# Modular Verification of Linearizability with Non-Fixed Linearization Points

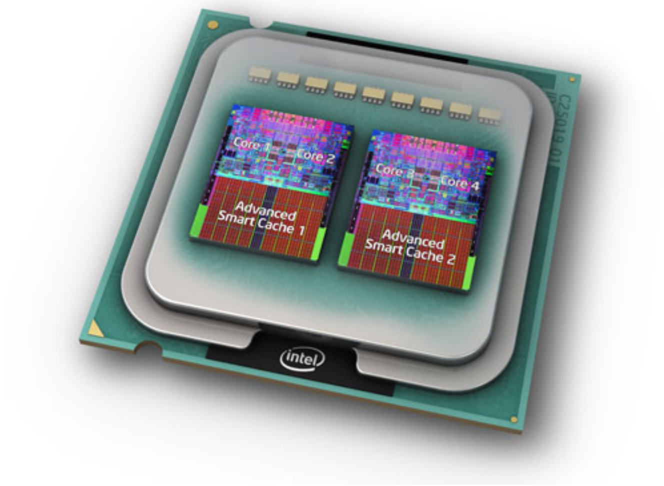
Xinyu Feng

USTC

Joint work with Hongjin Liang

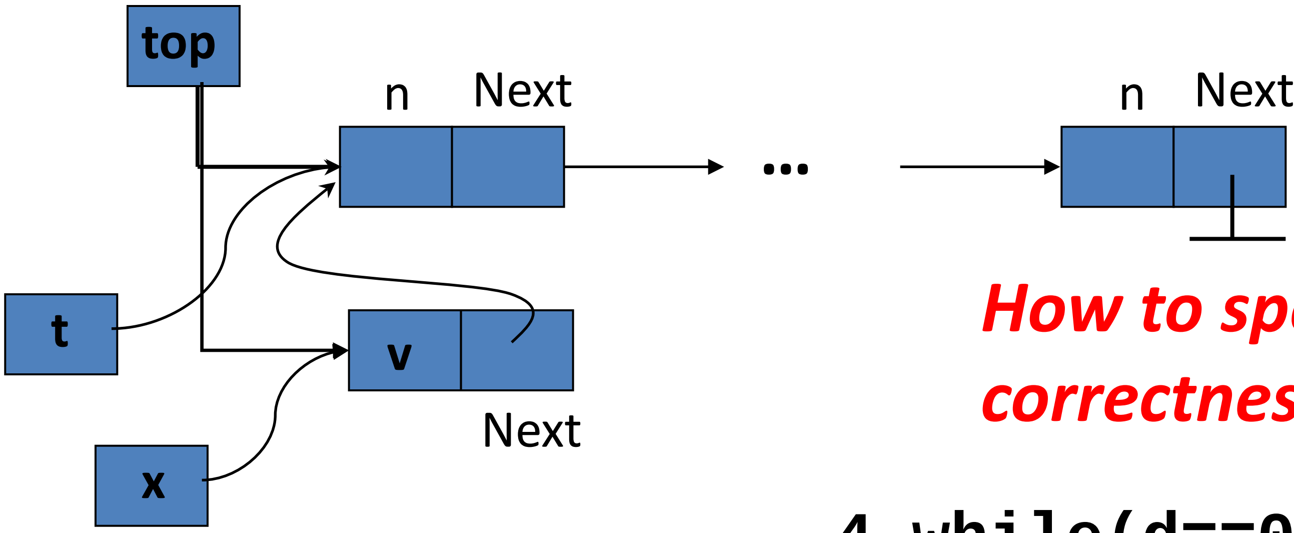
# Concurrency Verification

- Concurrency: a basic programming skill in multicore era



- Very difficult to ensure correctness
  - Very subtle interleaving
- This talk:
  correctness of (non-blocking) concurrent obj.

# Example: Treiber’s Non-blocking Stack



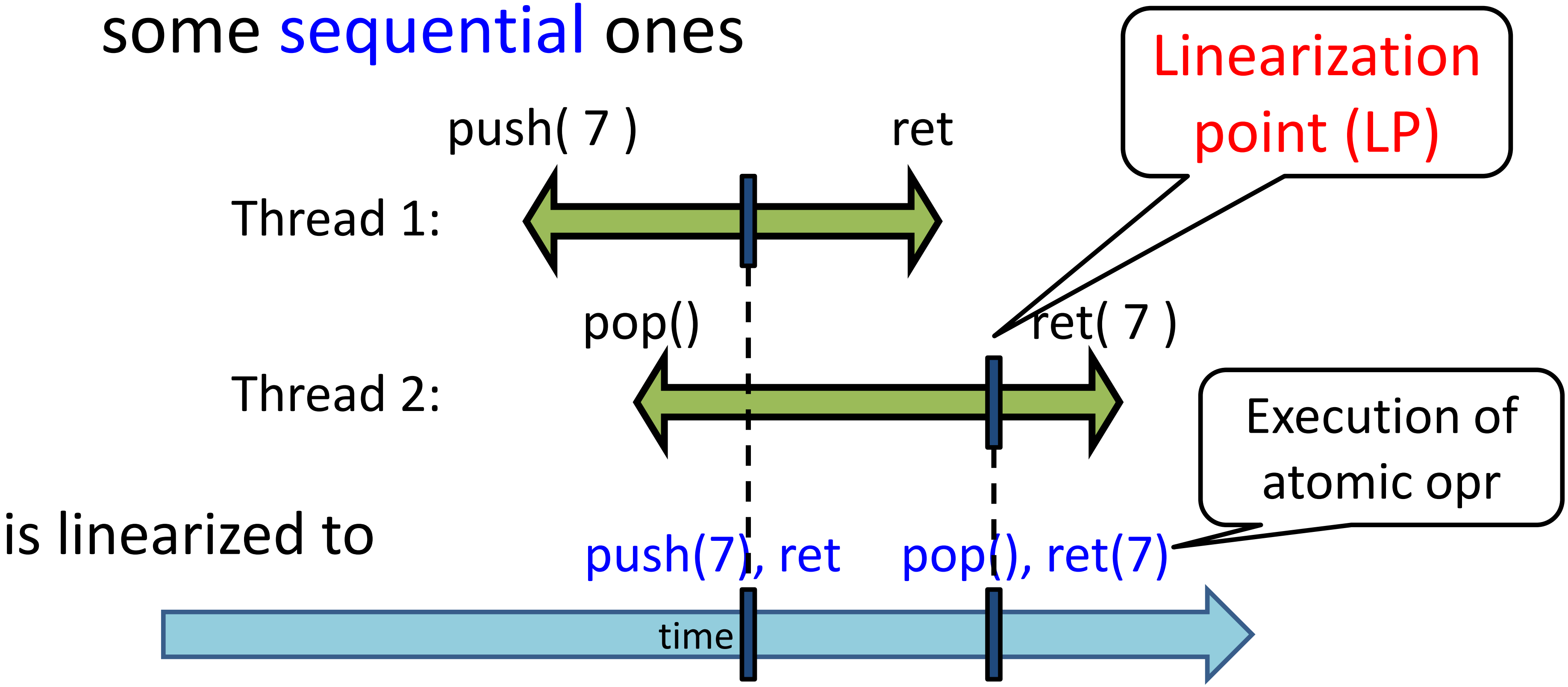
***How to specify/prove correctness?***

push(int v):

```
int d=0, x, t;
x = new Node();
x.data = v;
while(d==0){
  <t = top>;
  x.next = t;
  d = cas(&top, t, x);
}
```

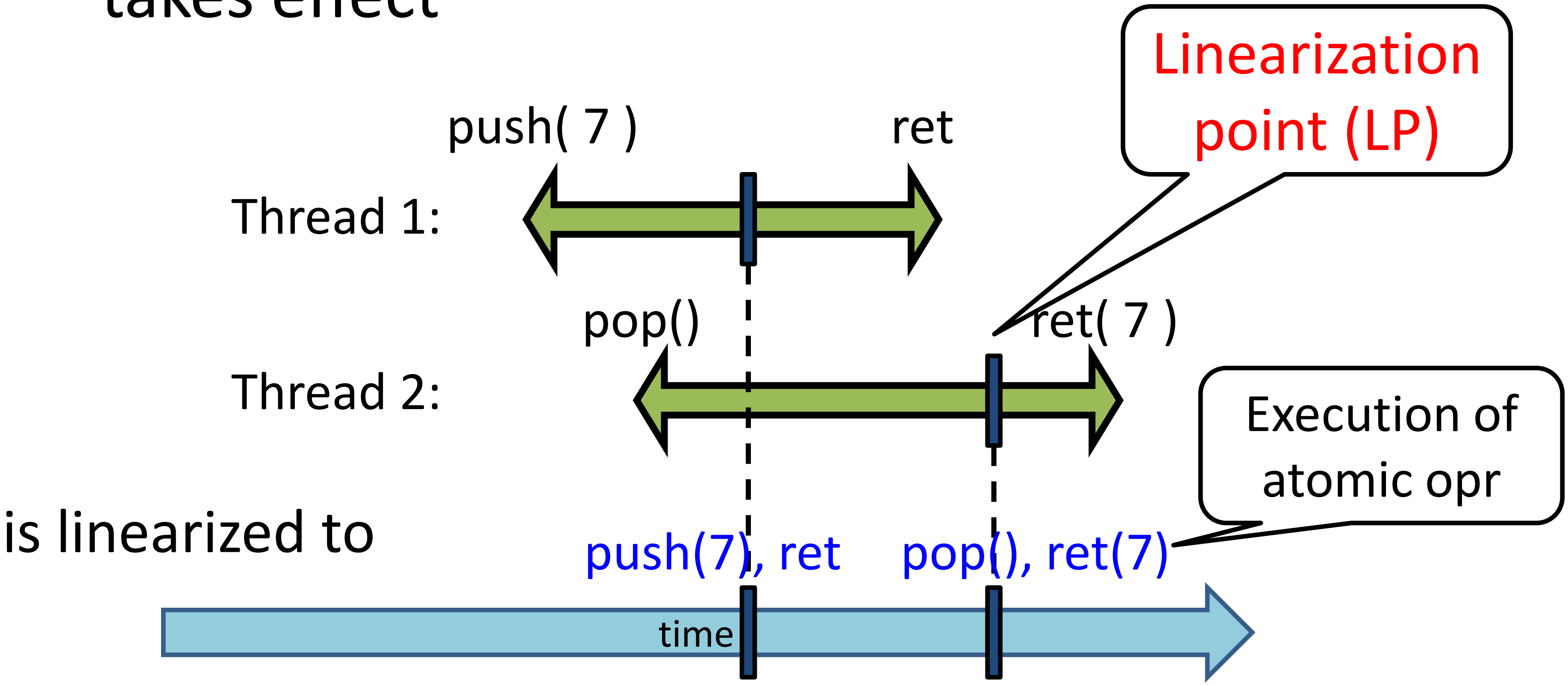
# Linearizability[Herlihy & Wing’90]

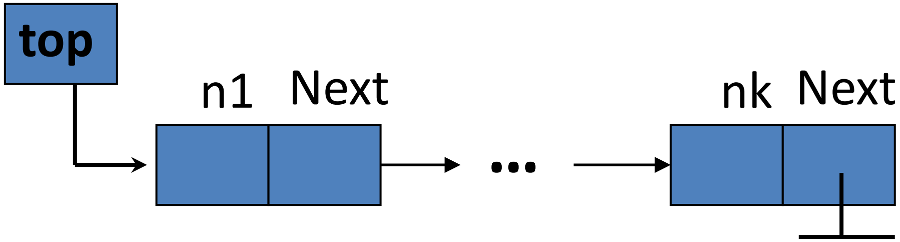
- Standard correctness criterion
- All concurrent executions are “equivalent” to some sequential ones



# LP Method to Prove Linearizability

- Locate LP in impl code O
- Show it is the single point where the method takes effect





# Treiber’s Stack

```
push(v):
1  local d:=0, x, t;
2  x := new Node(v);
3  while(d=0){
4    t := top;
5    x.next := t;
6    d := cas(&top, t, x);
7  }
```

Not update the shared list

LP

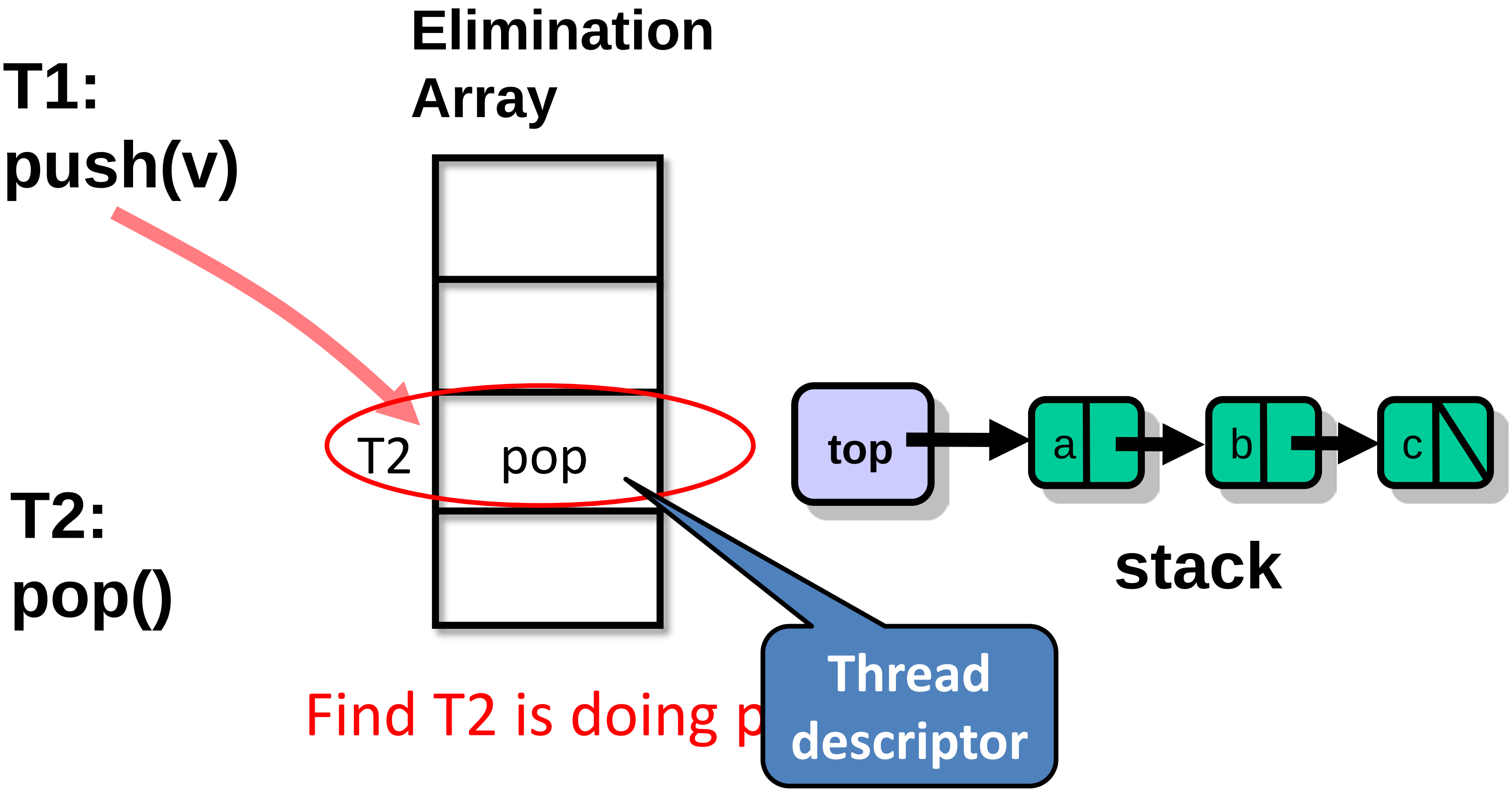
Line 6: the only command that changes the list

# Problems with LP Method

- Informal
  - Mostly folklore theorem

- Difficult to locate LP
  - LP cannot be statically located – non-fixed LP
  - Common in many wait-free algorithms

# Example: HSY Stack



T1: push(v)

Elimination Array

T2 pop

T2: pop()

top

a

b

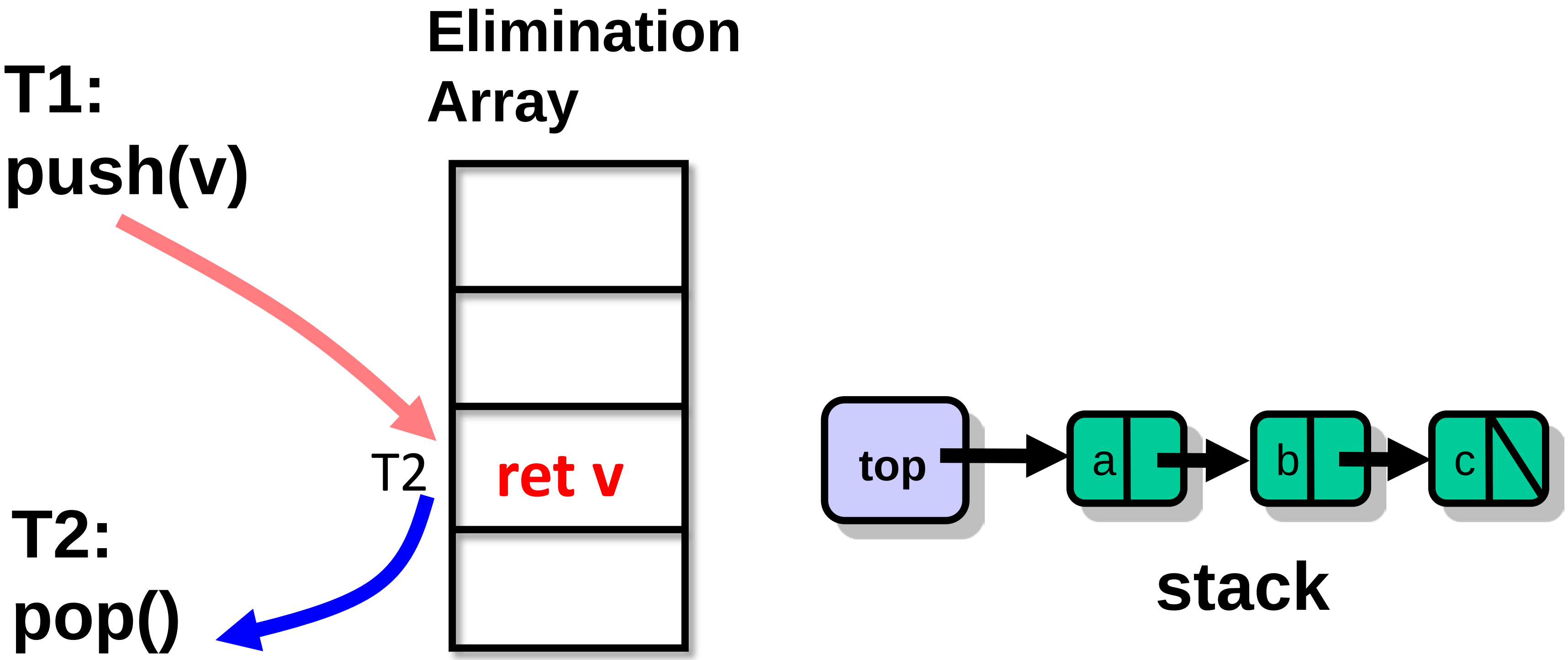
c

stack

Thread descriptor

Find T2 is doing p

# Example: HSY Stack



T1: push(v)

Elimination Array

T2

ret v

T2: pop()

top

a

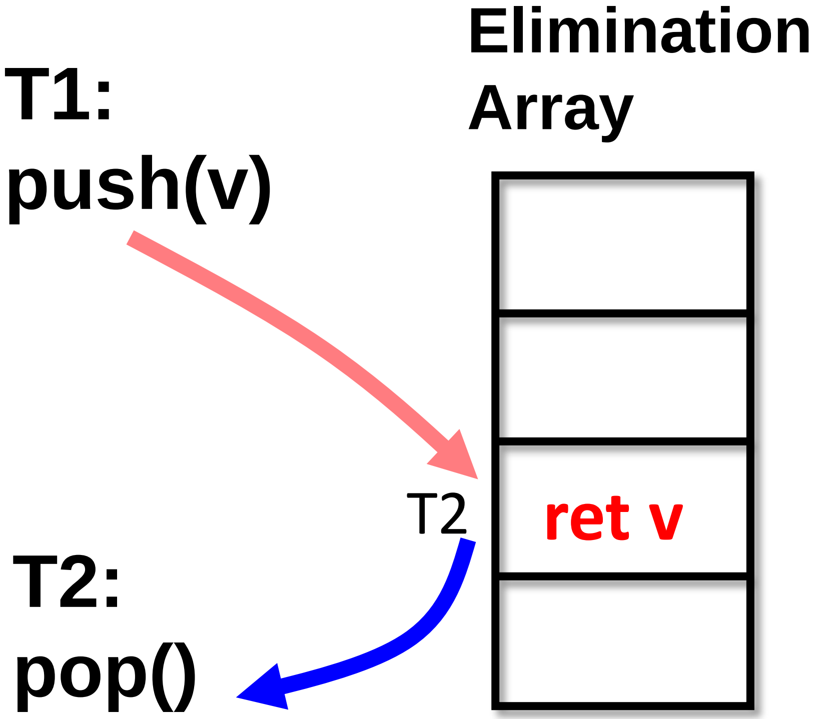
b

c

stack

T1 finishes not only its own opr, but also T2's

# Coordination Pattern in Wait-Free Alg.



**T1: push(v)**

**Elimination Array**

T2

**ret v**

**T2: pop()**

**T1 interrupts T2, and helps T2 to finish its pending opr.**

**When T2 comes, its job is done.**

***Difficult to find LP of T2's opr!***

# Problems with LP Method

- Informal
  - Mostly folklore theorem
- Difficult to locate LP
  - LP cannot be statically located – non-fixed LP
  - Common in many wait-free algorithms

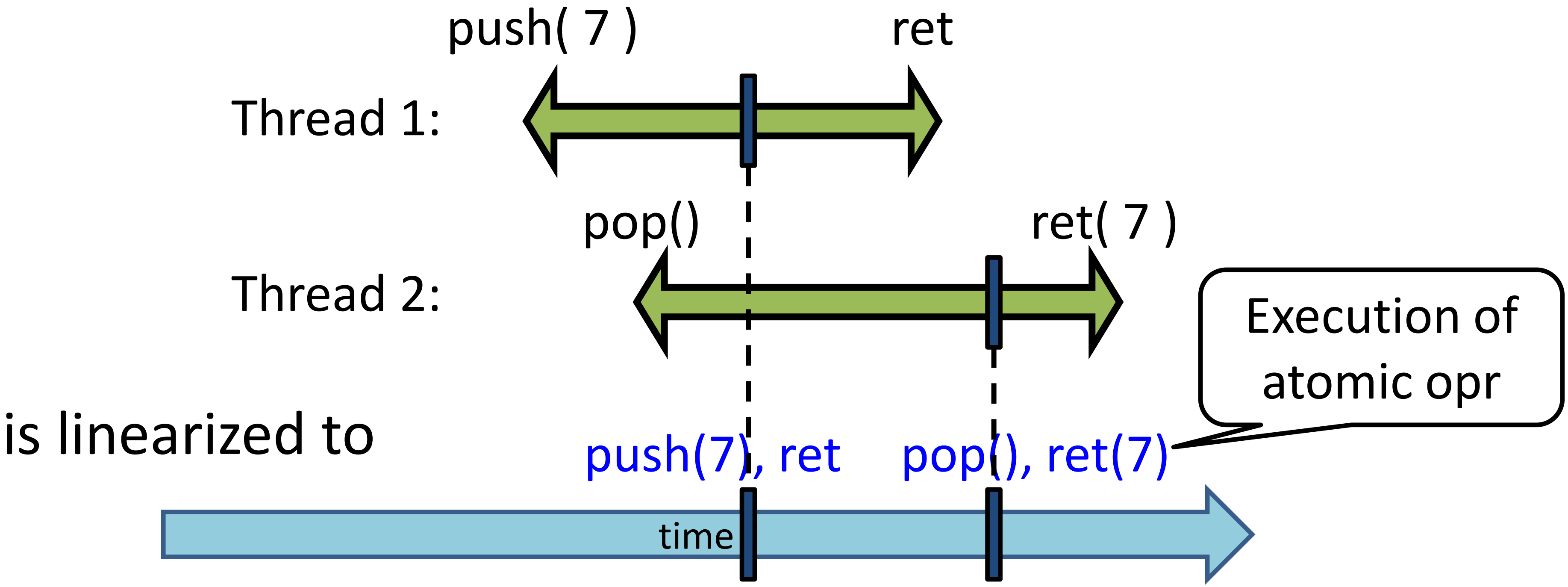
# This Talk

- Simulation-based proof for linearizability
  - Supports compositional verification
  - Formally justifies the folklore LP method

- Extending it for objects with non-fixed LP
  - HSY elimination stack, lazy set , etc.

# Linearizability and Program Refinement

**Observation:** Linearizable fine-grained impl. has the same effect as atomic operations



push( 7 )

ret

Thread 1:

pop()

ret( 7 )

Thread 2:

Execution of atomic opr

is linearized to

push(7), ret

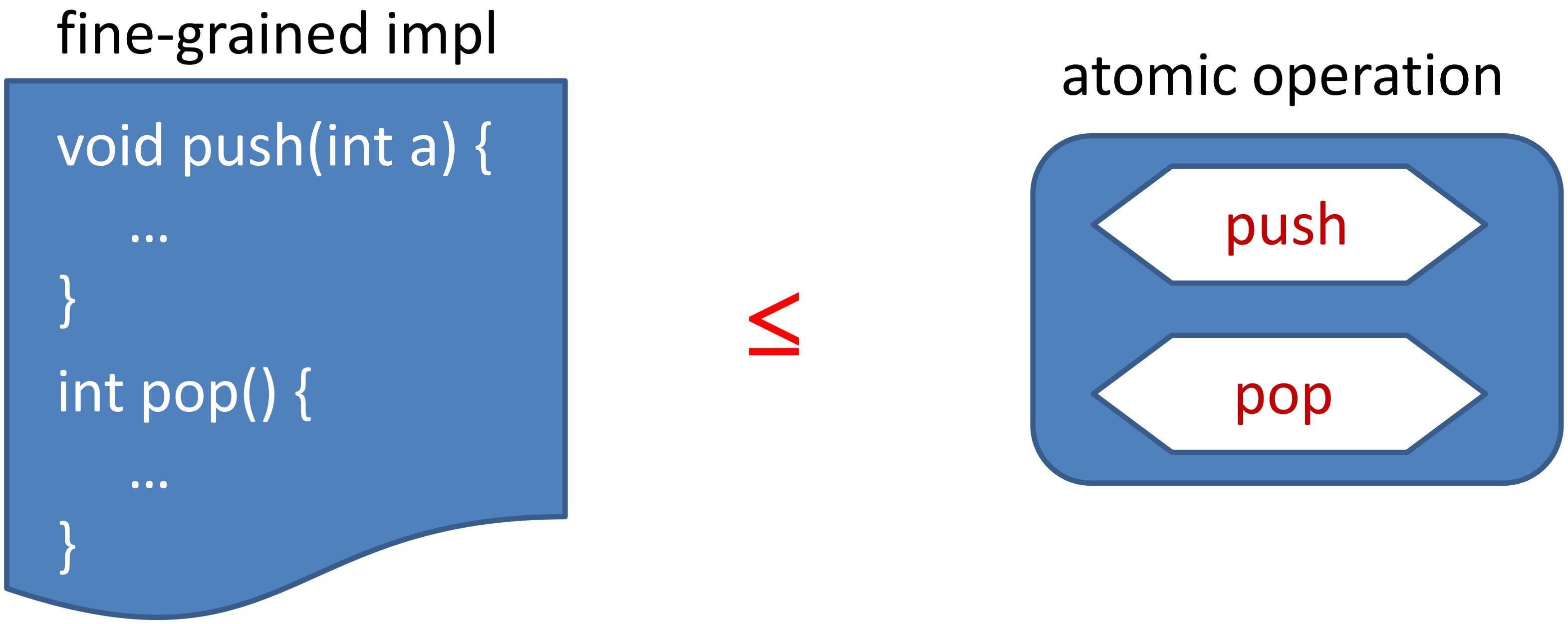
pop(), ret(7)

time

# Linearizability and Program Refinement

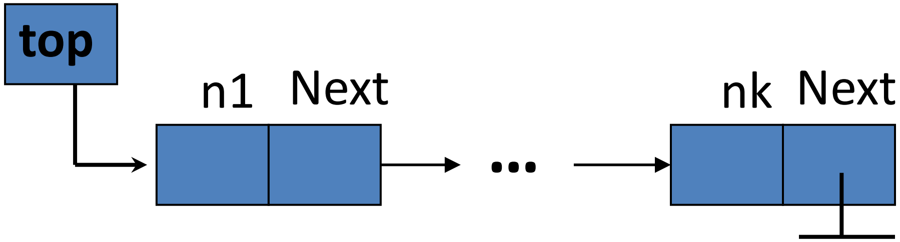
Observation: Linearizable fine-grained impl.
has the same effect as atomic operations

***Reduce linearizability to program equivalence:***



```
void push(int a) {
    …
}
int pop() {
    …
}
```

# Treiber’s Stack



```
push(v):
local d:=0, x, t;
x := new Node(v);
while(d=0){
  t := top;
  x.next := t;
  d := cas(&top, t, x);
}
```

object impl O

Abstract representation

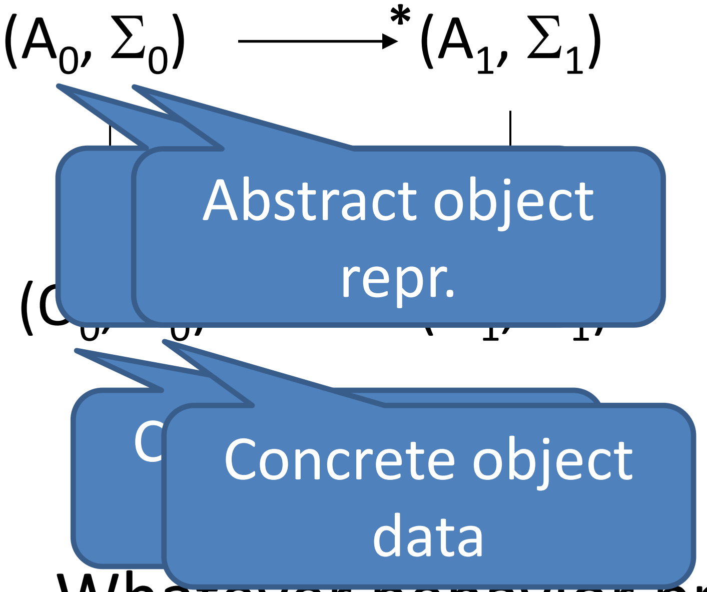
stk: n1 :: n2 :: … :: nk

```
push(v):
<stk := v::stk>;
```

Atomic operation

atomic opr as spec A

# Simulation for Refinement Proof

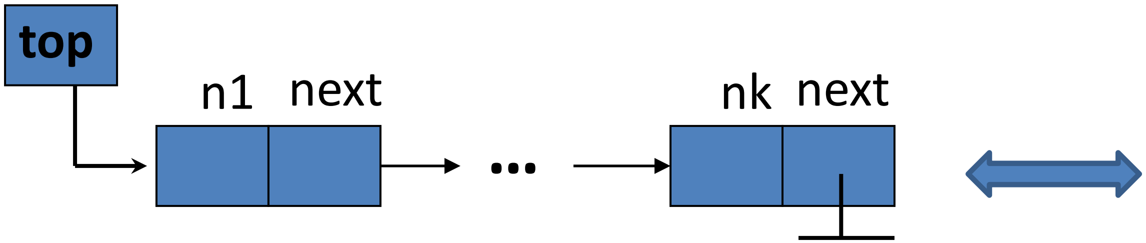


$$\begin{tikzcd}
(A_1, \Sigma_1) \arrow[r, "*"] \arrow[d, dash, "\precsim"] & (A_2, \Sigma_2) \arrow[d, dash, "\precsim"] & \cdots \\
(O_1, \sigma_1) \arrow[r] & (O_2, \sigma_2) & \cdots
\end{tikzcd}$$

Whatever behavior produced by low-level could be produced by high-level.

# Customized for Linearizability

$$\begin{tikzcd}
 & (A, \Sigma_0) \arrow[dl, dash, "\precsim"'] \arrow[d, dash, "\precsim"] \\
(O_0, \sigma_0) \arrow[r] & (O_1, \sigma_1)
\end{tikzcd}
\qquad
\begin{tikzcd}
(A, \Sigma_0) \arrow[r] \arrow[d, dash, "\precsim"] & (\bullet, \Sigma_1) \arrow[d, dash, "\precsim"] \\
(O_1, \sigma_1) \arrow[r, "\textit{LP}"'] & (O_2, \sigma_2)
\end{tikzcd}
\qquad
\begin{tikzcd}
(\bullet, \Sigma_1) \arrow[d, dash, "\precsim"] \arrow[dr, dash, "\precsim"] & \cdots \\
(O_2, \sigma_2) \arrow[r] & (O_3, \sigma_3)
\end{tikzcd}$$



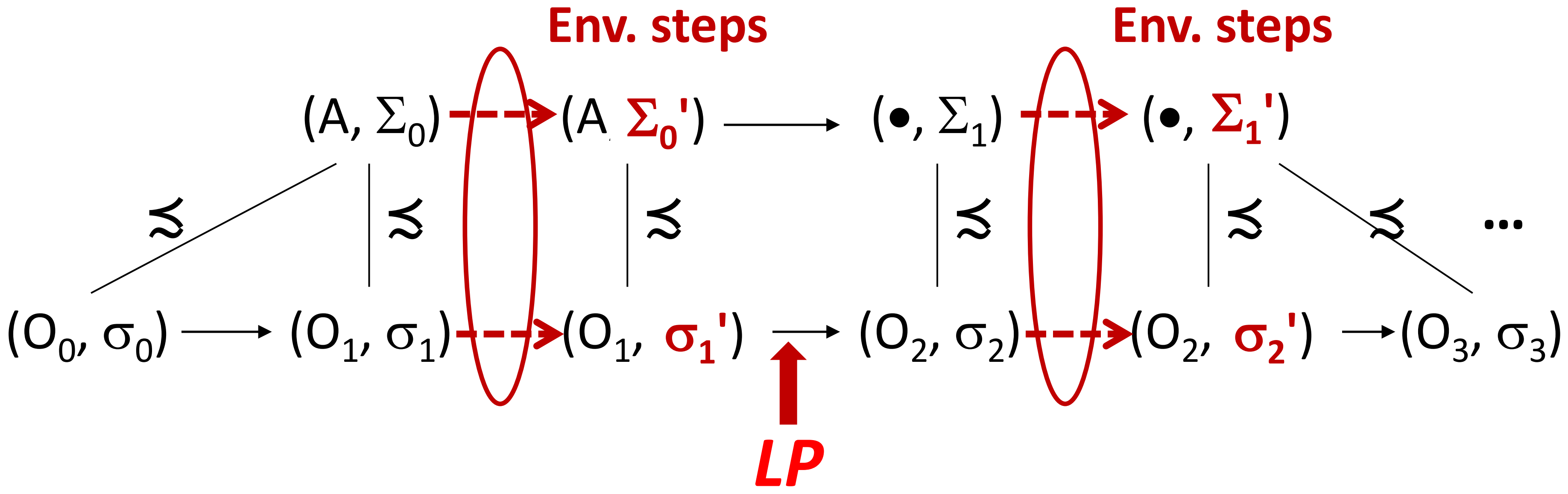
stk: n1 :: n2 :: … :: nk

```
push(v):
local d:=0, x, t;
x := new Node(v);
while(d=0){
  t := top;
  x.next := t;
  d := cas(&top, t, x);
}
```

**However, needs to consider env., otherwise unsound.**

<stk := v::stk>;

# Simulation with Env.



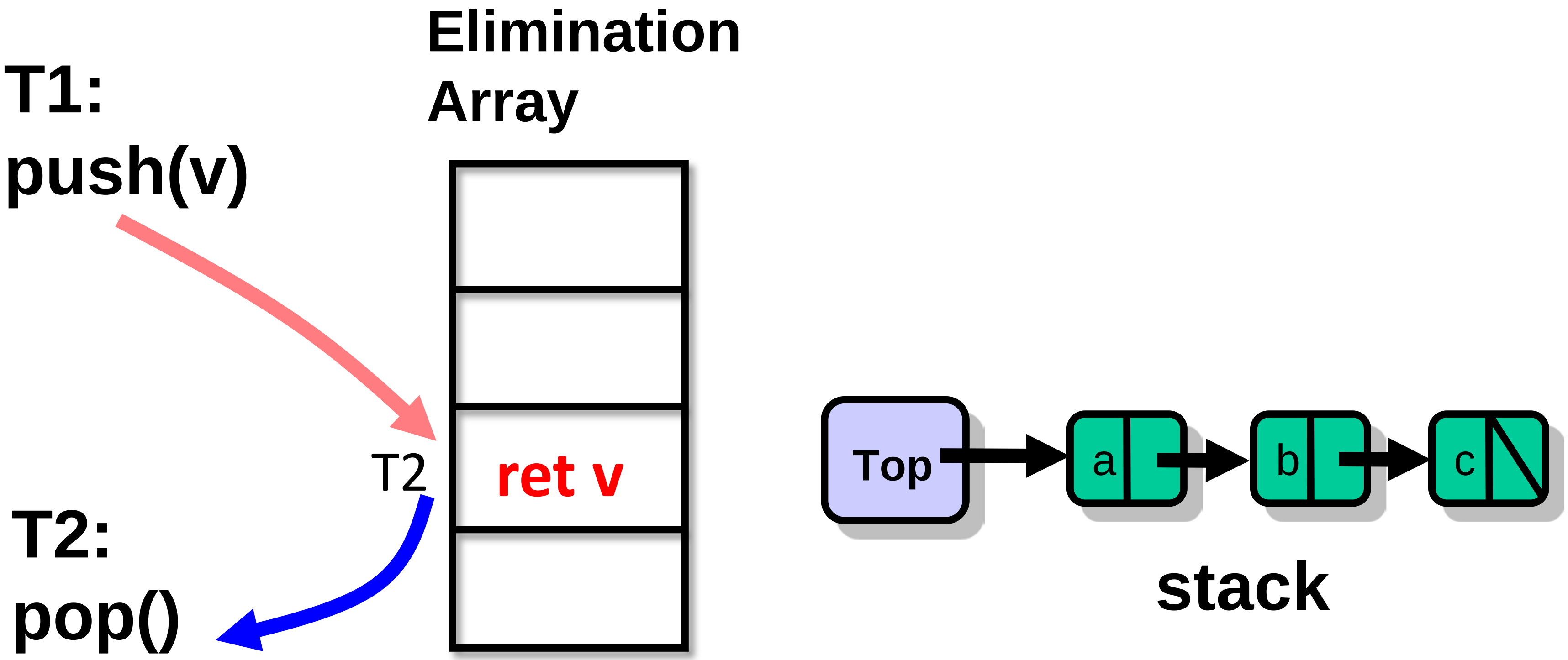
Needs to ensure the env. steps do not break simulation (environment doesn't do bad things)

# Simulation with Env.

- RGSim: Rely-Guarantee based simulation [Liang et al’12]
    - General method for concurrent program refinement
    - Adapted for linearizability proof
        - Formally justifies the LP method
        - Used to verify Treiber’s stack

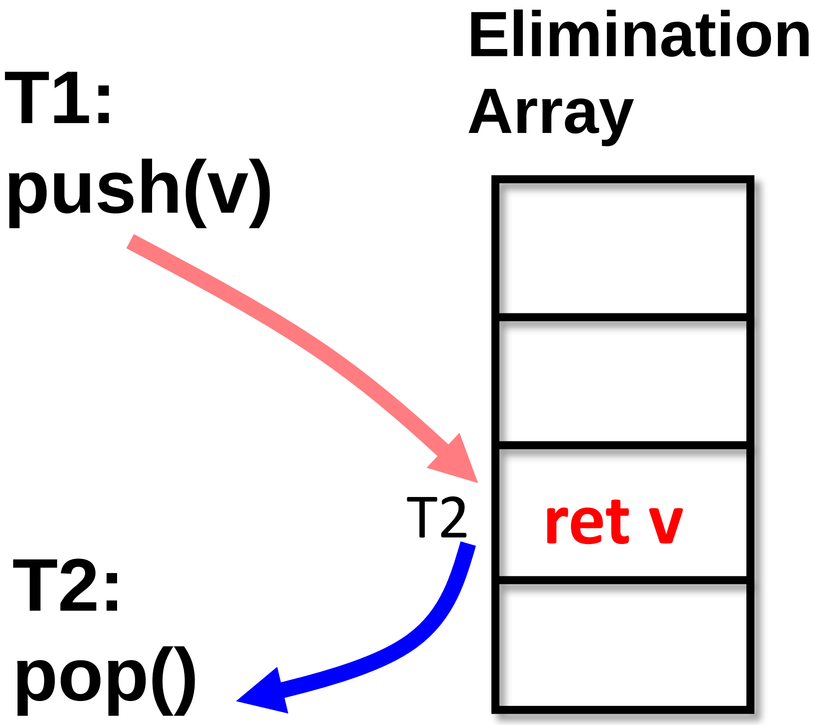
- Needs to know statically LP point
    - Does not support non-fixed LP as in HSY stack

# HSY Stack Implementation



T1 finishes not only its own opr, but also T2's

# HSY Stack Implementation



**T1: push(v)**

**Elimination Array**

T2

**ret v**

**T2: pop()**

What's the problem?

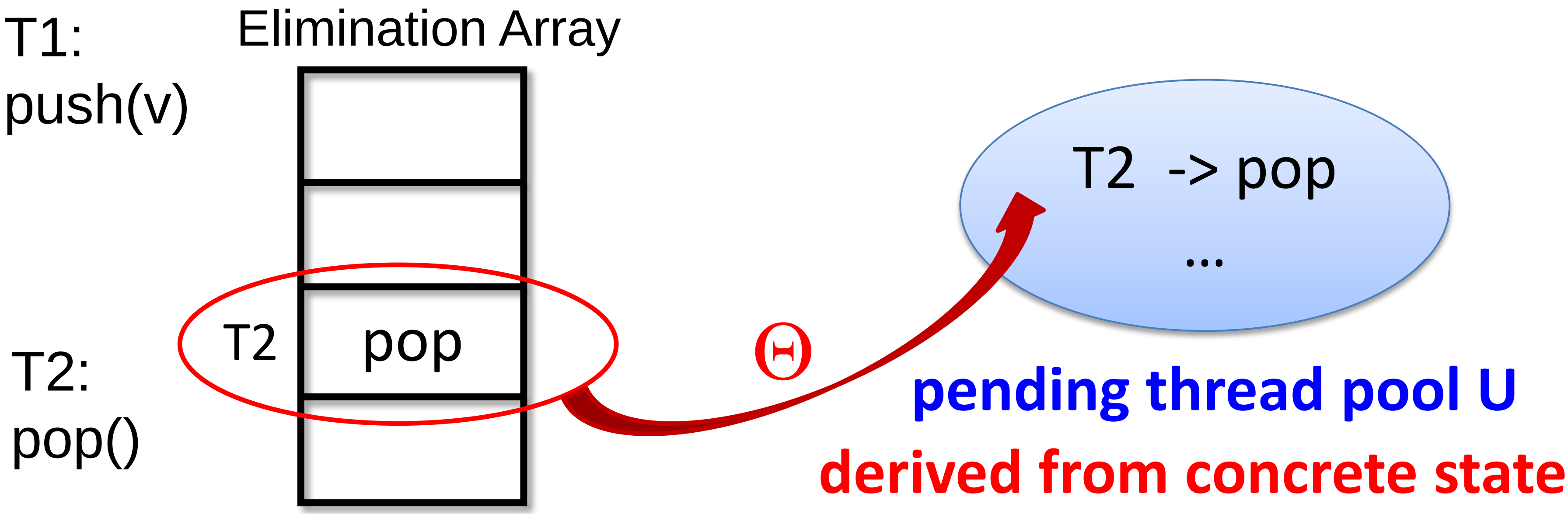
The LP of pop may not even be inside the method body.

LP corresponds to env. Step, not supported in previous sim.

T1 finishes not only its own opr, but also T2's

# Our Solution

- Parameterize RGSim with pending threads that might be helped by others



T1:
push(v)

Elimination Array

T2 pop

T2:
pop()

$\Theta$

T2 -> pop
…

**pending thread pool U**

**derived from concrete state**

- T1’s step may fulfill opr of T1 or threads in U
- T2 checks U to see if its opr has been done (by env)

# Our New Simulation O $\precsim_{\Theta}$ S

$$\begin{tikzcd}
(\mathbf{U_0} \cup A_0, \Sigma_0) \arrow[r, "*"] \arrow[d, dash, "\precsim_{\Theta}"'] & (\mathbf{U_1} \cup A_1, \Sigma_1) \arrow[r, dashed, "*"] \arrow[d, dash, "\precsim_{\Theta}"'] & (\mathbf{U_2} \cup A_2, \Sigma_2) \arrow[r, "*"] \arrow[d, dash, "\precsim_{\Theta}"'] & (\mathbf{U_3} \cup A_3, \Sigma_3) \arrow[d, dash, "\precsim_{\Theta}"'] \\
(O_0, \sigma_0) \arrow[r] & (O_1, \sigma_1) \arrow[r, dashed] & (O_1, \sigma_2) \arrow[r] & (O_2, \sigma_3)
\end{tikzcd}$$

**Env. steps**

U maps a thread ID t to a pending opr. A or •

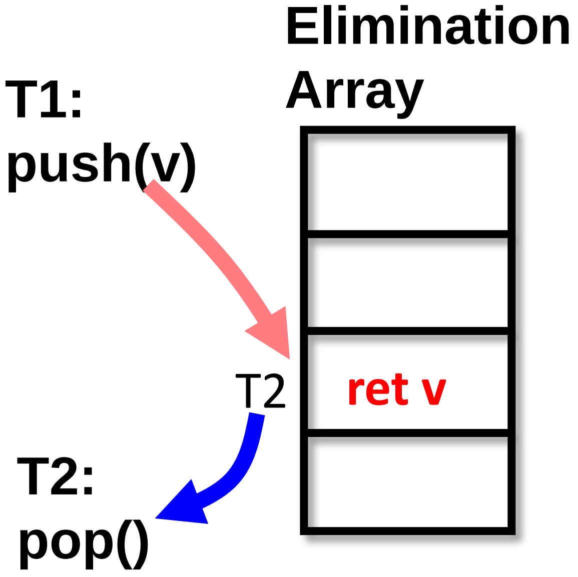
$$U_i \cup A_i \;=\; \Theta(\sigma_i)$$

Support non-fixed LPs: thread's LP can be in either $\longrightarrow$ or $\dashrightarrow$

$$\begin{tikzcd}
(U \cup push(v), \Sigma) \arrow[r, "push\ by\ T1"] \arrow[d, dash, "\precsim_{\Theta}"'] & (U \cup \{\bullet\}, \Sigma') \arrow[r, "pop\ by\ T2"] & (U' \cup \{\bullet\}, \Sigma'') \arrow[d, dash, "\precsim_{\Theta}"'] \\
\sigma \arrow[rr, "T1\ update\ the\ slot"'] & & \sigma'
\end{tikzcd}$$

**U(T2) = pop**

**U'(T2) = ●**



**Elimination Array**

**T1: push(v)**

T2

**ret v**

**T2: pop()**

Soundness:

If $O \precsim_{\Theta} A$ for some $\Theta$, then $O \leq_{\mathbf{lin}} A$

What's more:

Hoare-style syntactic logic for lin.

**http://home.ustc.edu.cn/~lhj1018/lin.pdf**

# Conclusion

- A new simulation O $\precsim_{\Theta}$ A for linearizability proof
  - Sim → refinement → linearizability
  - Supports non-fixed LP
    - HSY elimination-based stack, lazy set, etc
    - First refinement-based proof for HSY elimination-based stack
  - A program logic for syntactic verification

- Another dimension of complexity
  - LP depends on future
  - May need backward simulation (leave as future work)

# Thank you!