Programmazione concorrente

Laurea Magistrale in Ingegneria Informatica
Università Tor Vergata
Docente: Romolo Marotta

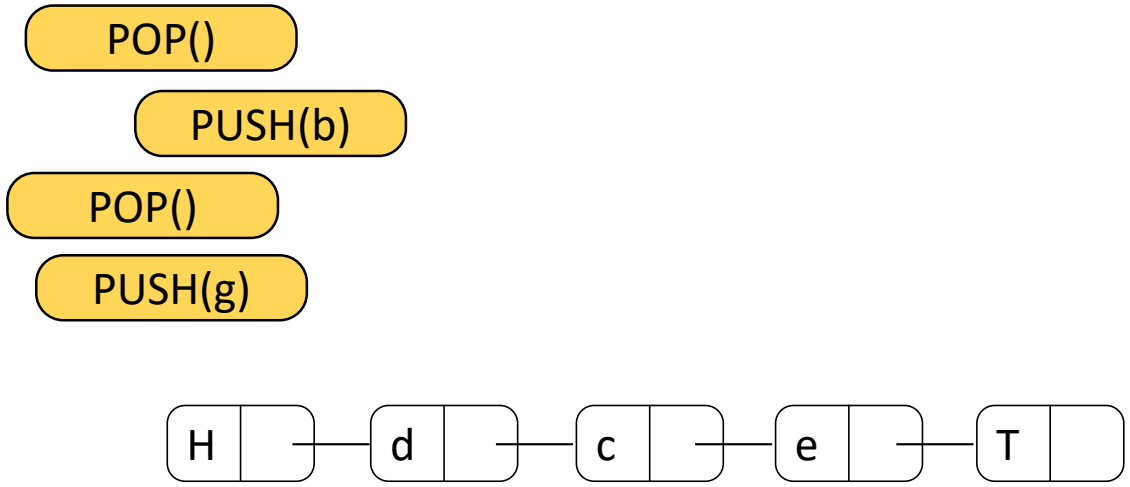
Concurrent data structures

1. Stack

Concurrent Data Structures: Stacks

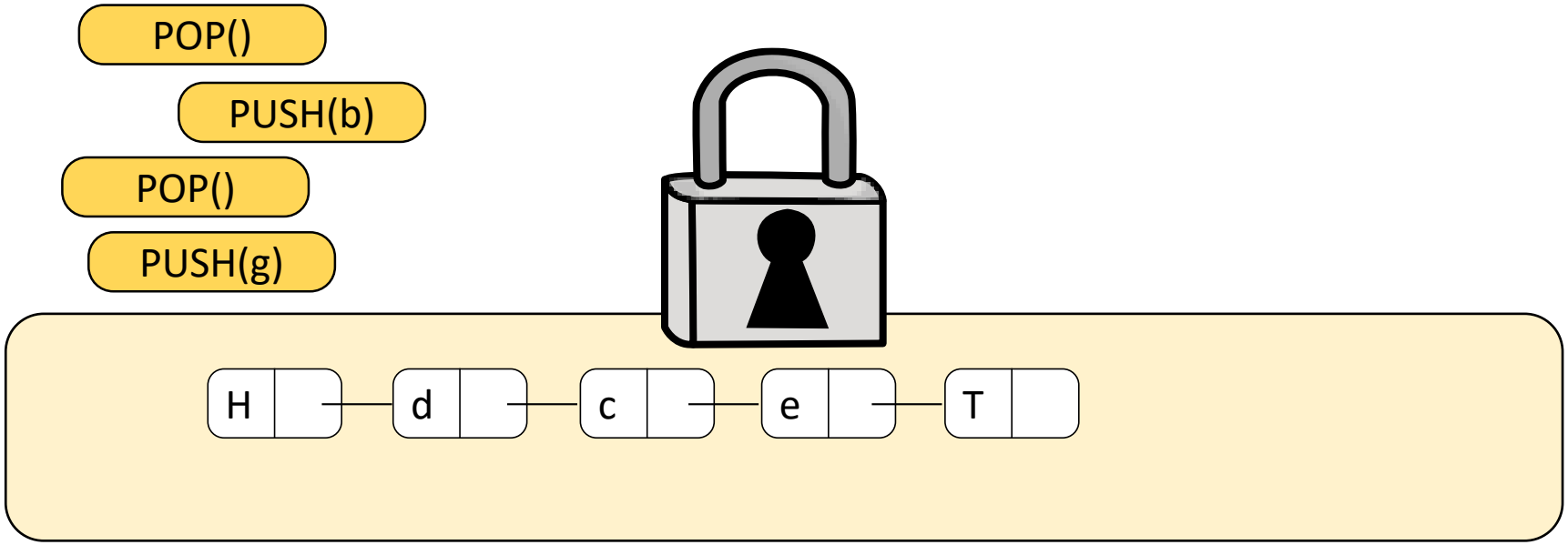
Stack implementation

- Stack methods:
  - `push(v)`
  - `pop()`
- Implemented as a linked list



Concurrent stack implementations

- Resort to a global lock



Read-Modify-Write

- RMW instructions allow to read memory and modify its content in an apparently instantaneous fashion.

```
1.RMW(MRegister *r, Function f){
2.   atomic{
3.     old = r;
4.     *r = f(r);
5.     return old;
6.   }
7.}
```

- Even conventional atomic Load and Store can be seen as RMW operations

Compare-And-Swap

- Compare-and-Swap (CAS) is an atomic instruction used in multithreading to achieve synchronization
  - It compares the contents of a memory area with a supplied value
  - If and only if they are the same
  - The contents of the memory area are updated with the new provided value
- Atomicity guarantees that the new value is computed based on up-to-date information
- If, in the meanwhile, the value has been updated by another thread, the update fails
- This instruction has been introduced in 1970 in the IBM 370 trying to limit as much as possible the use of spinlocks

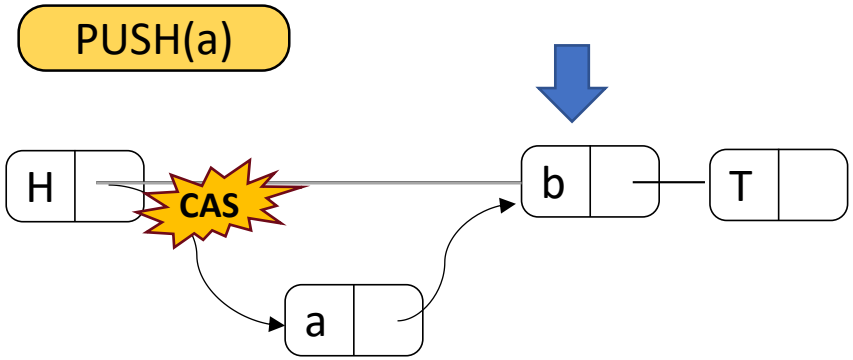
Attempt 1

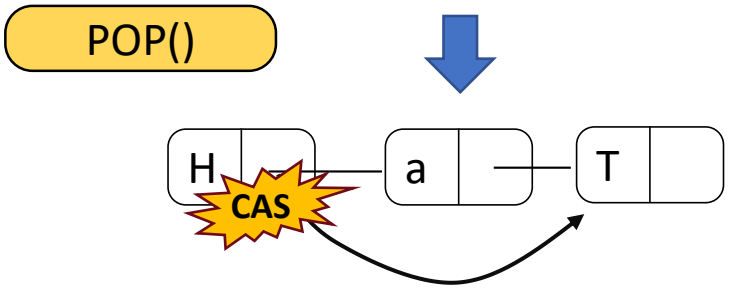
Push:

1. Get head next
2. Insert the new item with a CAS
3. If CAS fails, restart

Delete:

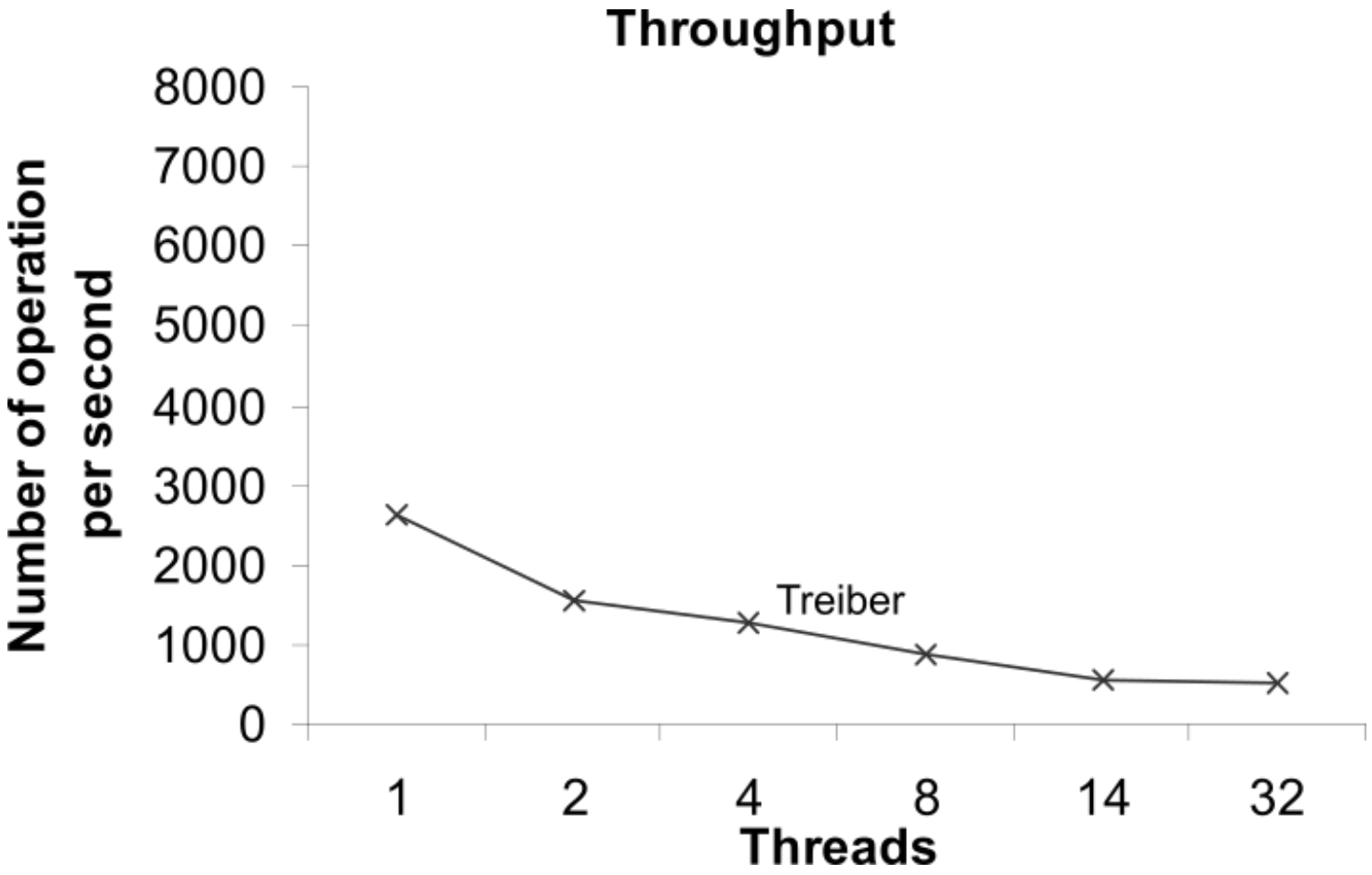
1. Get head next
2. Disconnect the item with a CAS
3. If CAS fails, restart





- Is it scalable?

Non-blocking stack – Attempt 2 [Treiber+BO]



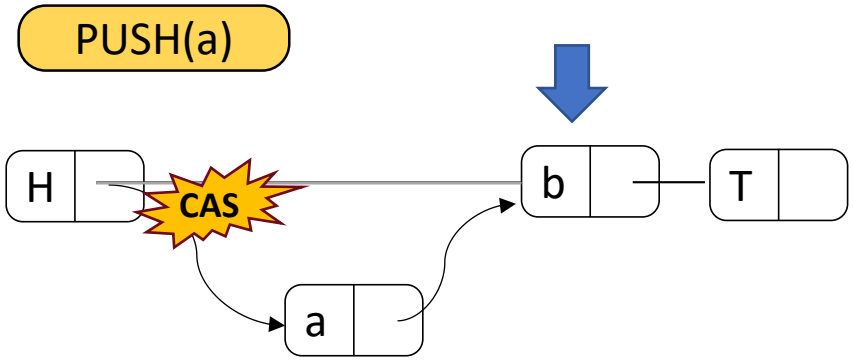
Non-blocking stack – Attempt 2 [Treiber+BO]

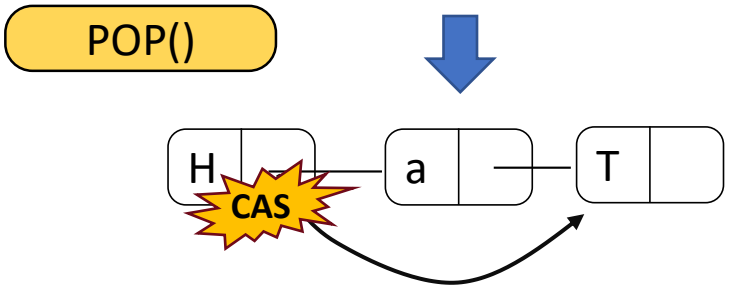
Push:

1. Get head next
2. Insert the new item with a CAS
3. If CAS fails, ~~restart~~ backoff and restart

Delete:

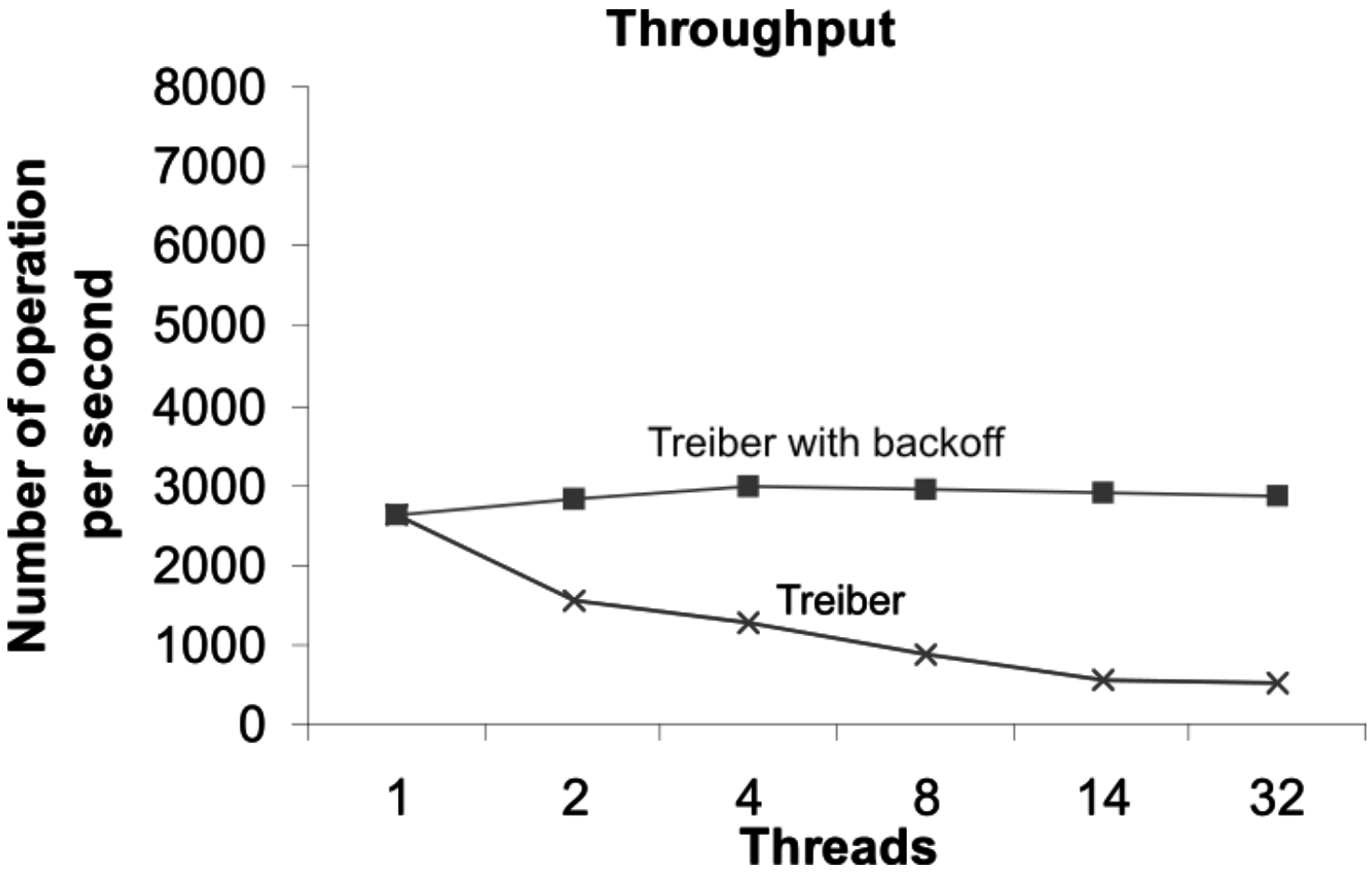
1. Get head next
2. Disconnect the item with a CAS
3. If CAS fails, ~~restart~~ backoff and restart





- Is it scalable?

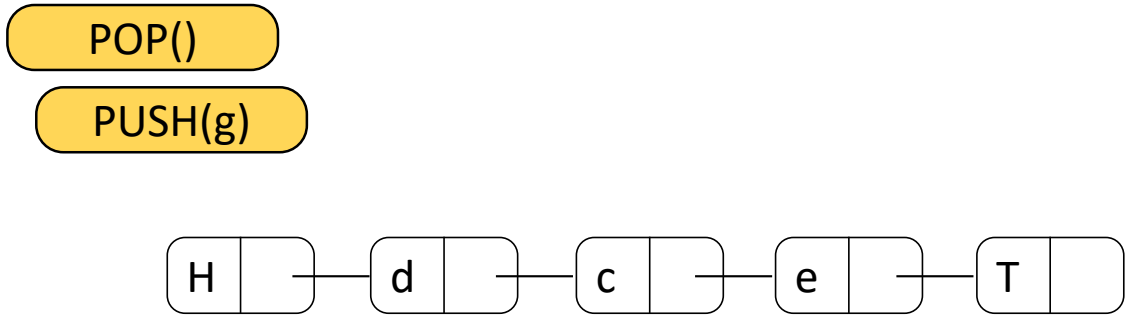
Non-blocking stack – Attempt 2 [Treiber+BO]



Concurrent stack implementations

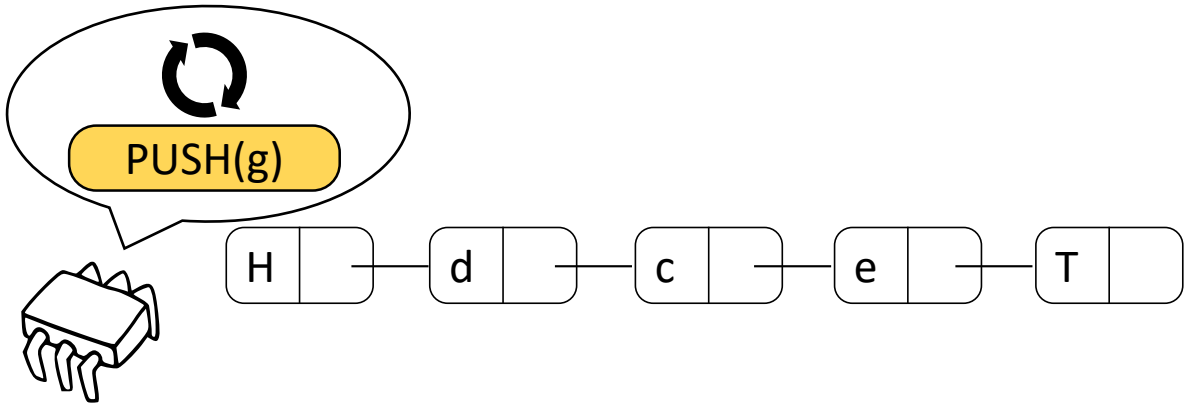
- Resort to a global lock
  - Do not scale
- Resort to a naïve non-blocking approach
  - Do not scale
- Resort to a naïve non-blocking approach + Back off
  - Do not scale, but conflict resilient
- How achieve scalability?

Make back-off times useful



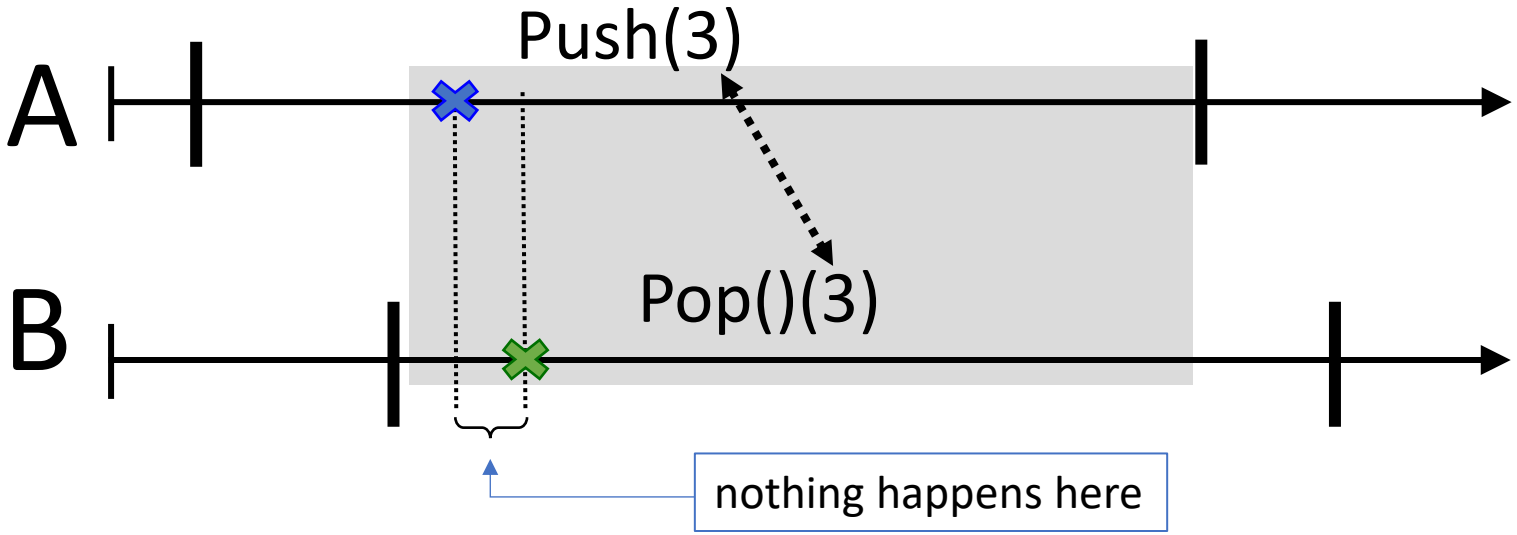
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?



Observation

- Concurrent matching push/pop pairs are always linearizable



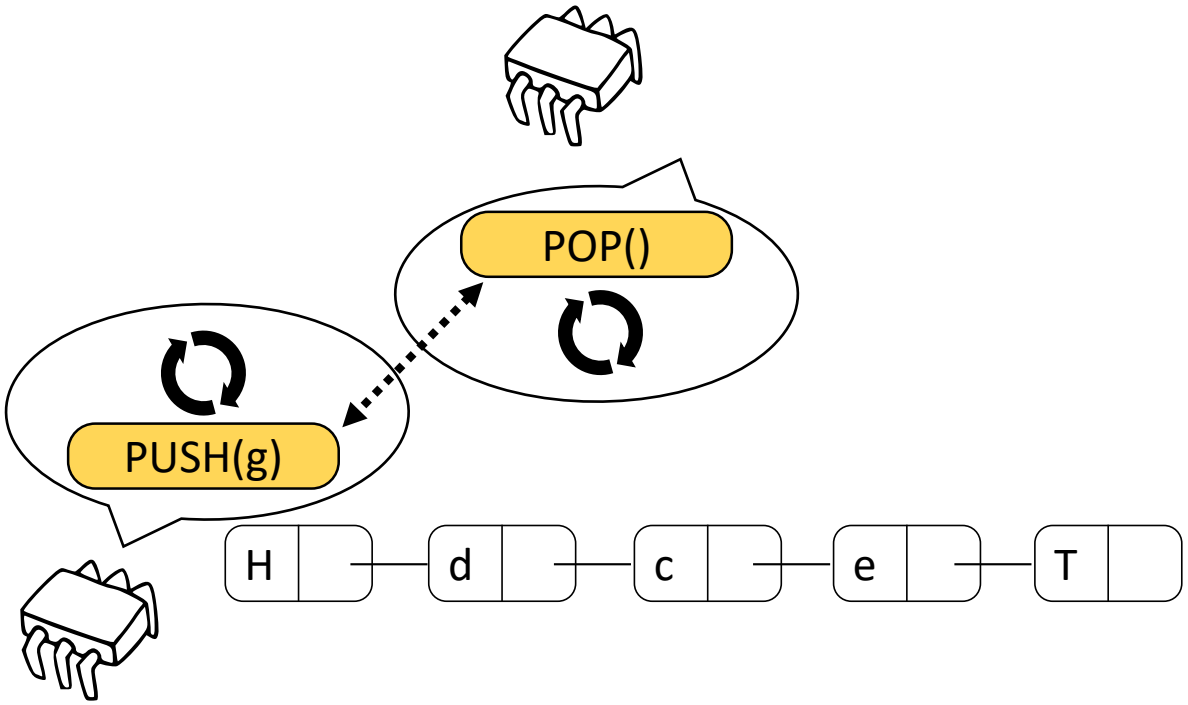
- A push A and a pop B are:
  - concurrent to each other
  - B returns the item inserted by A

$\Rightarrow$ we can always take two points such that:

  - A is the last one to insert an item before A linearizes
  - B appears to extract the last item inserted (by A)

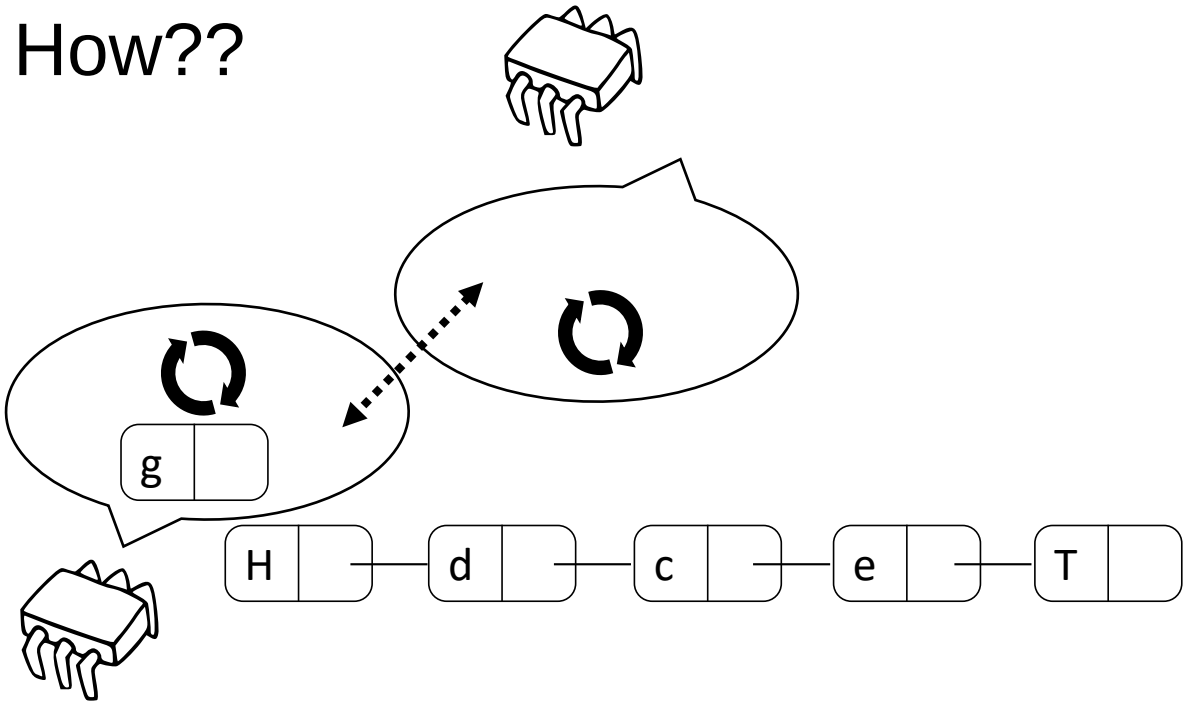
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?
- Hope that an opposite operation arrives while waiting
- Match the two without interacting with the stack



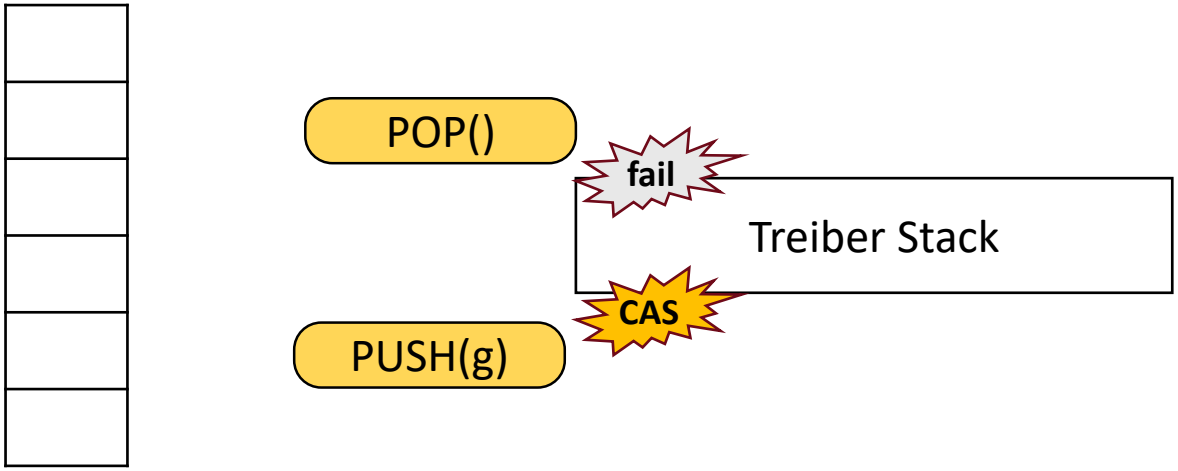
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?
- Hope that an opposite operation arrives while waiting
- Match the two without interacting with the stack
- How??



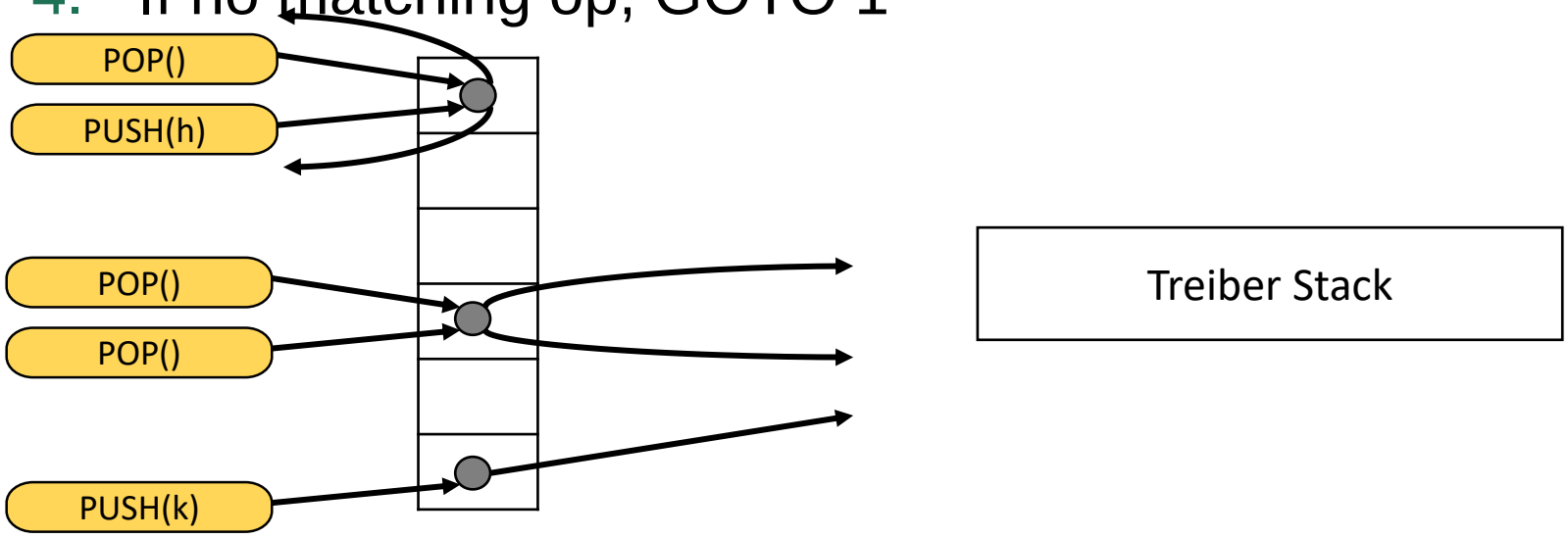
Non-blocking stack – Elimination stack

- Pair the Treiber stack with an array
- Algorithm:
  1. Update the original stack via CAS
  2. If CAS fails, publish the operation in a random cell of the array



Non-blocking stack – Elimination stack

- Pair the Treiber stack with an array
- Algorithm:
  1. Update the original stack via CAS
  2. If CAS fails, publish the operation in a random cell of the array
  3. Wait for a matching operation
  4. If no matching op, GOTO 1



Non-blocking stack – Attempt 3