic,Gupta CAV05

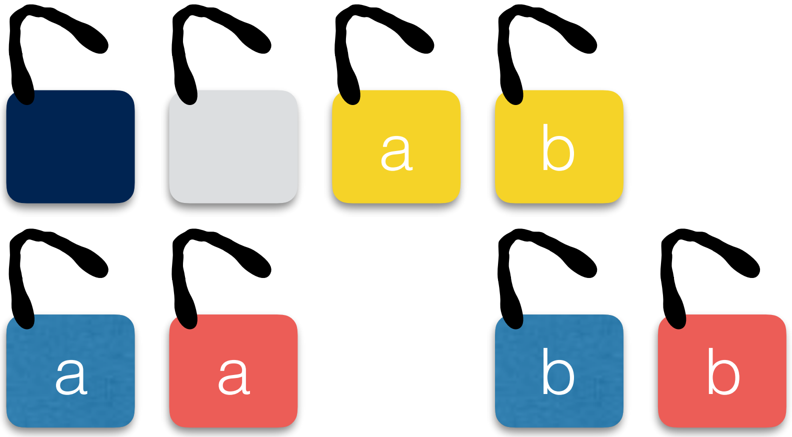
# Initial Condition on Locks:

What if we need all locks to be free at the beginning?

A somewhat more elaborate protocol with additional locks works.

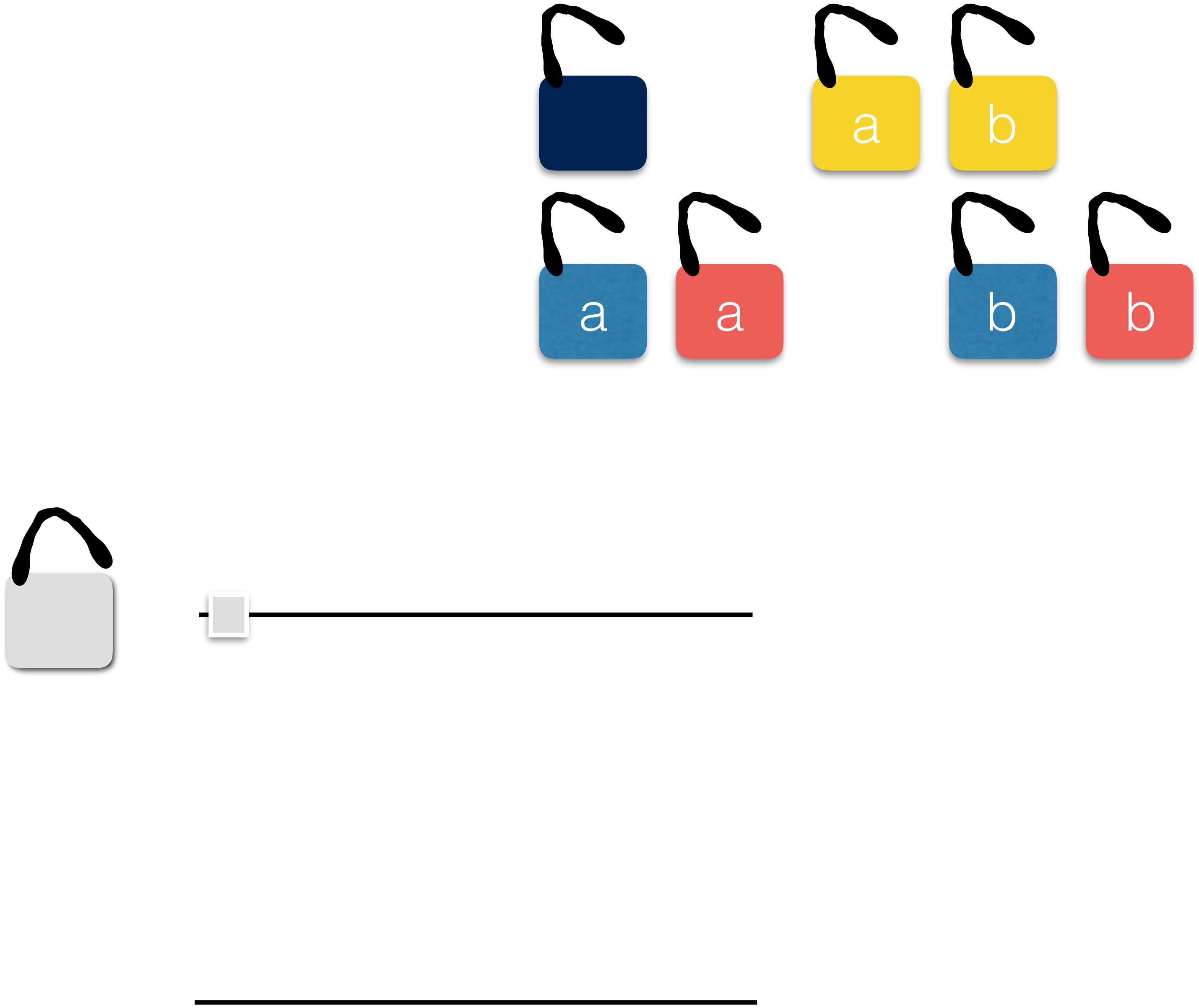
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



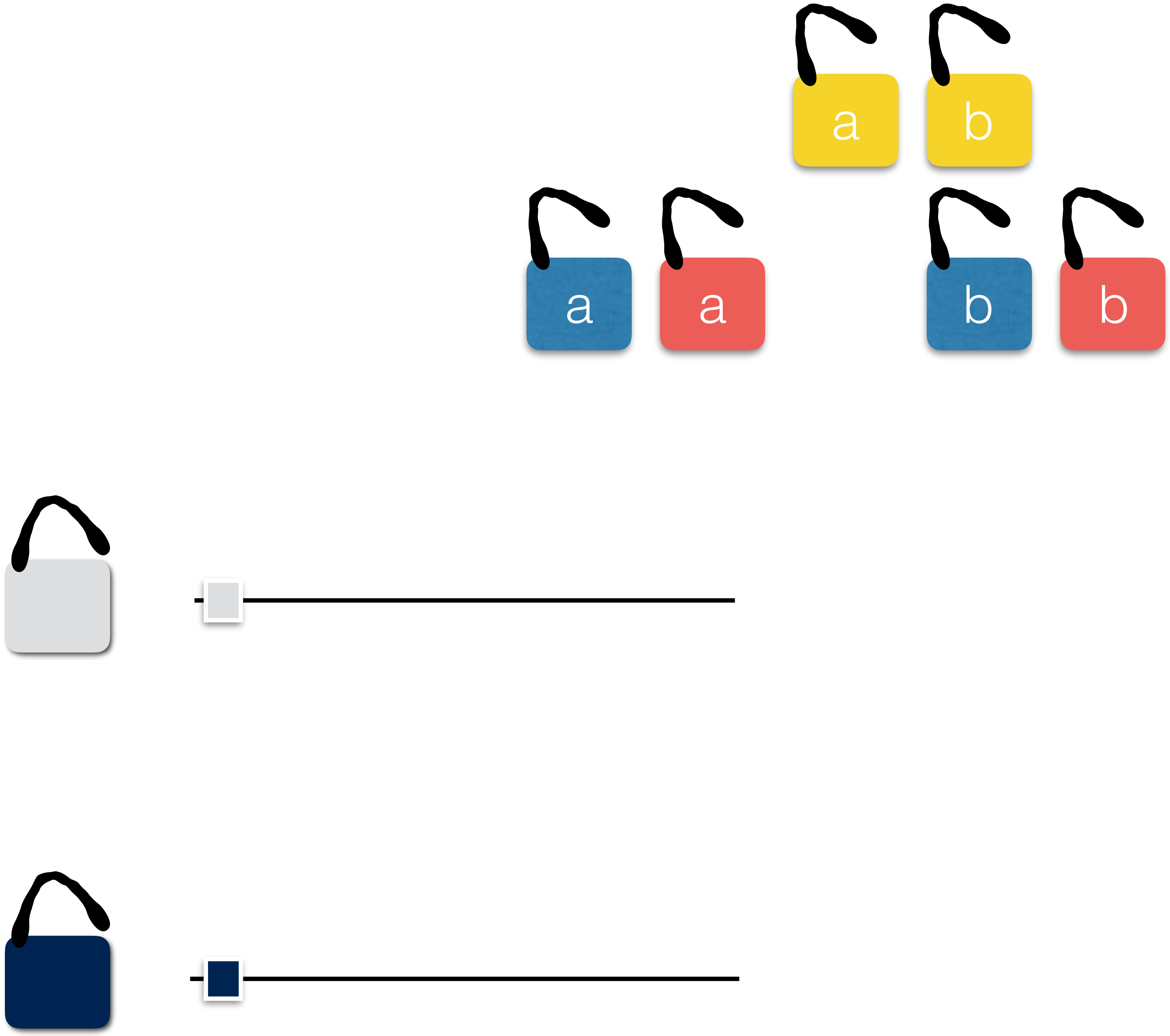
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



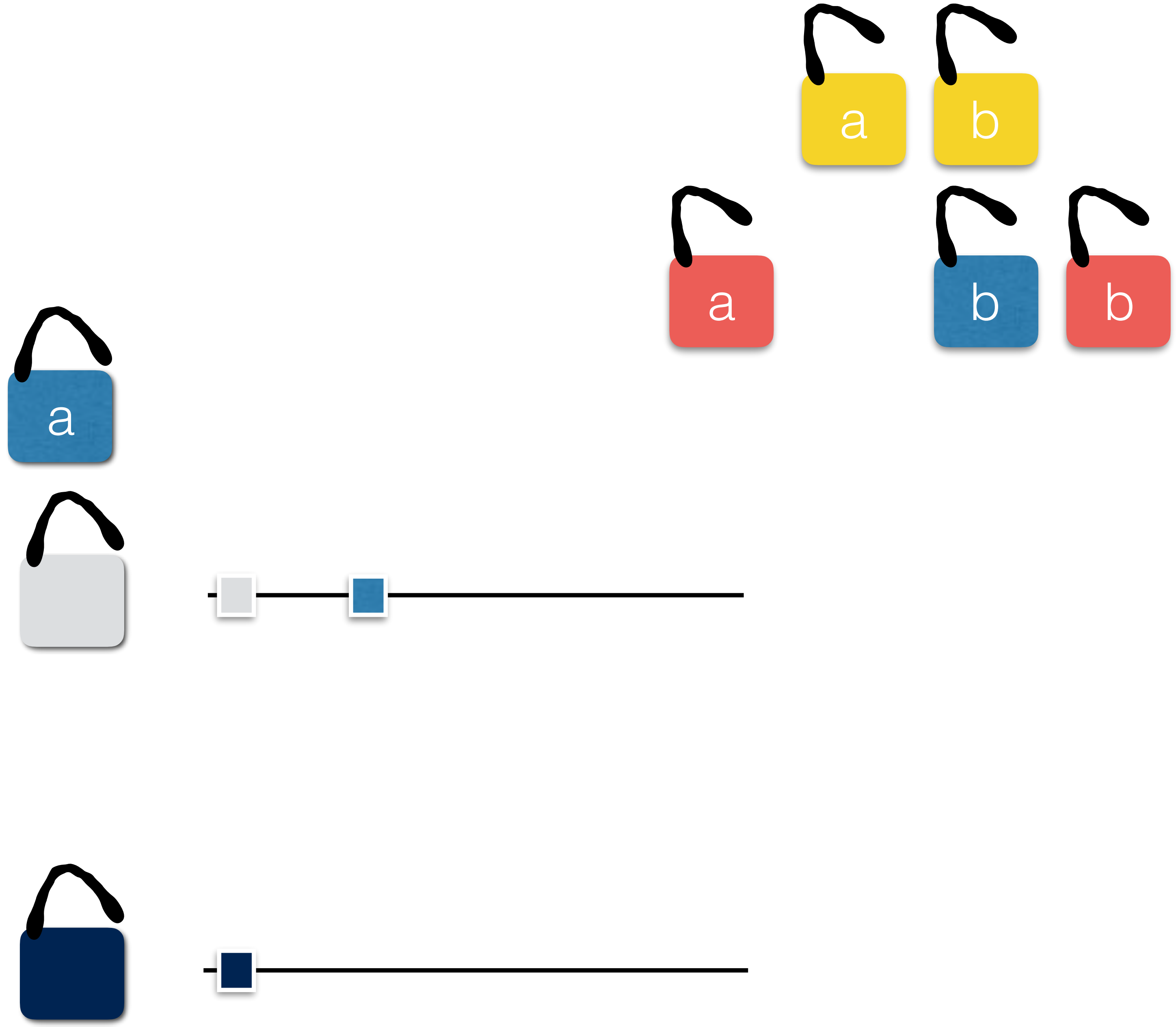
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



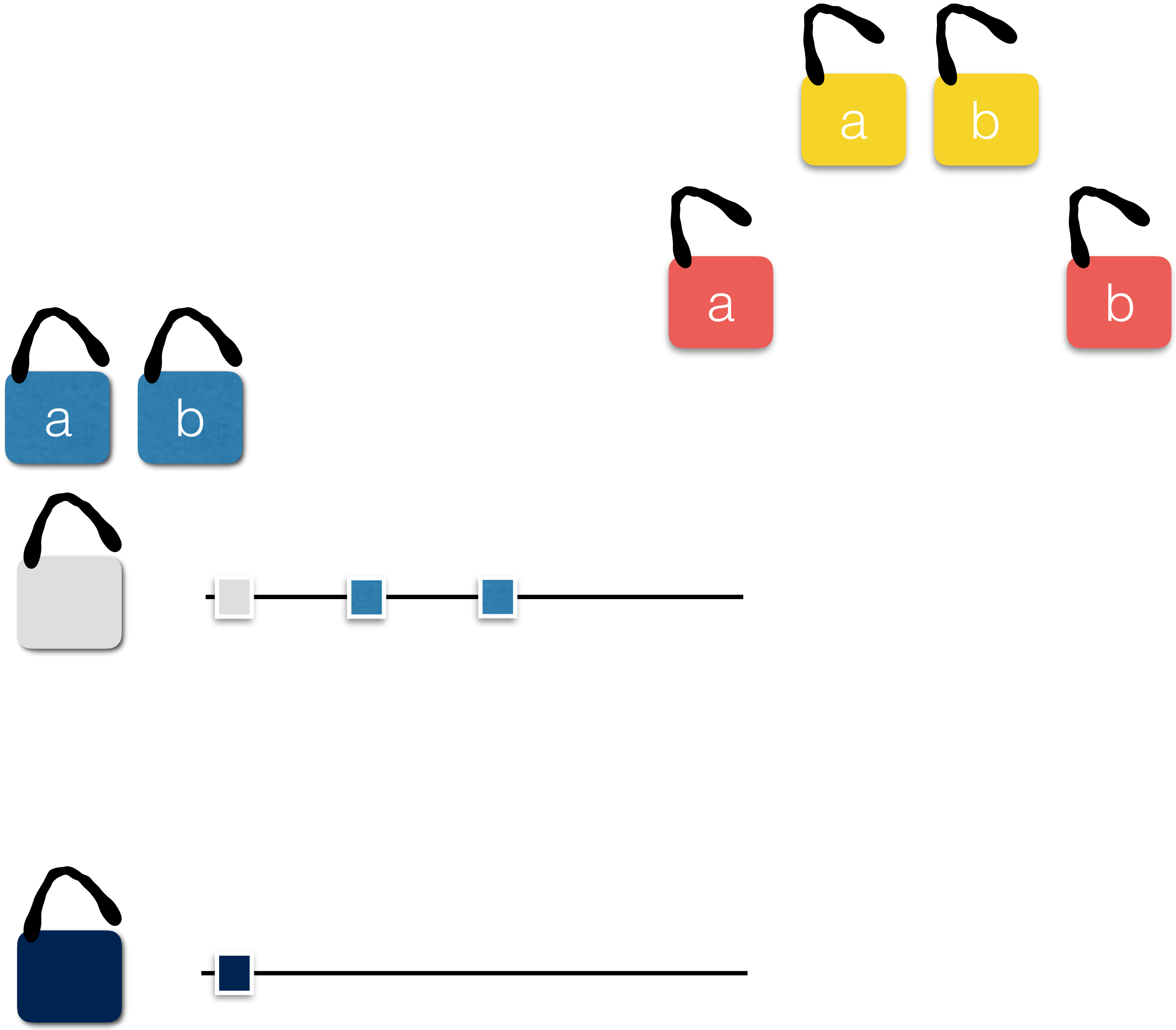
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



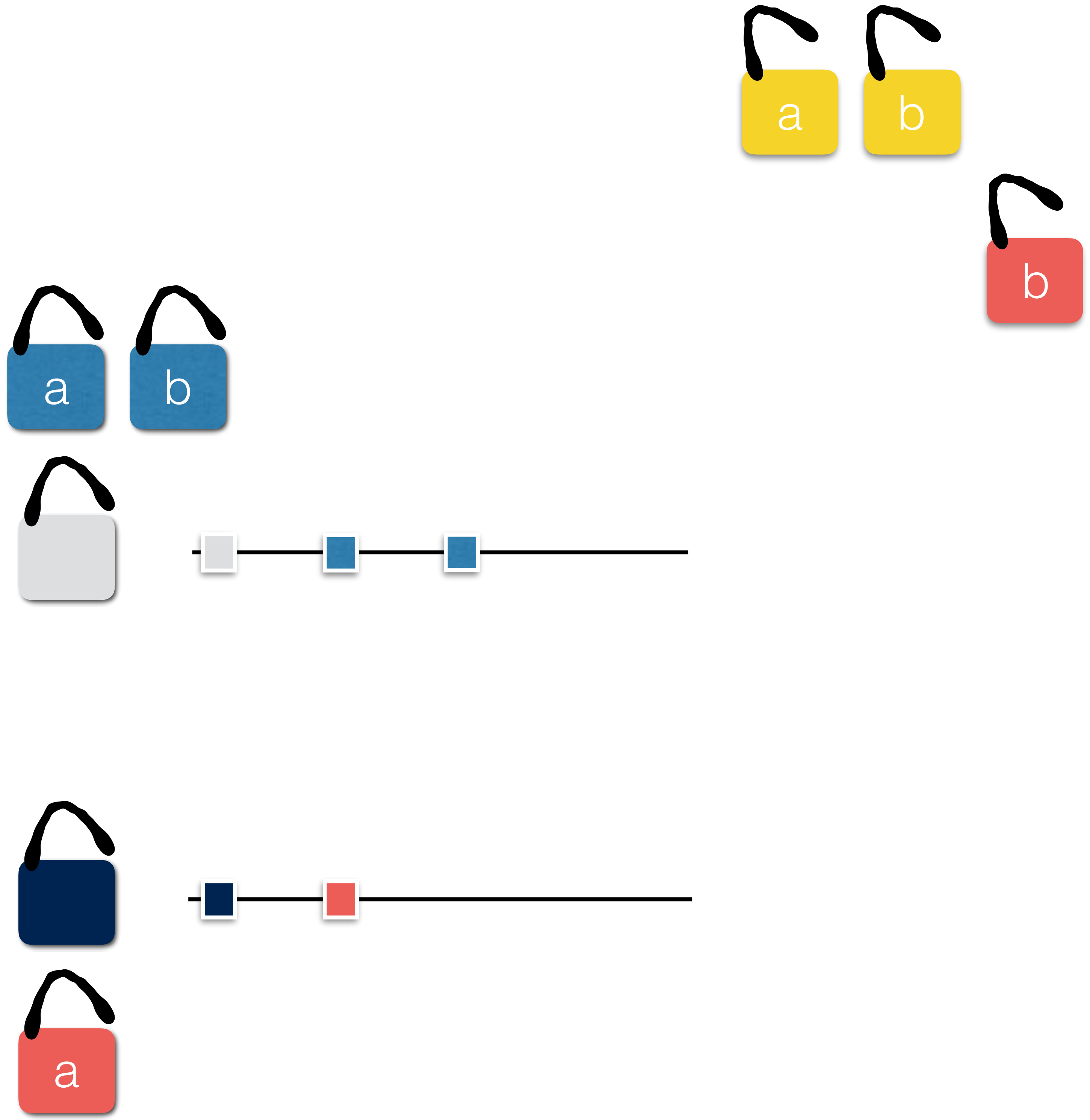
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



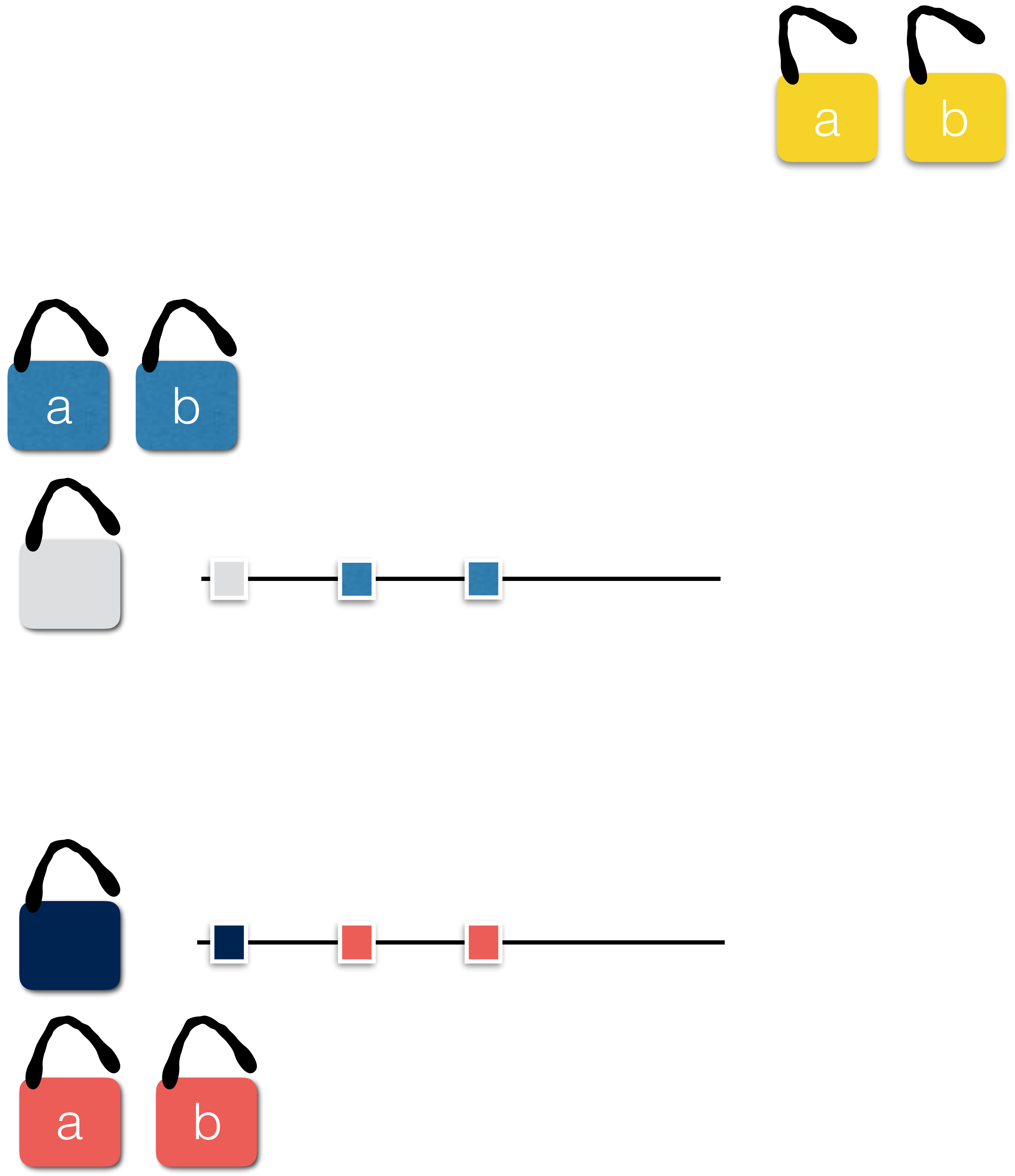
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



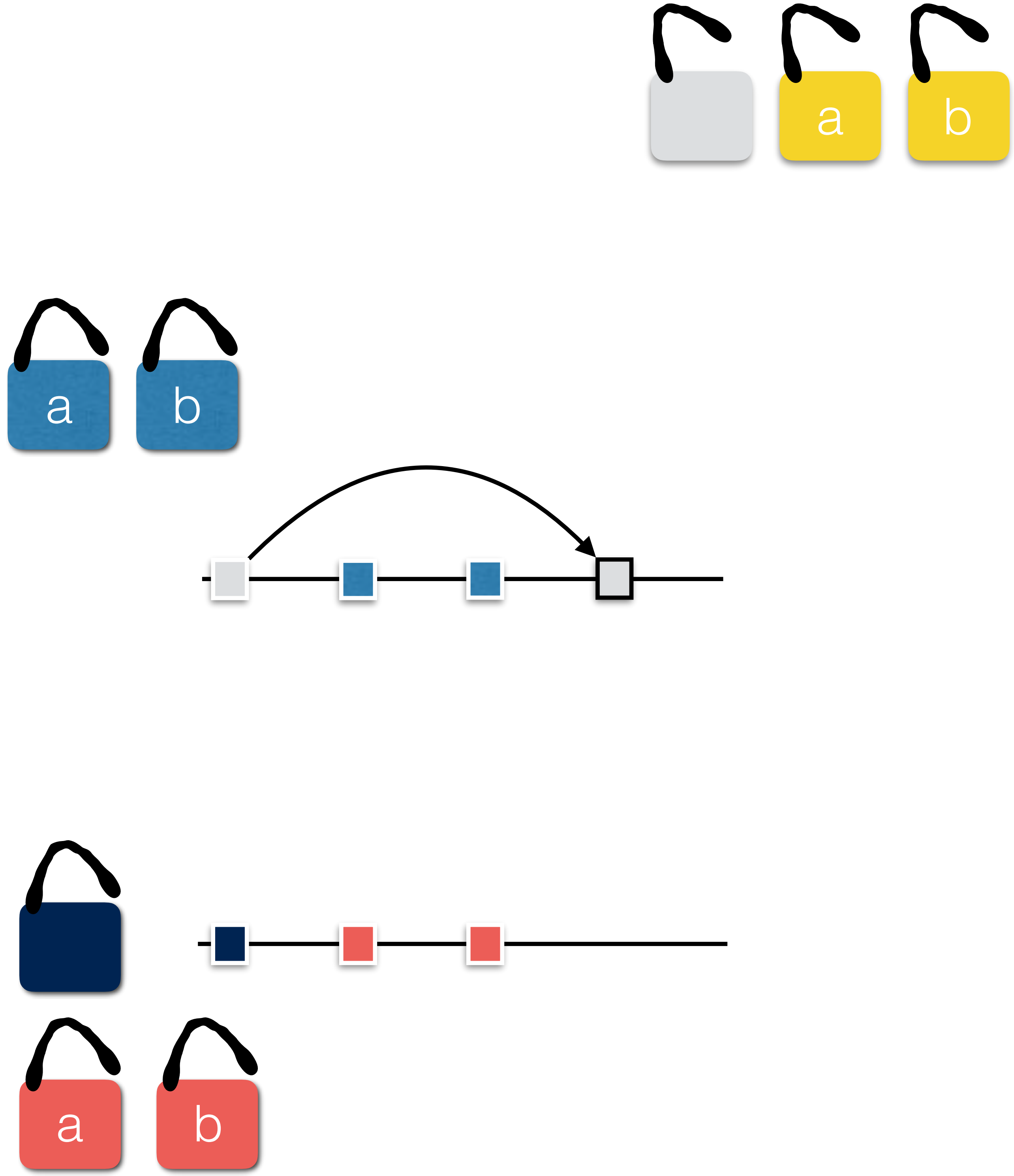
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



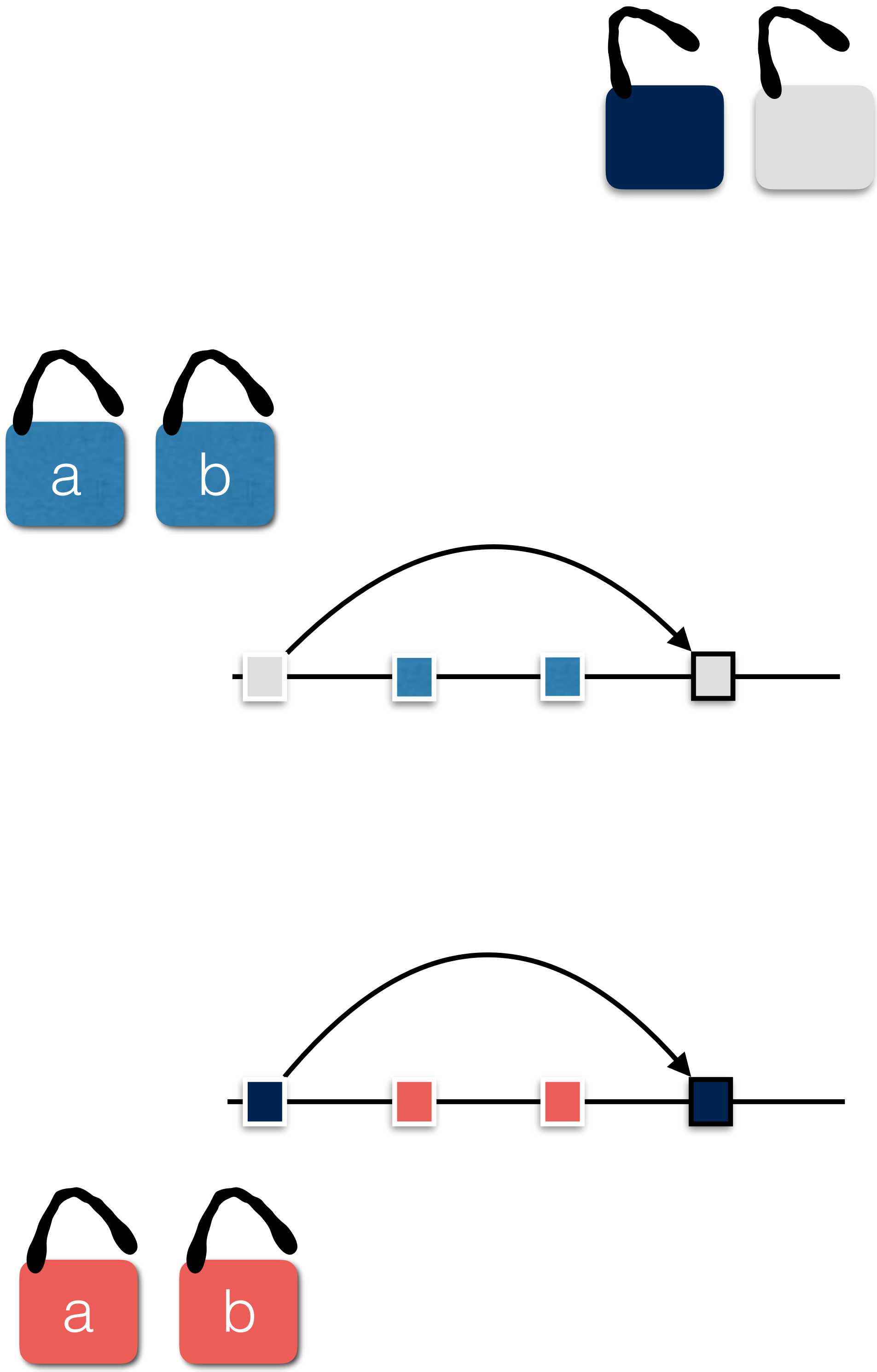
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



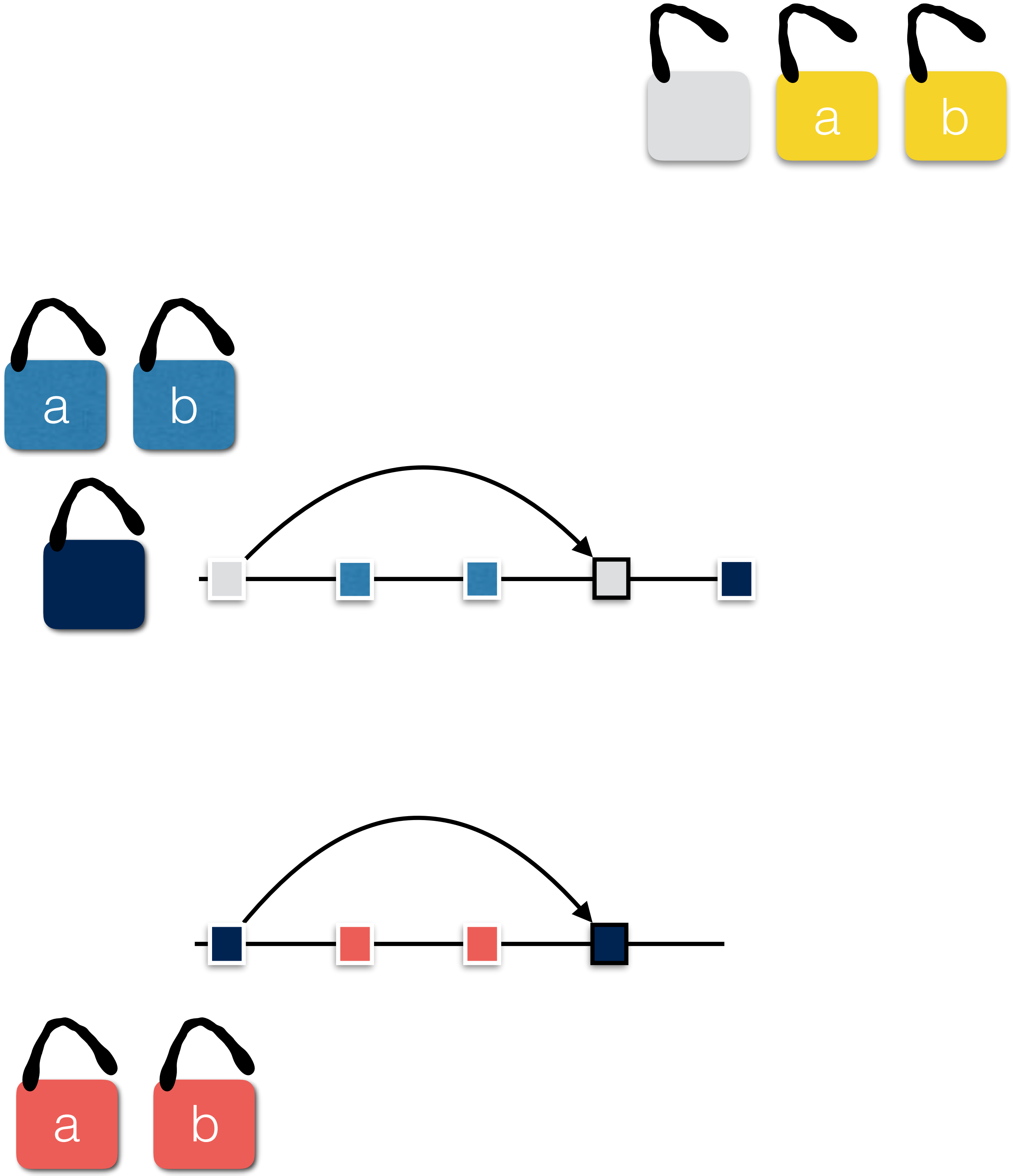
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



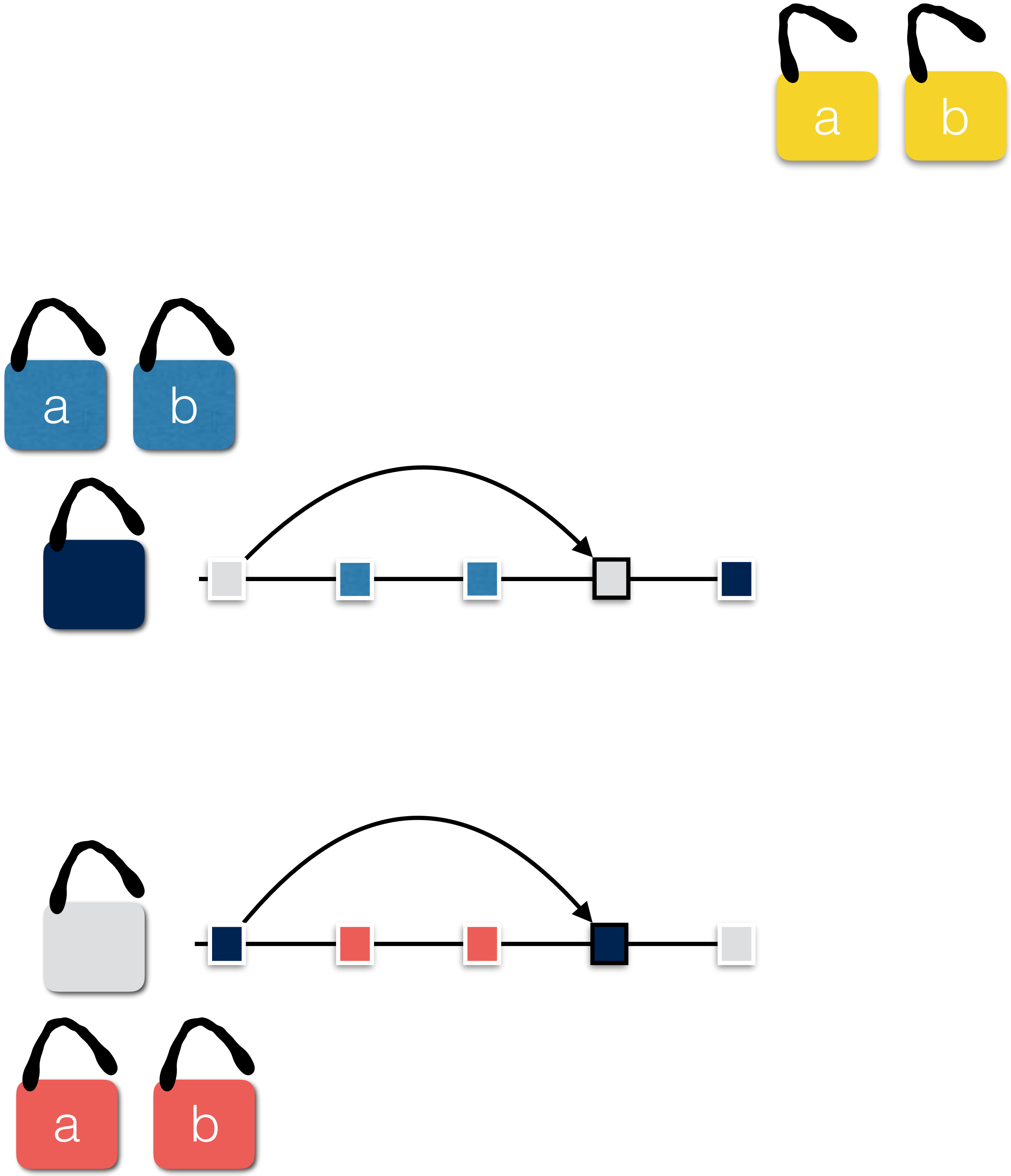
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



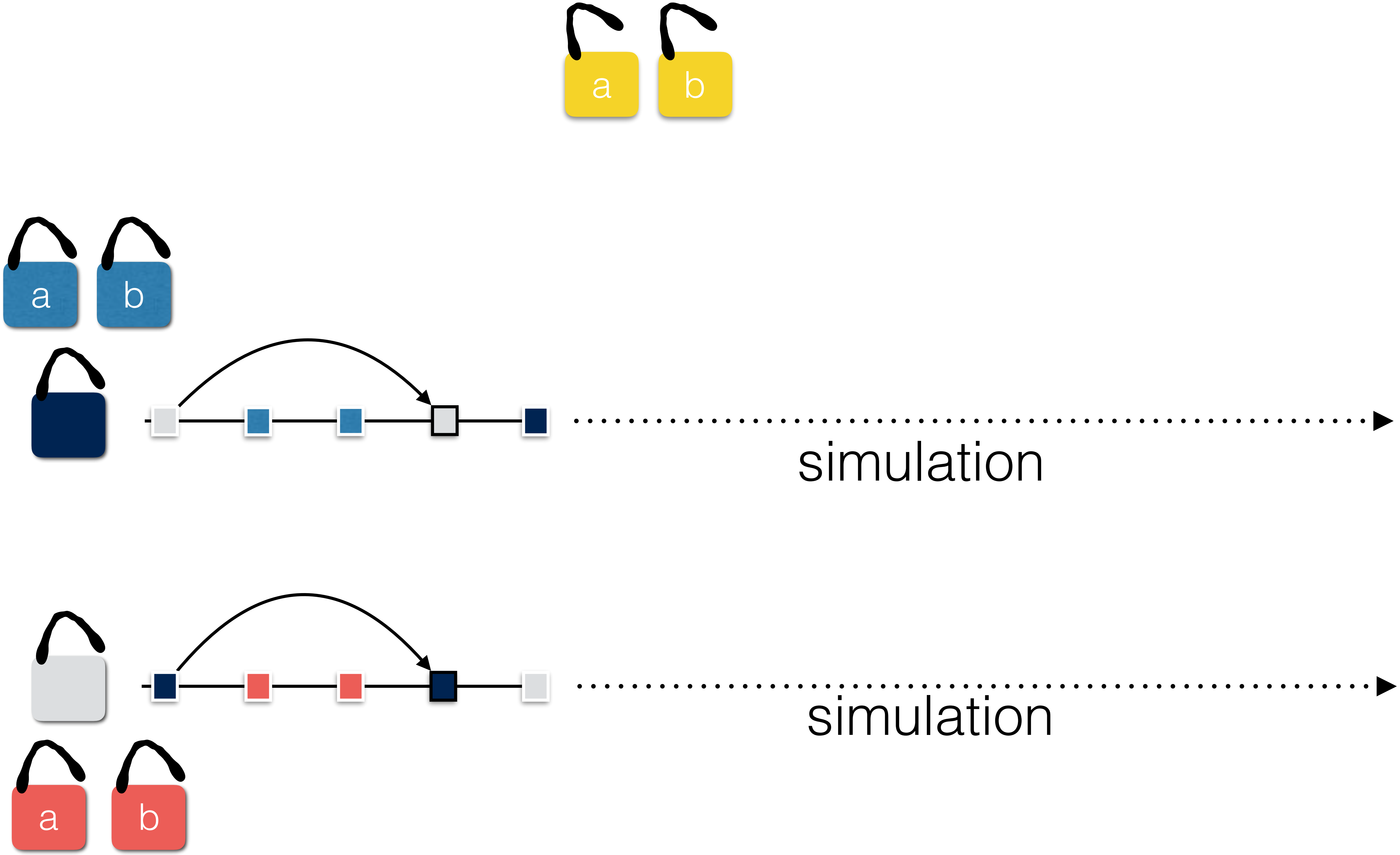
Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



Kahlon,Ivancic,Gupta CAV05

# Initializing the Locks:



simulation

simulation

Kahlon,Ivancic,Gupta CAV05

# Decidable Underapproximations:

Nested Locking

Locks are taken and released by each process in well-nested (last in first out/stack-like) manner

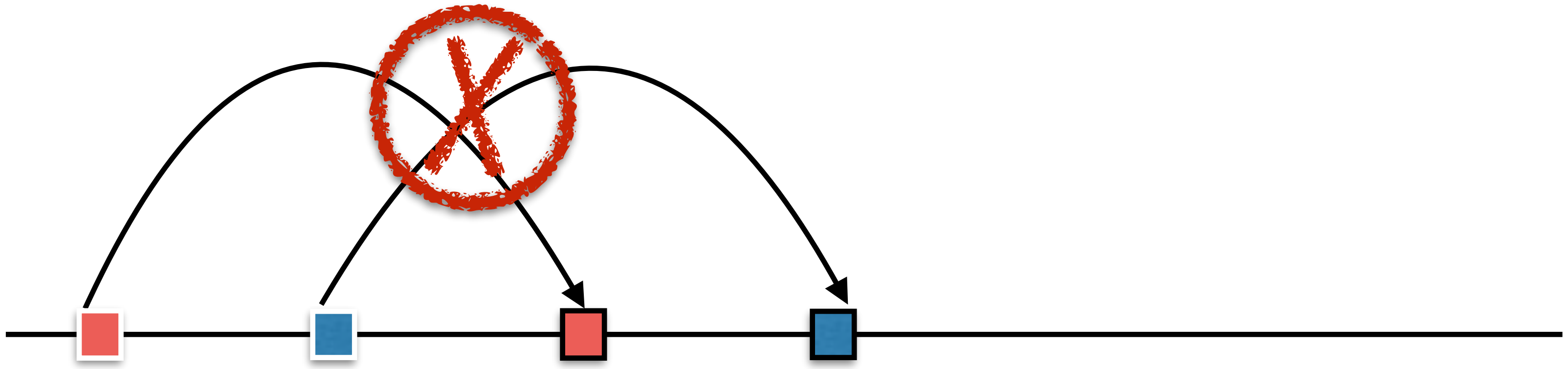
Kahlon,Ivancic,Gupta CAV05

# Decidable Underapproximations:

Nested Locking

Locks are taken and released by each process in well-nested (last in first out/stack-like) manner

Kahlon,Ivancic,Gupta CAV05



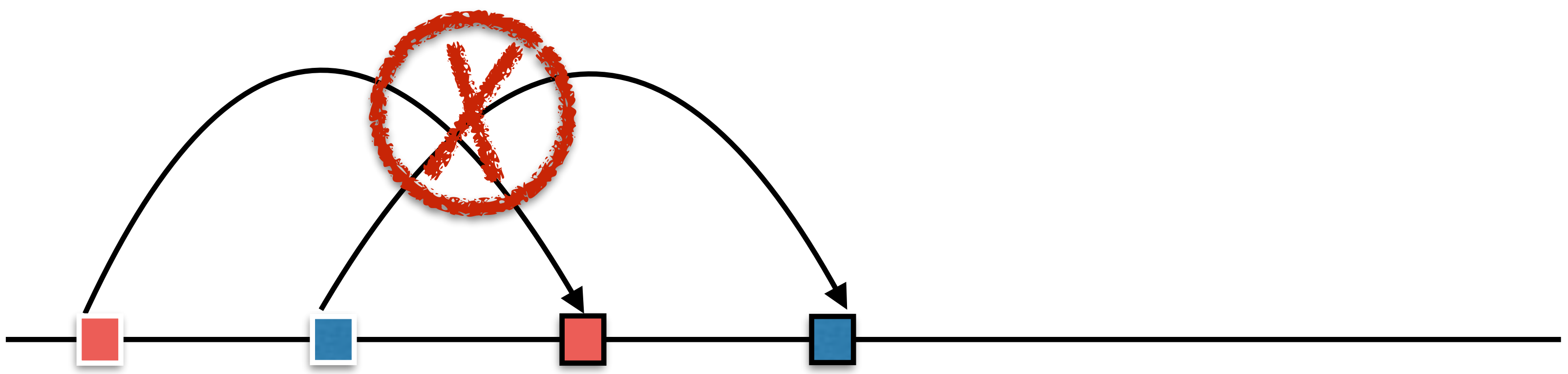
The well-nested assumption is per process (not global).

# Decidable Underapproximations:

Nested Locking

Locks are taken and released by each process in well-nested (last in first out/stack-like) manner

Kahlon,Ivancic,Gupta CAV05



The well-nested assumption is per process (not global).

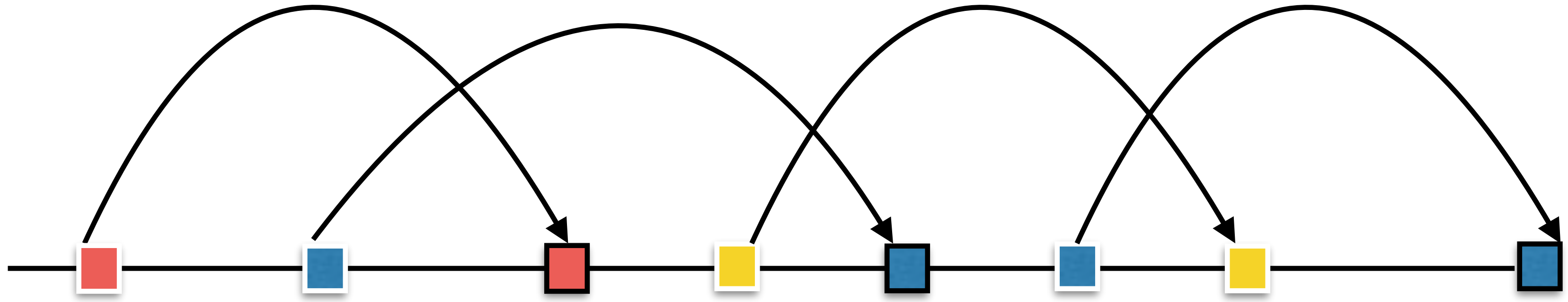
More on nested locking later …

# Decidable Underapproximations:

Bounded Lock Chains

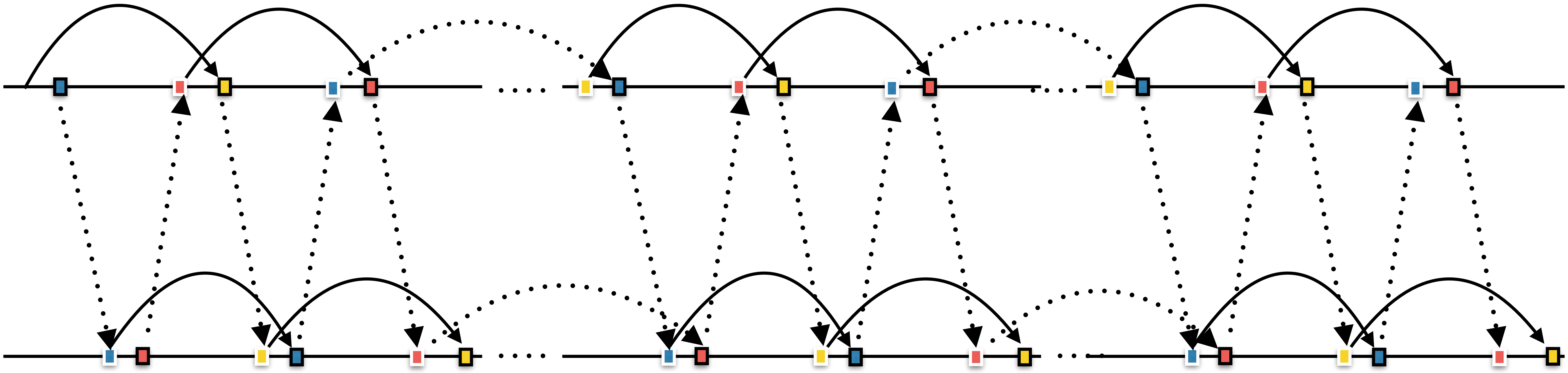
Lock chaining is permitted but there is a priori bound on length of such chains.

Kahlon LICS09

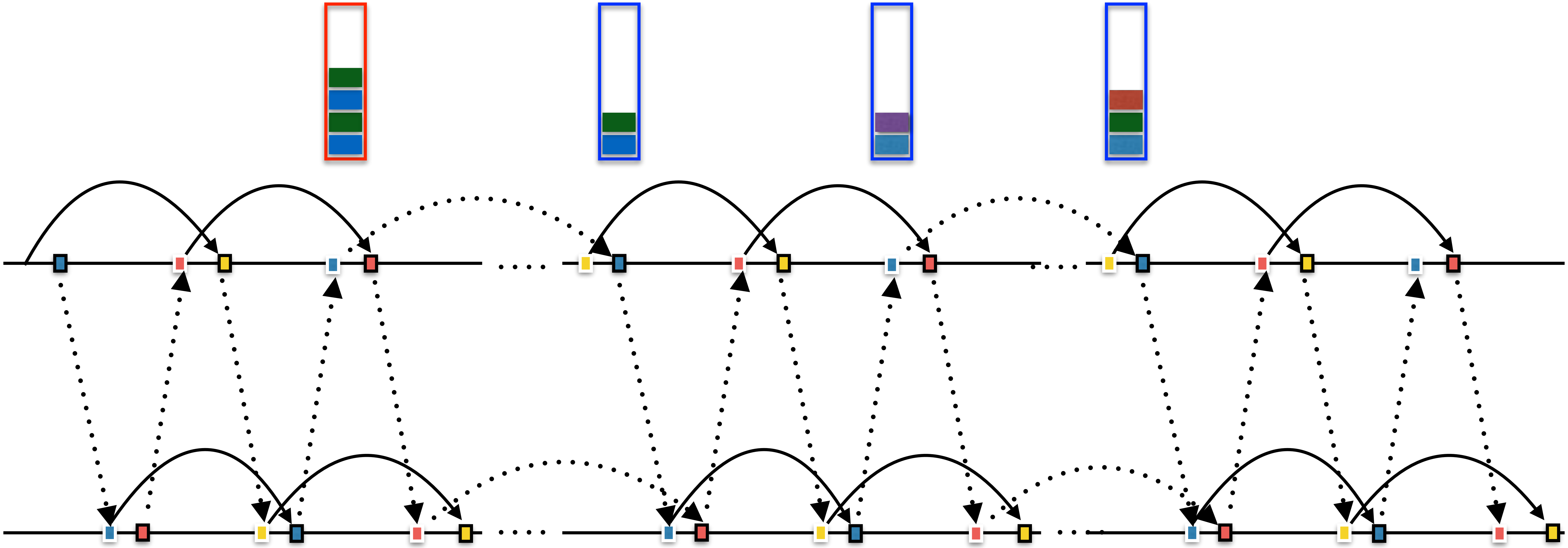


A length 4 lock-chained run

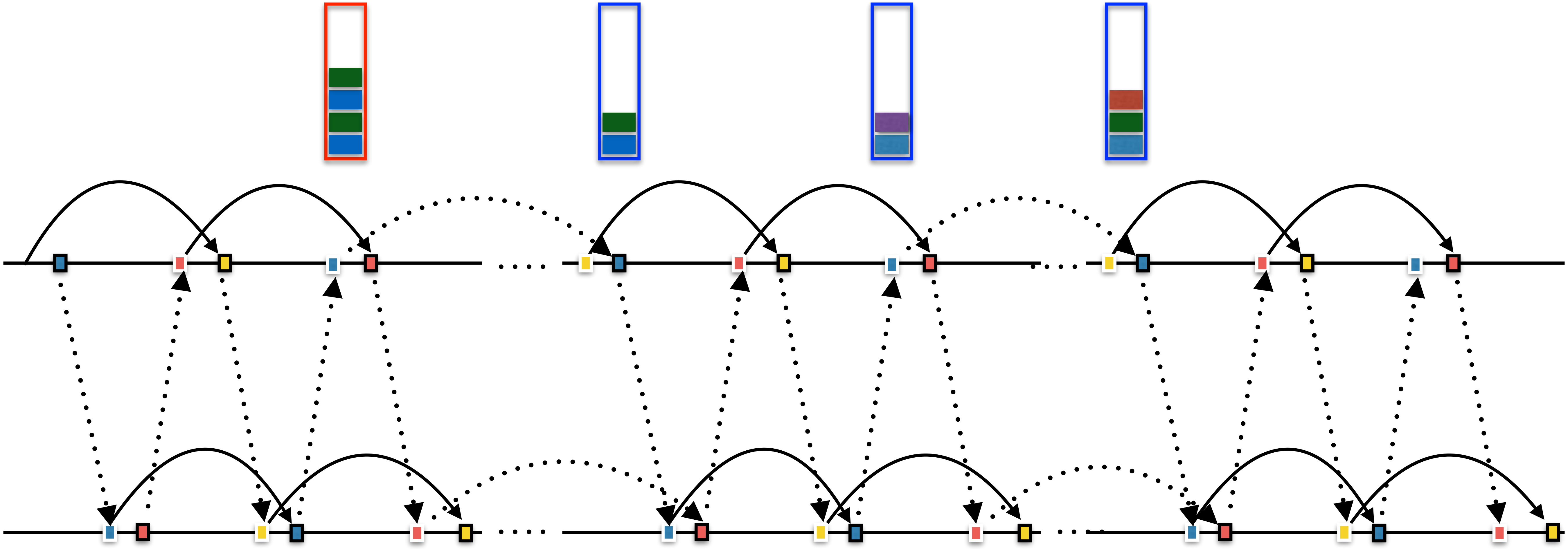
# Recursive Programs with Locks



# Recursive Programs with Locks

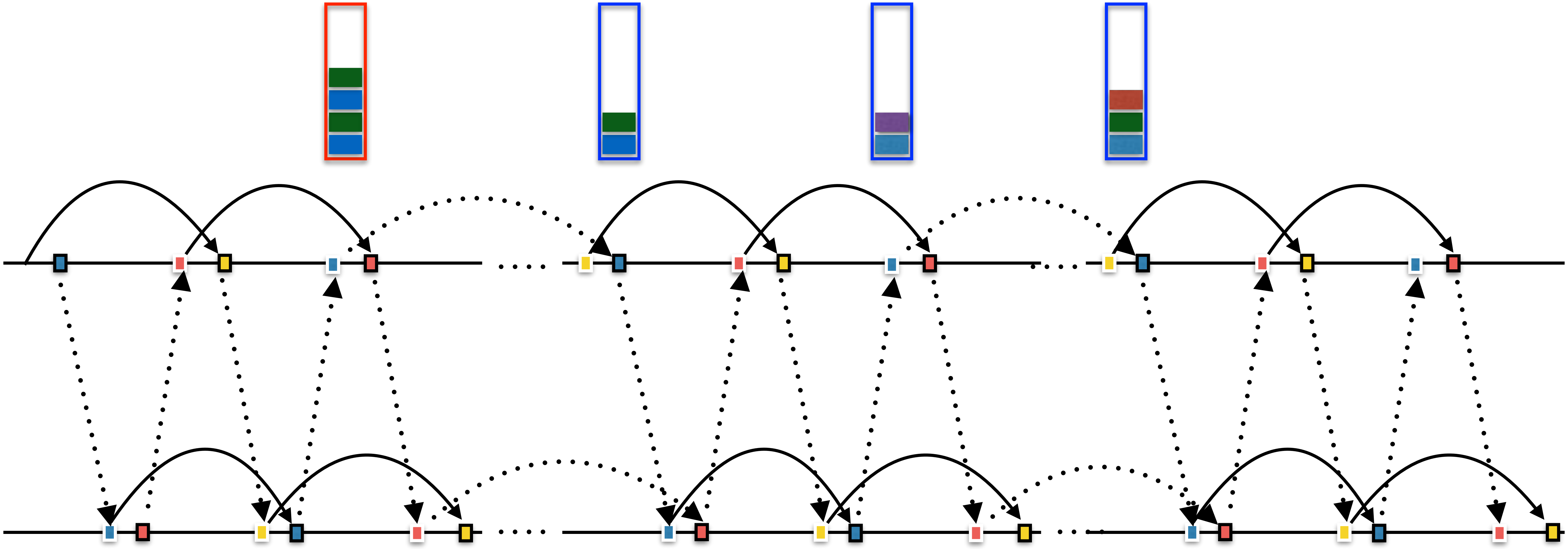


# Recursive Programs with Locks



Locks taken in procedure may be released after the procedure terminates

# Recursive Programs with Locks



Procedures may return locks they did not take

# Contextual Locking

Contextual Locking

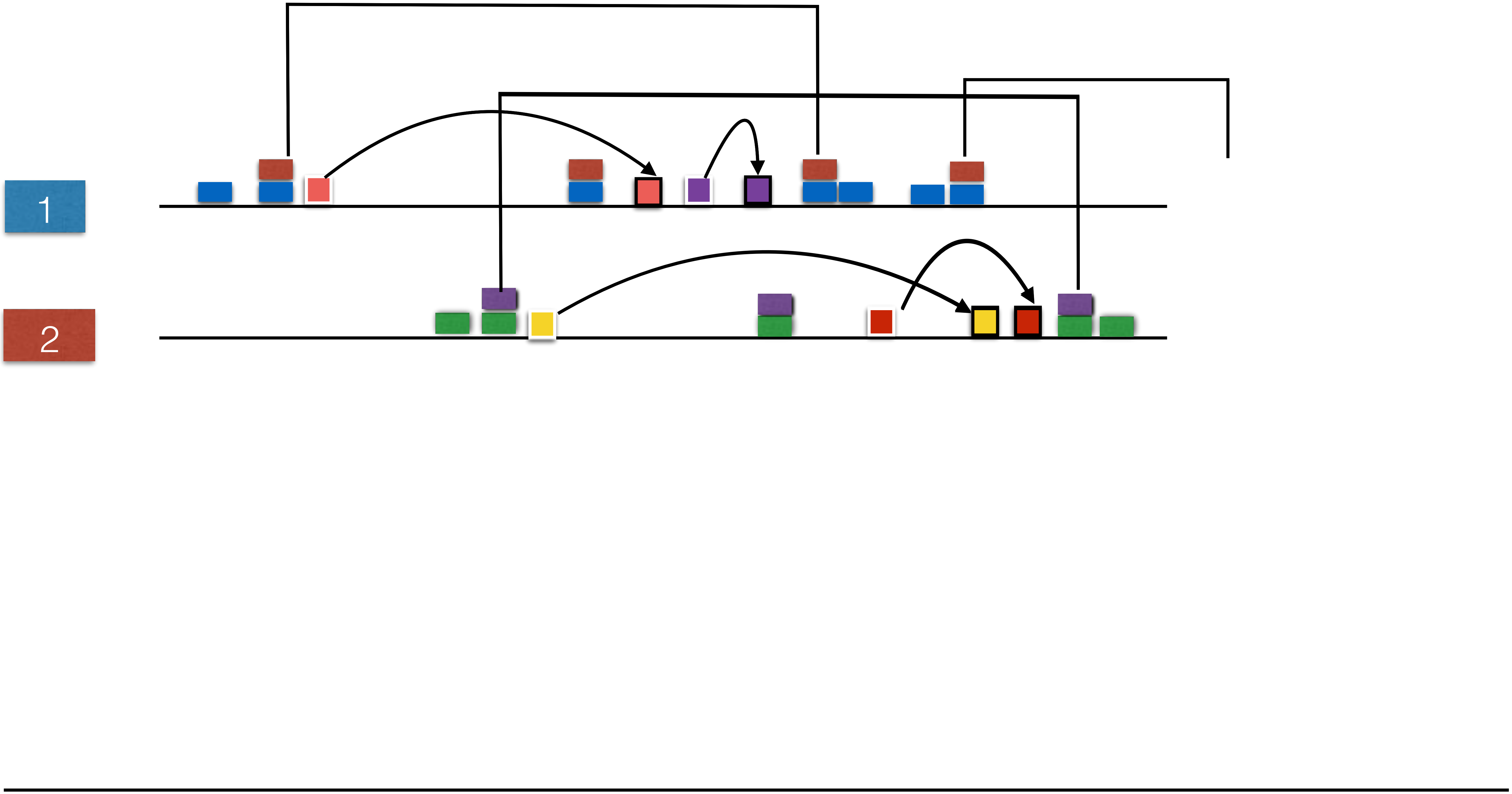
Locks taken by a procedure call are returned during the execution of that very procedure call.

Chadha,Madhusudan,Vishwanathan
TACAS12

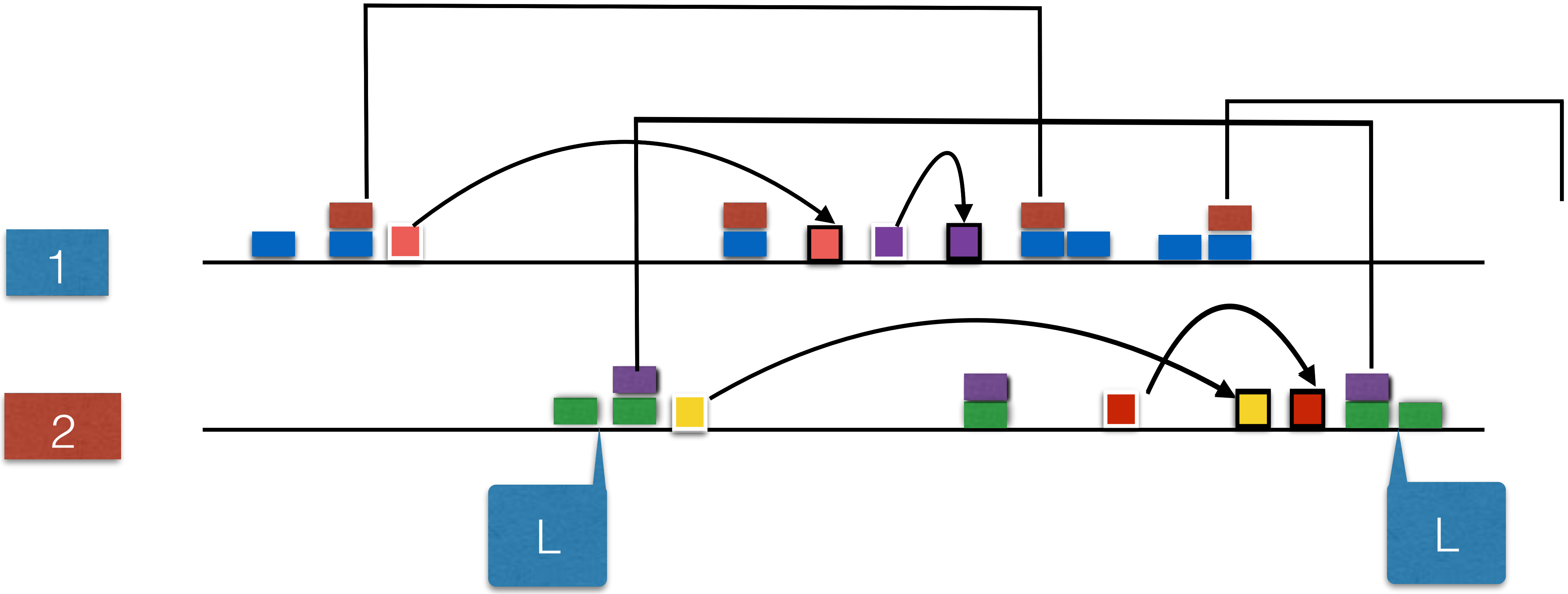
Reachability is decidable for 2 processes under contextual locking

Chadha, Madhusudan, Vishwanathan
TACAS12
Bonnet, Chadha, Madhusudan, Viswanathan
LMCS 2013

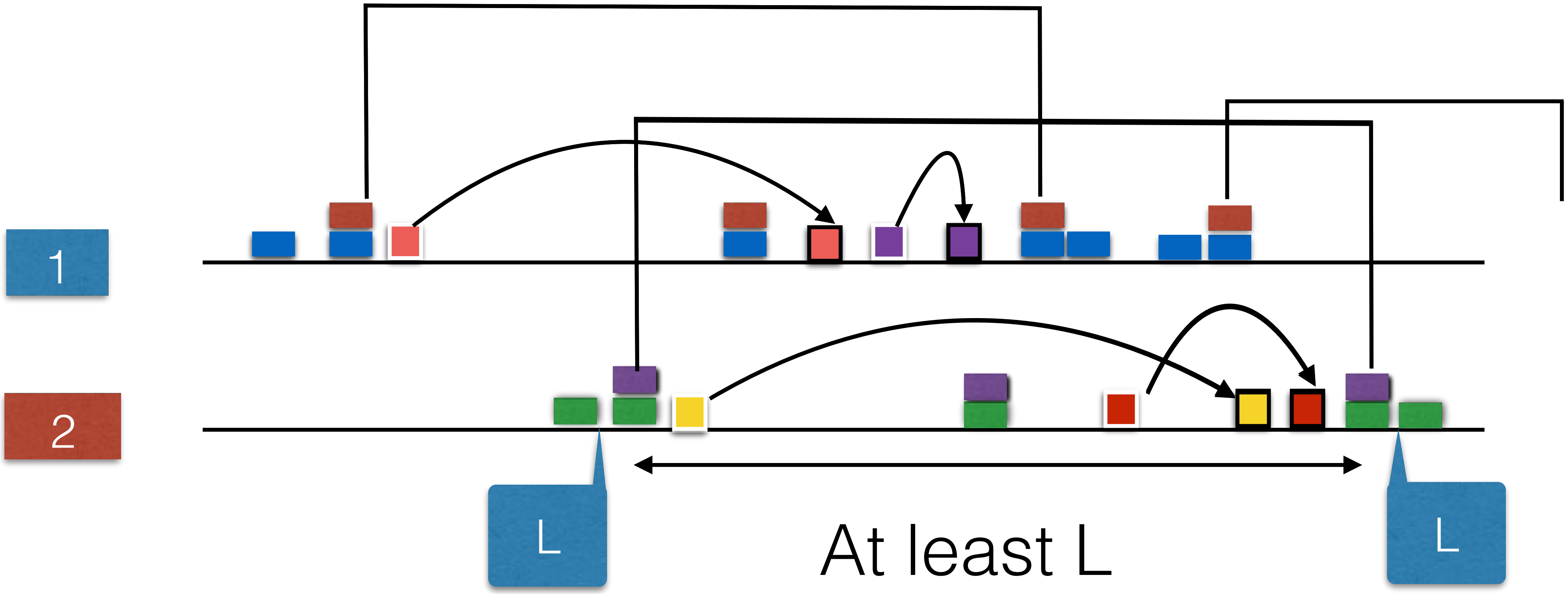
# Sequentializing the runs:



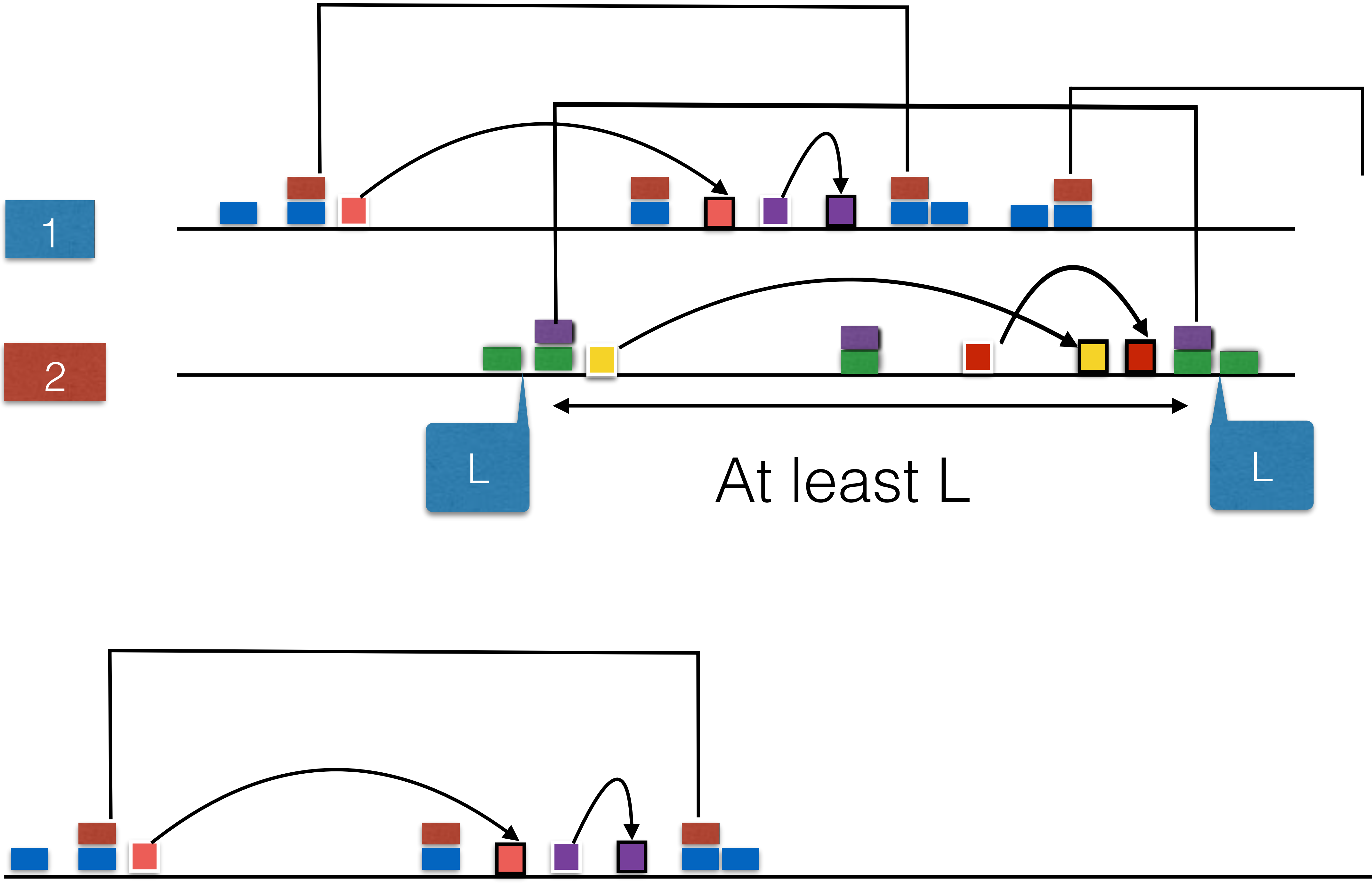
# Sequentializing the runs:



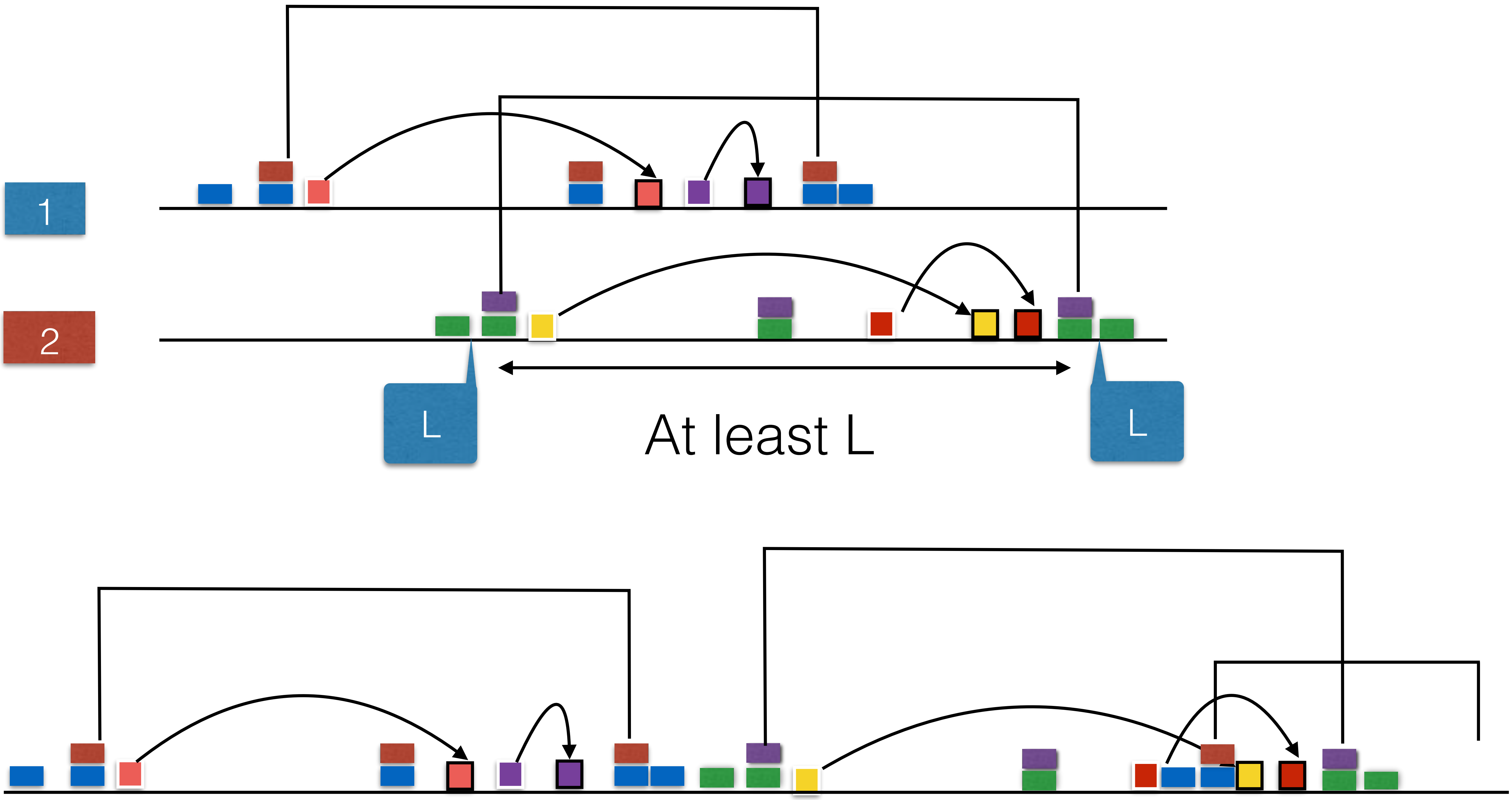
# Sequentializing the runs:



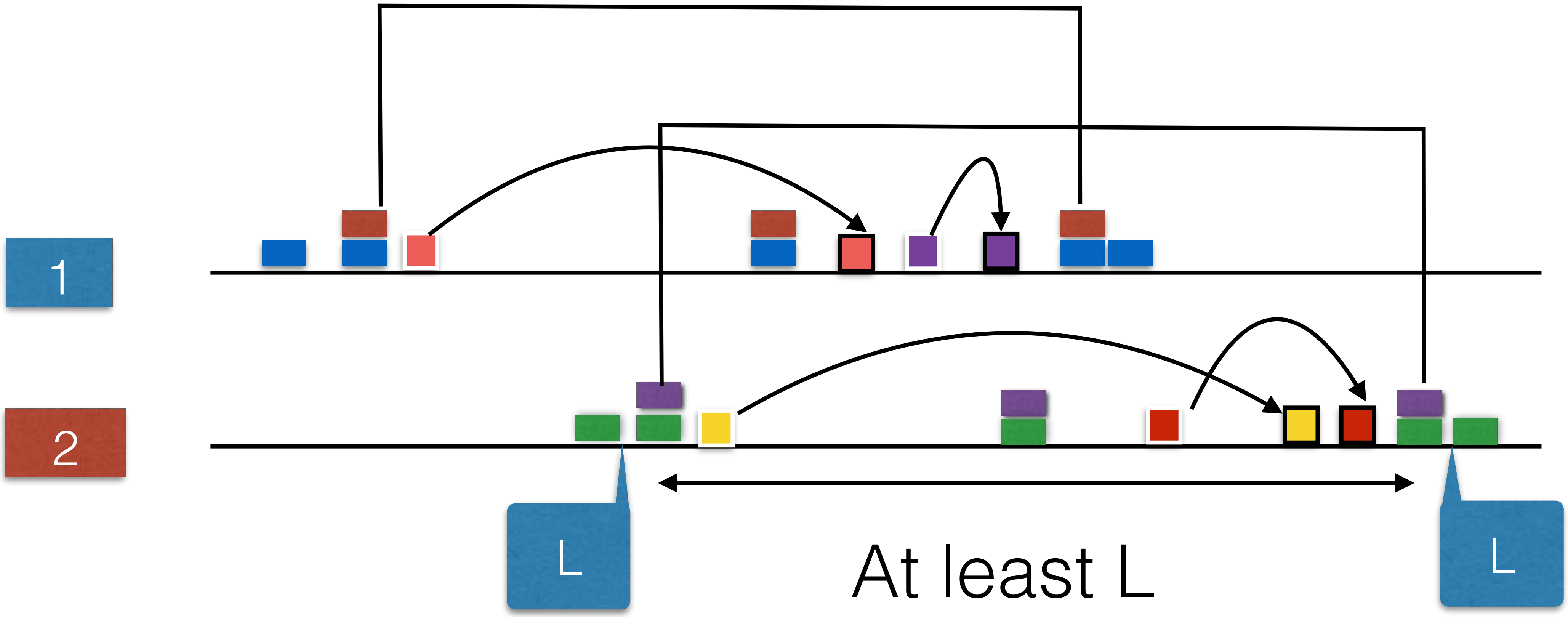
# Sequentializing the runs:



# Sequentializing the runs:



# Sequentializing the runs:



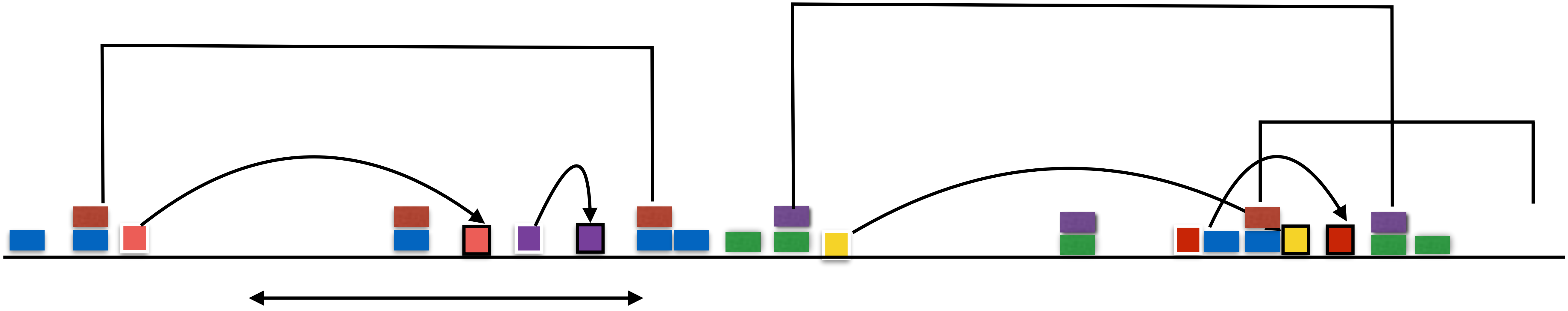
1

2

L

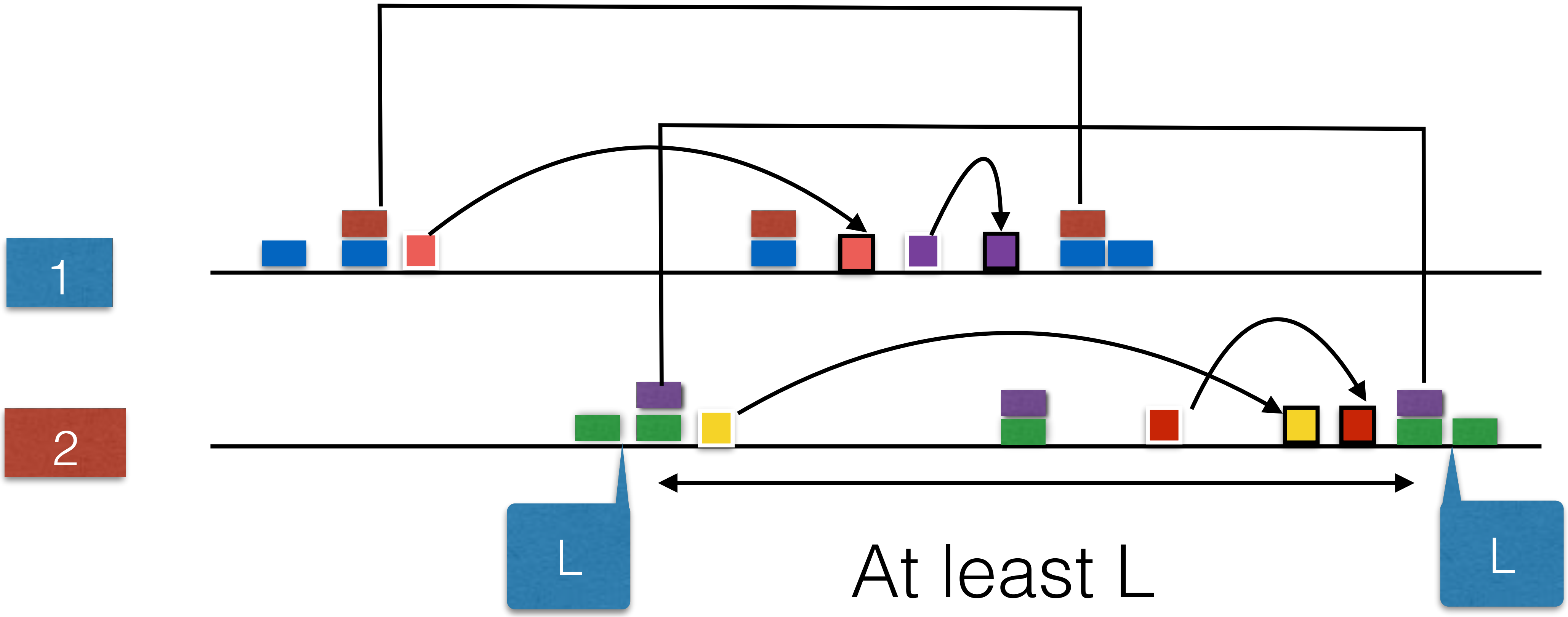
At least L

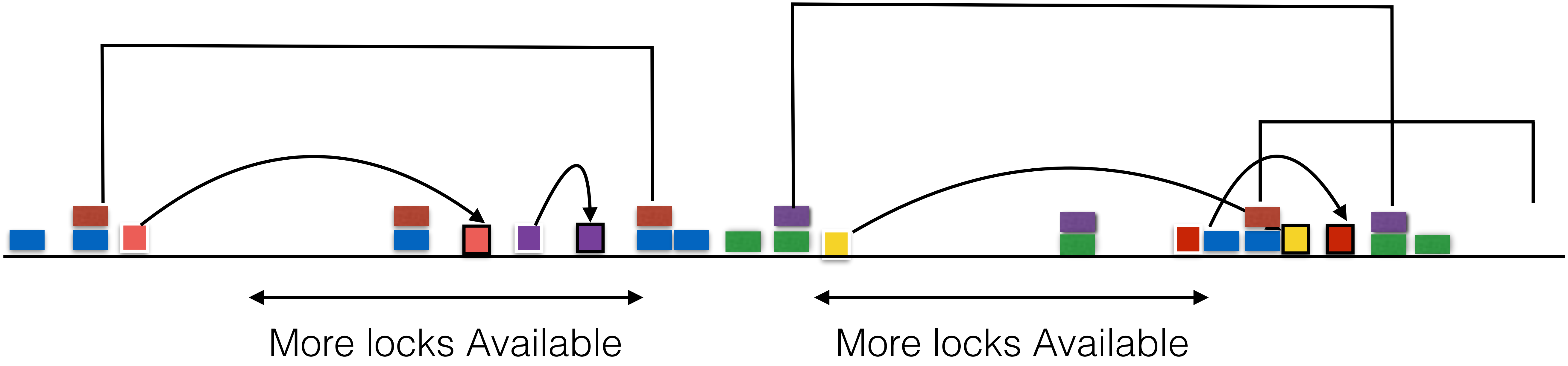
L



More locks Available

# Sequentializing the runs:





# Contextual Locking: 2 processes

Contextual  Locking with 2 processes

It suffices to consider runs where the procedure calls of the two processes are also well-nested. Can be simulated by a single PDS.

Chadha,Madhusudan,Vishwanathan
TACAS12

This does not work if there are 3 processes or more.

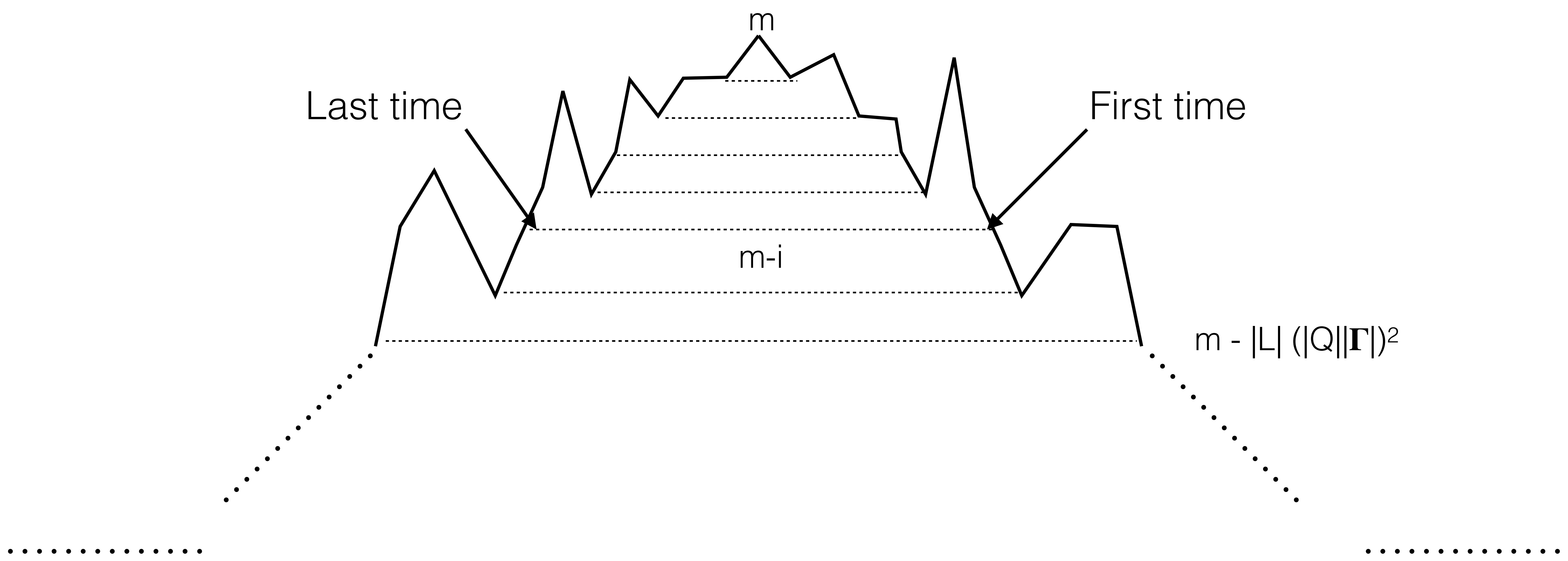
# Contextual Locking: >2 processes

The reachability problem for any number of pushdown systems synchronising via contextual locks is decidable.

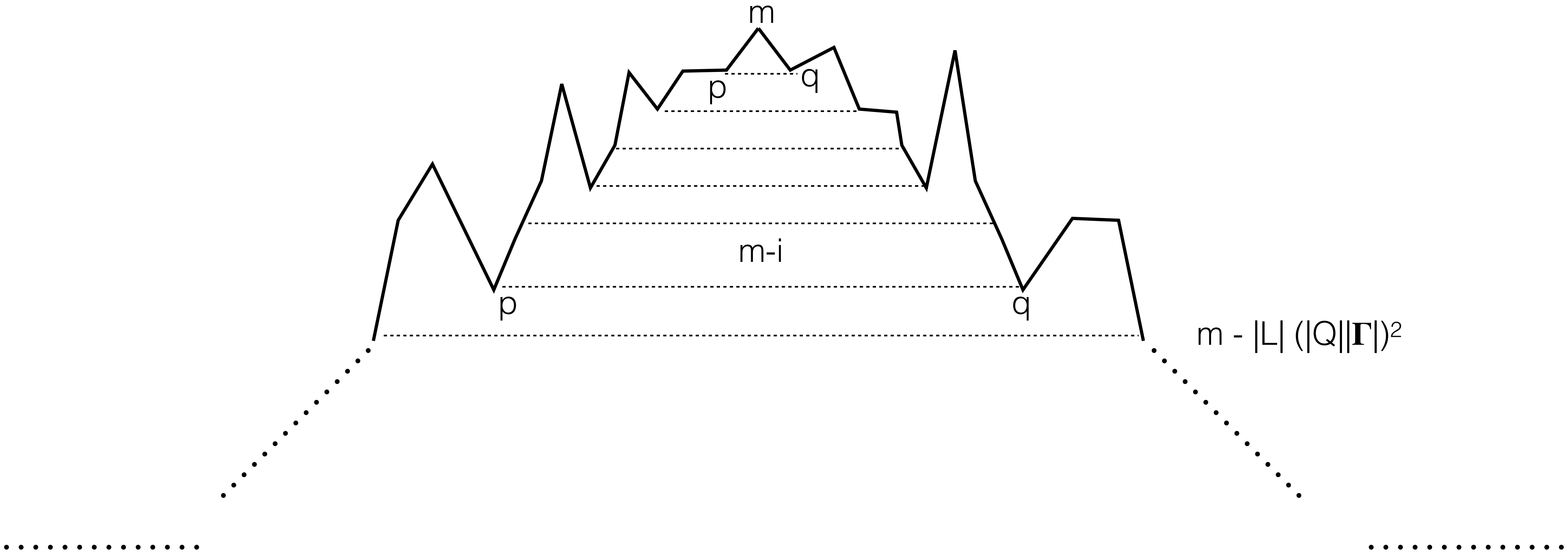
Lammich, Muller-Olm, Seidl, Werner SAS13

Stack height bounding argument.

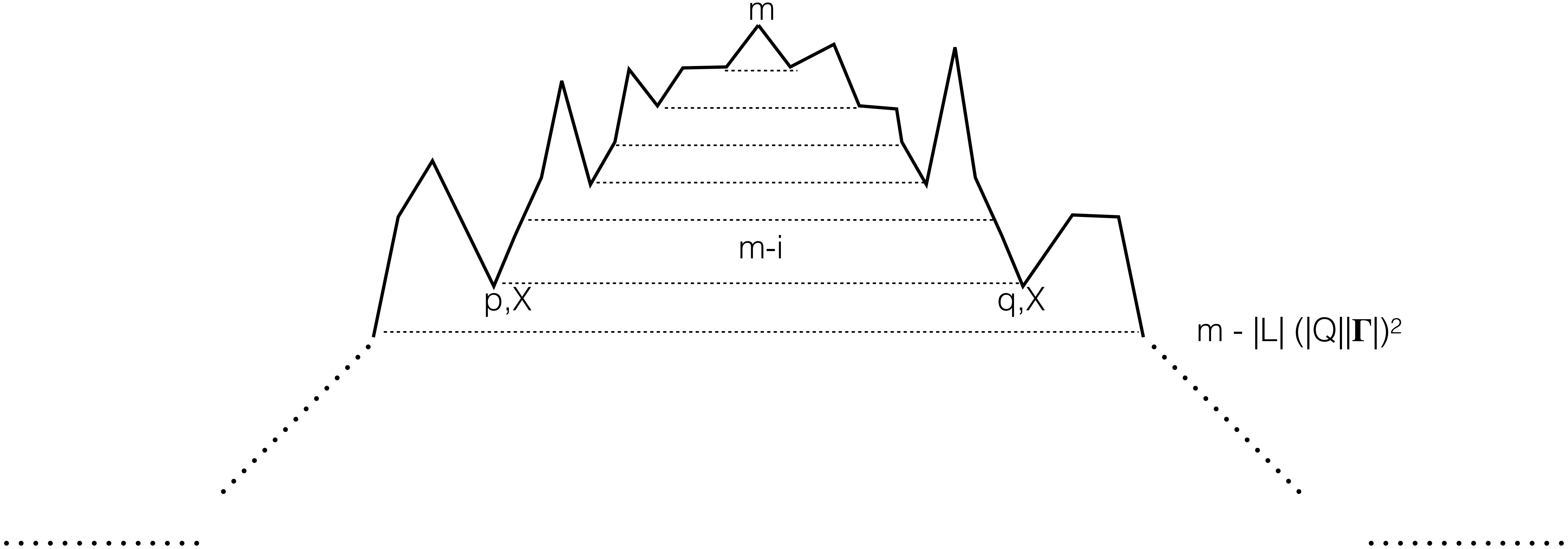
# Stack height bounding:



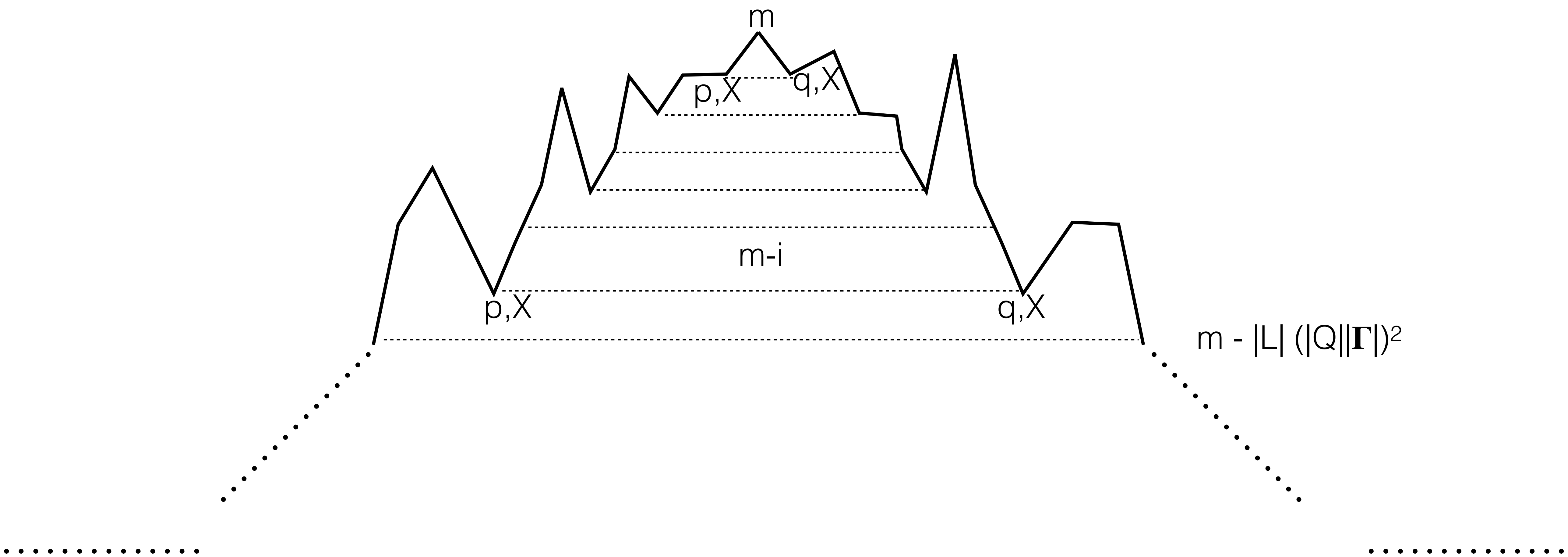
# Stack height bounding...



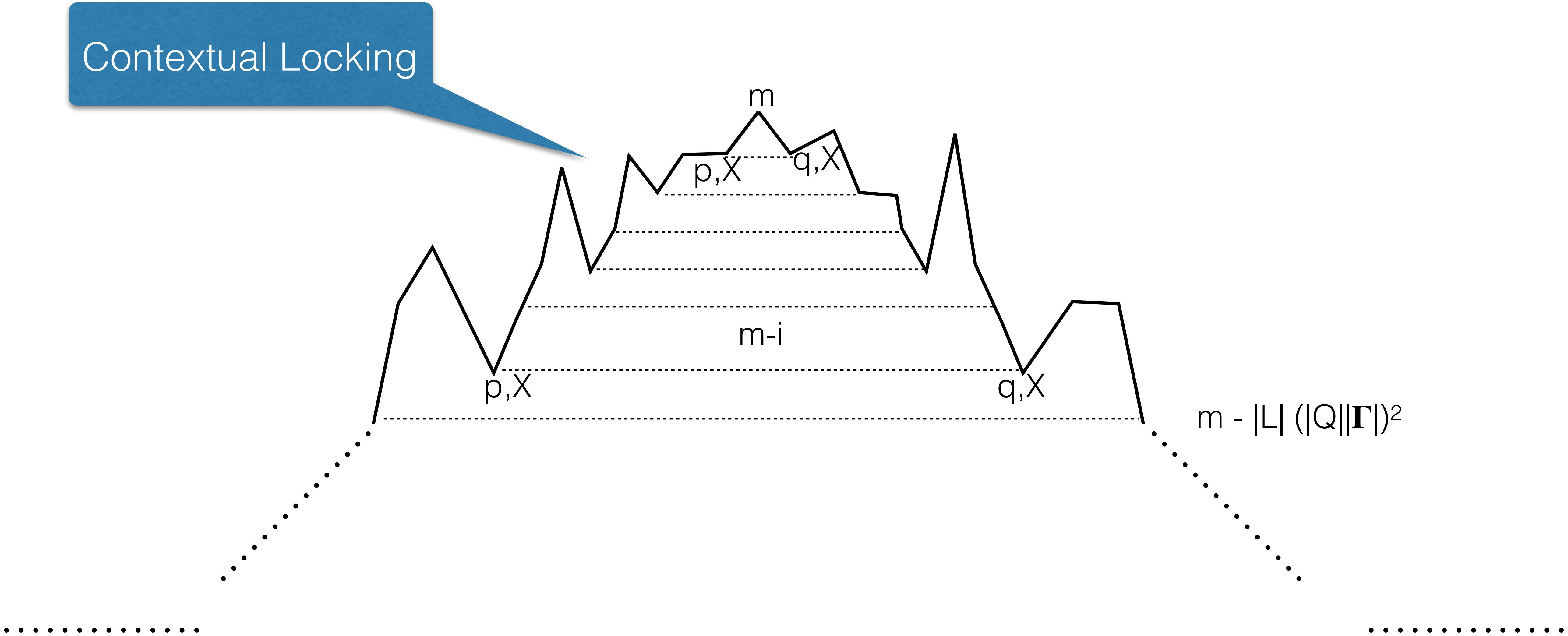
# Stack height bounding…



# Stack height bounding...

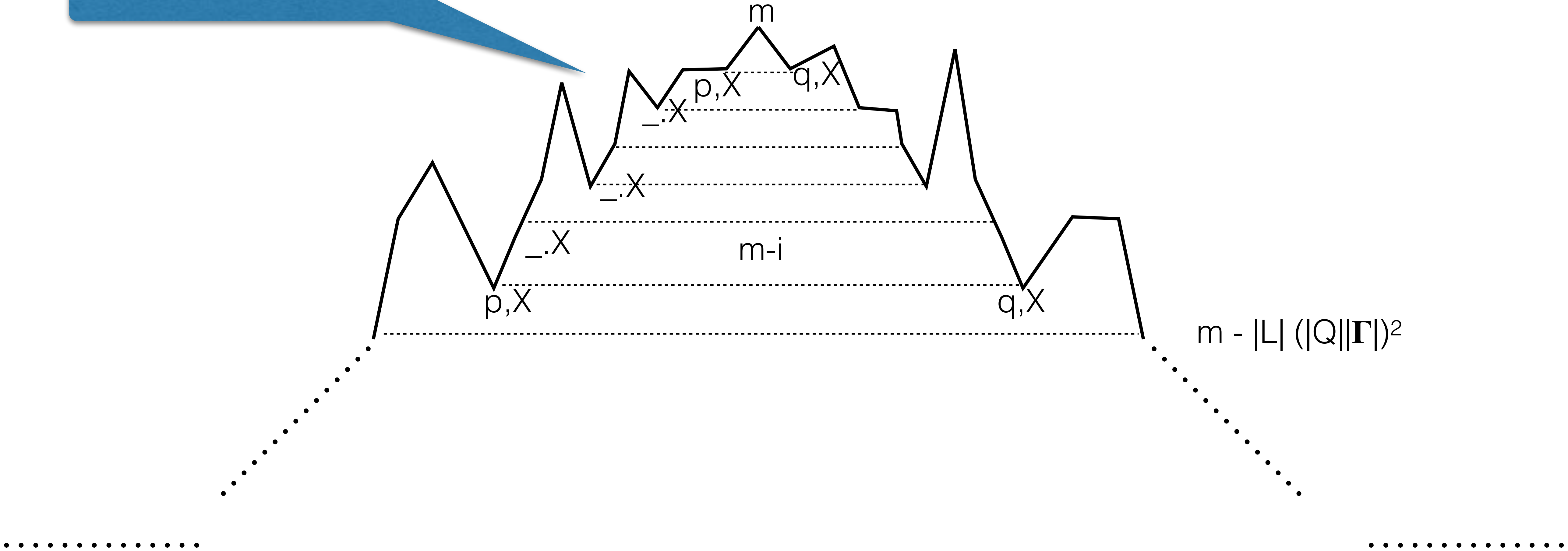


# Stack height bounding...



Contextual Locking

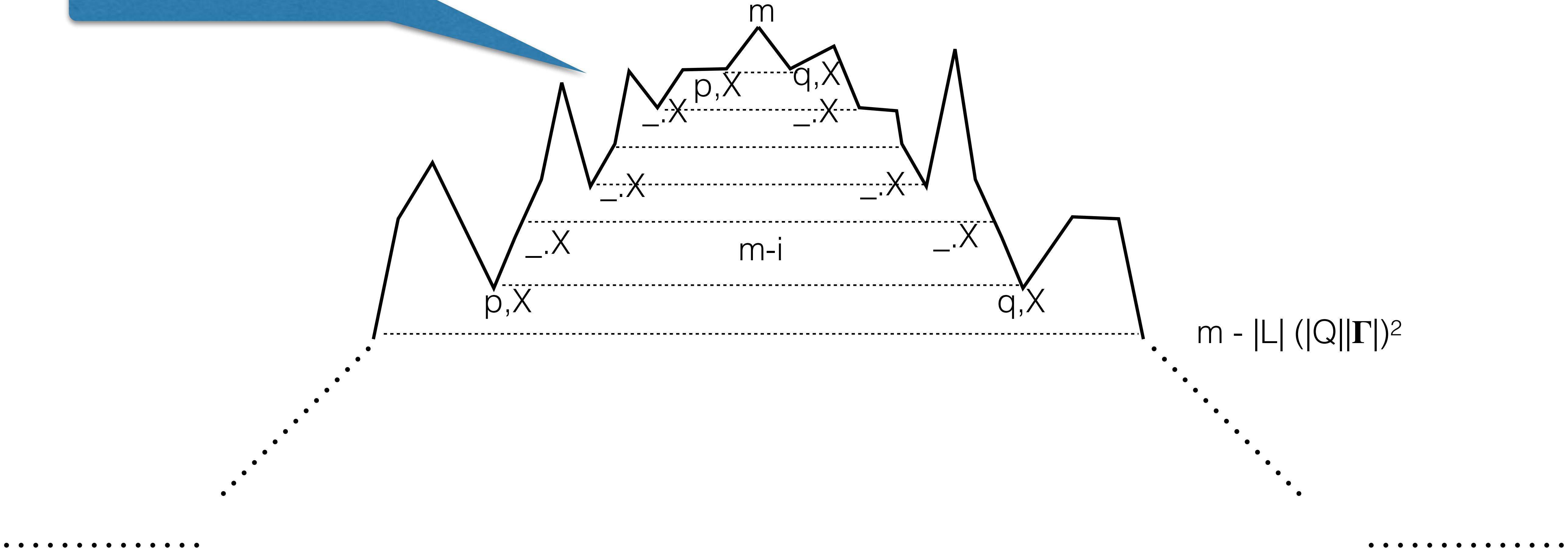
# Stack height bounding...



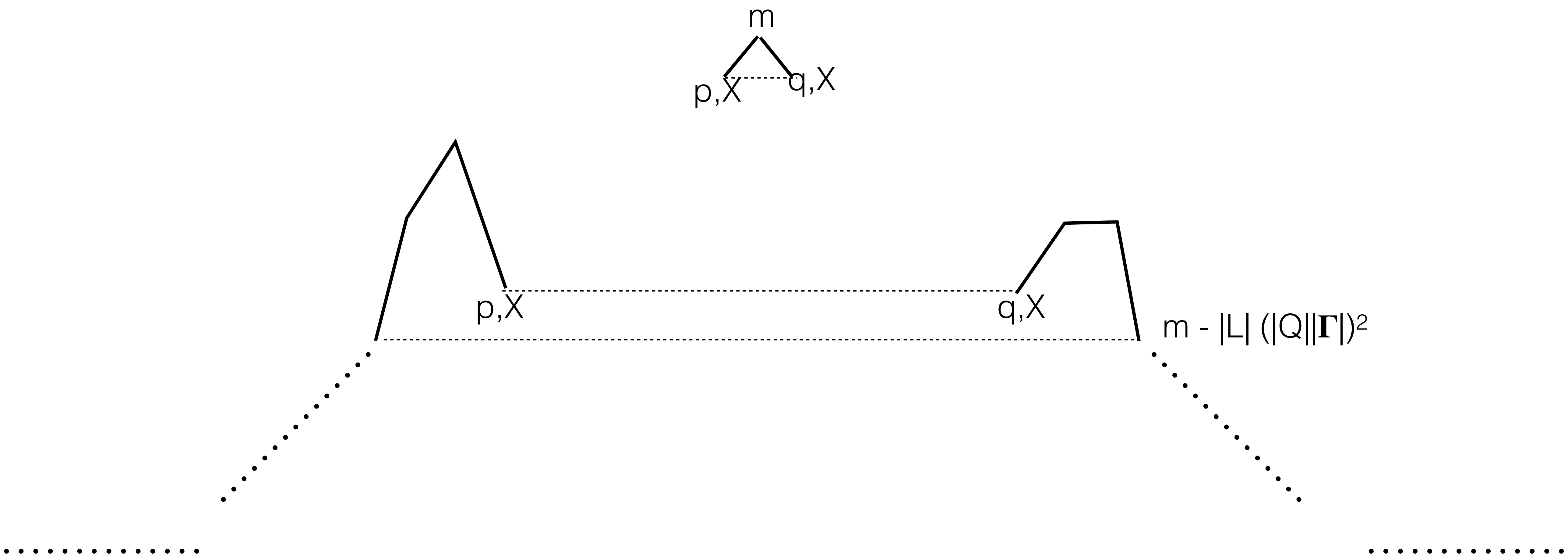
Contextual Locking

# Stack height bounding...

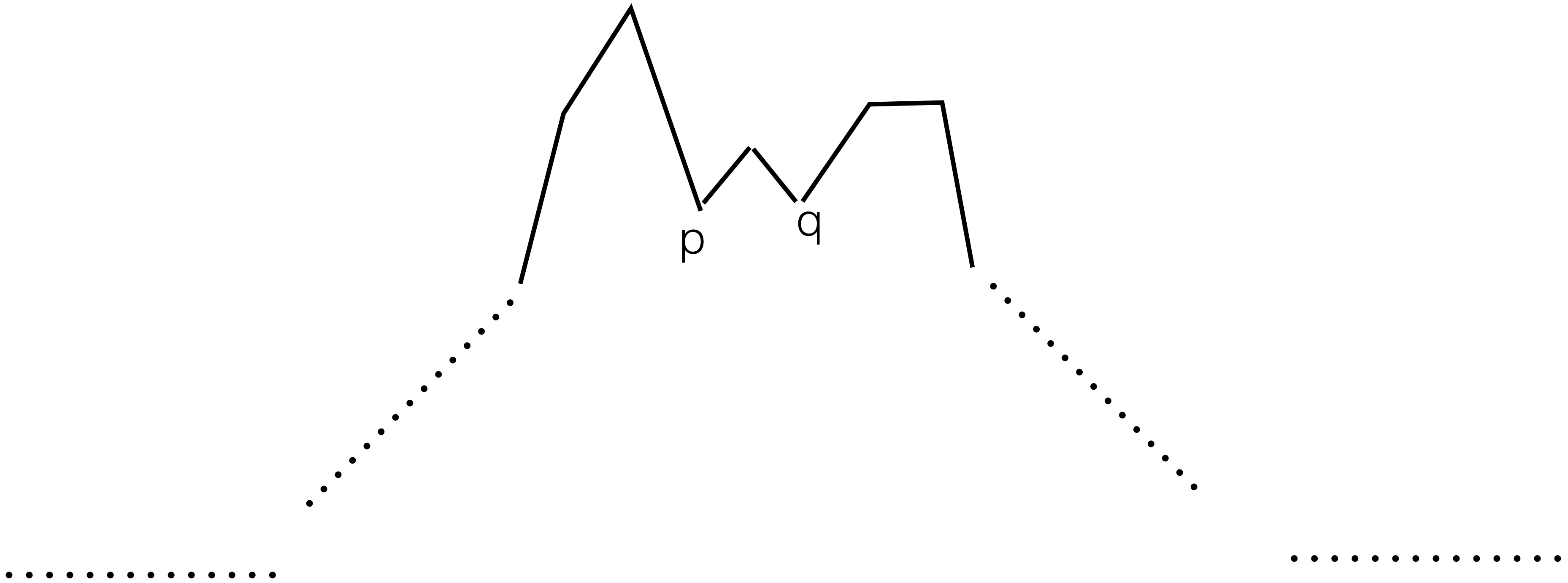
Contextual Locking



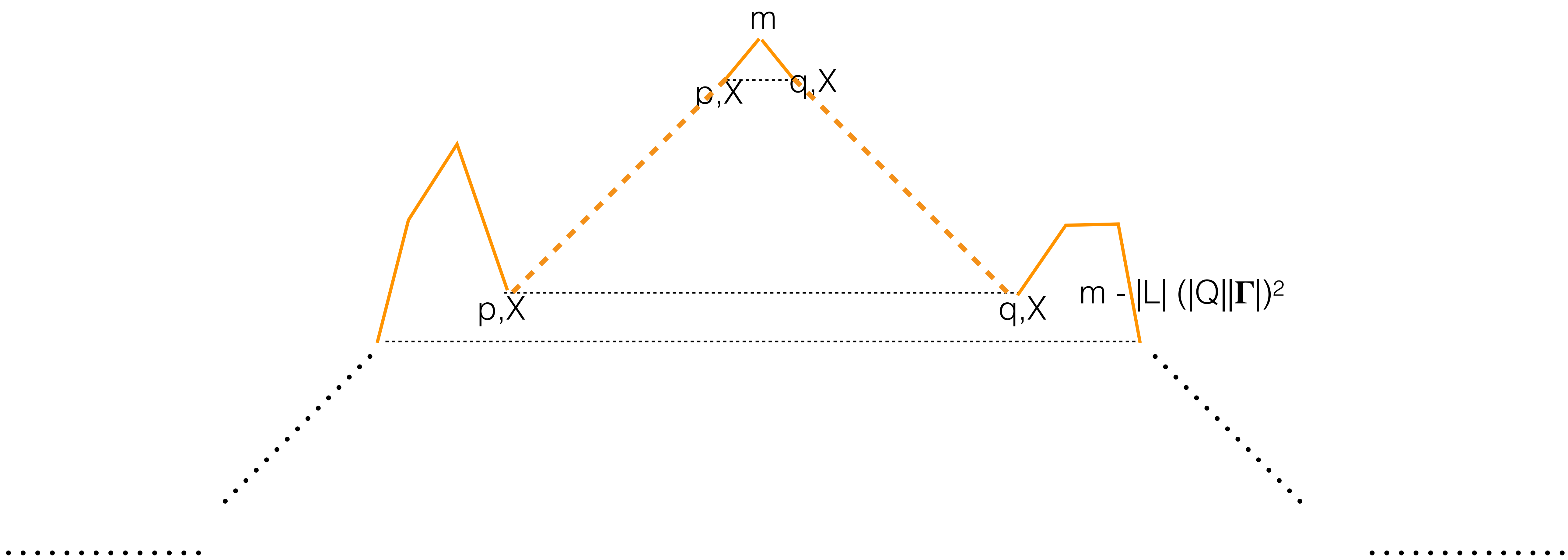
# Stack height bounding...



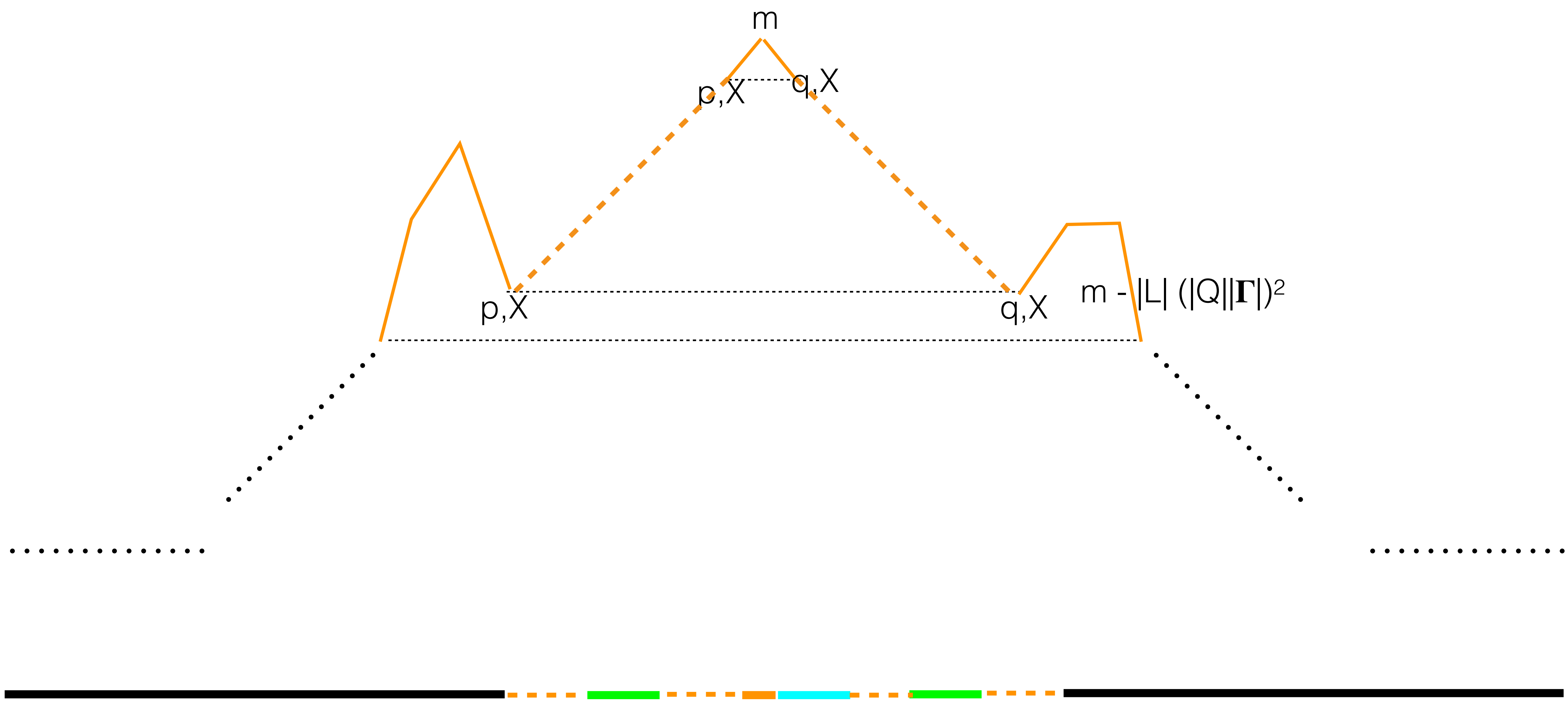
# Stack height bounding...



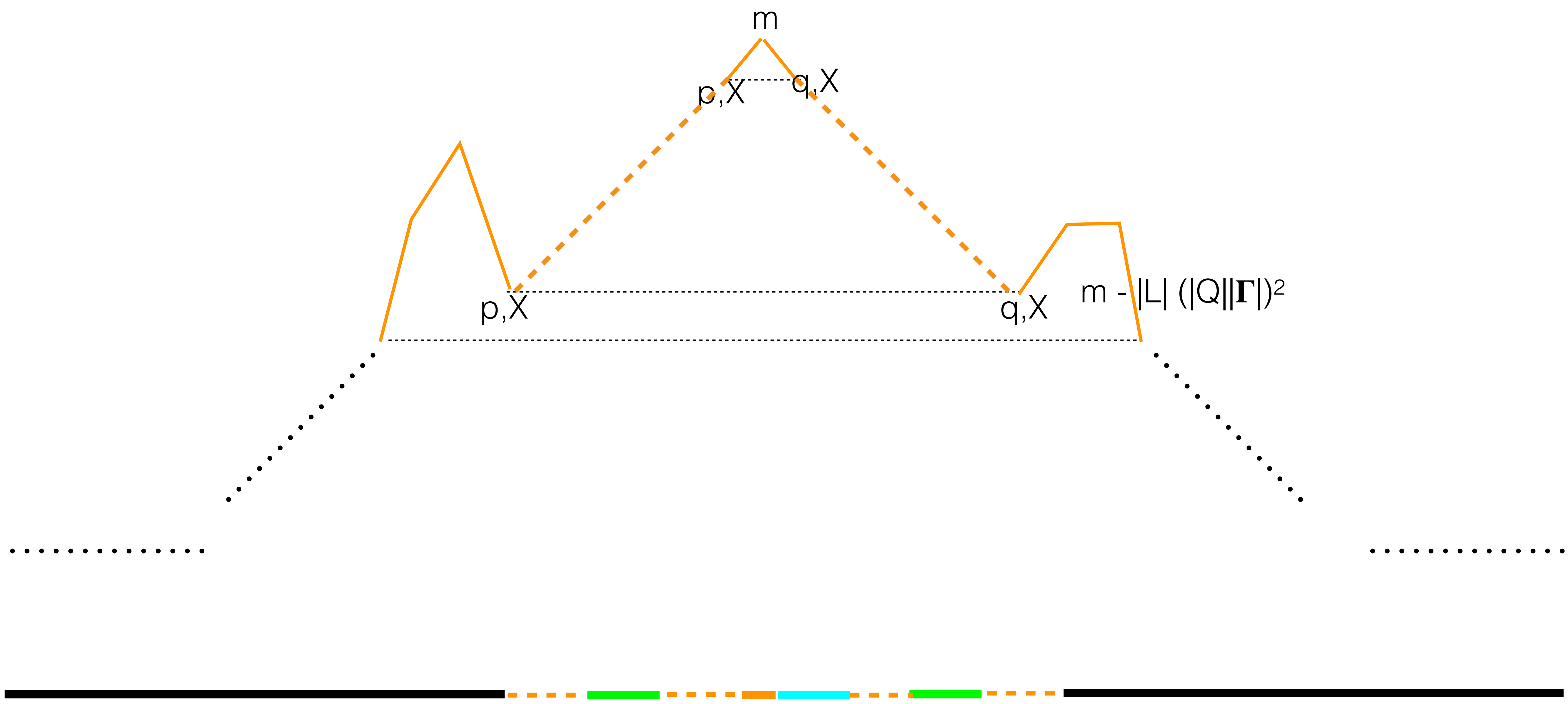
# Stack height bounding...



# Stack height bounding…



# Stack height bounding...



More locks available for other processes below

# Contextual Locking: >2 processes

The reachability problem for any number of pushdown systems synchronising via contextual locks is decidable.

Exponential (in states, stack alphabet, locks) length paths suffice.
In PSPACE.

Extension to systems with Dynamic thread creation.

Lammich, Muller-Olm, Seidl, Werner SAS13

# Asynchronous programs:

Sen and Vishwanathan CAV06, Ganty and Majumdar TOPLAS12 …,

```
Proc one()
{
        < …. >
         call function();
         async-call function();

}
```

# Asynchronous programs:

Sen and Vishwanathan CAV06, Ganty and Majumdar TOPLAS12 …,

```
Proc one()
{
         < …. >
          call function();
          async-call function();

}
```

- Recursive programs with option to invoke asynchronous calls.
- The asynchronous calls are stored as tasks that can be retrieved later and executed
- The stored tasks have no specific order.
- The tasks are executed atomically when there are no other pending calls.

# Asynchronous programs:

Sen and Vishwanathan CAV06, Ganty and Majumdar TOPLAS12 …,

```
Proc one()
{
        < …. >
         call function();
         async-call function();

}
```

- Recursive programs with option to invoke asynchronous calls.
- The asynchronous calls are stored as tasks that can be retrieved later and executed
- The stored tasks have no specific order.
- The tasks are executed atomically when there are no other pending calls.

Modeled as a PDS augmented with a multi-set. (MPDS)

# Decidability:

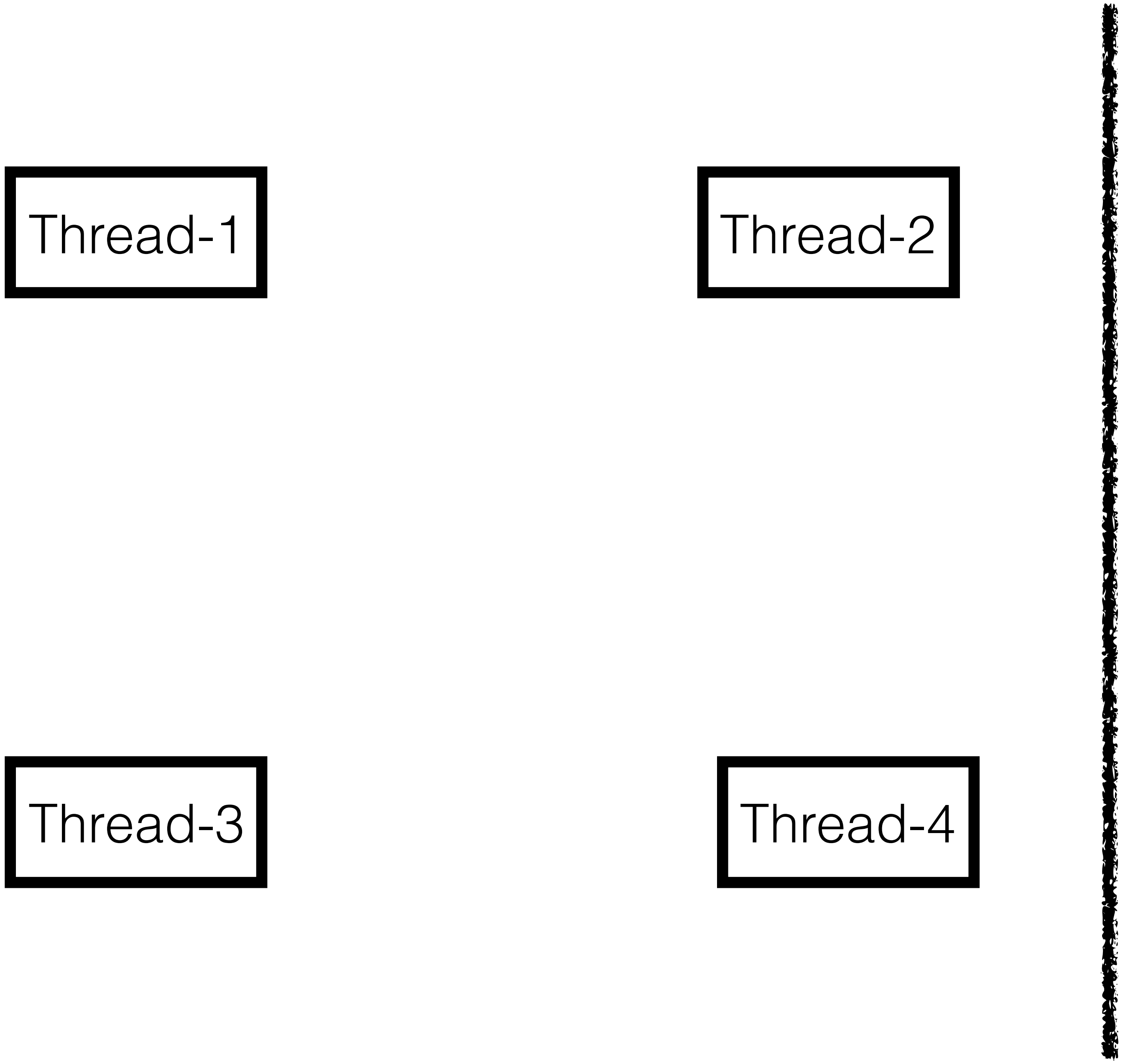
Control state reachability for pushdown systems equipped with a multi-set is EXPSPACE-Complete

# Decidability:

Control state reachability for  pushdown systems equipped with a multi-set is EXPSPACE-Complete

Sen and Vishwanathan CAV06
Atig, Bouajjani, Touili FSTTCS08
Ganty and Majumdar TOPLAS12

# Multi-threaded version:



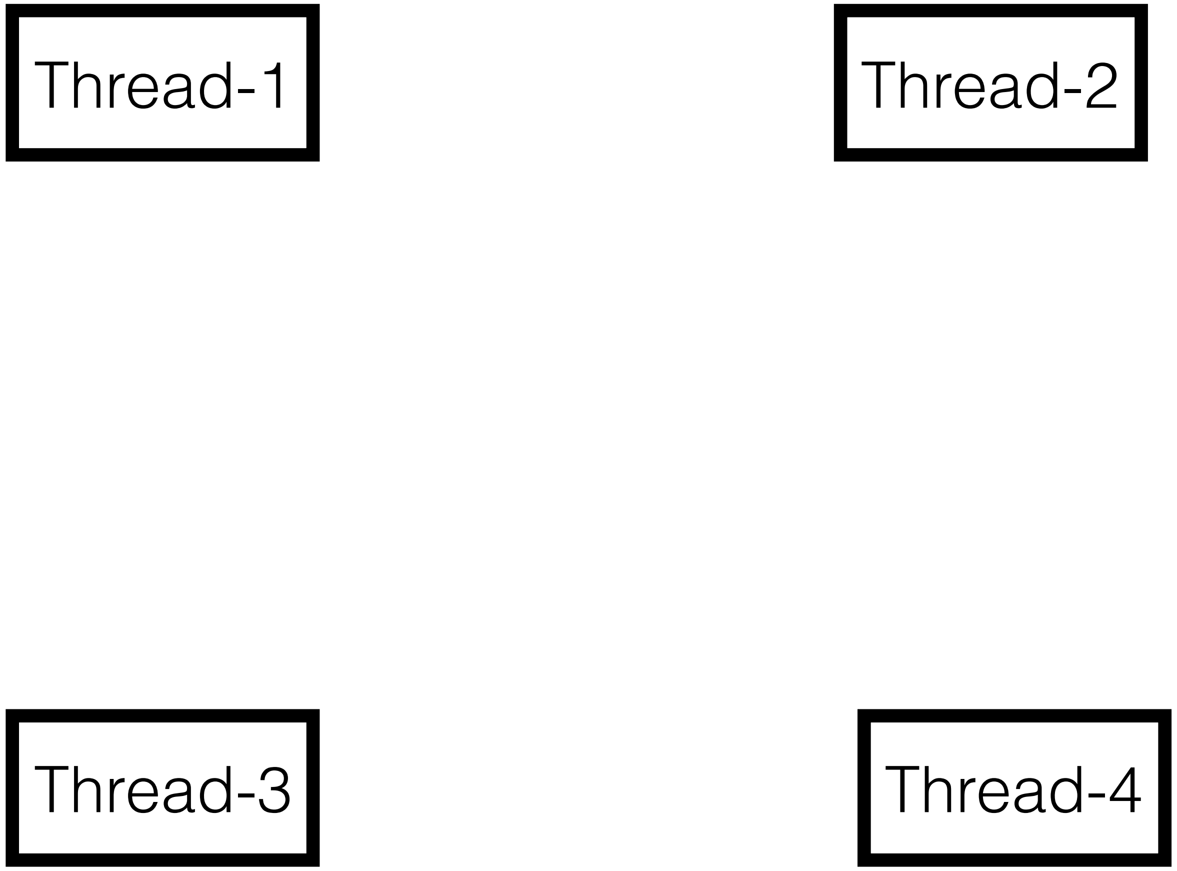
Thread-1

Thread-2

Thread-3

Thread-4

# Multi-threaded version:



Thread-1

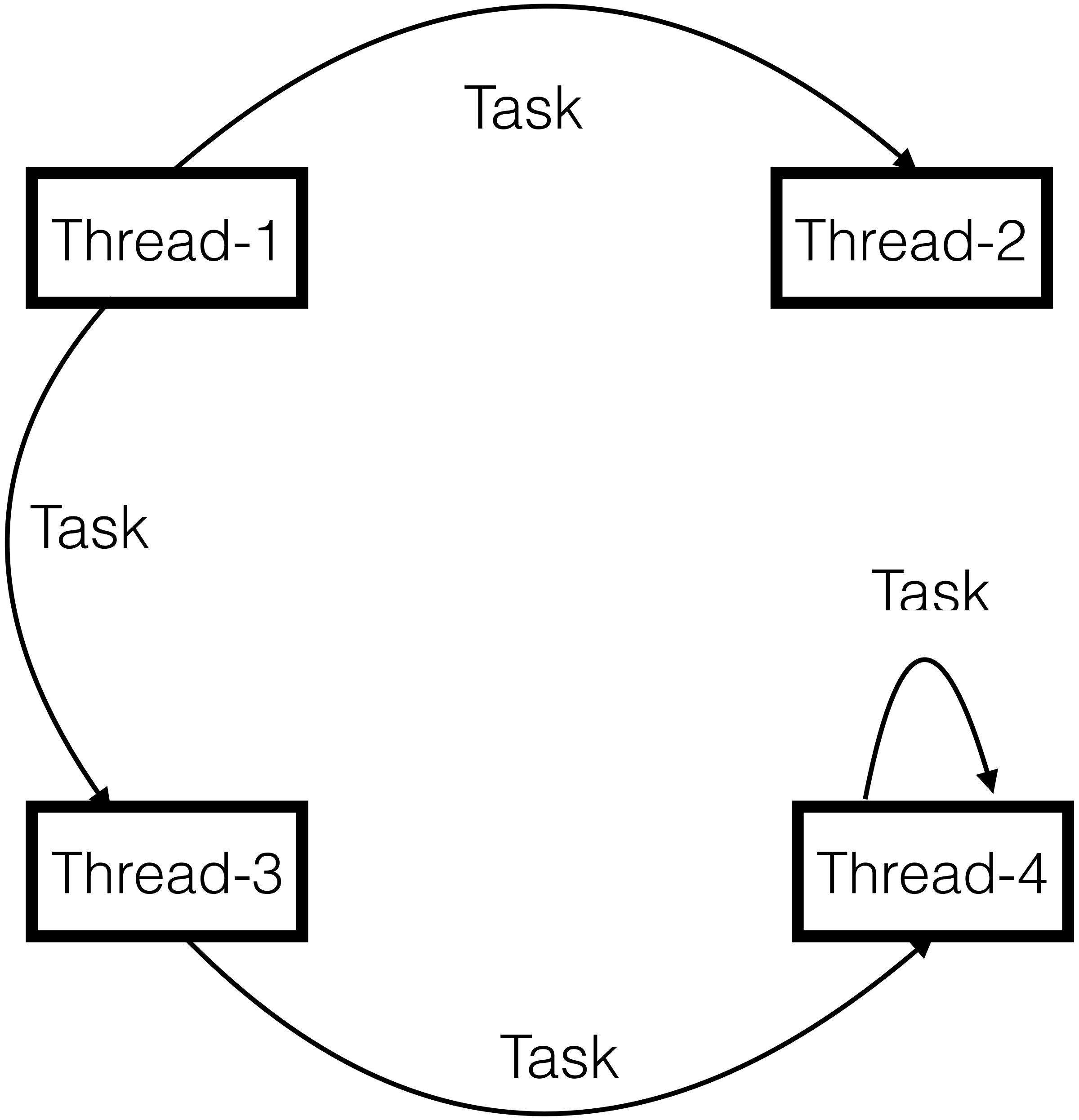
Thread-2

Thread-3

Thread-4

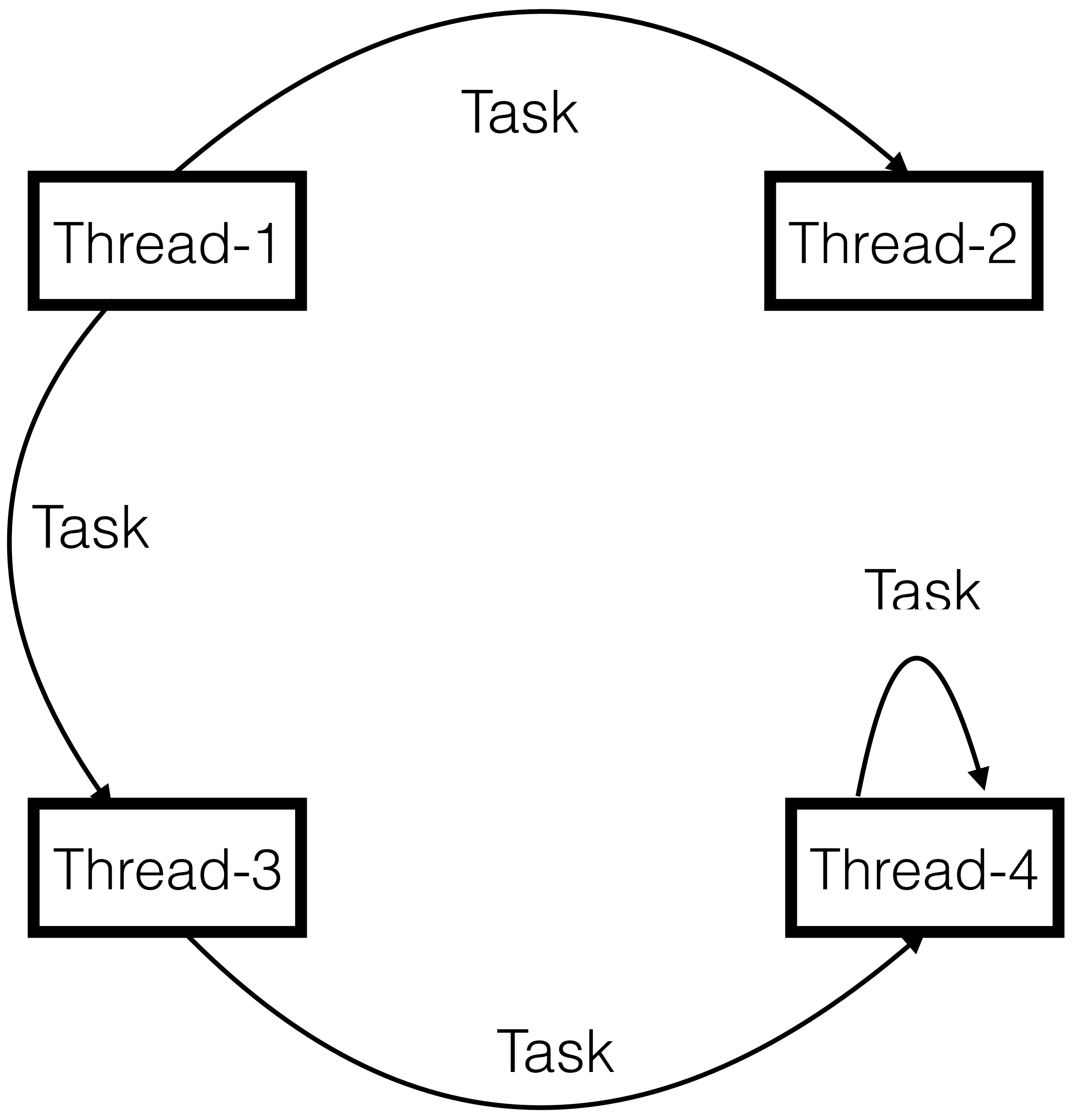
- Programs with multiple threads running in parallel

# Multi-threaded version:



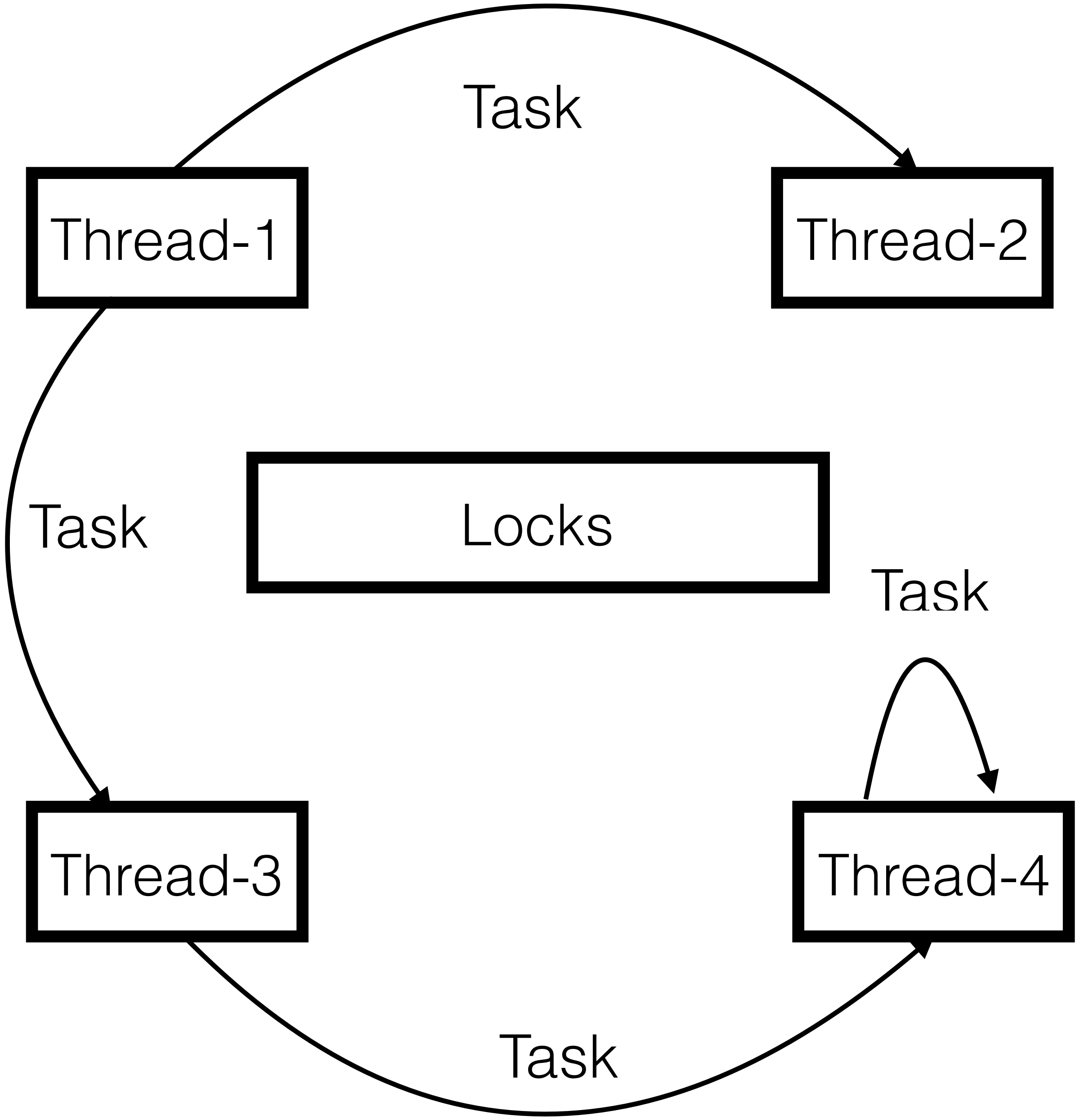
- Programs with multiple threads running in parallel
- Threads can either make a synchronous call or an asynchronous call by delegating it to some thread

# Multi-threaded version:

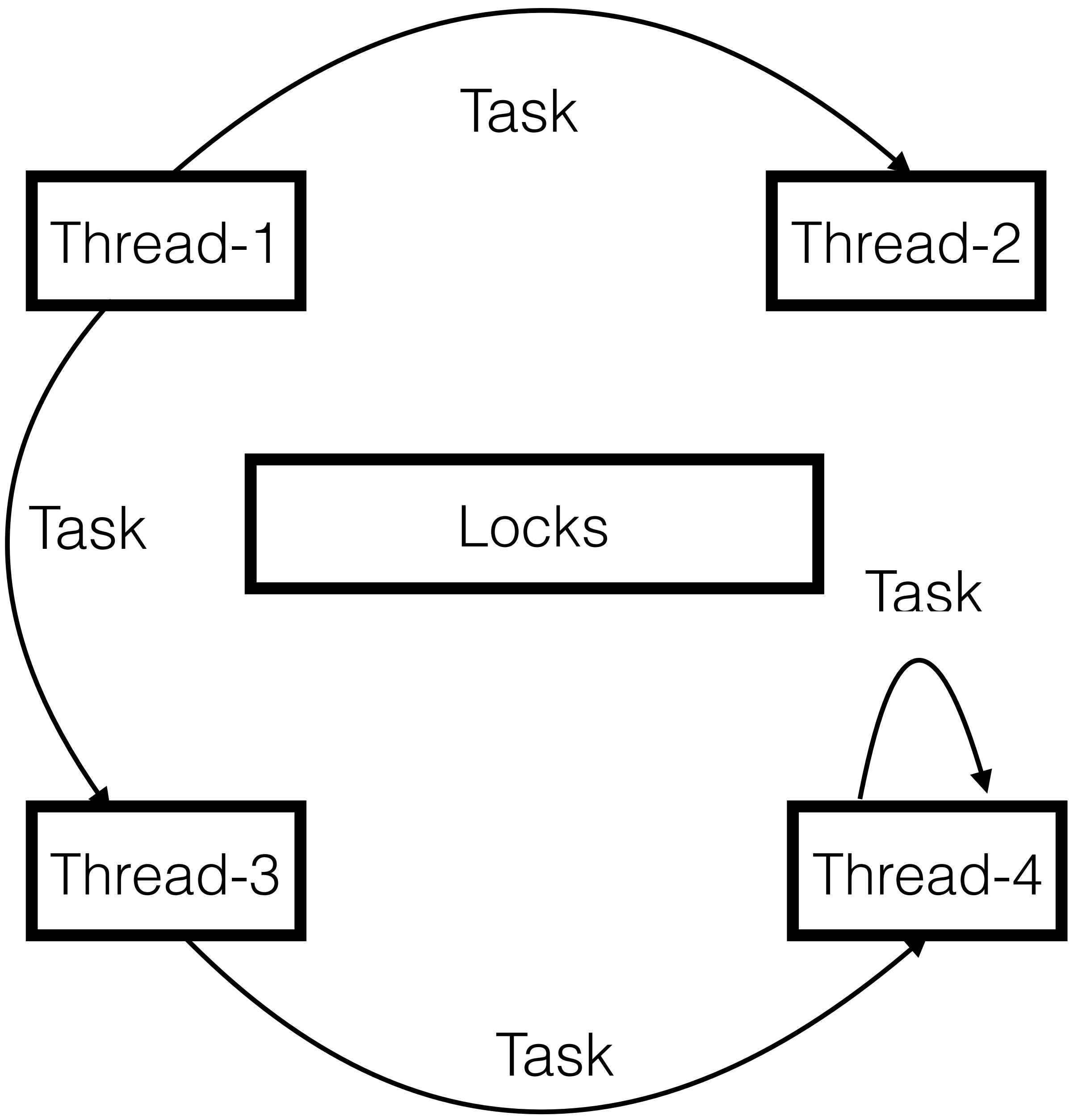


- Programs with multiple threads running in parallel
- Threads can either make a synchronous call or an asynchronous call by delegating it to some thread
- Threads have unbounded unordered buffers to store the tasks

# Communication:

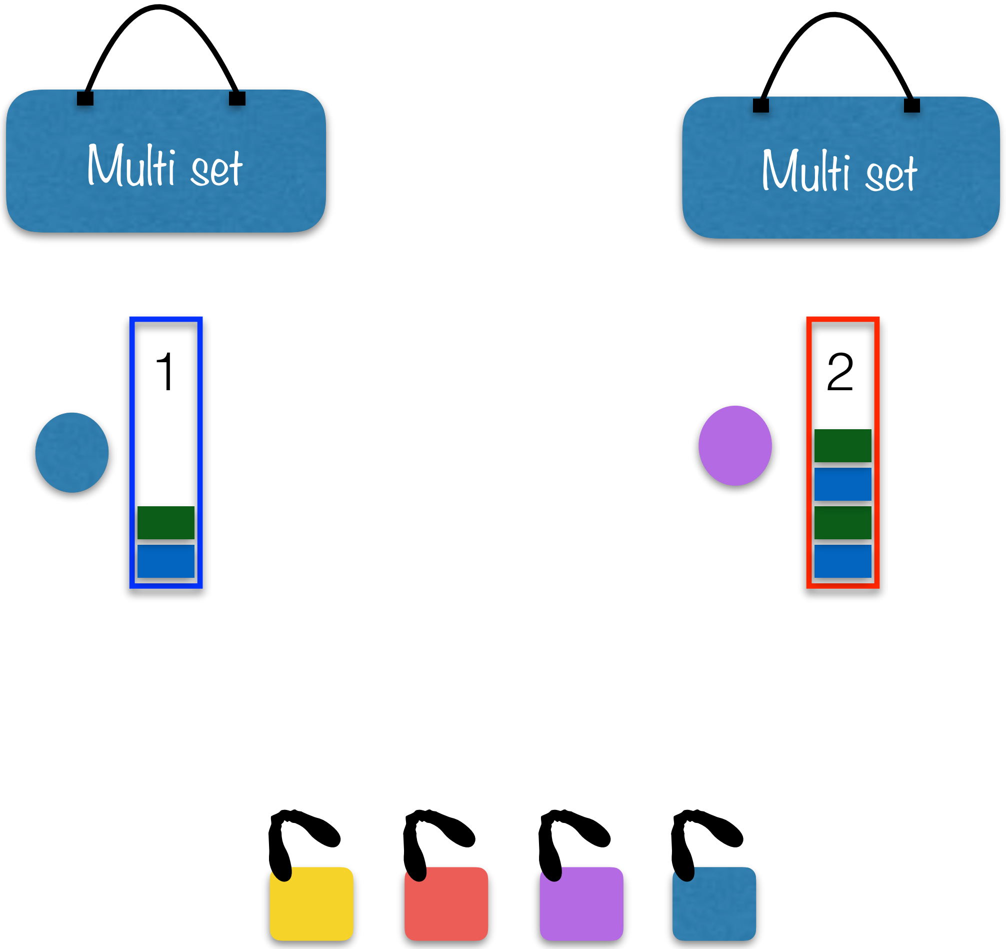


# Communication:



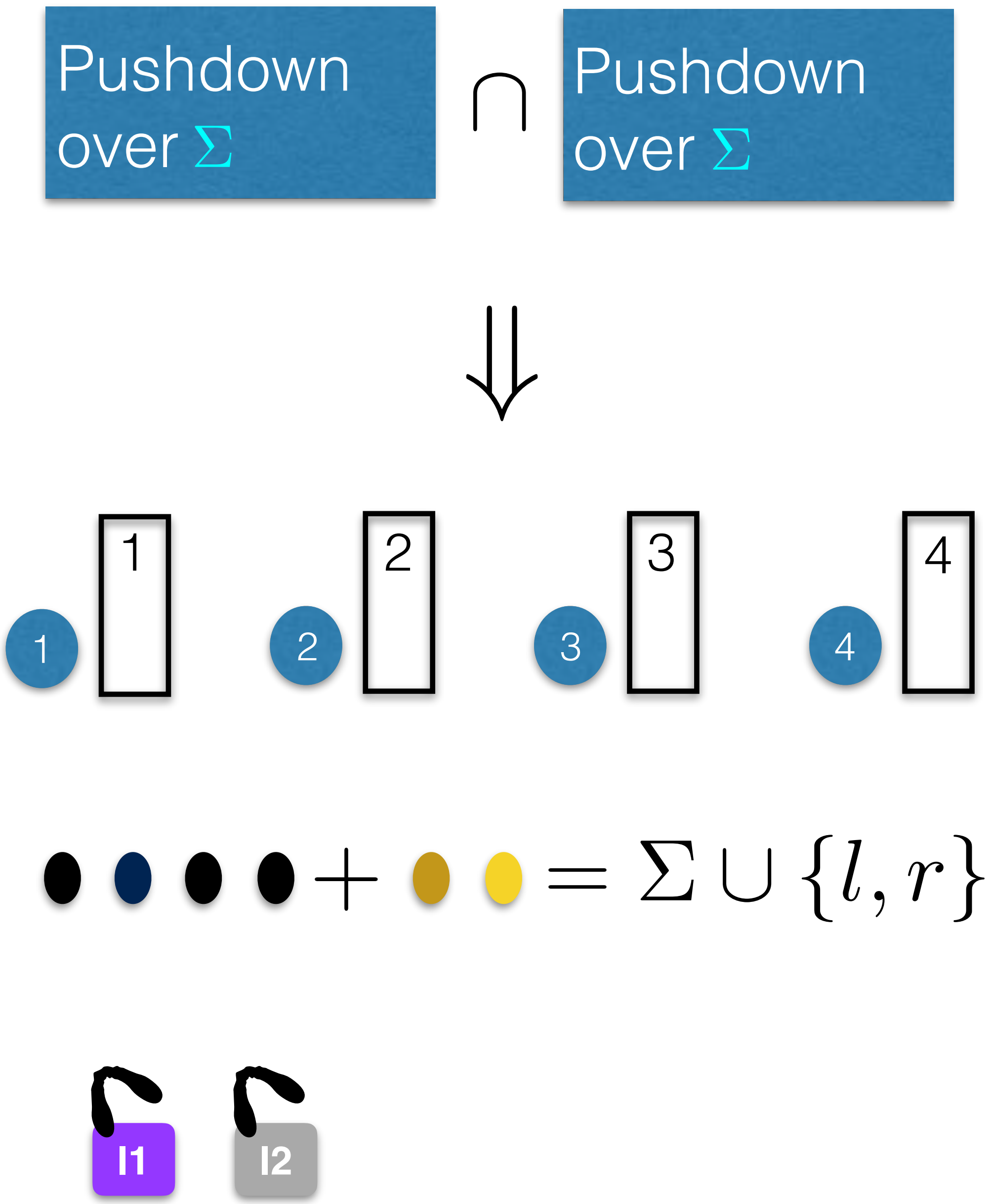
We consider in asynchronous programs synchronising through locks

# Asynchronous programs + Locks:



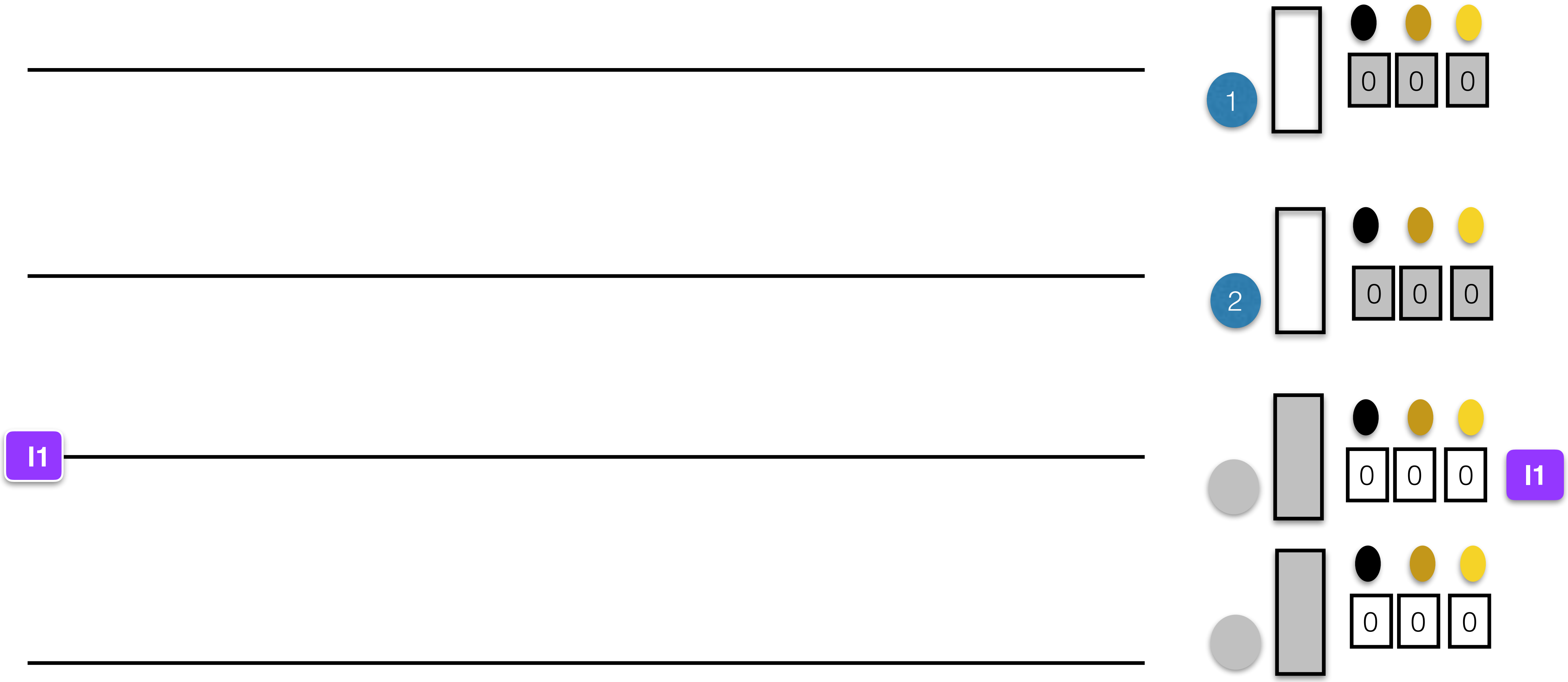
- Pushdown systems with
    - Multi-sets to hold tasks
- A finite set of global locks

# Undecidability under nested locking



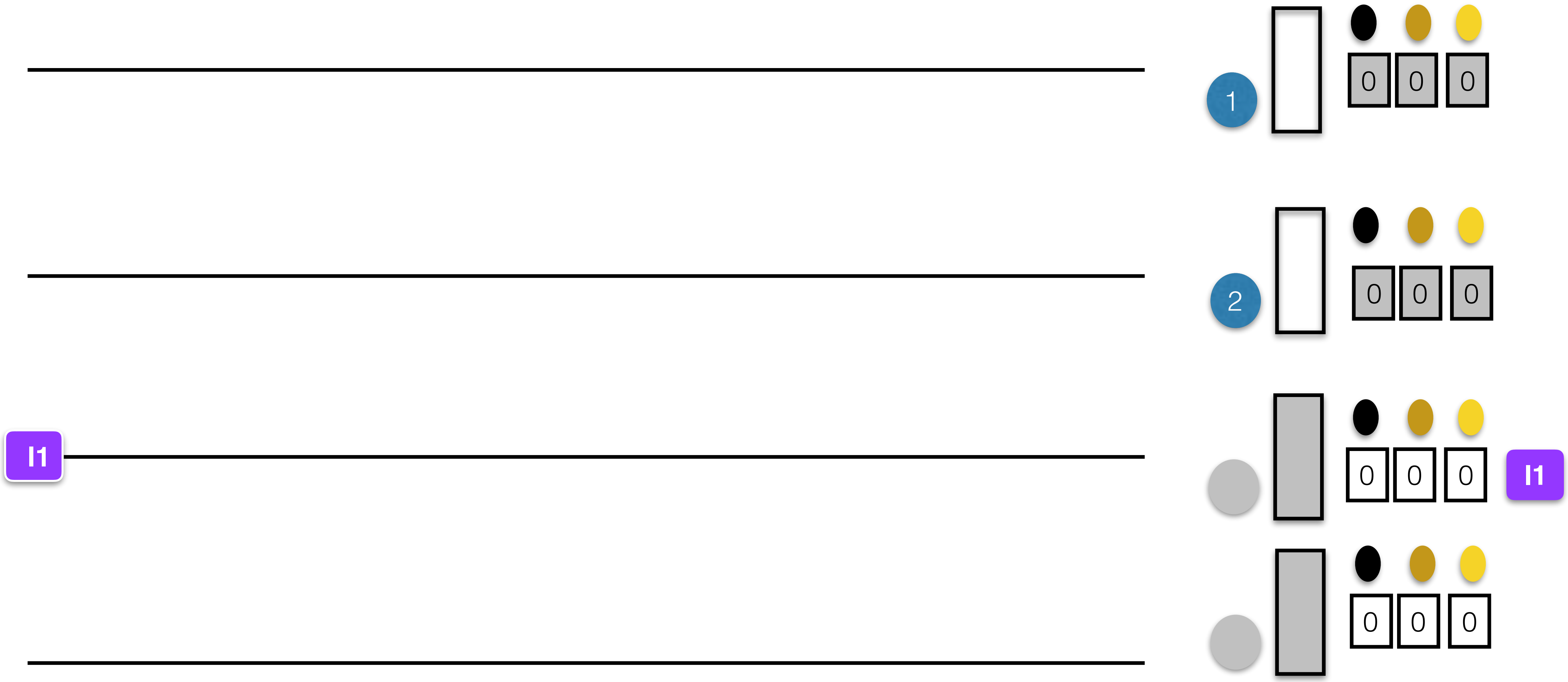
- Reduce intersection of two pushdown systems
- 4 threads along with two locks and set of tasks
- The set of tasks is the alphabet of pushdown systems along with two additional tasks

# Simulation of a move:



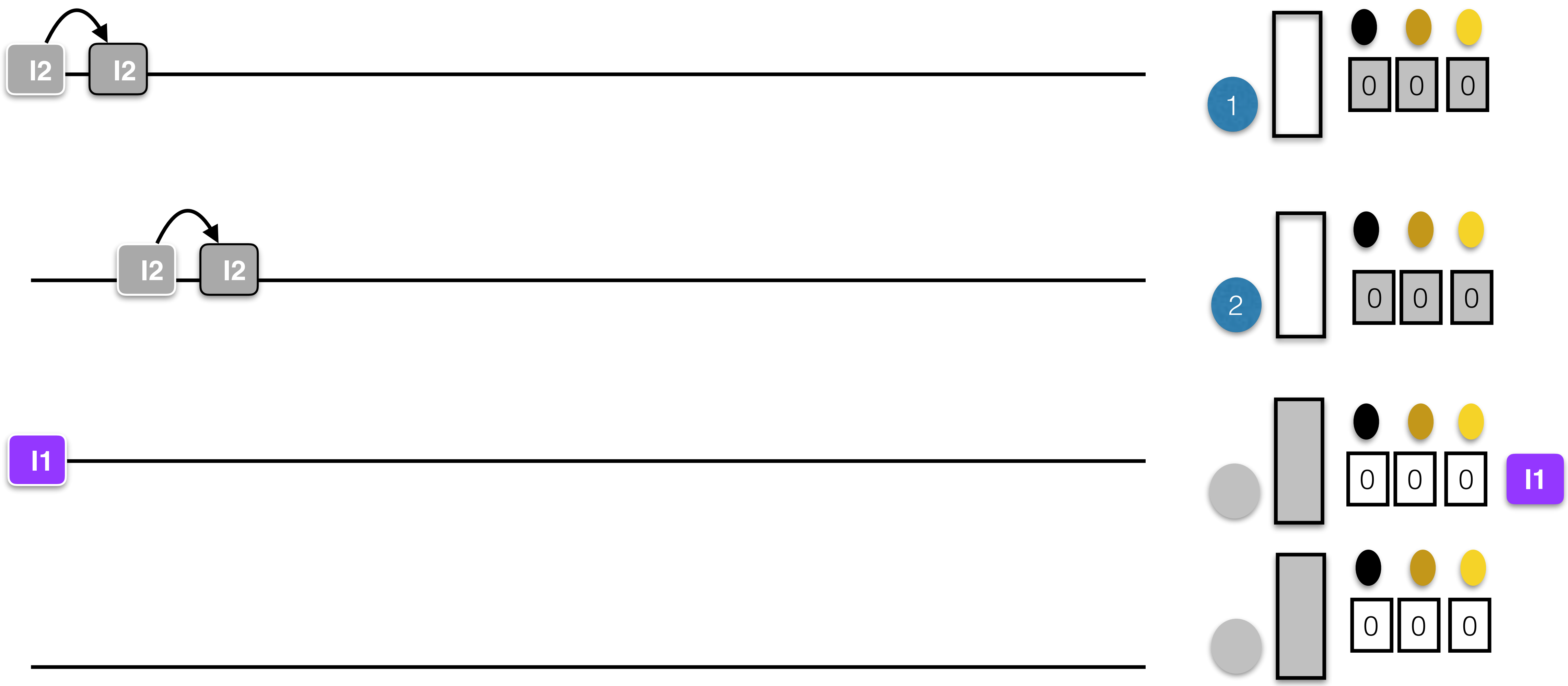
We will show how to simulate a single move of each of the pushdown systems

# Simulation of a move:



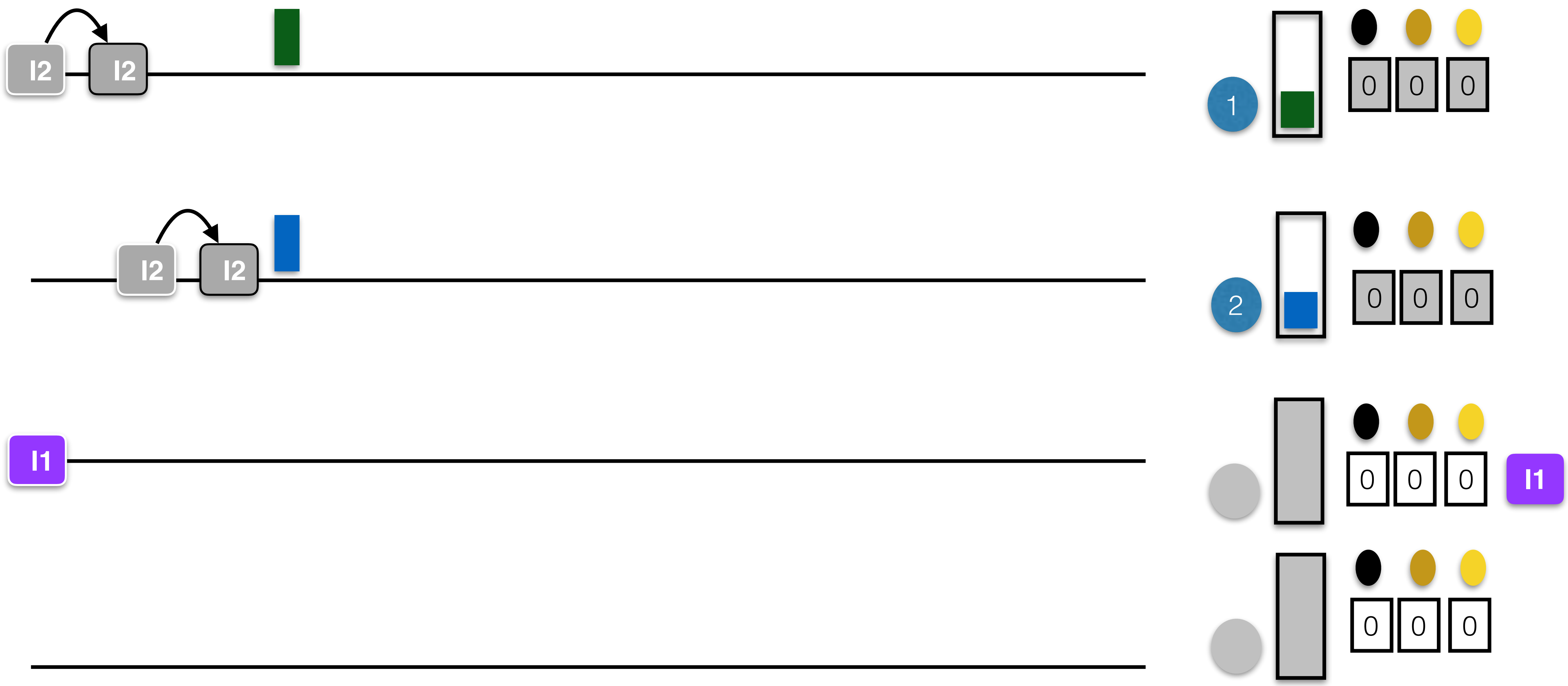
The Simulation starts with process 3 holding I1

# Simulation of a move:



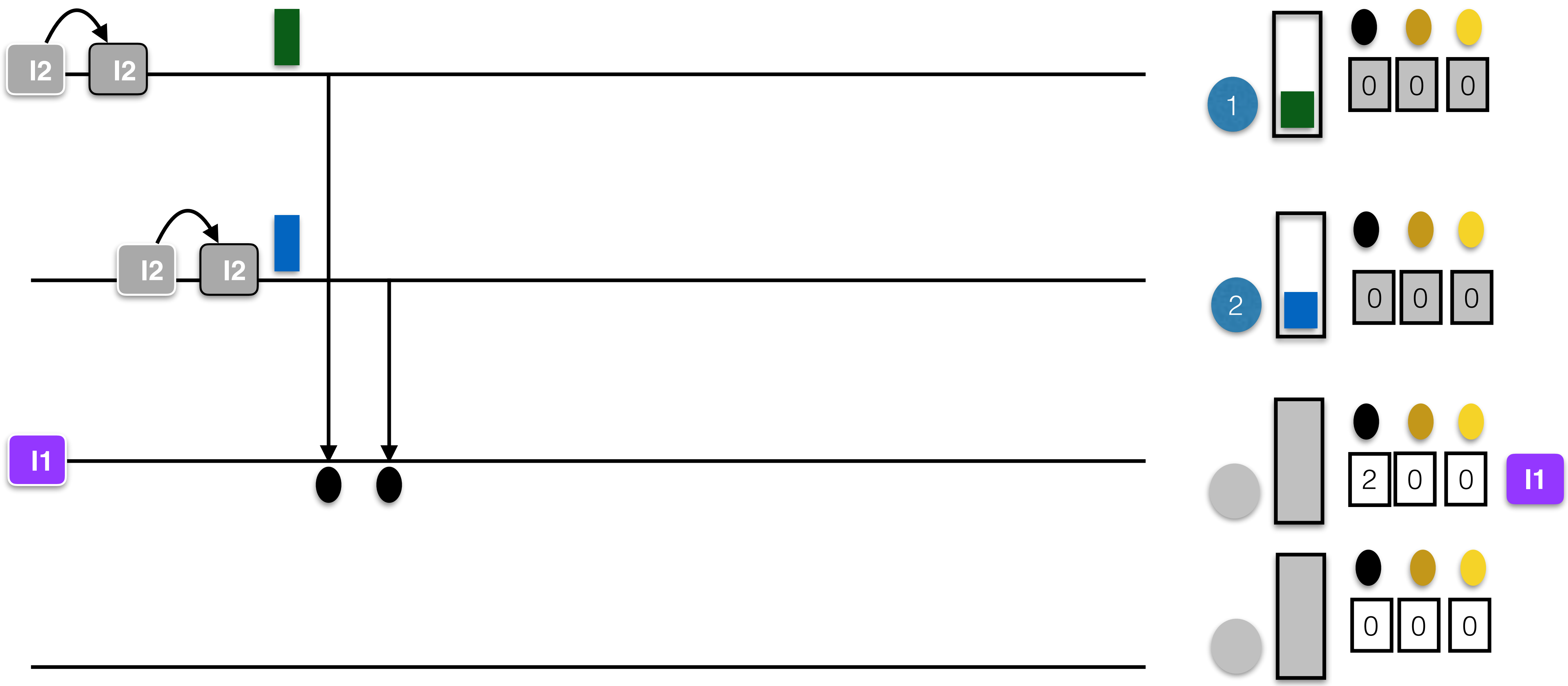
Process 1 and 2 test lock l2

# Simulation of a move:



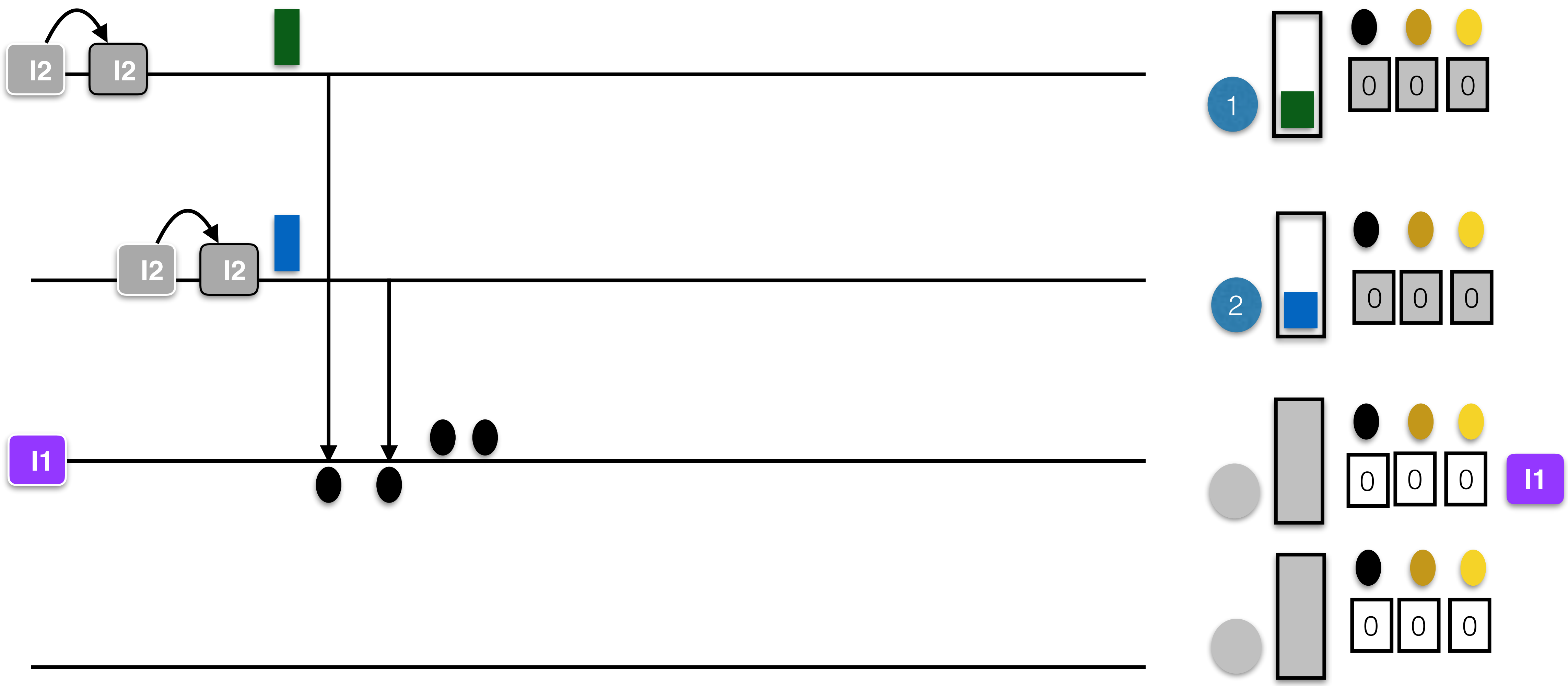
Process 1 and 2 guess an letter and simulate the move

# Simulation of a move:



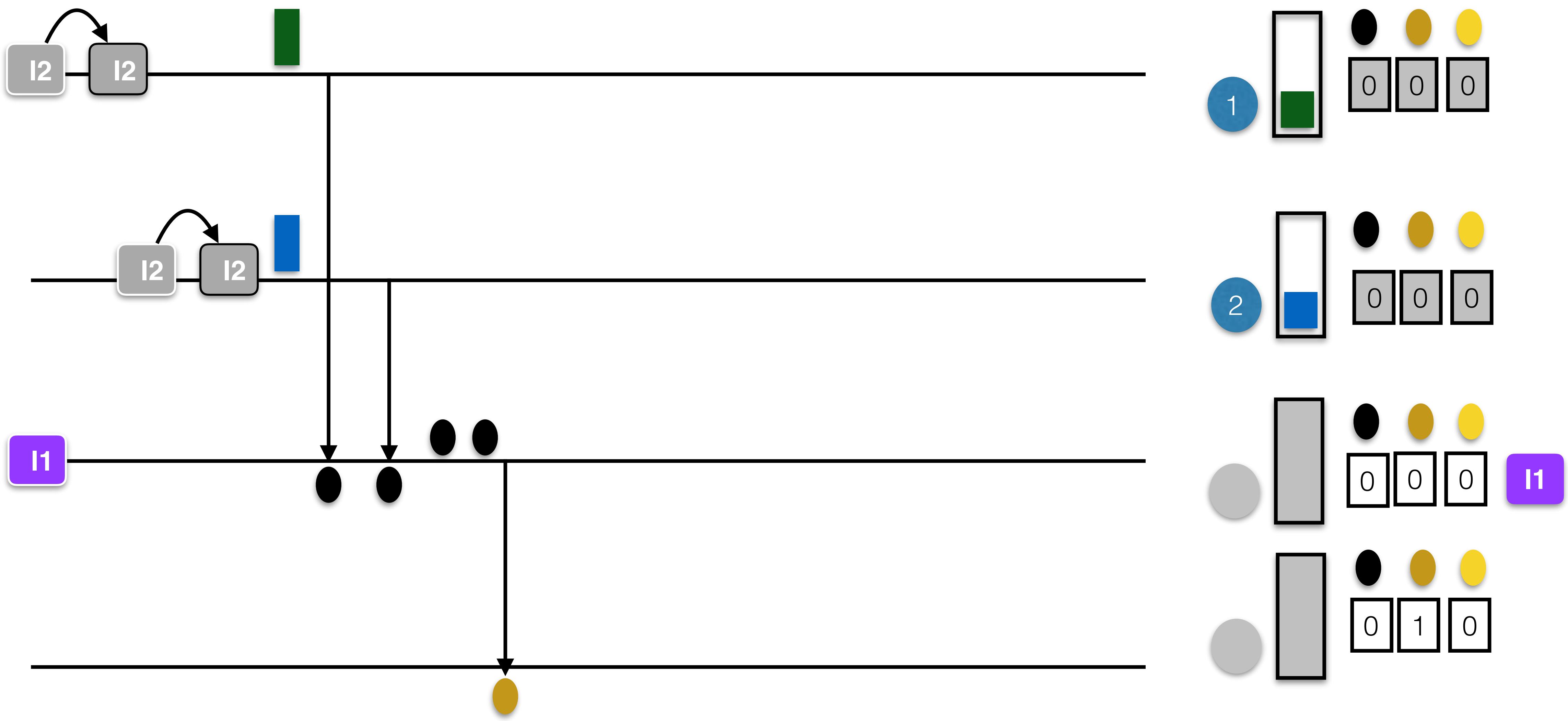
Process 1 and 2 sends the guessed letter to 3

# Simulation of a move:



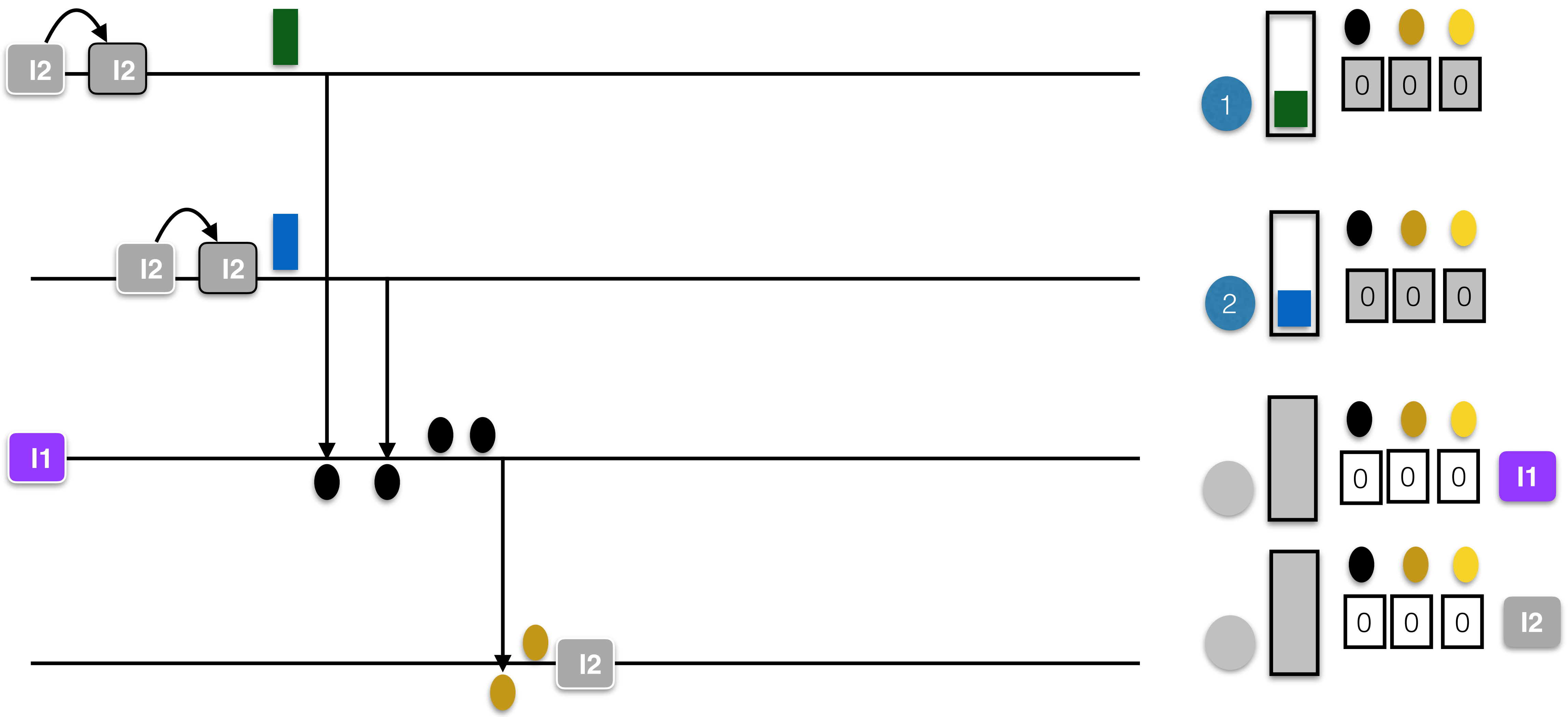
Process 3 reads and verifies that the letters match

# Simulation of a move:



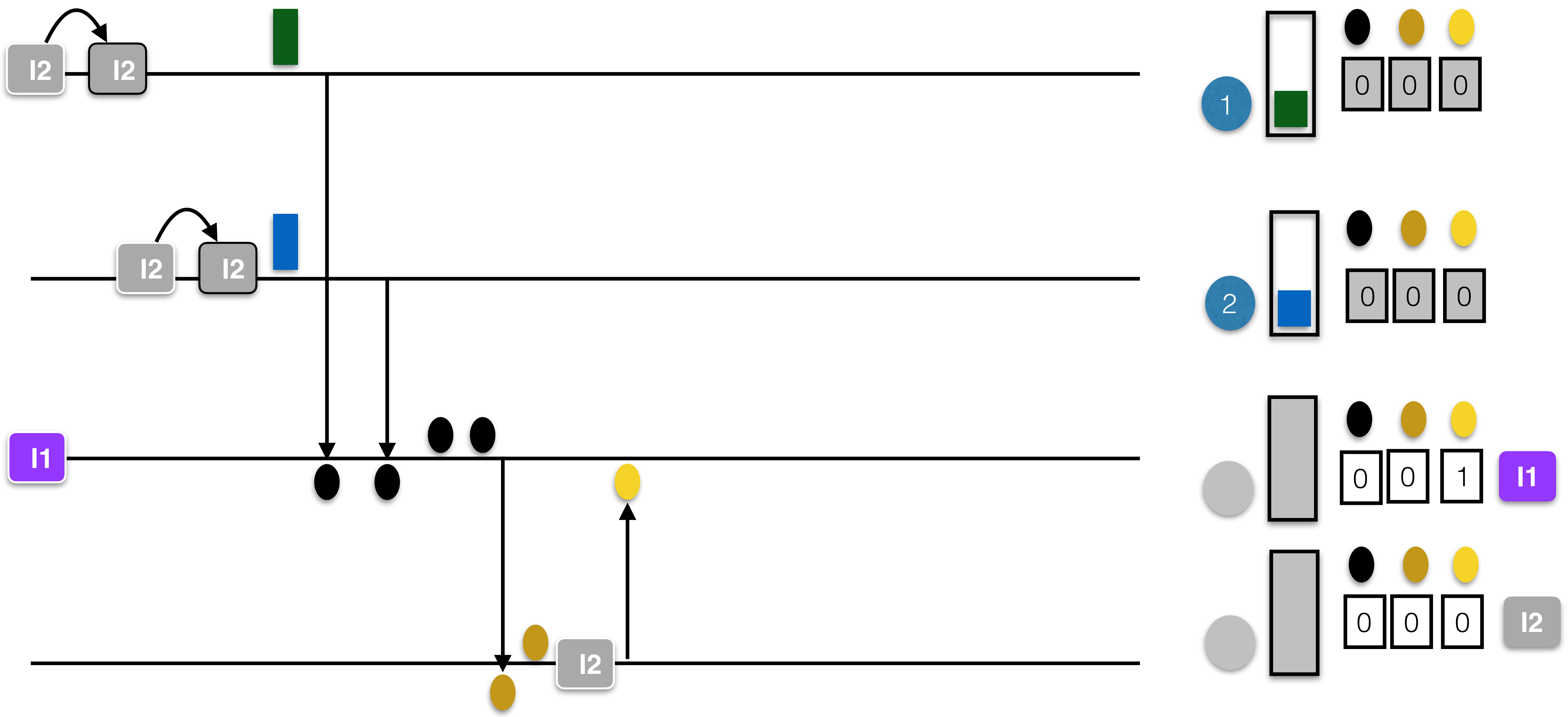
Process 3 requests 4 to hold lock l2

# Simulation of a move:



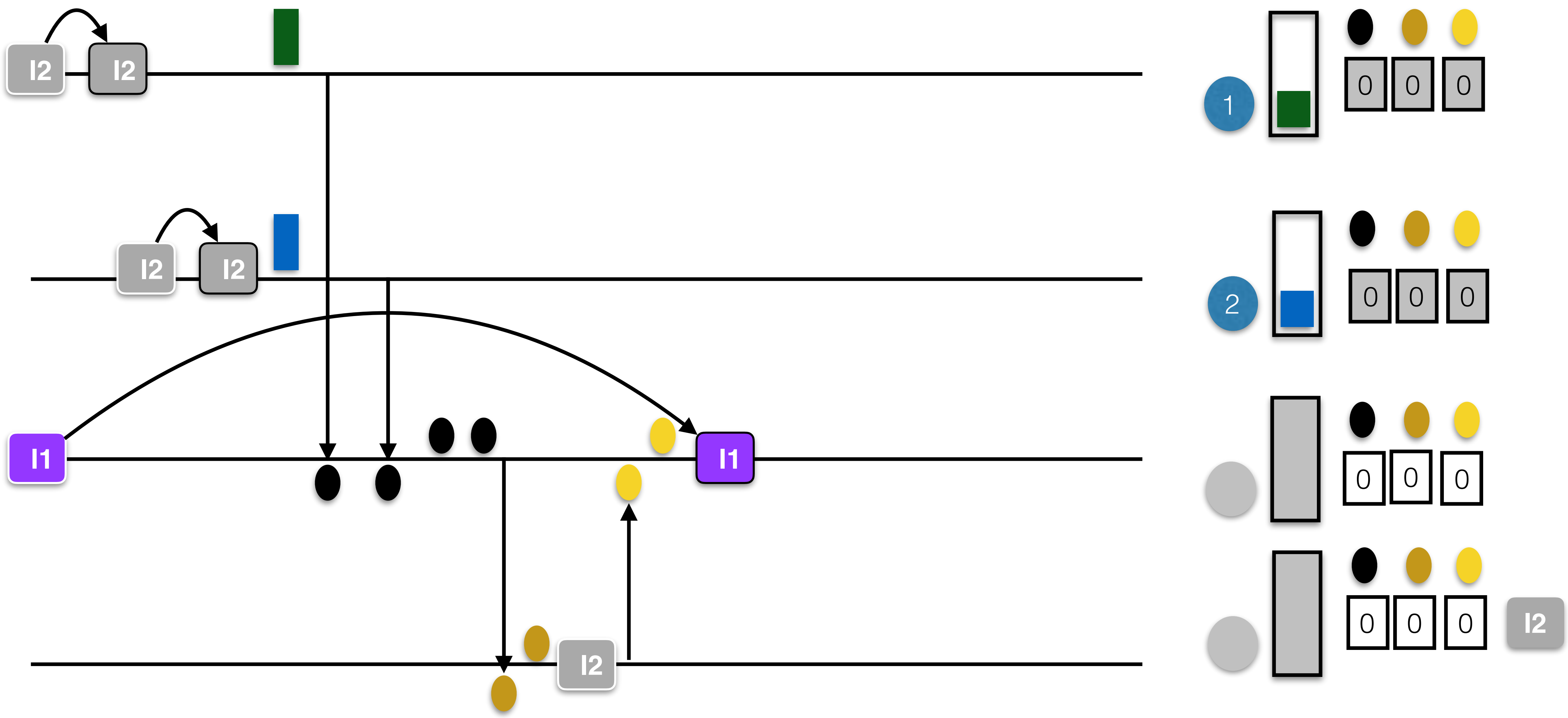
Process 4 reads the request and holds lock l2

# Simulation of a move:



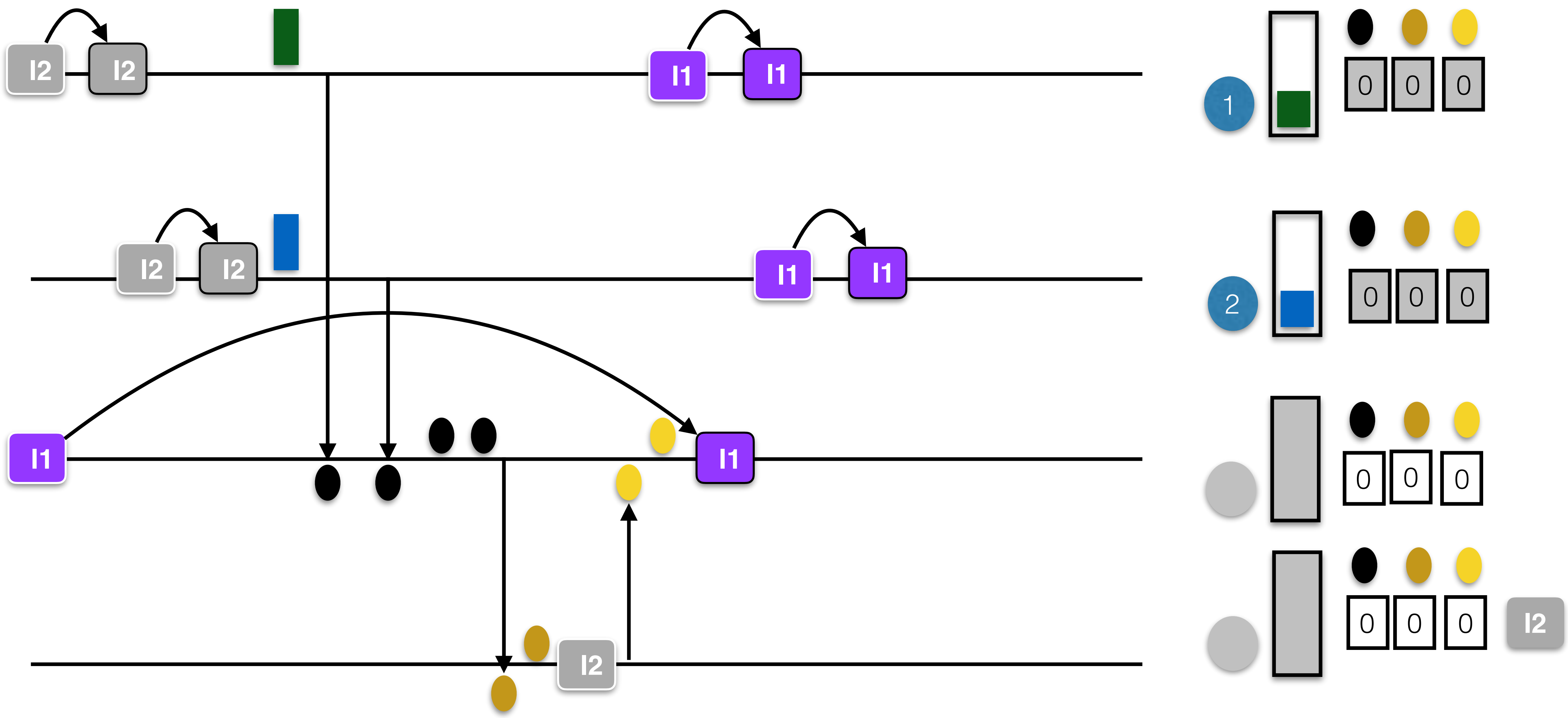
Process 4 reads the request and holds lock l2

# Simulation of a move:



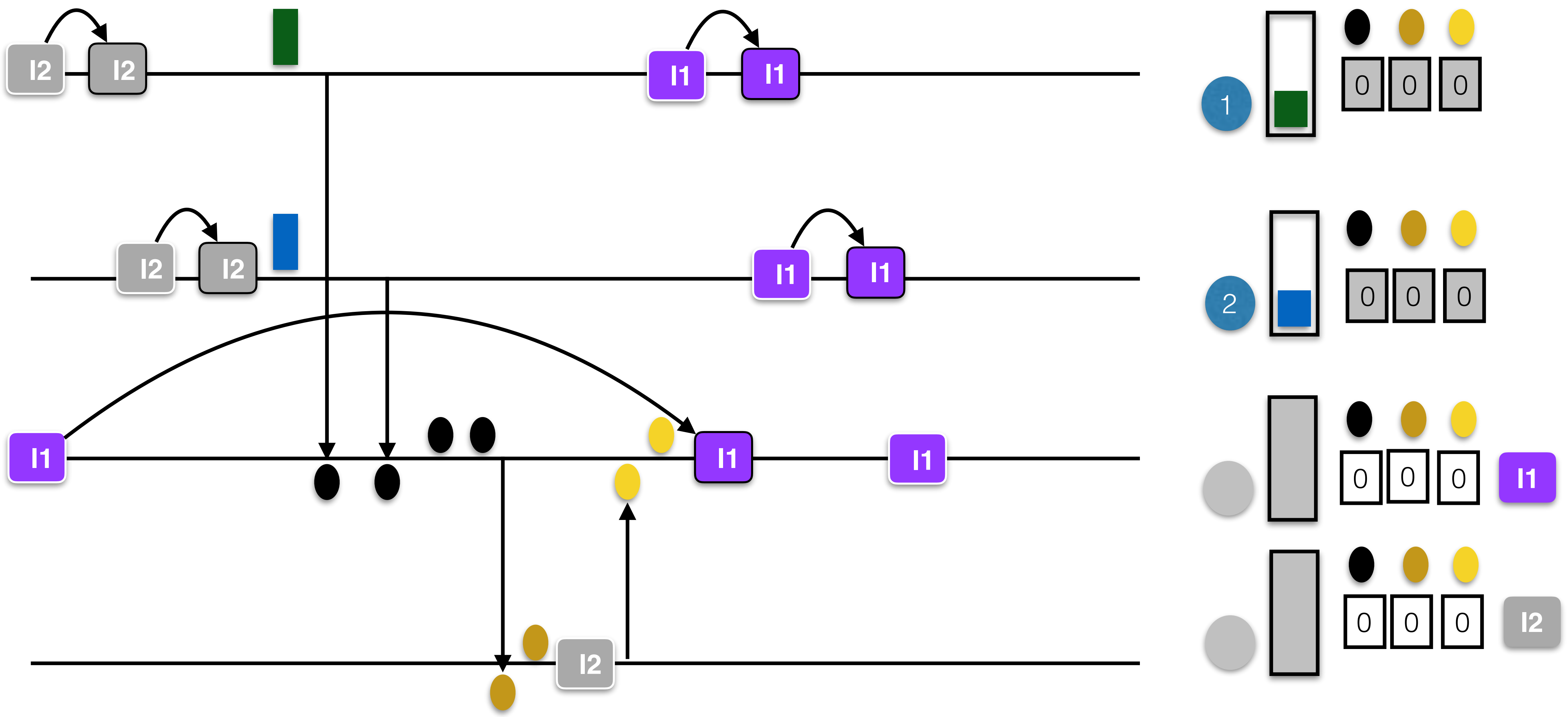
Process process 3 releases I1 on learning I2 is taken

# Simulation of a move:



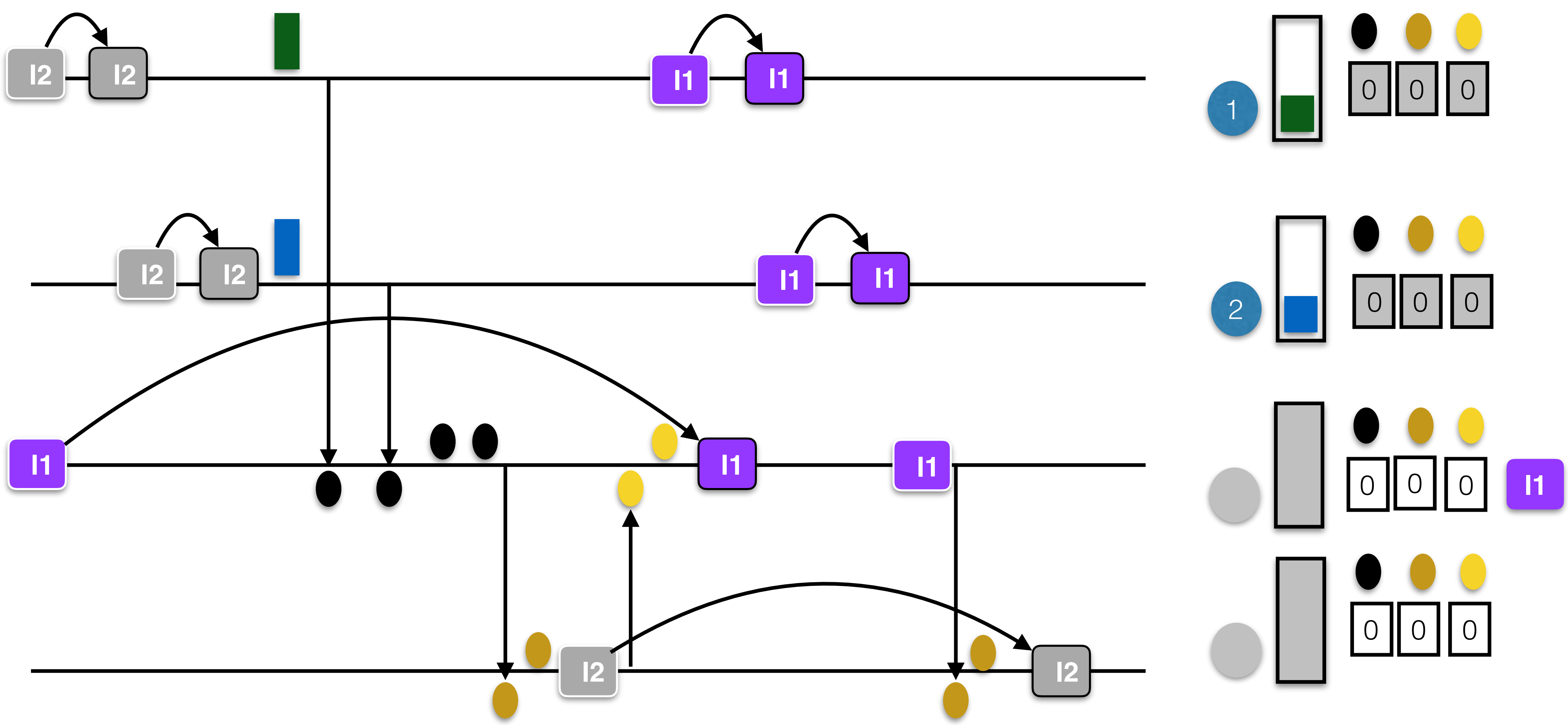
Process 1 and 2 tests lock l1

# Simulation of a move:



Process 3 retakes lock l1 and asks 4 to release l2

# Simulation of a move:

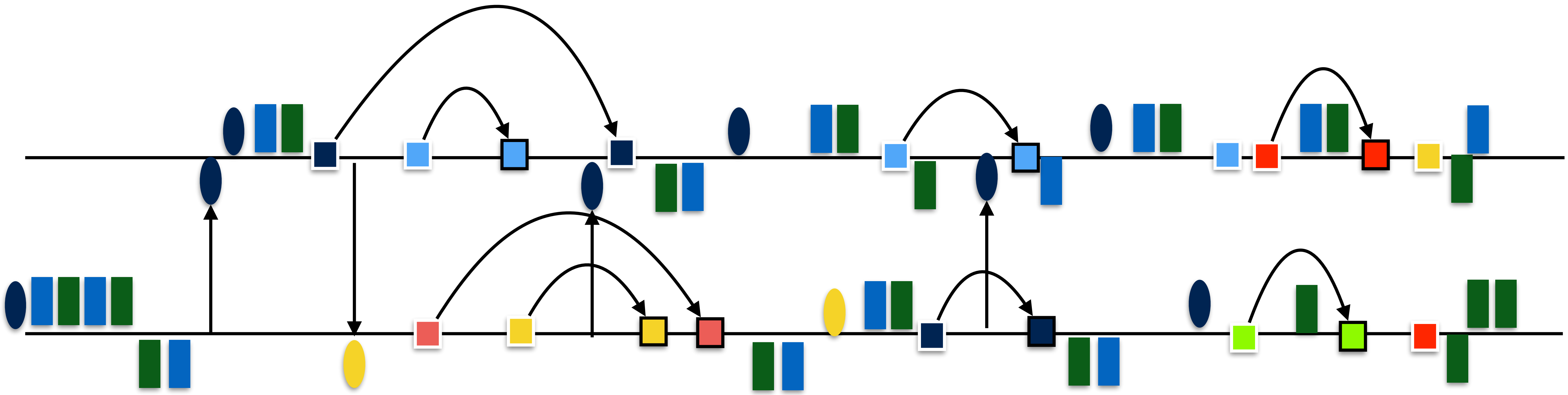


Process 3 retakes lock l1 and asks 4 to release l2

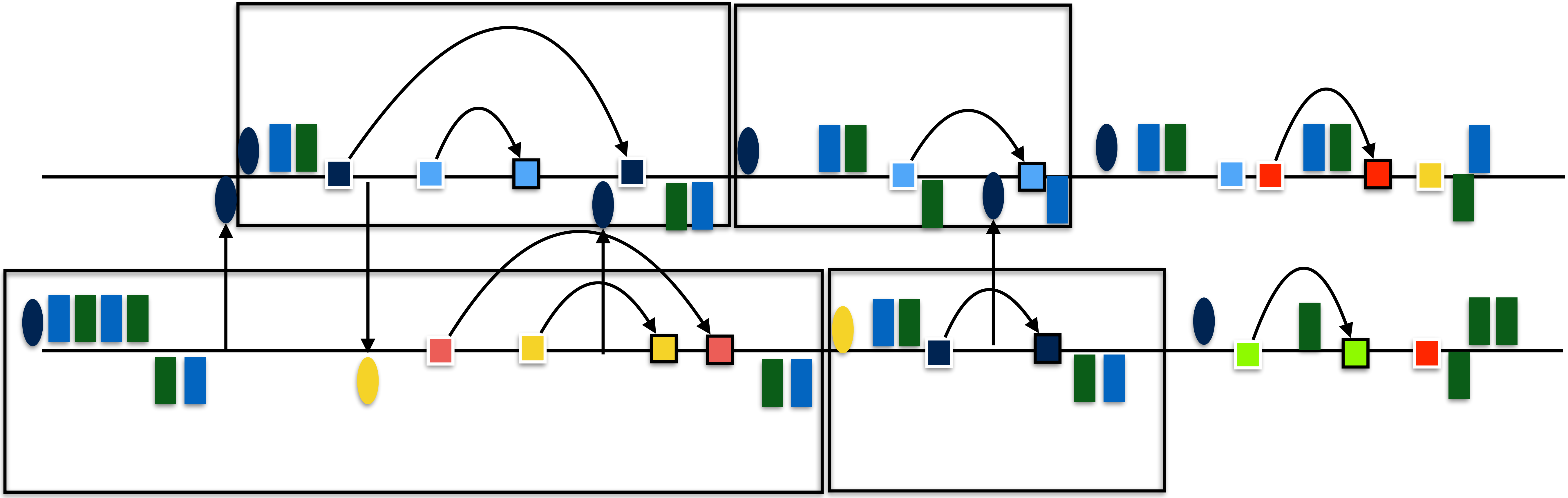
# Task locking restriction:

Locks can be held only by task. That is, locks are held only when the stack is not empty

# Phases of a thread:

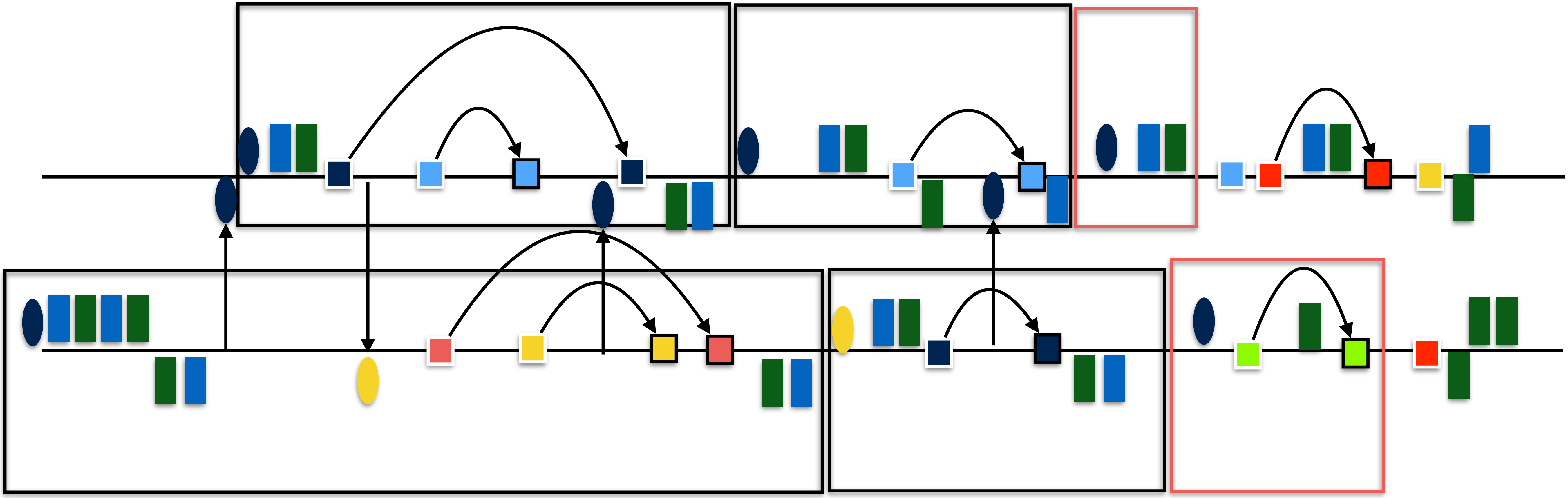


# Phases of a thread:



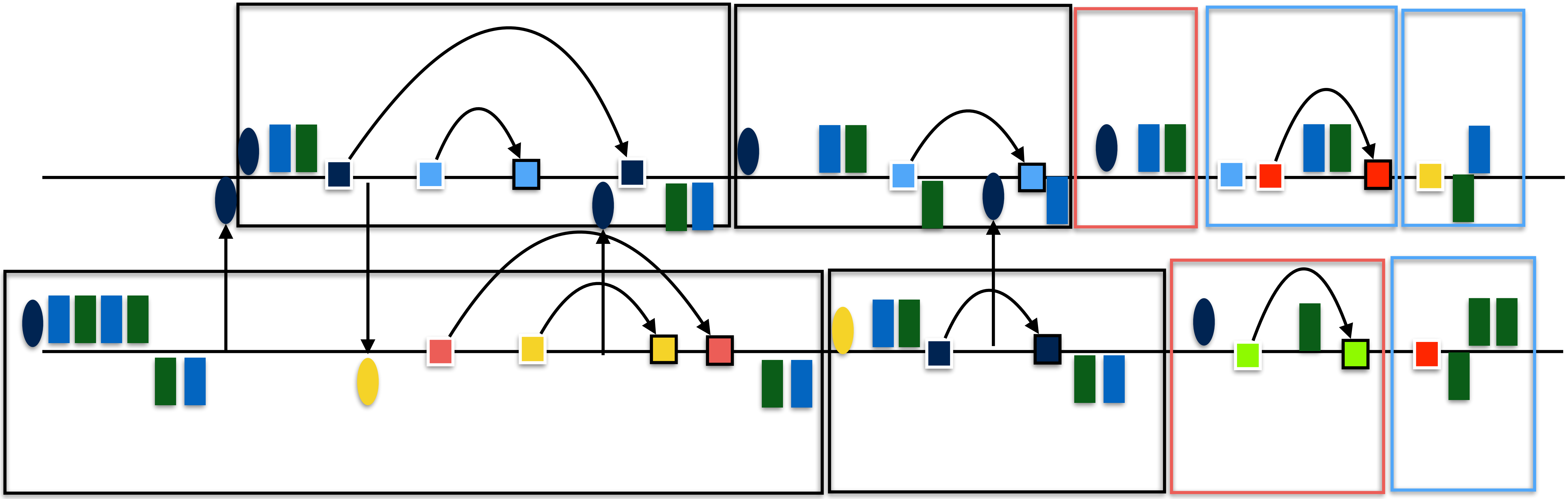
- Task Phases: Complete execution of one task

# Phases of a thread:



- Task Phases: Complete execution of one task
- Boundary Phase: Initial part of a nonterminating task where all locks are returned

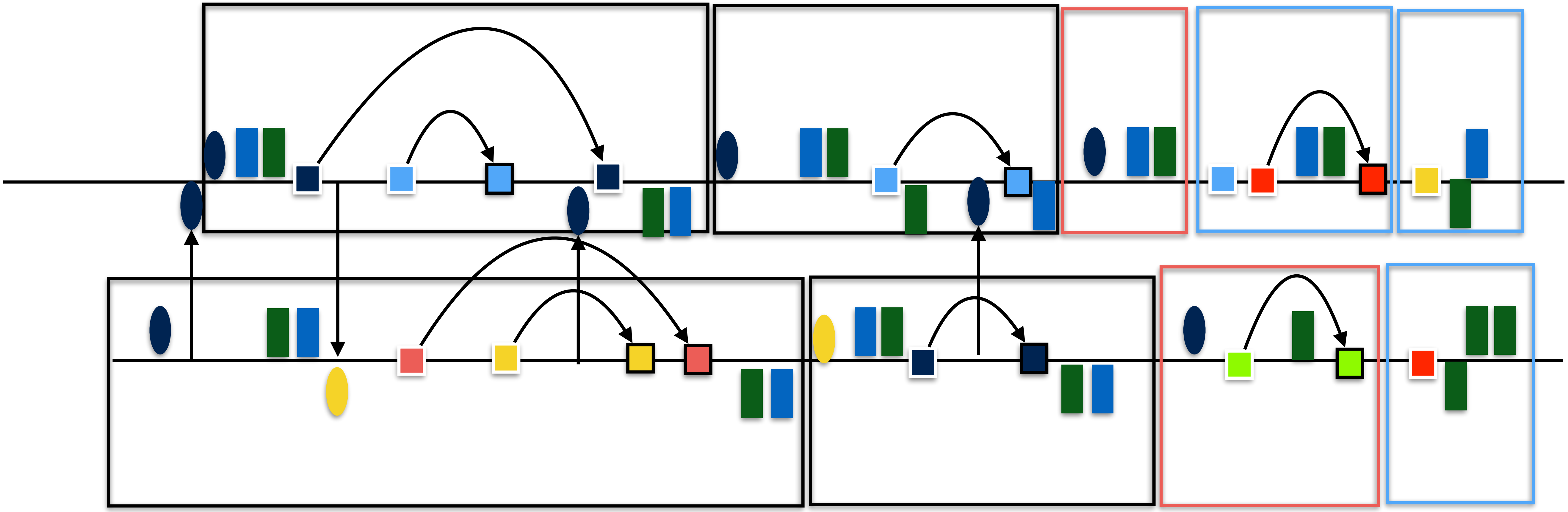
# Phases of a thread:



- Task Phases: Complete execution of one task
- Boundary Phase: Initial part of a nonterminating task where all locks are returned
- Lock phases: Part of a nonterminating task that begins with a lock that is never returned, until the next such action.

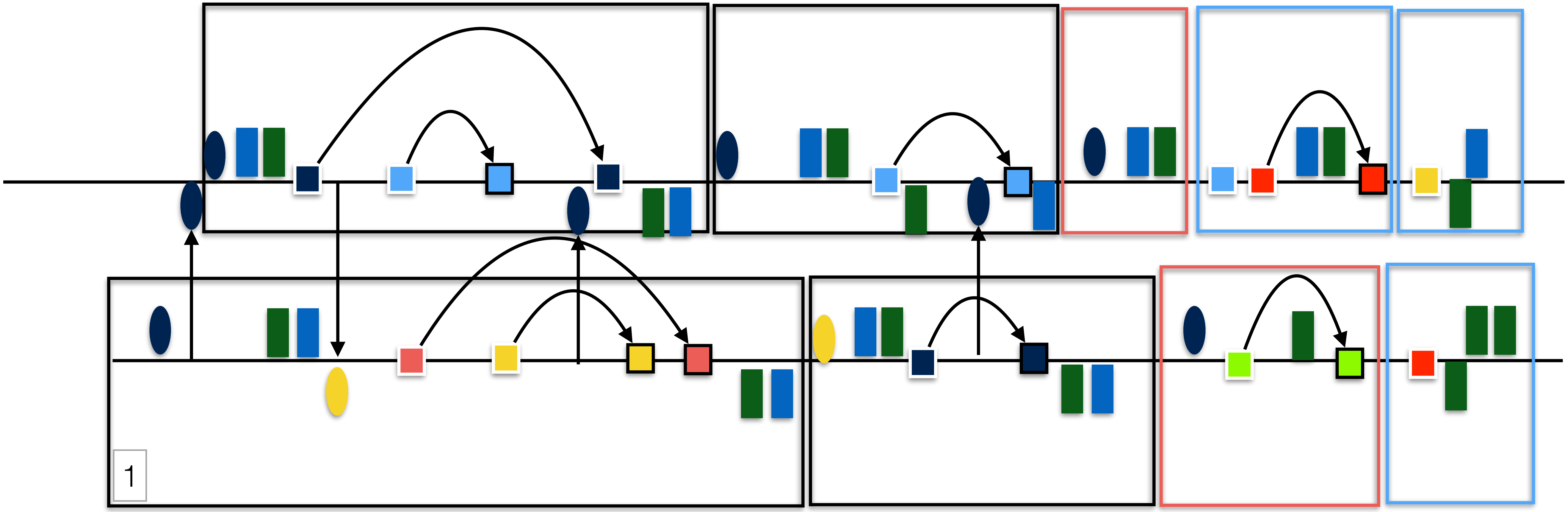
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



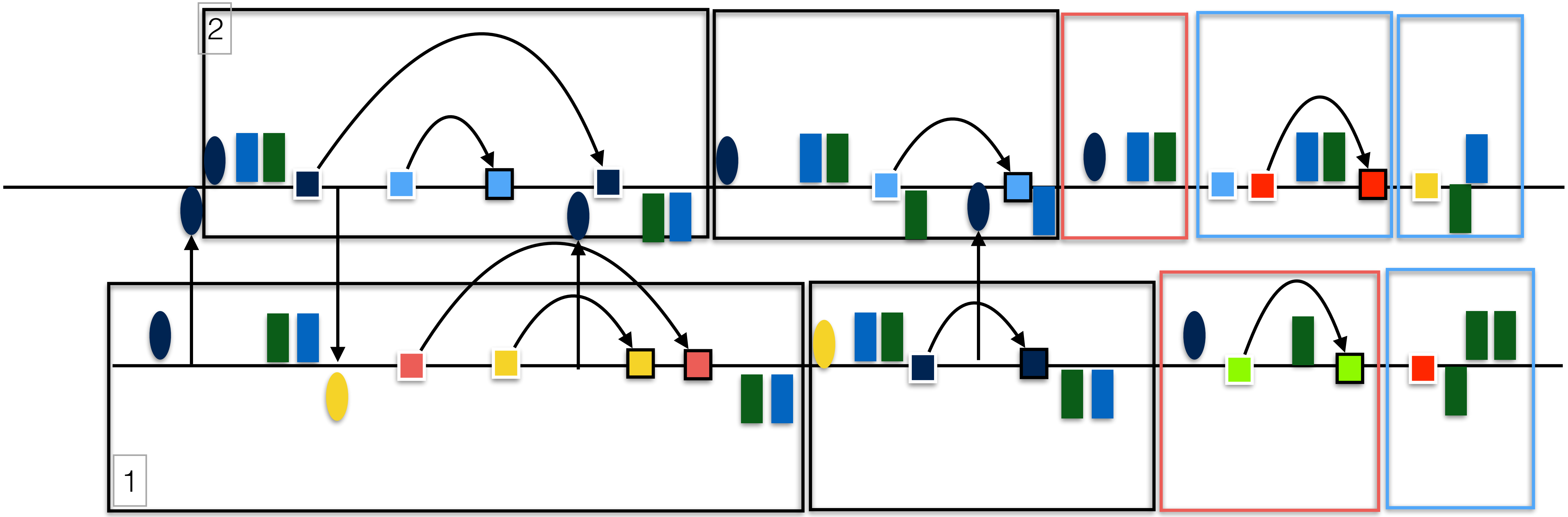
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



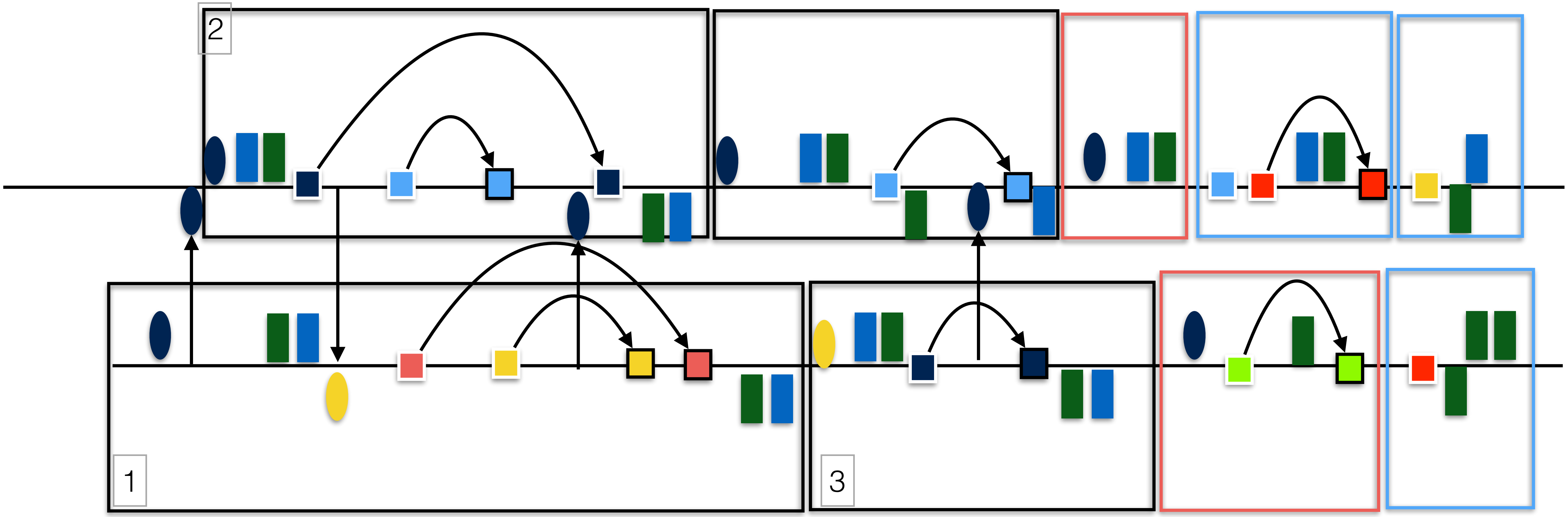
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



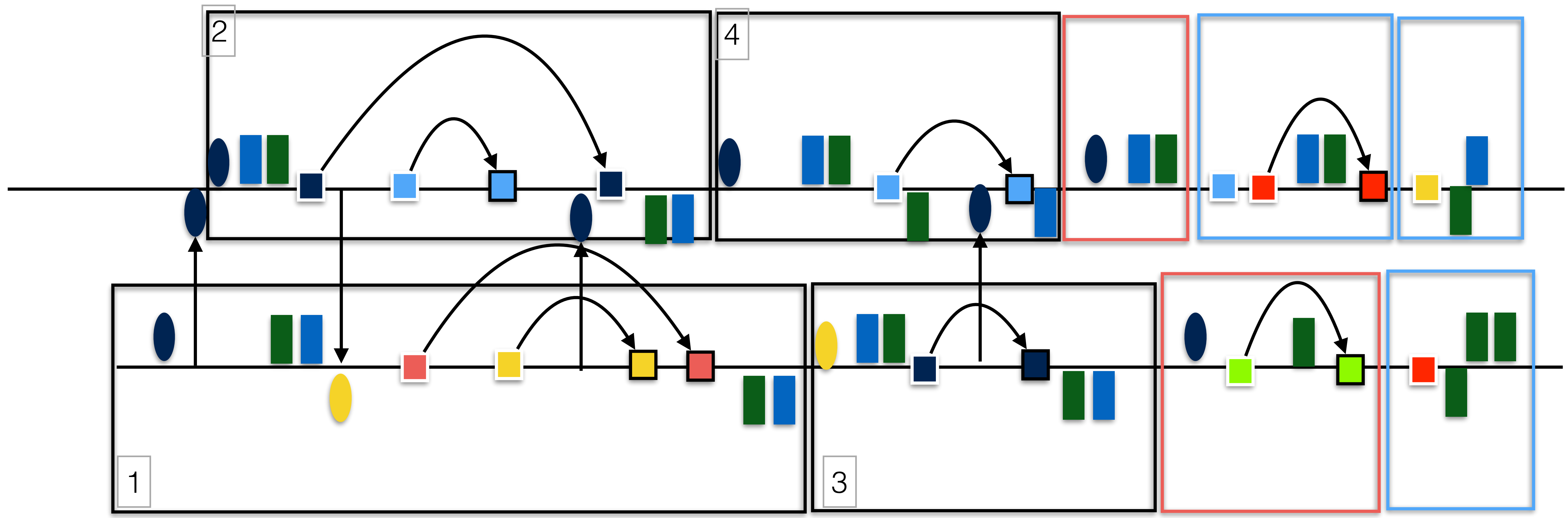
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



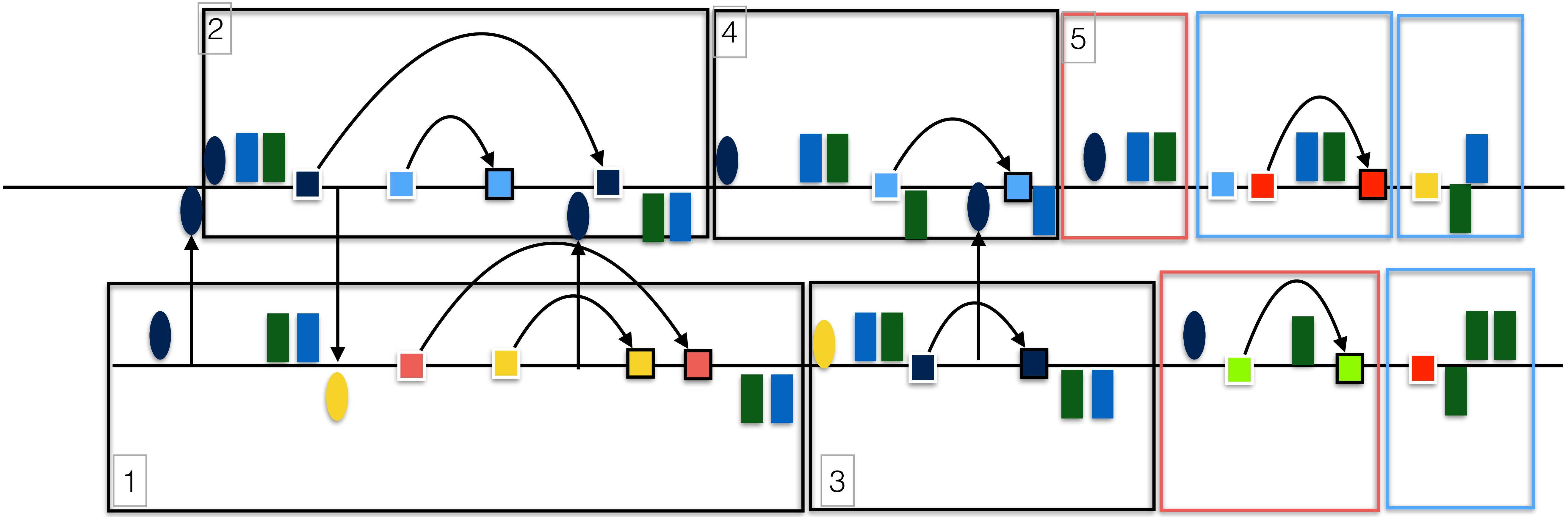
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



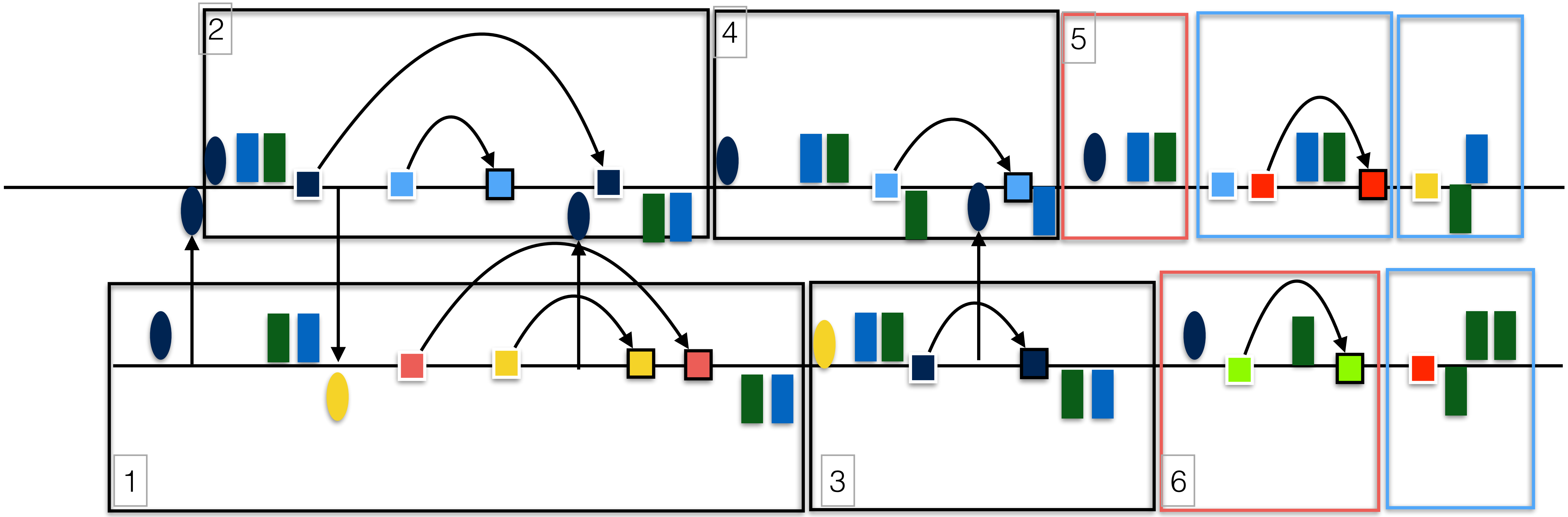
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



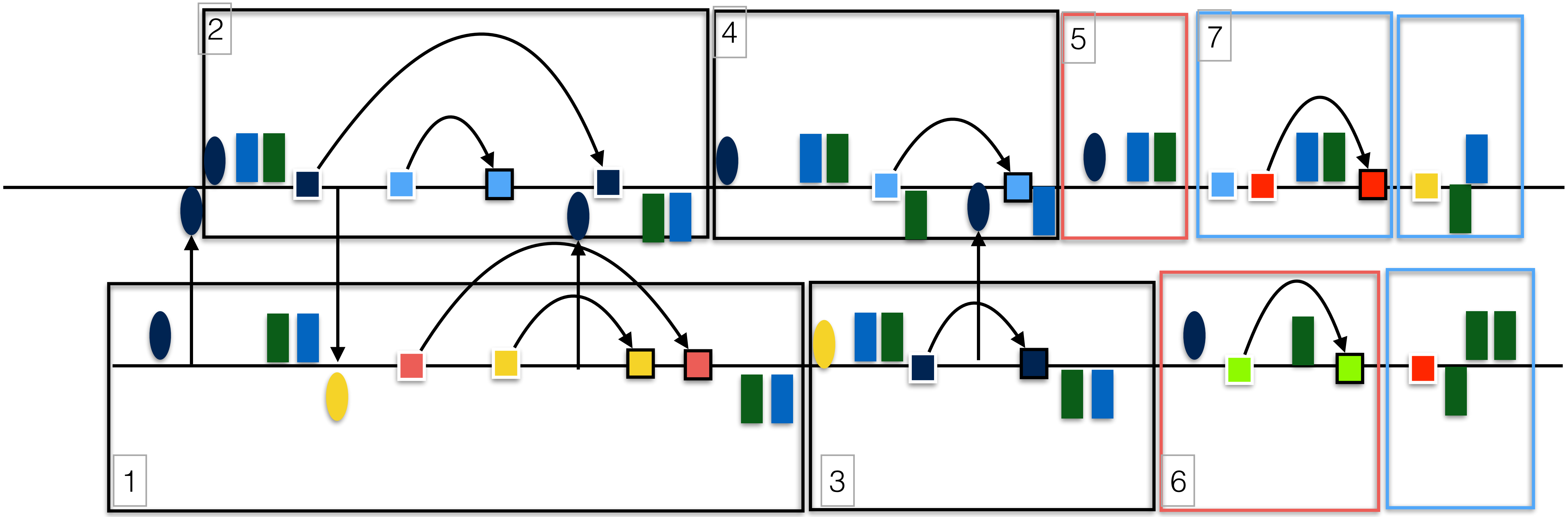
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



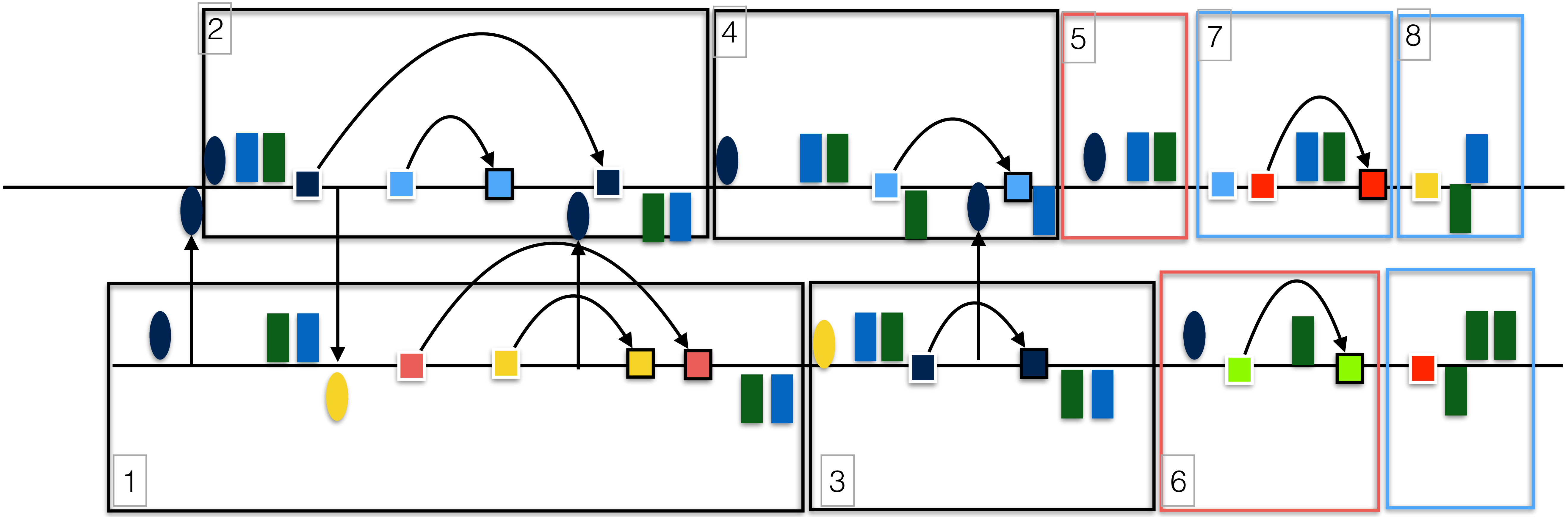
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



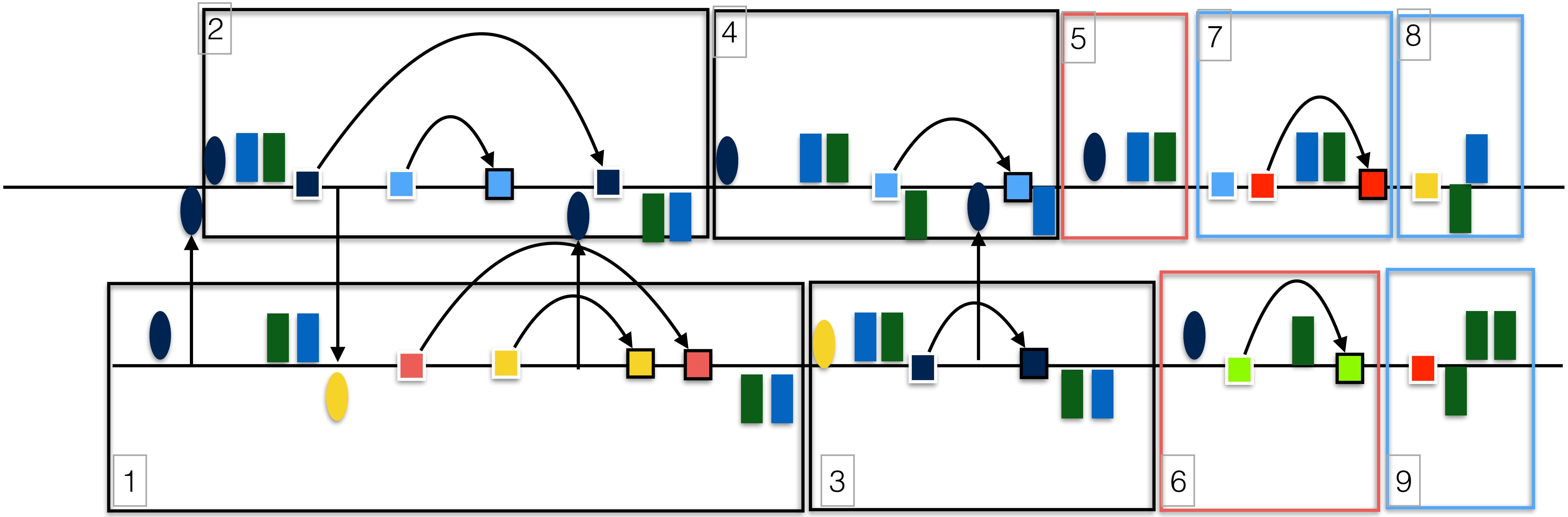
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



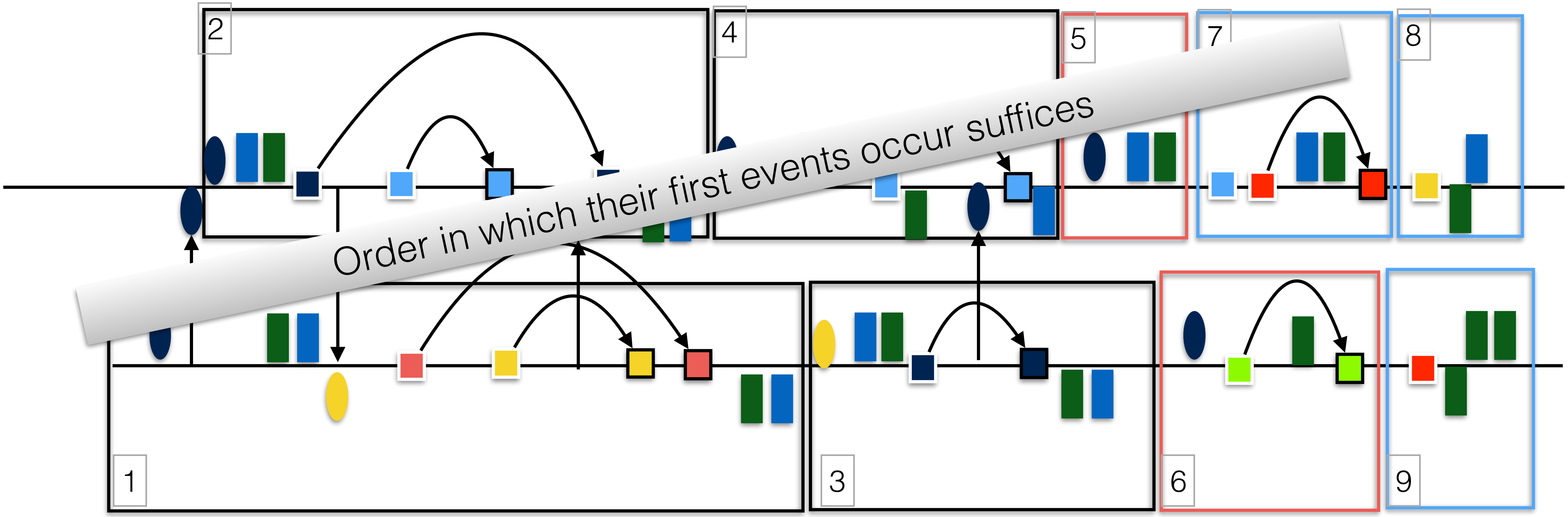
# Sequentialisation Lemma:

Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



# Sequentialisation Lemma:

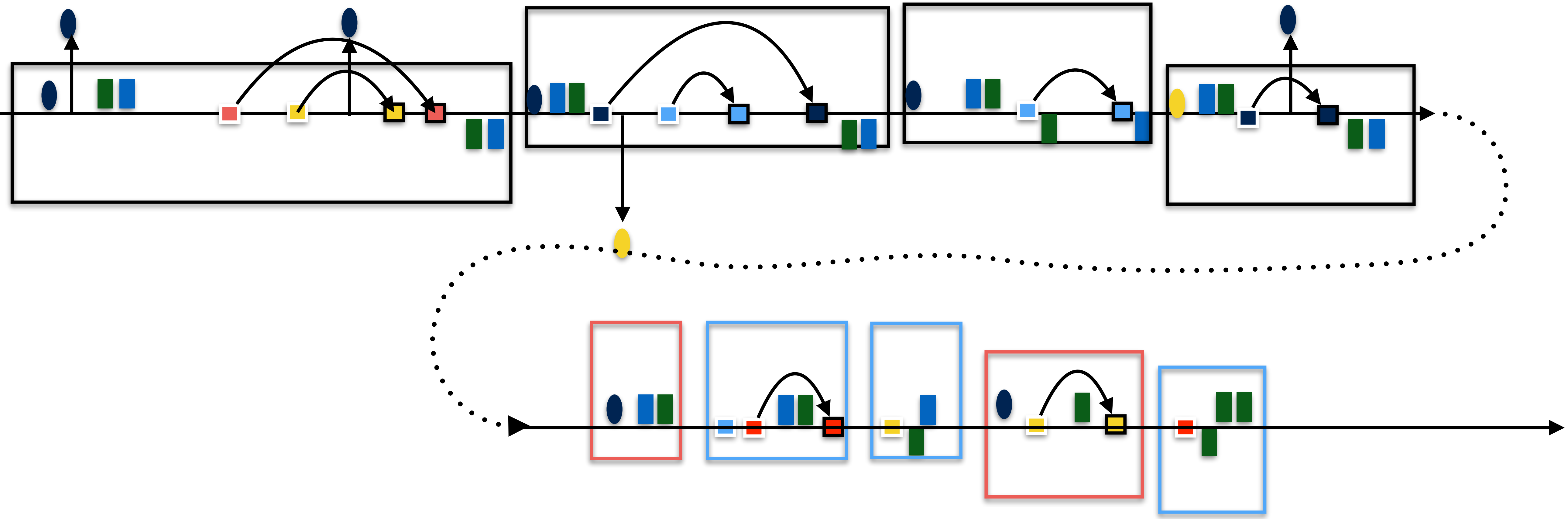
Every reachable configuration can be reached via a run that is a sequence of phases (of the different threads). That is, phases can be executed atomically.



Order in which their first events occur suffices

# N-threads to 1-thread:

(Guess and) Simulate the phases of all the threads using a single thread.



# N-threads to 1-thread:

- States have to be consistent across phases of a thread.
  - Maintain states

# N-threads to 1-thread:

- States have to be consistent across phases of a thread.
  - Maintain states
- Tasks picked for thread i have to be “available” at thread i.

# N-threads to 1-thread:

- States have to be consistent across phases of a thread.
    - Maintain states
- Tasks picked for thread i have to be “available” at thread i.
    - Easy. Use single multiset, but now tasks are tagged with the associated thread.

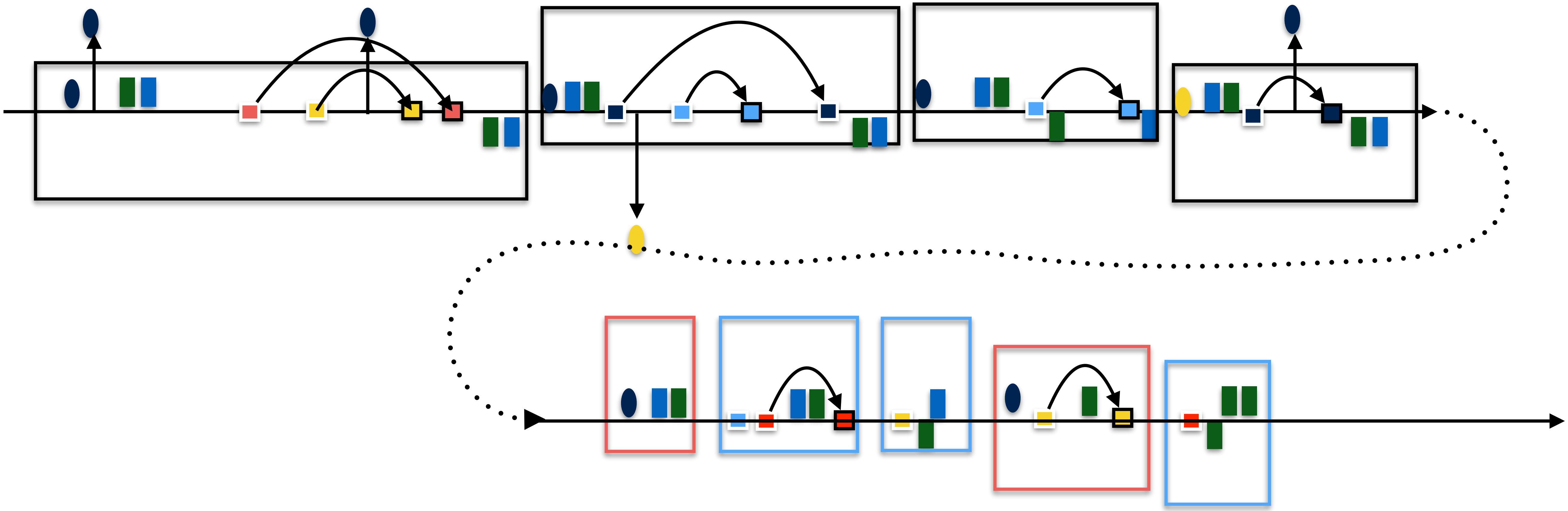
# N-threads to 1-thread:

- States have to be consistent across phases of a thread.
    - Maintain states
- Tasks picked for thread i have to be “available” at thread i.
    - Easy. Use single multiset, but now tasks are tagged with the associated thread.
- Locks should be handled correctly (taken only when available …)

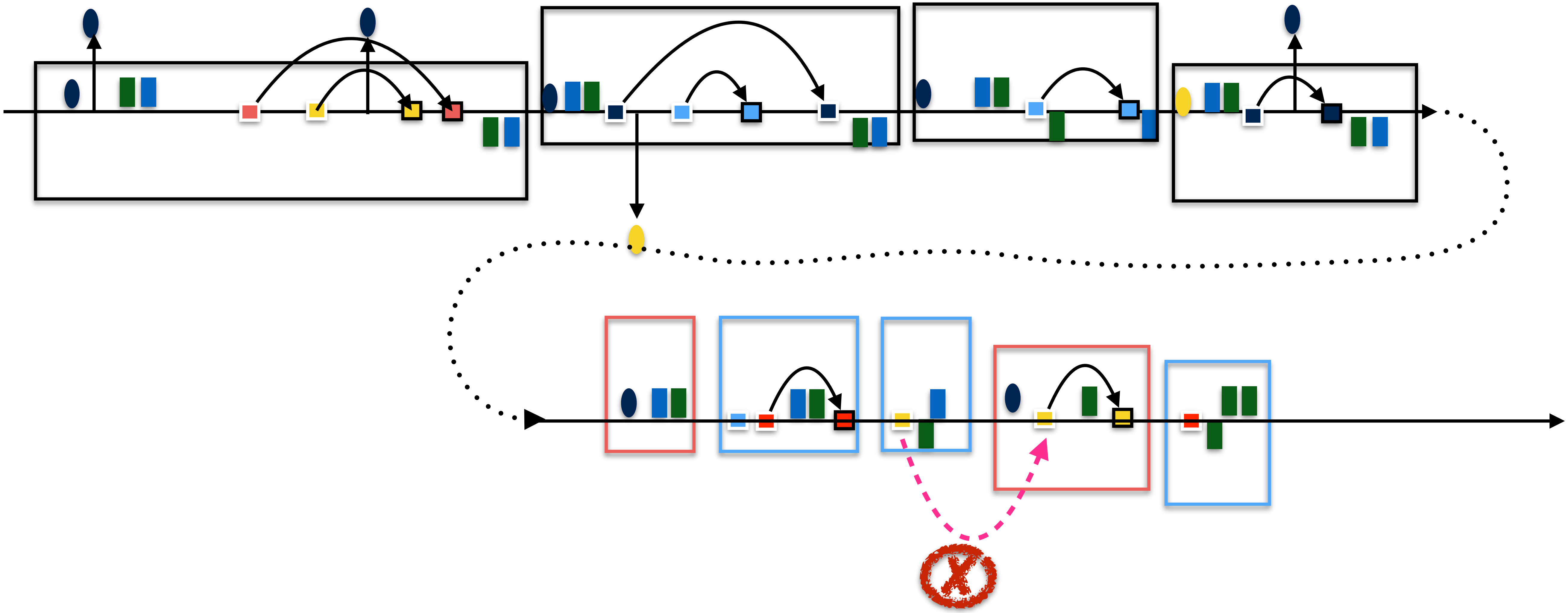
# N-threads to 1-thread:

- States have to be consistent across phases of a thread.
    - Maintain states
- Tasks picked for thread i have to be “available” at thread i.
    - Easy. Use single multiset, but now tasks are tagged with the associated thread.
- Locks should be handled correctly (taken only when available …)
- Handle multiple pushdown stores

# N-threads to 1-thread: locks

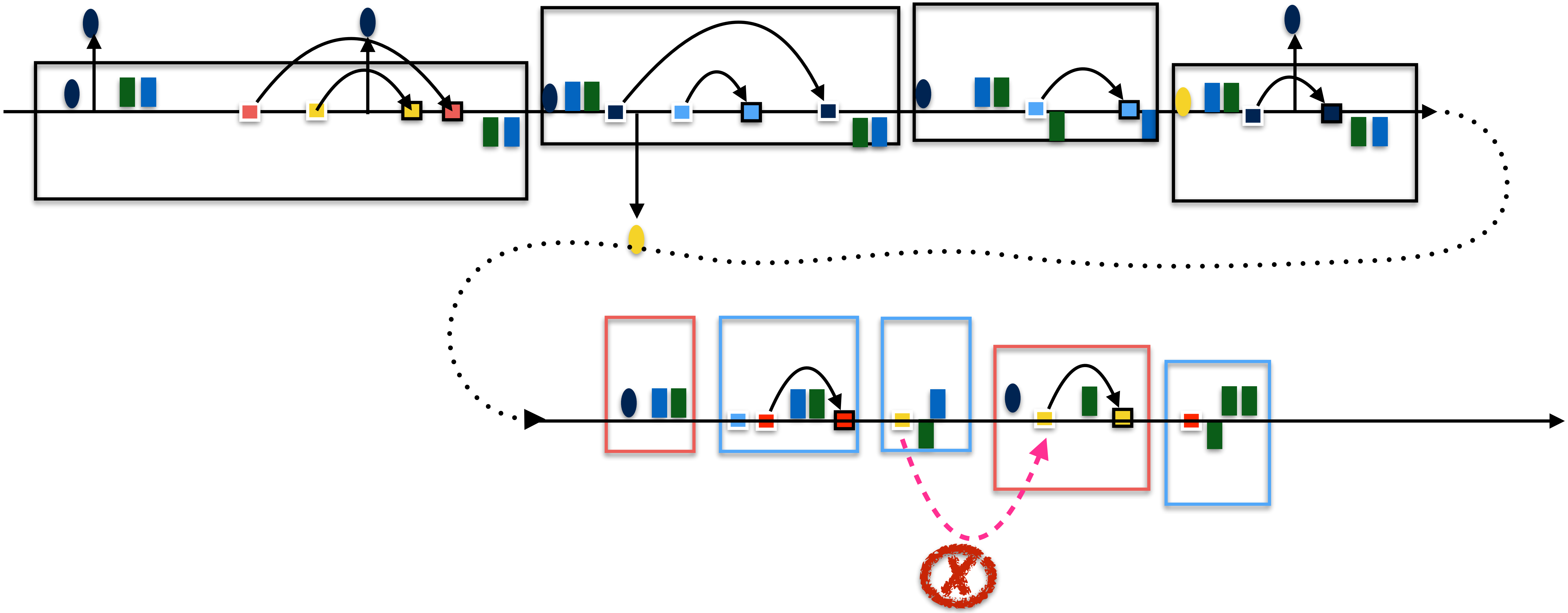


# N-threads to 1-thread: locks



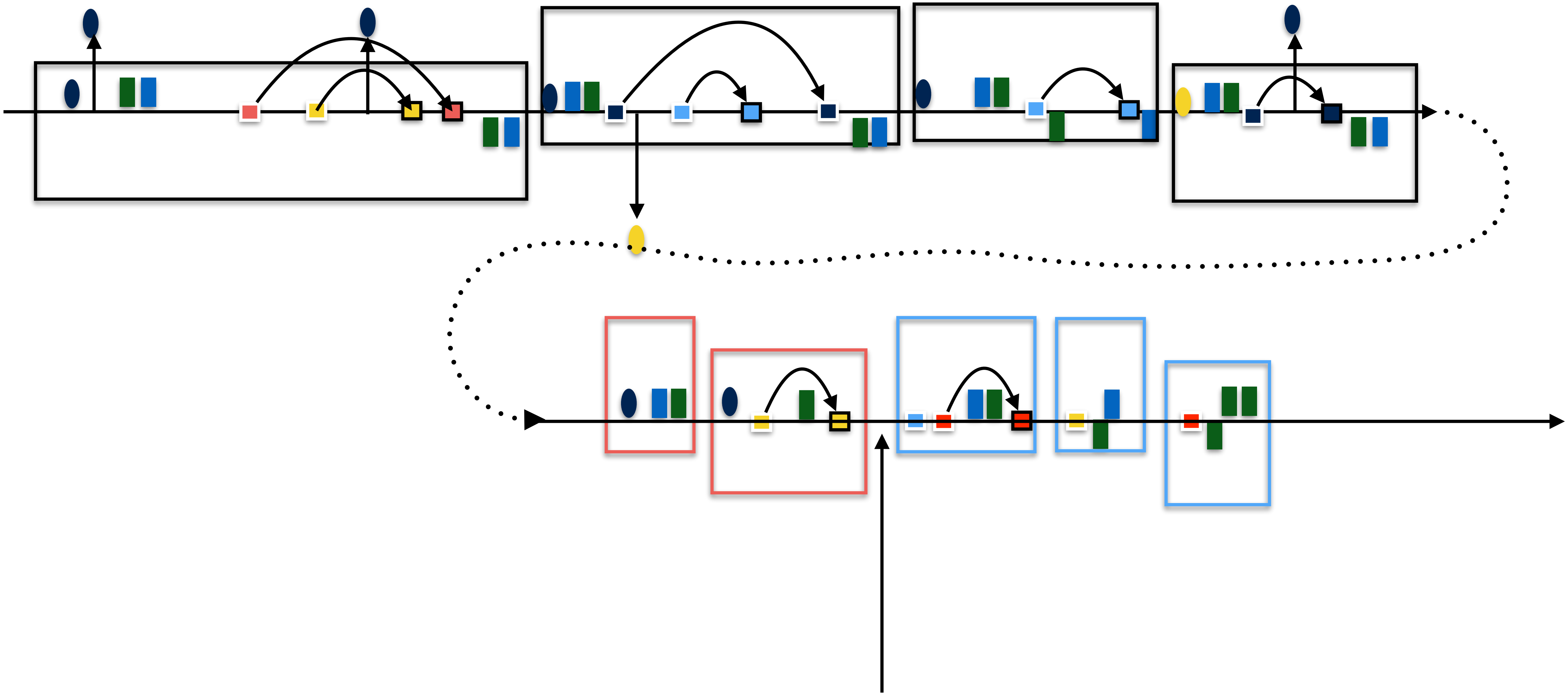
- Lock phases impose restrictions on availability of locks to future phases.

# N-threads to 1-thread: locks

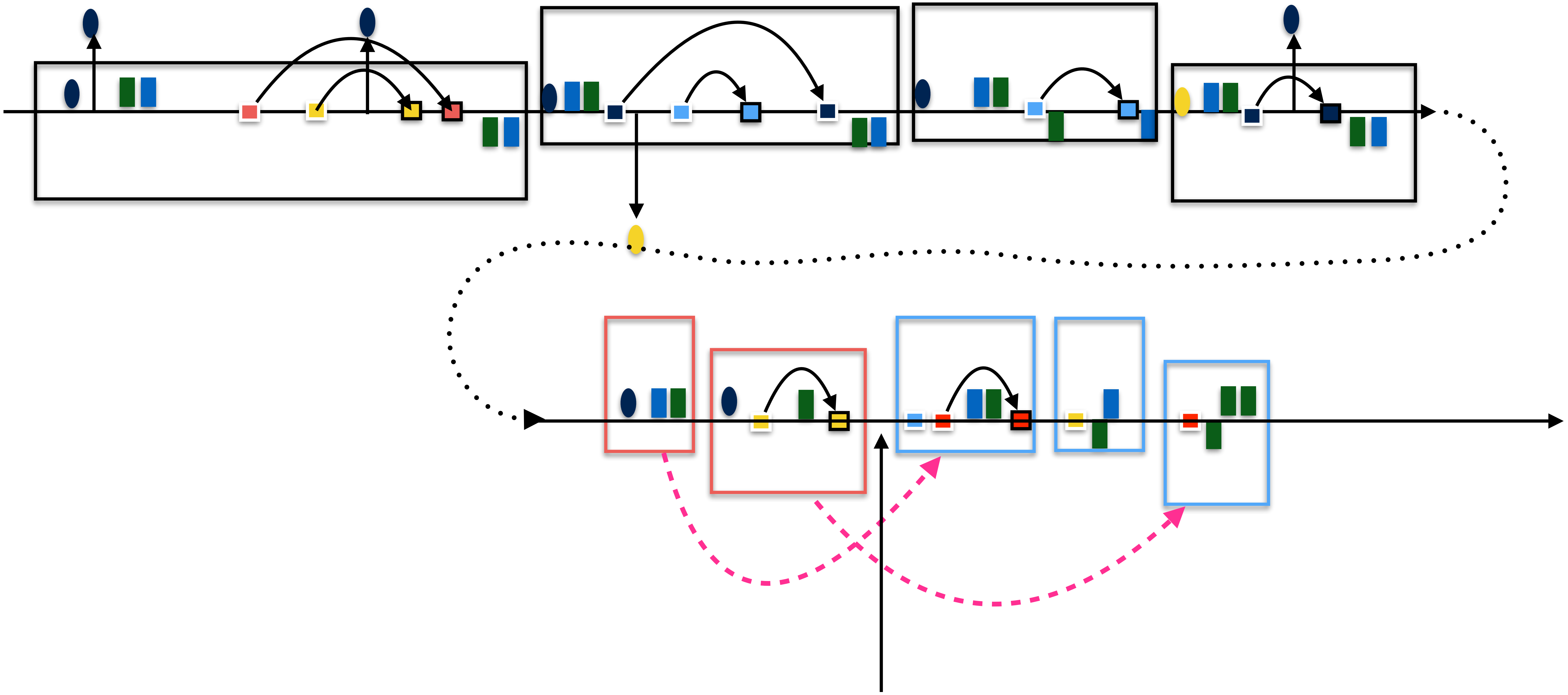


- Lock phases impose restrictions on availability of locks to future phases.
    - Maintain information on availability of locks

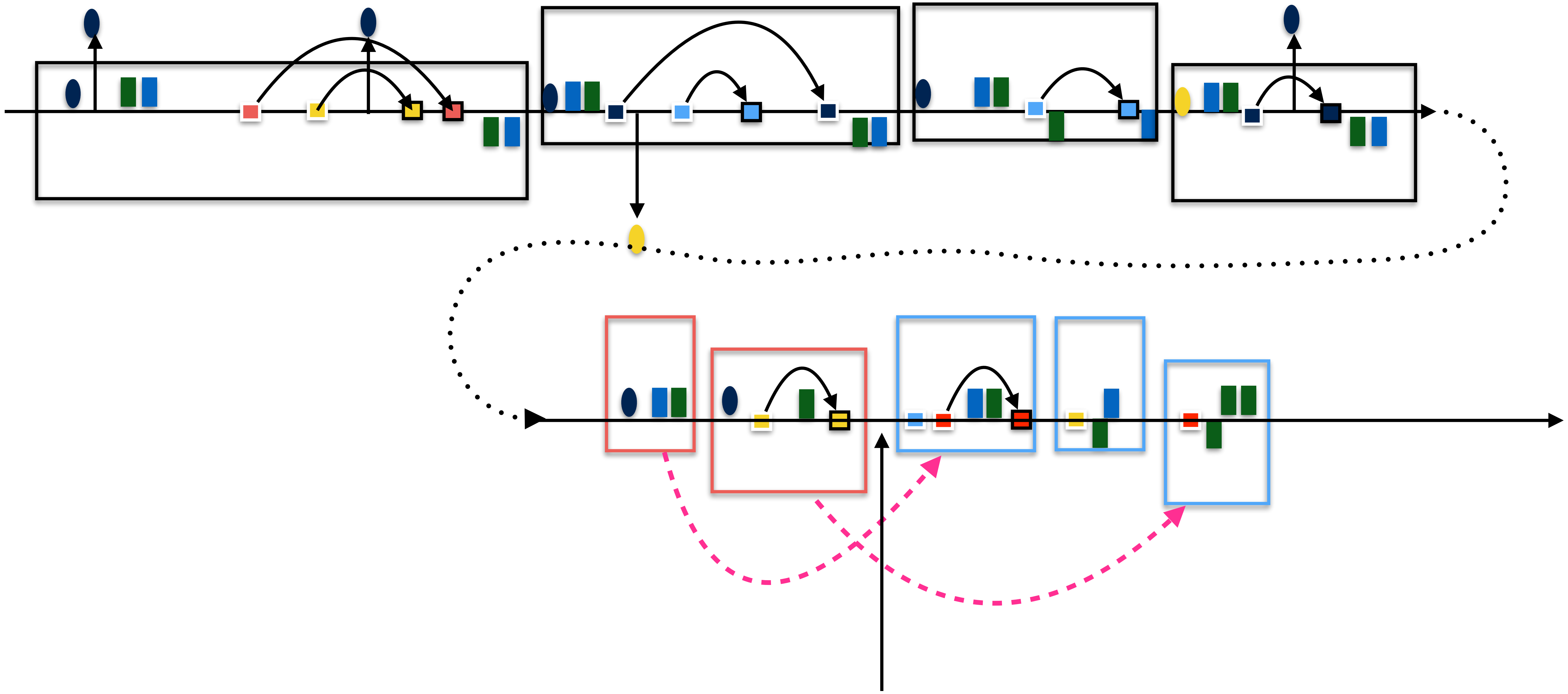
# N-threads to 1-thread: stacks



# N-threads to 1-thread: stacks



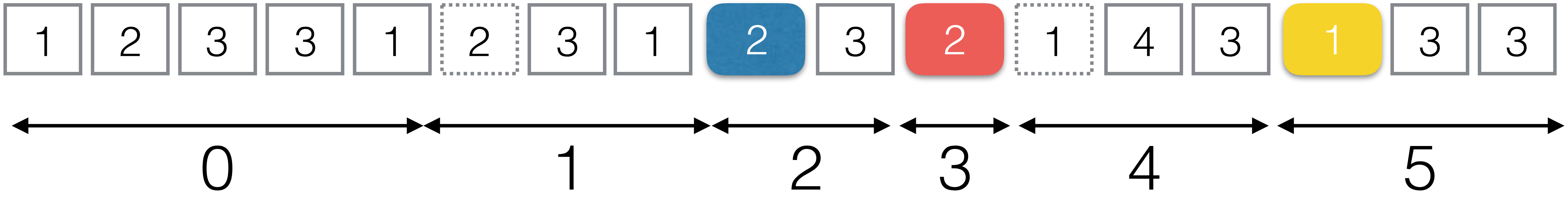
# N-threads to 1-thread: stacks



- Multiple stacks have to be maintained simultaneously.

# Segments of phases:

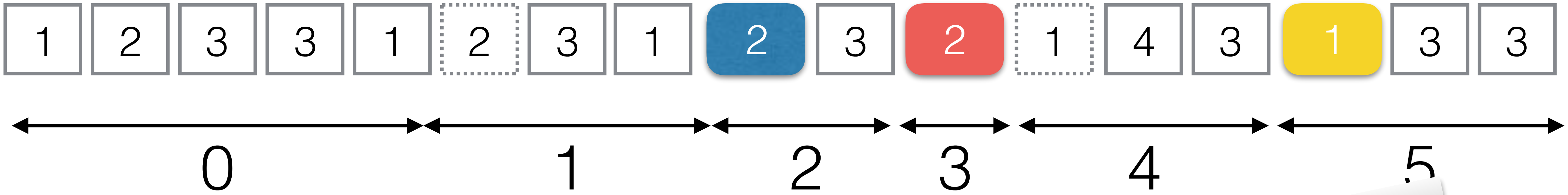
| | |
|---|---|
| i | A task phase of thread i |
| i | Boundary phase of thread i |
| i | A lock phase of thread i with lock 🔒 |



- Segment 0 — only task phases
- Segment i+1 — begins with boundary or lock phase, rest are task phases.

# Segments of phases:

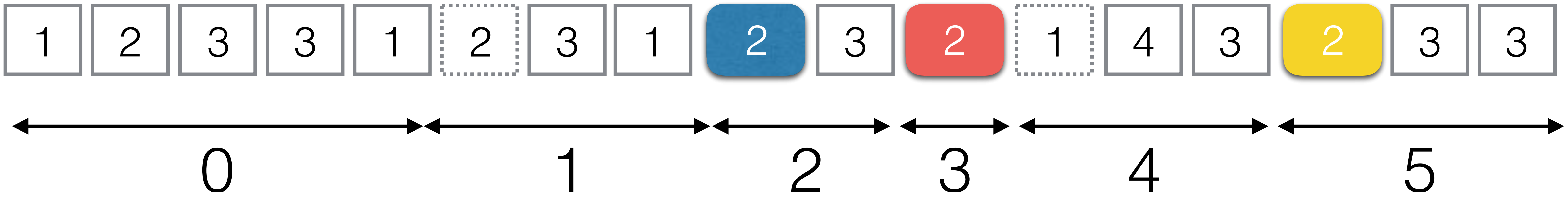
| | |
|---|---|
| ☐ i | A task phase of thread i |
| ⬚ i | Boundary phase of thread i |
| ■ i | A lock phase of thread i with lock 🔒 |



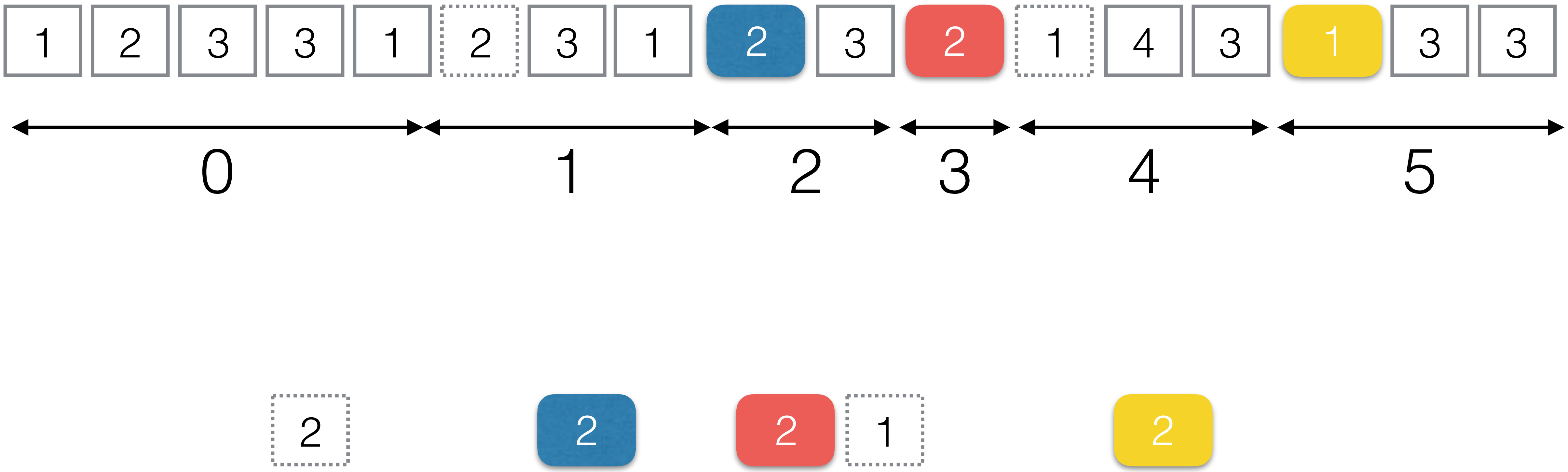
Number of segments is bounded by locks + threads

- Segment 0 — only task phases
- Segment i begins with boundary or lock phase, rest are task phases.

# Guiding Sequences:

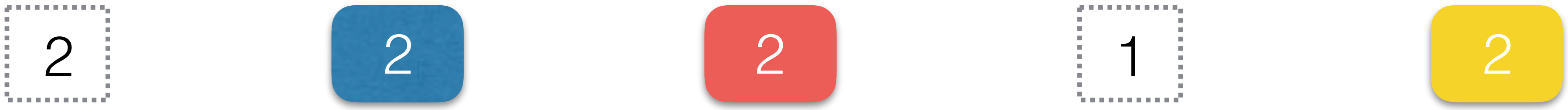


# Guiding Sequences:



A sequence identifying the first element of each segment

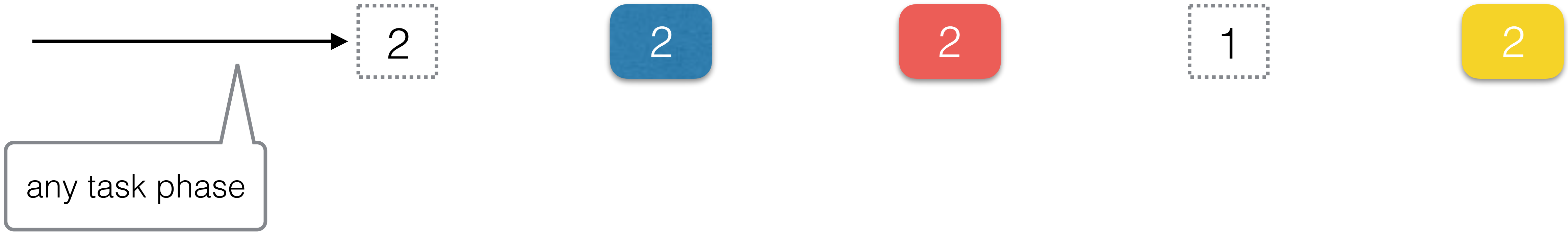
# Simulation with a single stack:



2 2 2 1 2

Seg No = 0

# Simulation with a single stack:



2

2

2

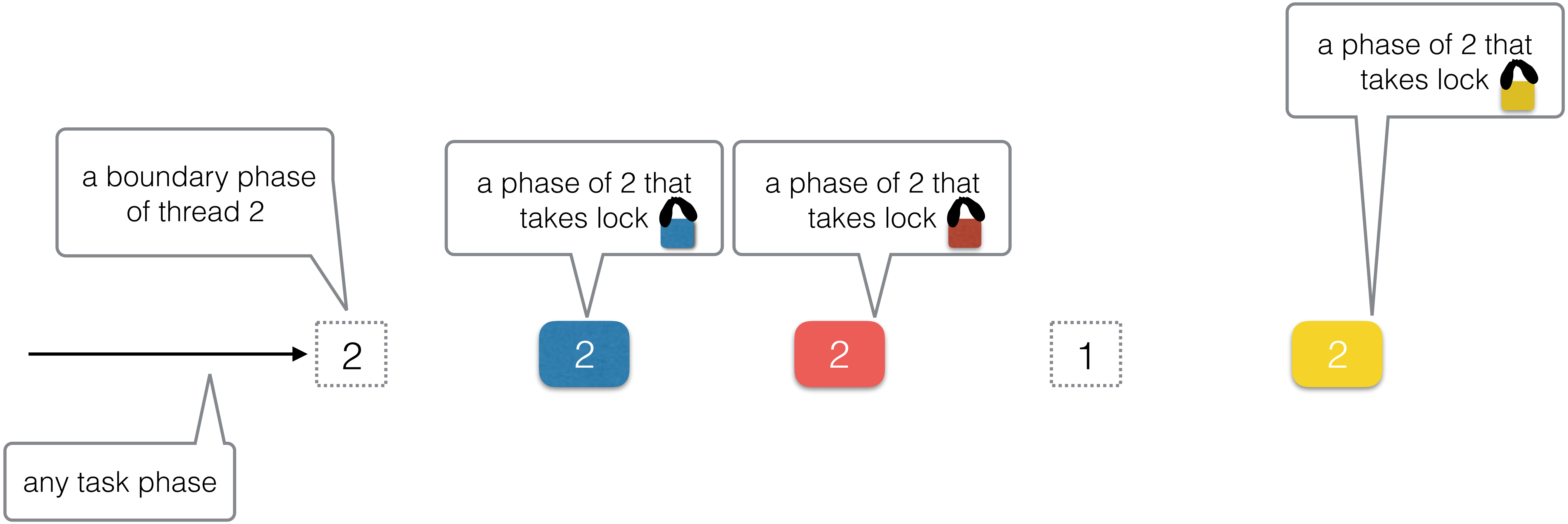
1

2

any task phase

Seg No = 0

# Simulation with a single stack:



a boundary phase of thread 2

any task phase

2

a phase of 2 that takes lock

2

a phase of 2 that takes lock

2

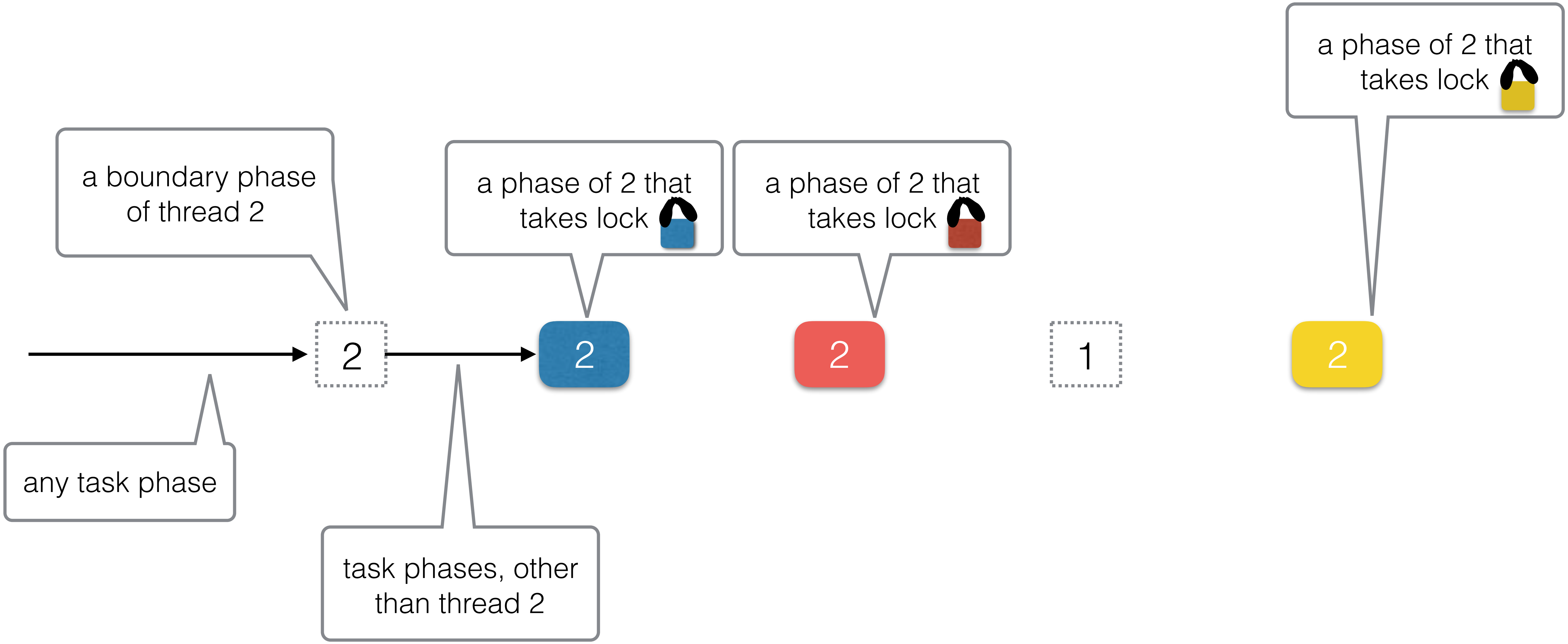
1

a phase of 2 that takes lock

2

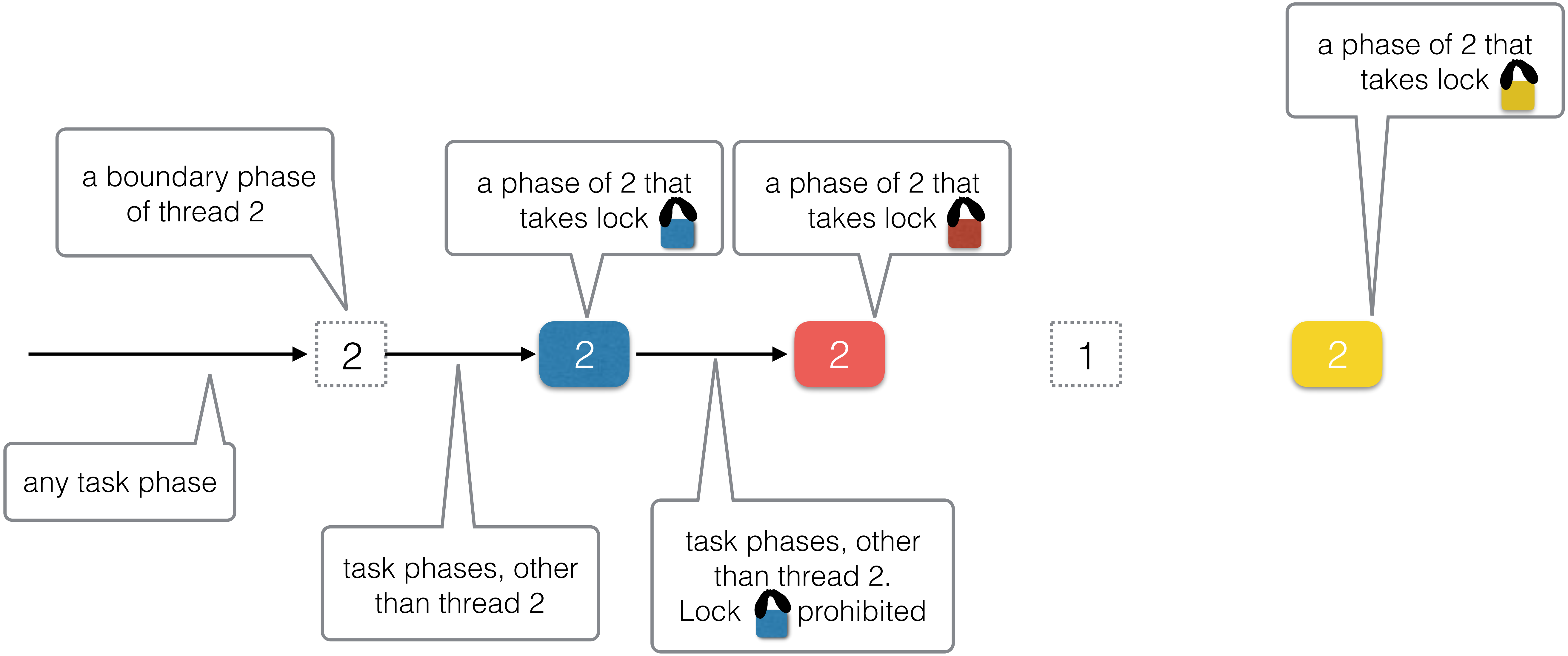
Seg No = 1

# Simulation with a single stack:


a boundary phase of thread 2
a phase of 2 that takes lock
a phase of 2 that takes lock
a phase of 2 that takes lock
2
2
2
1
2
any task phase
task phases, other than thread 2

Seg No = 1

# Simulation with a single stack:



a boundary phase of thread 2

a phase of 2 that takes lock 🔒

a phase of 2 that takes lock 🔒

a phase of 2 that takes lock 🔒

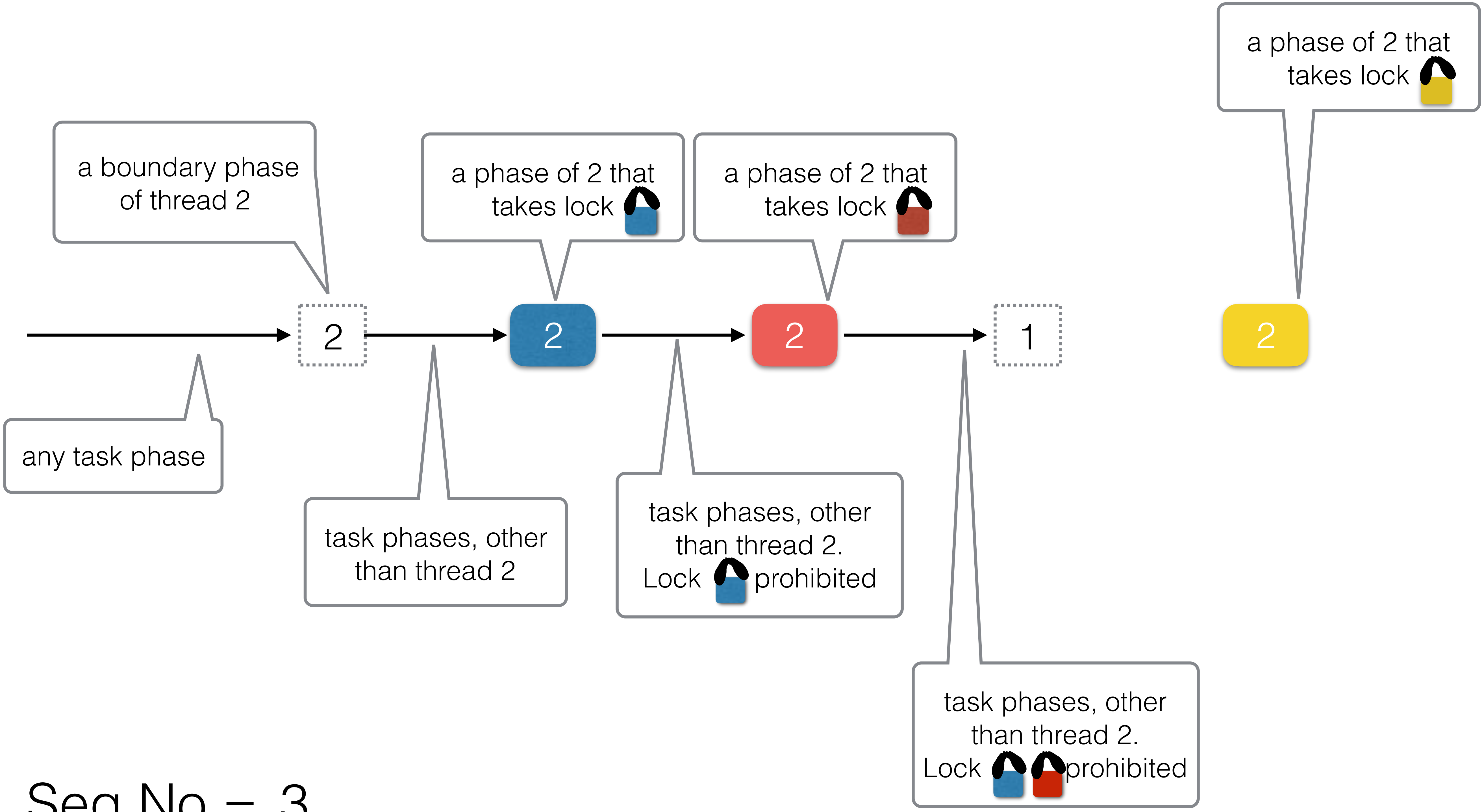
any task phase

task phases, other than thread 2

task phases, other than thread 2. Lock 🔒 prohibited

Seg No = 2

# Simulation with a single stack:



a boundary phase of thread 2

a phase of 2 that takes lock 🔒

a phase of 2 that takes lock 🔒

a phase of 2 that takes lock 🔒

any task phase

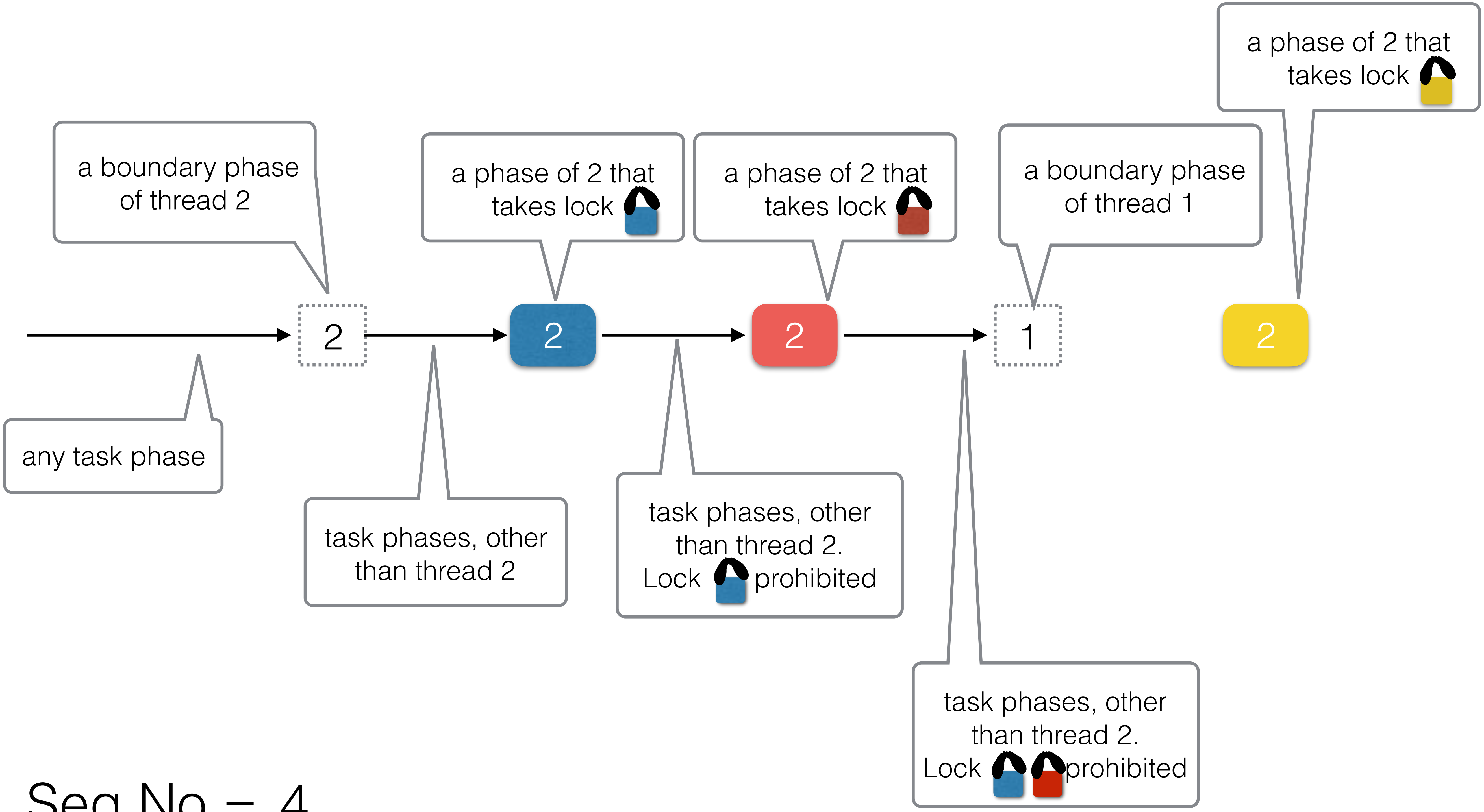
task phases, other than thread 2

task phases, other than thread 2. Lock 🔒 prohibited

task phases, other than thread 2. Lock 🔒 🔒 prohibited

Seg No = 3

# Simulation with a single stack:



a boundary phase of thread 2

a phase of 2 that takes lock 🟦

a phase of 2 that takes lock 🟥

a boundary phase of thread 1

a phase of 2 that takes lock 🟨

2

2

2

1

2

any task phase

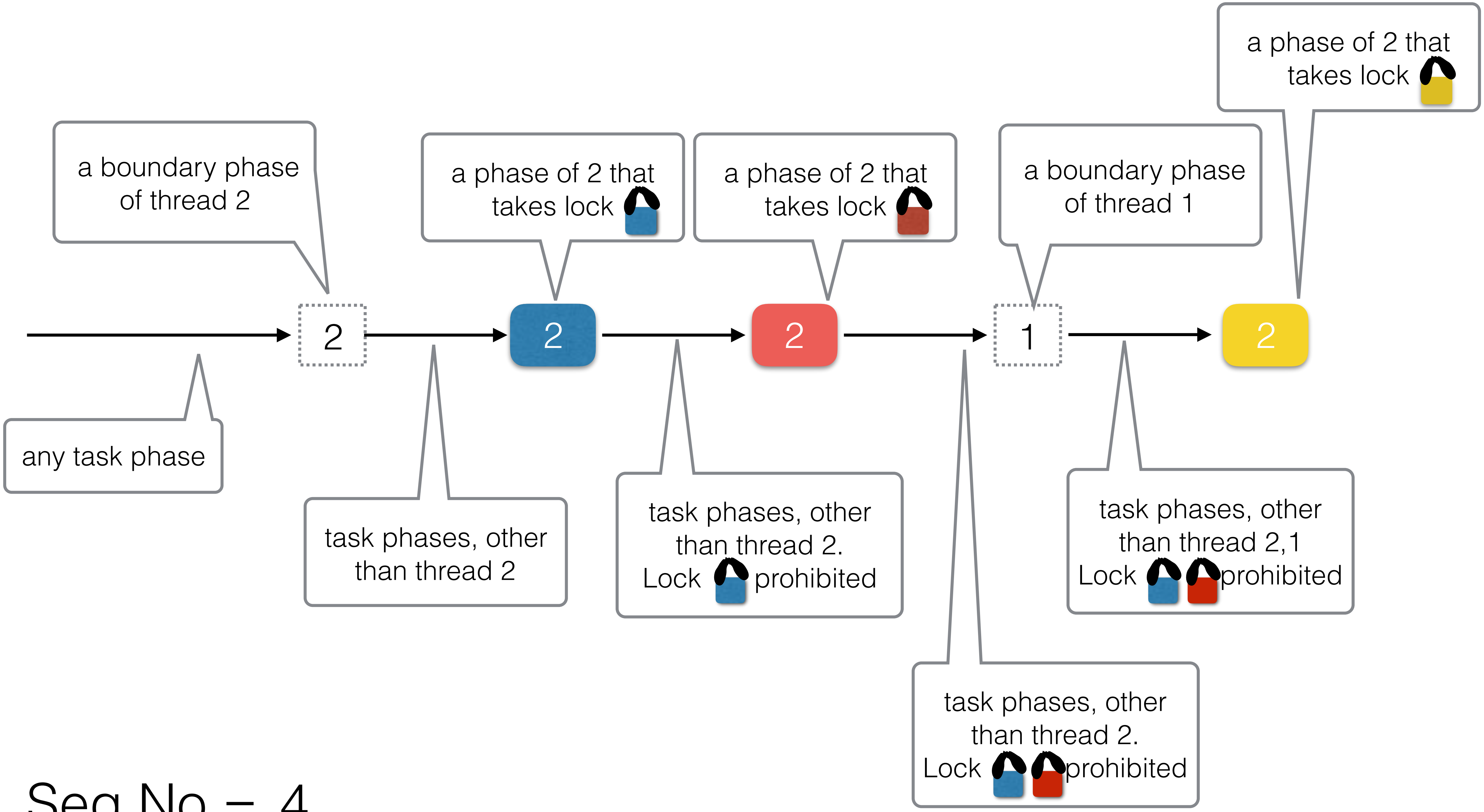
task phases, other than thread 2

task phases, other than thread 2. Lock 🟦 prohibited

task phases, other than thread 2. Lock 🟦 🟥 prohibited

Seg No = 4

# Simulation with a single stack:



a boundary phase of thread 2

a phase of 2 that takes lock 🔒

a phase of 2 that takes lock 🔒

a boundary phase of thread 1

a phase of 2 that takes lock 🔒

any task phase

task phases, other than thread 2

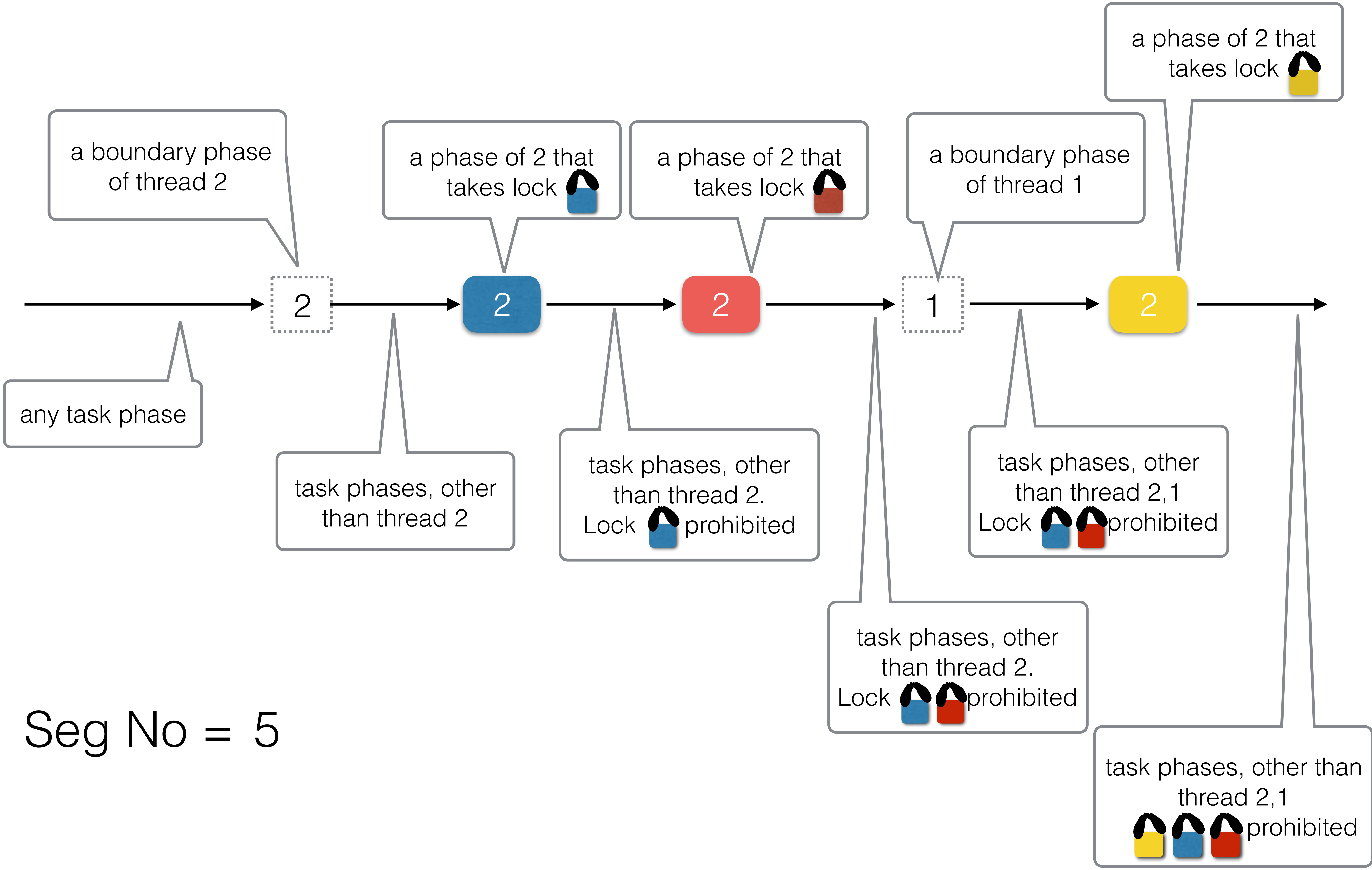
task phases, other than thread 2.
Lock 🔒 prohibited

task phases, other than thread 2.
Lock 🔒🔒prohibited

task phases, other than thread 2,1
Lock 🔒🔒prohibited

Seg No = 4

# Simulation with a single stack:



a boundary phase of thread 2

a phase of 2 that takes lock 🟦

a phase of 2 that takes lock 🟥

a boundary phase of thread 1

a phase of 2 that takes lock 🟨

any task phase

task phases, other than thread 2

task phases, other than thread 2. Lock 🟦 prohibited

task phases, other than thread 2. Lock 🟦 🟥 prohibited

task phases, other than thread 2,1 Lock 🟦 🟥 prohibited

task phases, other than thread 2,1 🟨 🟦 🟥 prohibited

Seg No = 5

# Complexity:

- For a given guiding sequence
  - Exponential blow up due to product of state spaces

# Complexity:

- For a given guiding sequence
  - Exponential blow up due to product of state spaces

Maintain the local states in the multiset

# Complexity:

- For a given guiding sequence
  - Exponential blow up due to product of state spaces

Maintain the local states in the multiset

Reachability via runs consistent with a given guiding sequence reduces to a polynomially larger 1-Thread system.

# Complexity …

- For a given guiding sequence

- There are only exponentially many guiding sequences

# Complexity ...

- For a given guiding sequence

Reachability via runs consistent with a given guiding sequence is in EXPSPACE.

- There are only exponentially many guiding sequences

Theorem: Reachability for Asynchronous programs with locks under well-nested, task locking is EXPSPACE-Complete

# Complexity: underapproximation

- What if we also want to verify that the system uses nested locking?
  - Exponential blow up due to set of locks to be maintained.
  - Locks are accessed when the stack is not empty, so can’t be simply moved to the multi-set.
  - Using Parikh’s theorem transform this into FA with multi-sets with 2-EXP number of states, but same multi-set alphabet as in the input.
  - Treat as a VASS with 2-EXP number of states and polynomial number of places.
  - Yen-Rosier show that coverability for VASS can be solved space logarithmic in the number of states and exponential in the number of places.

# Stateless task scheduling:

Each thread may schedule a new task only from a fixed local state.

- Tasks cannot “communicate” via local state of threads
- A thread just schedules and runs  tasks.

# Stateless task scheduling:

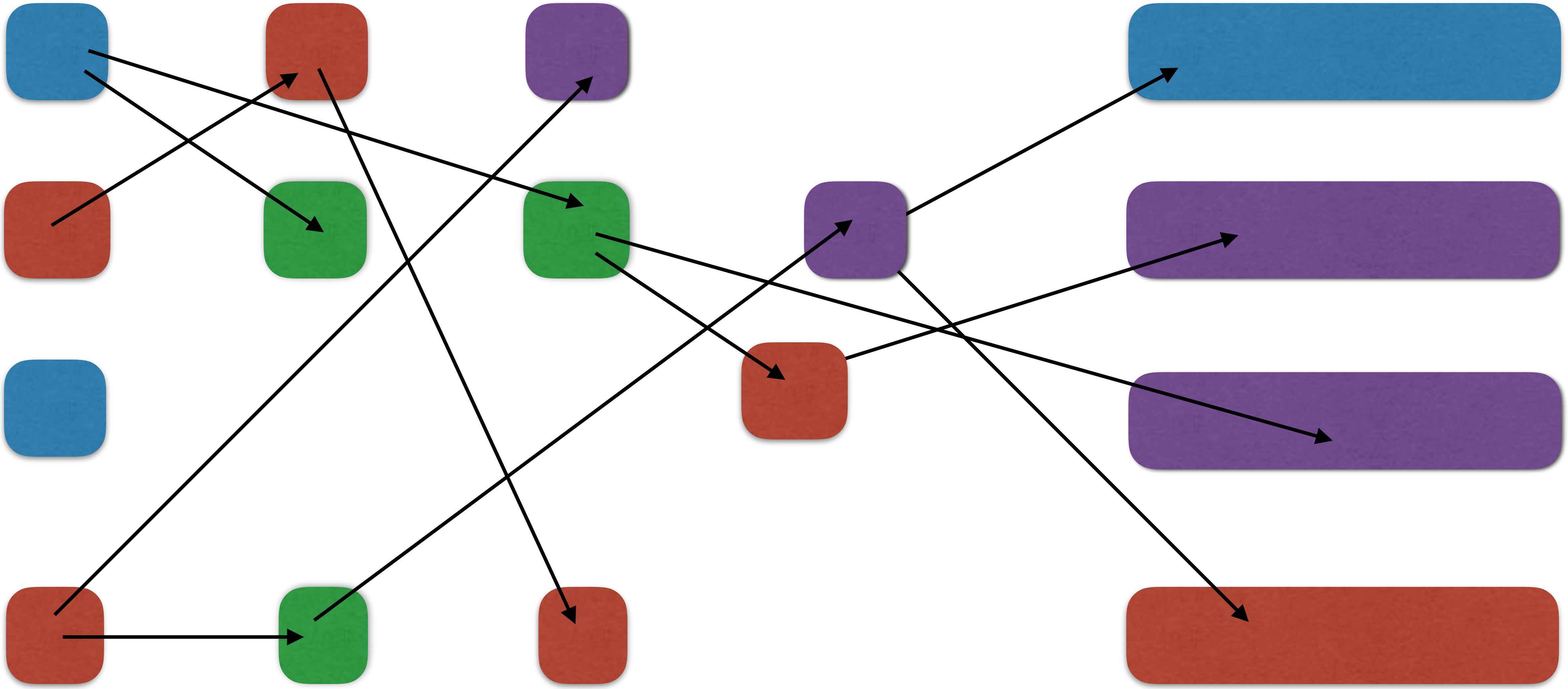
Theorem: Reachability for Asynchronous programs with locks under state-less scheduling,  well-nested locks and  task locking is NP-Complete

# Stateless task scheduling:

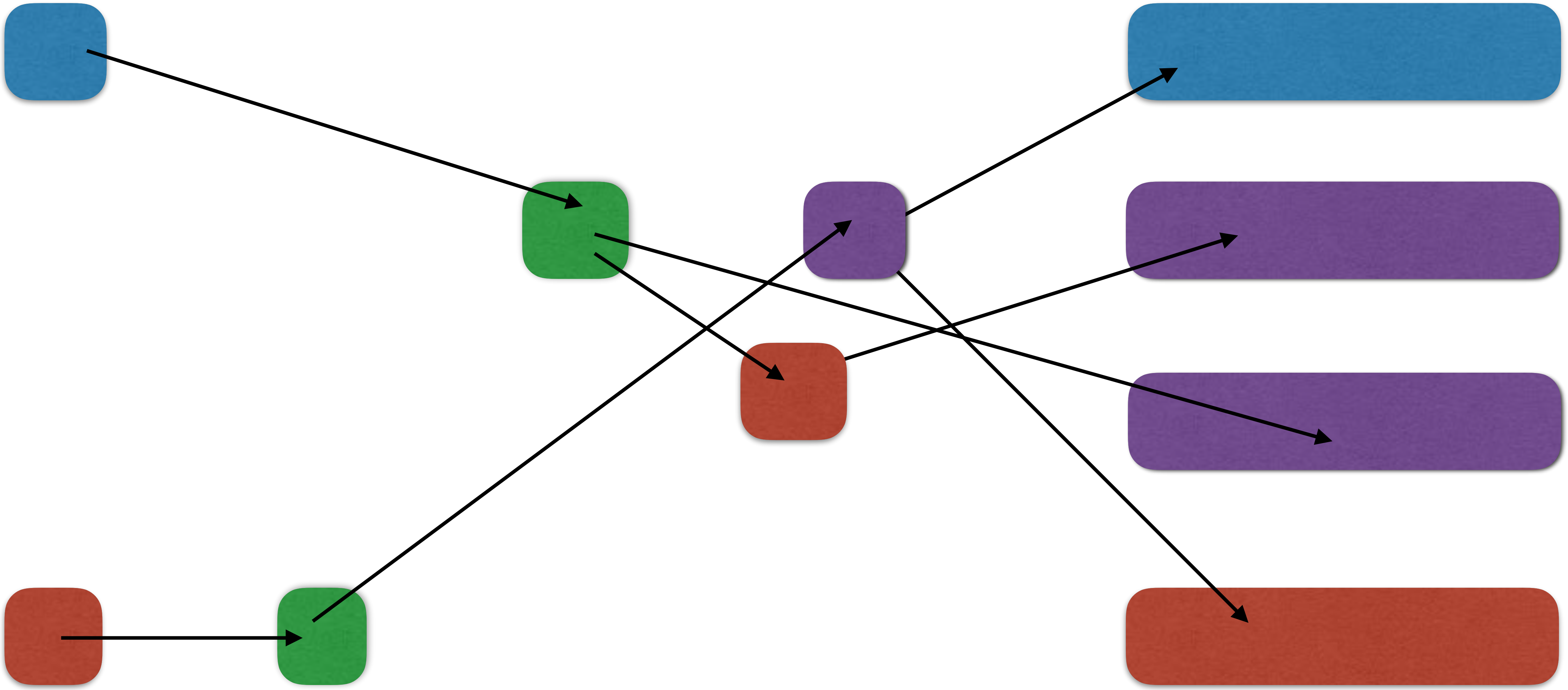
Theorem: Reachability for Asynchronous programs with locks under state-less scheduling, well-nested locks and task locking is NP-Complete

- A polynomial bound on the number of tasks that need to be scheduled to reach any (reachable) state.

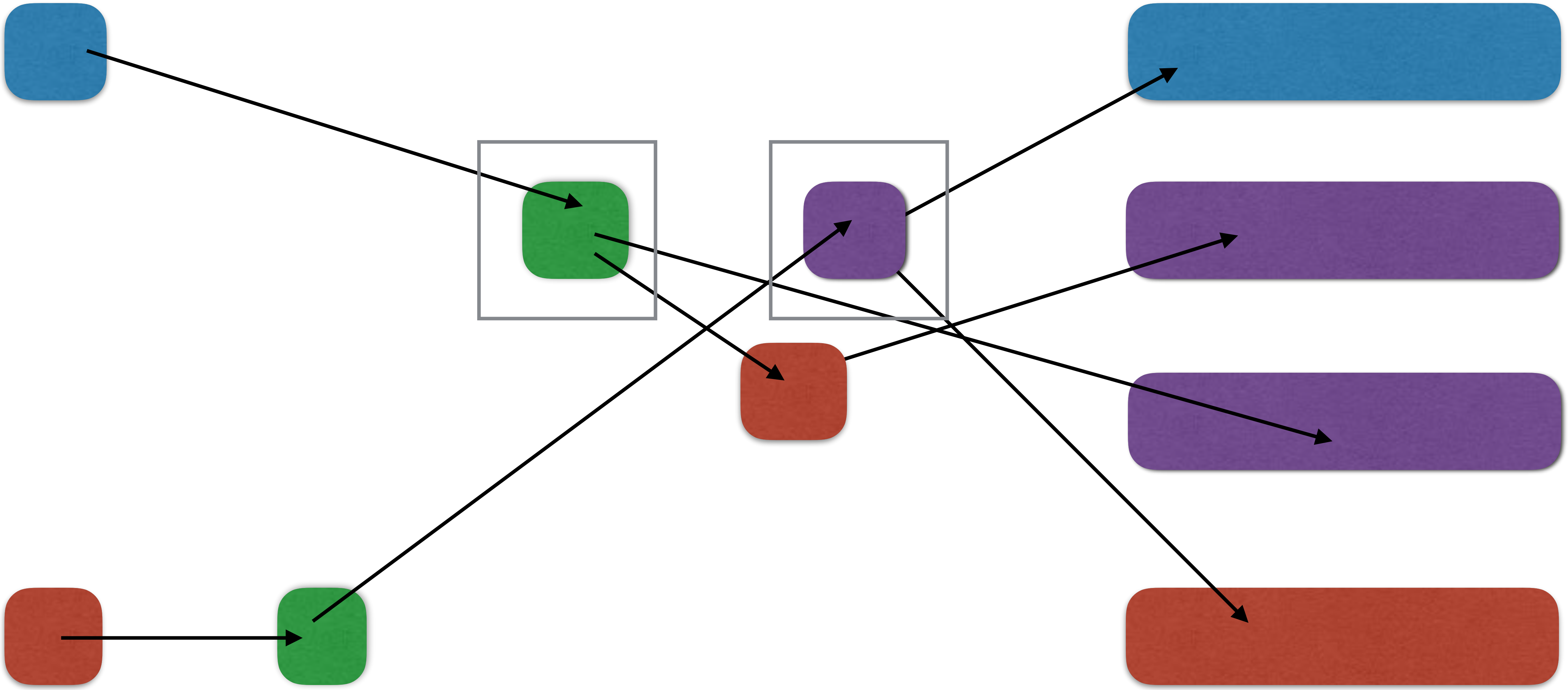
# Bounding the number of tasks



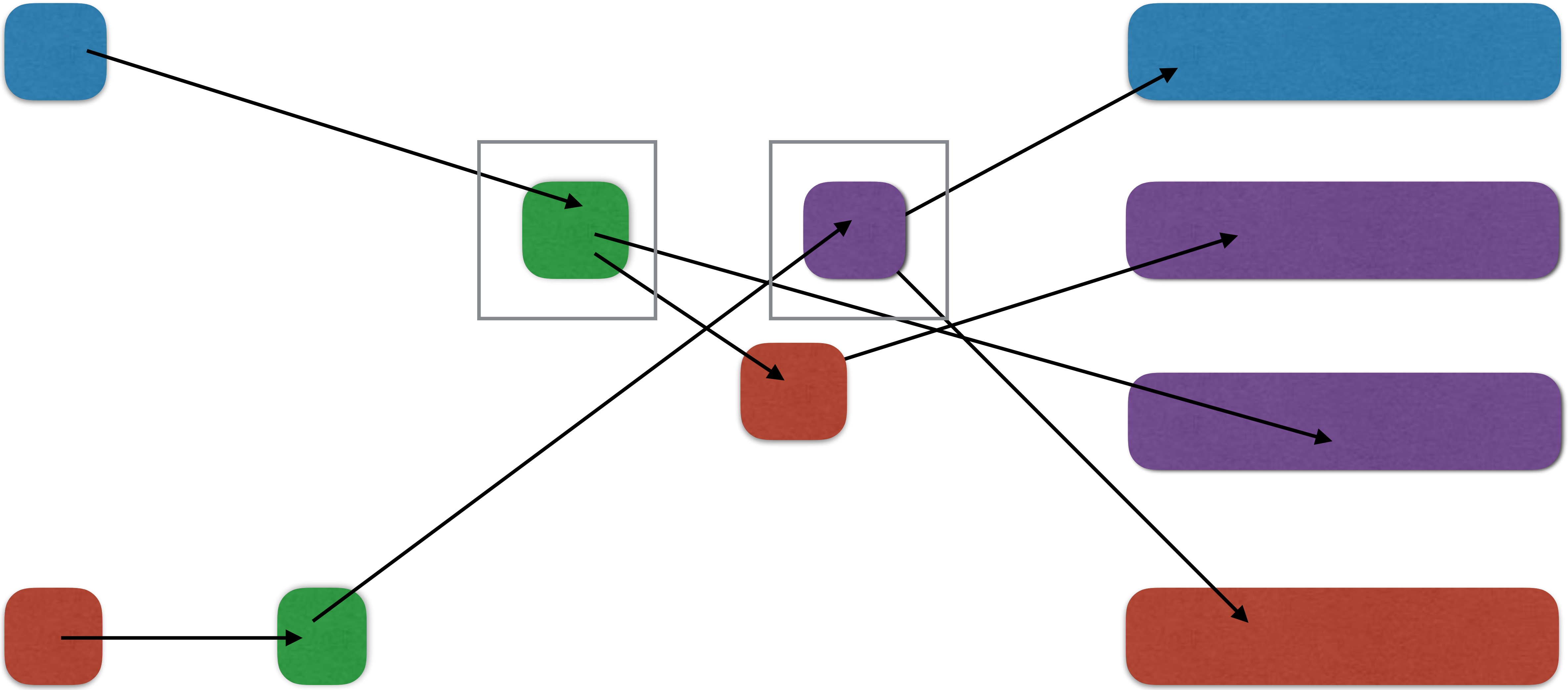
# Bounding the number of tasks



# Bounding the number of tasks

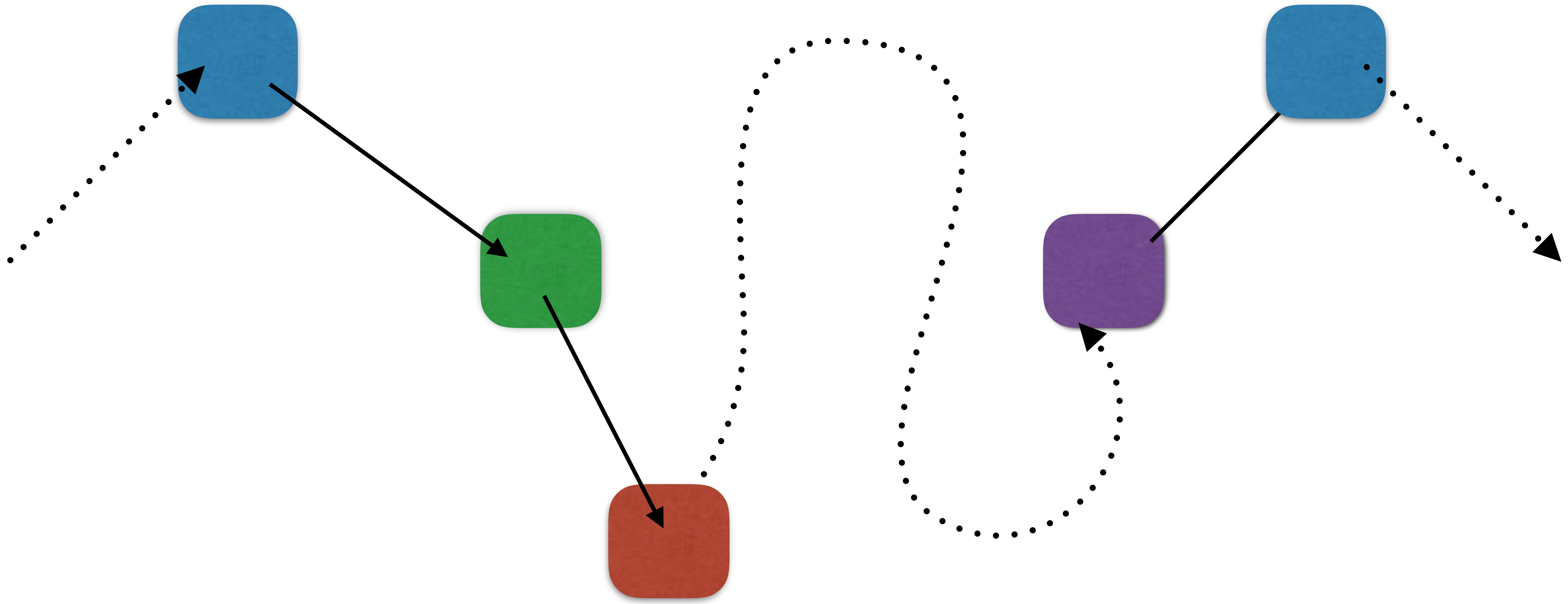


# Bounding the number of tasks

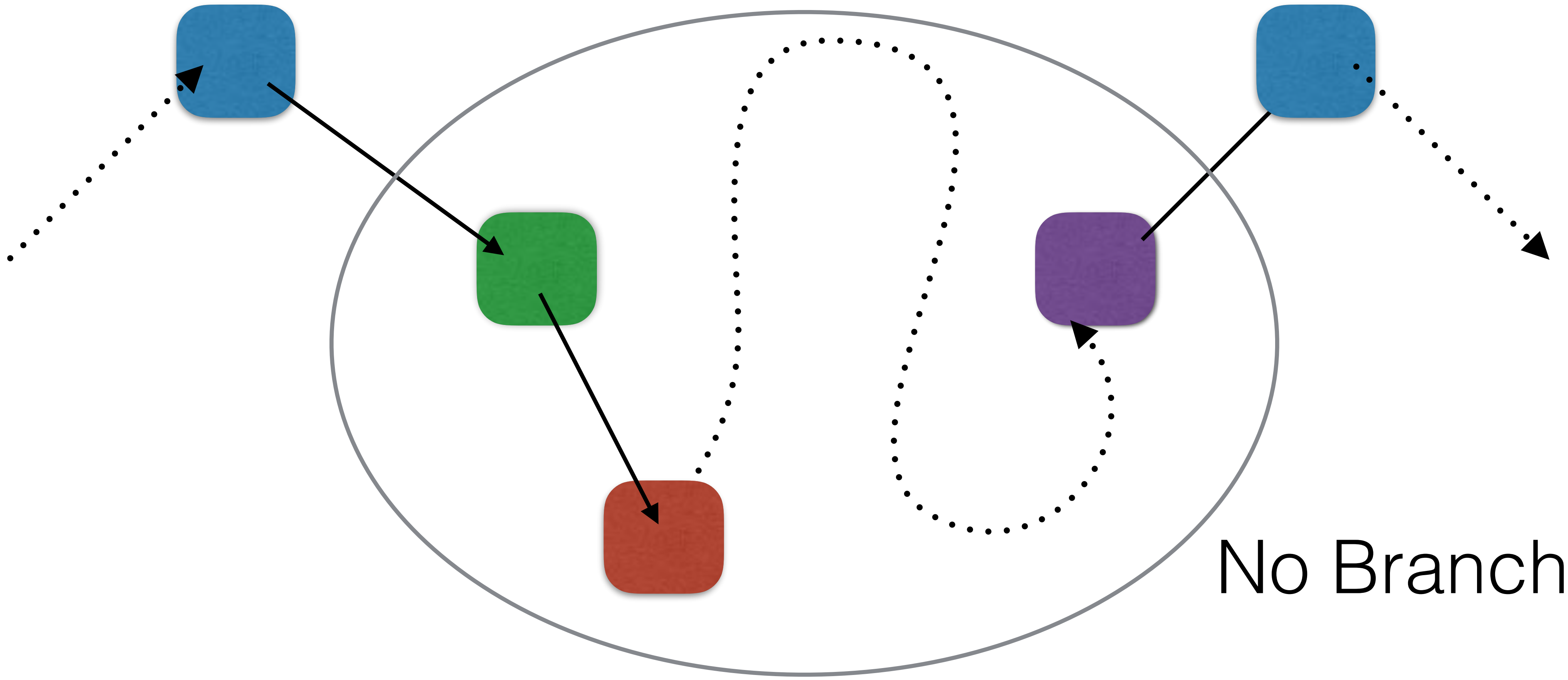


Number of branching points bounded by threads

# Bounding Path length

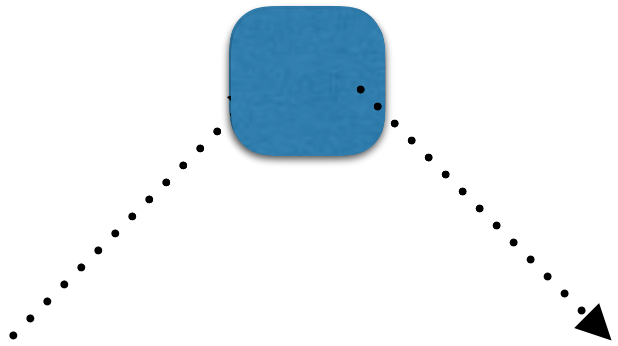


# Bounding Path length

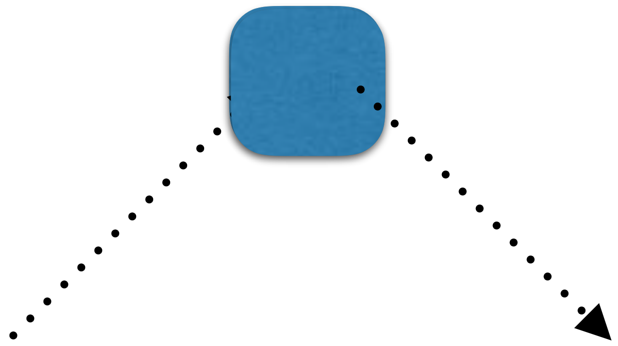


No Branching

# Bounding Path length



# Bounding Path length



Path length bounded by Poly(threads, tasks)

Width also bounded by threads.

# Stateless task scheduling:

# Stateless task scheduling:

- Only a polynomial bound on the number of tasks that need to be scheduled.

# Stateless task scheduling:

  - Only a polynomial bound on the number of tasks that need to be scheduled.

- 1-Thread simulation can work with the same number of tasks.

# Stateless task scheduling:

- Only a polynomial bound on the number of tasks that need to be scheduled.

- 1-Thread simulation can work with the same number of tasks.

Complexity of emptiness of Asynchronous Programs with at most polynomial number of operations on the multi-set.

# Stateless task scheduling:

- Only a polynomial bound on the number of tasks that need to be scheduled.

- 1-Thread simulation can work with the same number of tasks.

Complexity of emptiness of Asynchronous Programs with at most polynomial number of operations on the multi-set.

- Guess and write down a consistent sequence of Multi-set operations (consistent: add >= remove at each point for each task)

# Stateless task scheduling:

- Only a polynomial bound on the number of tasks that need to be scheduled.

- 1-Thread simulation can work with the same number of tasks.

Complexity of emptiness of Asynchronous Programs with at most polynomial number of operations on the multi-set.

- Guess and write down a consistent sequence of Multi-set operations (consistent: add >= remove at each point for each task)
- Simulate the Asynchronous program as a pushdown on this input.

# Stateless task scheduling:

Theorem: Reachability for Asynchronous programs with locks under state-less scheduling, well-nested locks and task locking is NP-Complete

# Stateless task scheduling:

Theorem: Reachability for Asynchronous programs with locks under state-less scheduling, well-nested locks and task locking is NP-Complete

- Lower-bound —- reduction from SAT.
    - Take locks to decide on valuation (taking lock x if x = False)
    - Cycle through clauses and check that at least one literal is true.

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.

- Decidable under a further task locking restriction. EXPSPACE-Complete.

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.

- Decidable under a further task locking restriction. EXPSPACE-Complete.

- Stateless scheduling is decidable in NP.

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.
- Decidable under a further task locking restriction. EXPSPACE-Complete.
- Stateless scheduling is decidable in NP.

## Possible Extensions

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.
- Decidable under a further task locking restriction. EXPSPACE-Complete.
- Stateless scheduling is decidable in NP.

## Possible Extensions

- Locks + Shared memory. Reasonable restrictions for decidability?

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.
- Decidable under a further task locking restriction. EXPSPACE-Complete.
- Stateless scheduling is decidable in NP.

## Possible Extensions

- Locks + Shared memory. Reasonable restrictions for decidability?
- Other locking subclasses: bounded lock chains, contextual locking

# Conclusion:

- Asynchronous programs with nested locks: reachability is undecidable.
- Decidable under a further task locking restriction. EXPSPACE-Complete.
- Stateless scheduling is decidable in NP.

## Possible Extensions

- Locks + Shared memory. Reasonable restrictions for decidability?
- Other locking subclasses: bounded lock chains, contextual locking

**Thank you**