

**Note: Information on these slides is not updated during the course,  
please refer to the course webpage for news.**

# **TDDD56**

## **Lesson 1: Lab series intro**

**August Ernstsson**

august.ernstsson@liu.se

# Staff

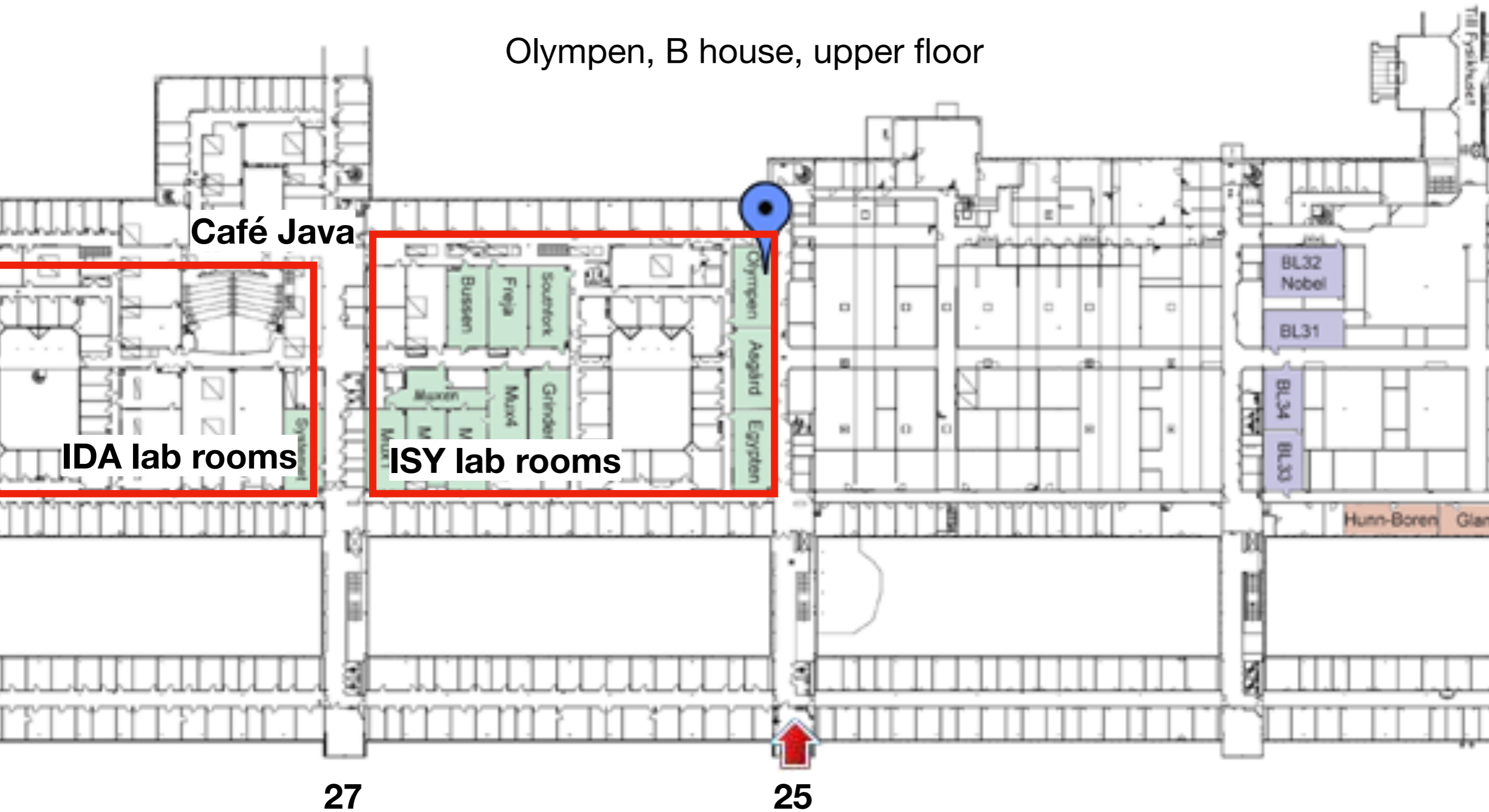
- **August Ernstsson**, course/lab assistant, lessons  
Contact person for CPU labs  
[august.ernstsson@liu.se](mailto:august.ernstsson@liu.se)
- **Ingemar Ragnemalm**, lab assistant, GPU lectures  
Contact person for GPU labs  
[ingemar.ragnemalm@liu.se](mailto:ingemar.ragnemalm@liu.se)

# Lab groups

- Two main groups: **A** and **B**
  - Different schedule slots.
- Subgroups of two students. Work in pairs.
- Each session will be attended by one assistant.
  - For the latter half (GPU part), Ingemar takes over supervision of group A.

# Lab room

Olympen, B house, upper floor



# Lab equipment

- Olympen has special lab computers for the course
  - May be able to use other IDA systems or own equipment for development, but use **Olympen machines** for performance testing and demonstration.
- 16 seats for groups of 2 students = 32 students at once in room
  - Intel Xeon CPU W-2145
  - 8 cores, 3.70 GHz
  - 16 GiB memory

# Lab schedule

	WebReg	Week		
Responsible: August	CPU	Lab 1	v45	Load Balancing
		Lab 2	v46	Non-blocking Data Structures
		Lab 3	v47	High-level Parallel Programming <i>Lesson 2</i>
Responsible: Ingemar	GPU	Lab 4	v48	CUDA 1
		Lab 5	v49	CUDA 2
		Lab 6	v50	OpenCL

# General info

- **Be prepared** when coming to labs, use time with teachers well!
- Lab compendiums and resources (code skeletons etc.) on course webpage.
- **Ask** if something is unclear.
- **Demonstrate** your solutions and provide answers to any questions asked in lab material, as well as questions asked by assistant.
- No written lab reports, so demonstration is thorough!
- **Both** members of a group should be actively contributing and be prepared to answer questions during demonstration.
- It is allowed to discuss among groups, but don't share solutions. Plagiarism is taken seriously!

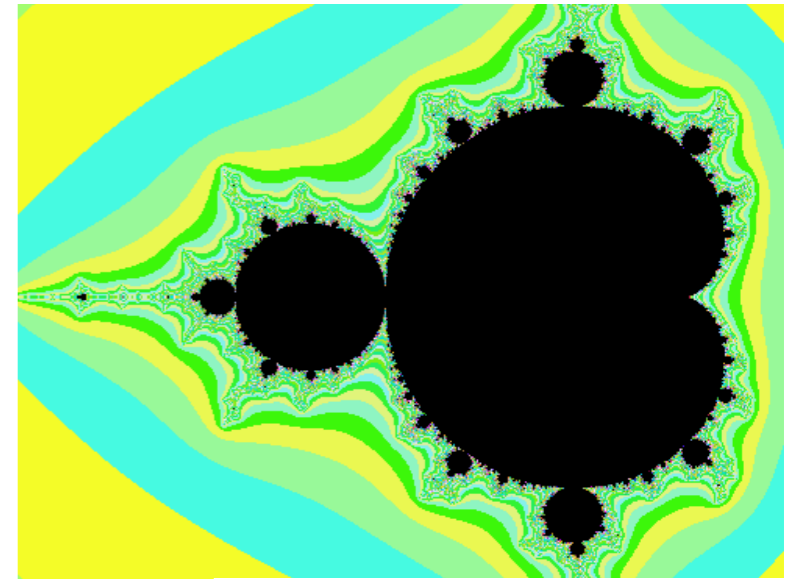
# Information Resources

- Lab instructions
- Source files
- TDDD56 lecture, lesson slides



# Lab 1 – Load Balancing

- Working with threads (Pthreads) on multicore CPU
- Mandelbrot fractal image generation

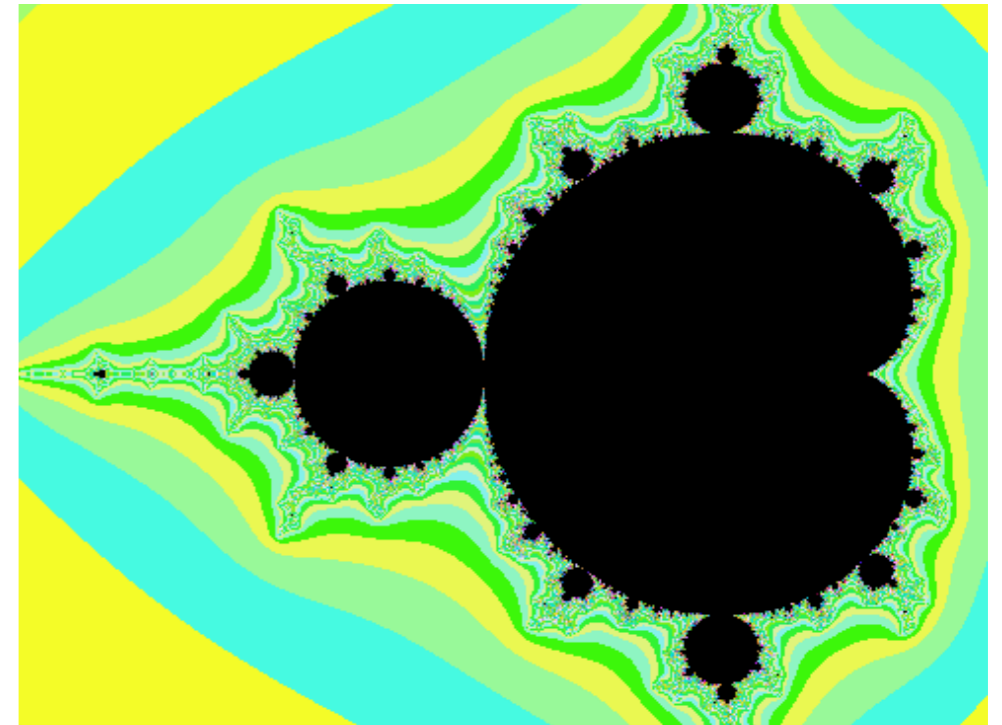


- Test if a complex number is in the Mandelbrot set  $f_c(z) = z^2 + c$
- For those interested in the maths, check out:
  - [https://en.wikipedia.org/wiki/Mandelbrot\\_set](https://en.wikipedia.org/wiki/Mandelbrot_set)
  - <https://www.youtube.com/watch?v=NGMRB4O922I>

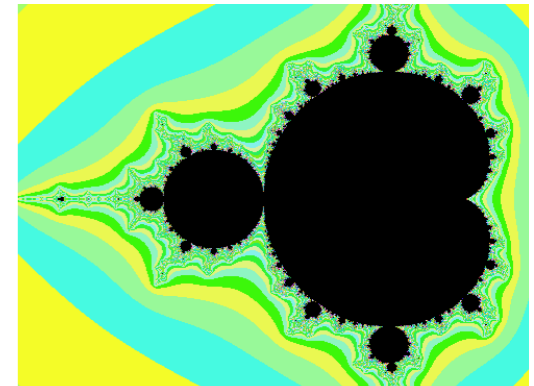
# Mandelbrot Algorithm

```
int is_in_Mandelbrot(float Cre, float Cim)
{
    int iter;
    float x=0.0, y=0.0, xto2=0.0, yto2=0.0, dist2;

    for (iter = 0; iter <= MAXITER; iter++)
    {
        y = x * y;
        y = y + y + Cim;
        x = xto2 - yto2 + Cre;
        xto2 = x * x;
        yto2 = y * y;
        dist2 = xto2 + yto2;
        if ((int)dist2 >= MAXDIV)
            break; // diverges
    }
    return iter;
}
```

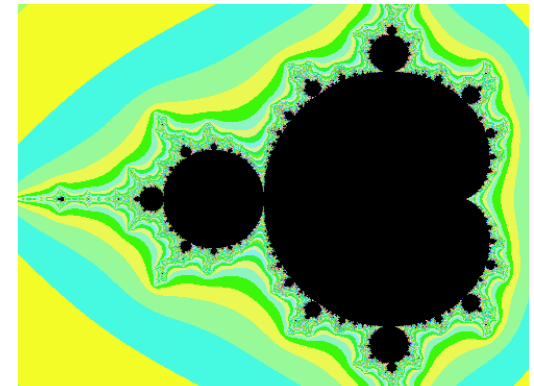


# Load Balancing



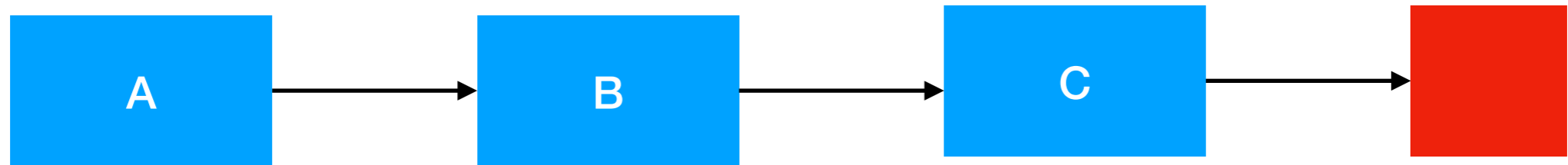
- Each image pixel is an **independent** unit of work
  - => embarrassingly parallel!
- However, all pixels are not **equal amount** of work!
  - Load balancing becomes a problem.

# Lab 1



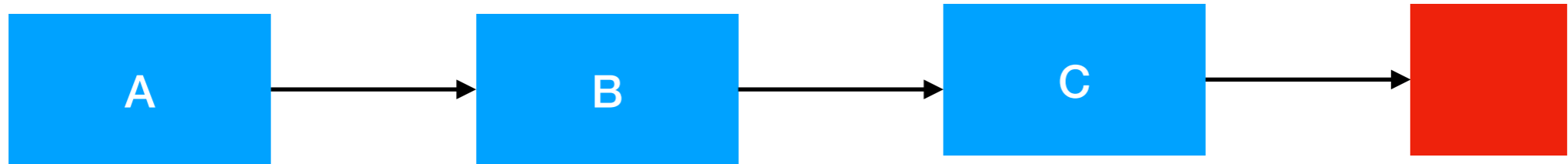
- Goal for the lab:
  - Implement a solution with near-equal load
  - Try different approaches
  - Utilize properties of the domain
    - How well will your solution work in a general case?

# Lab 2: Non-blocking Stack



- Working with Pthreads on multicore CPU
- Using atomic operations (CAS)
- Implementing efficient parallel data structures

# Unbounded Stacks



- Stacks implemented as **linked lists**
- Non-blocking: **NO LOCKS!**
- **Push** and **Pop** operations with atomic instructions

# Compare-and-Swap

- Do atomically:
  - If *pointer*  $\neq$  *old pointer*: do nothing  
Else: swap *pointer* to *new pointer*
- Typically used only for compare + assign, no swap

```
CAS(void** pointer, void* old, void* new)
{
    atomic {
        if(*pointer == old)
            *pointer = new;
    }
    return old;
}
```

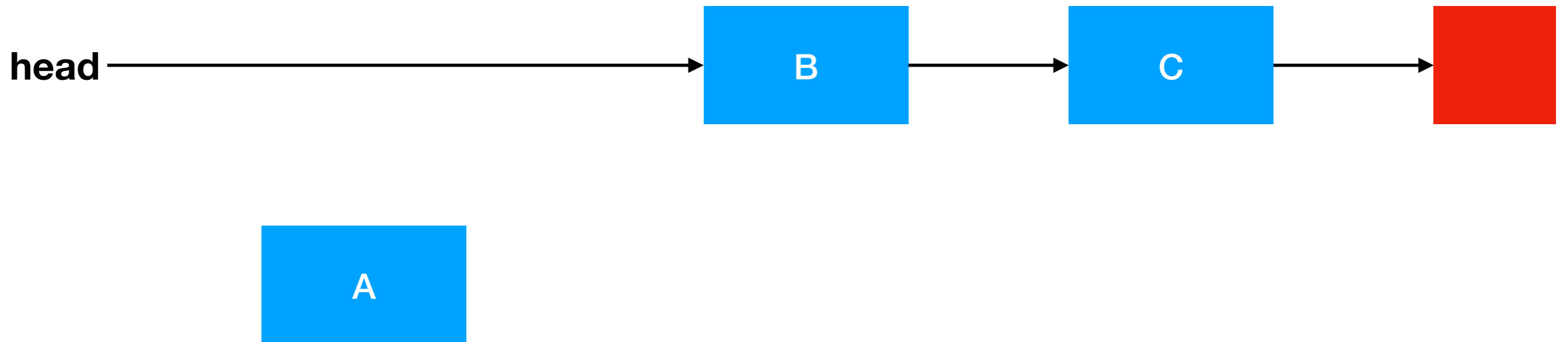
# CAS for Stack

- Push
  - Keep track of old head
  - Set new elements next pointer to old head
  - **Atomically:**
    - Compare current head with saved old head
    - If still equal, set list head to new element

```
do {  
    old = head; elem.next = old;  
} while(CAS(head, old, elem) != old);
```

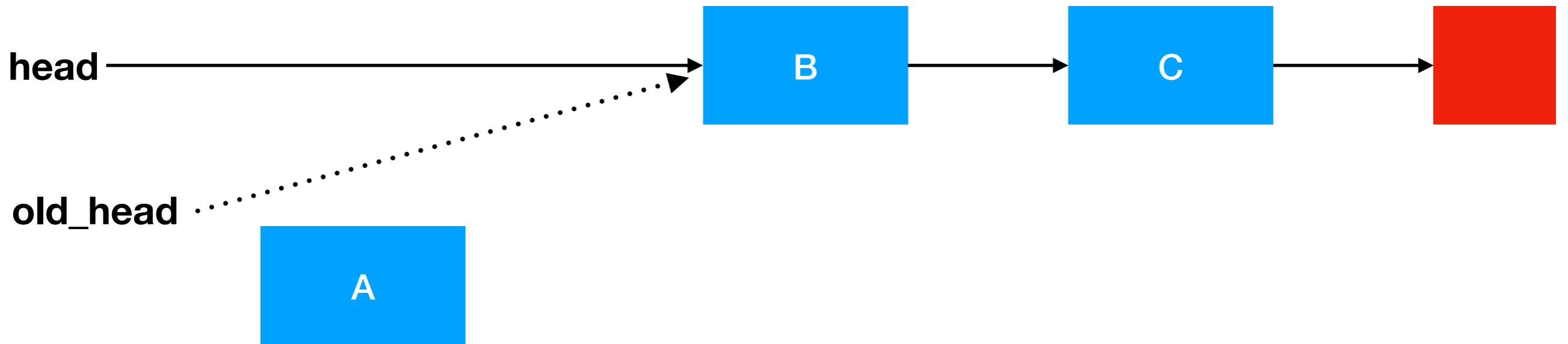


# CAS push



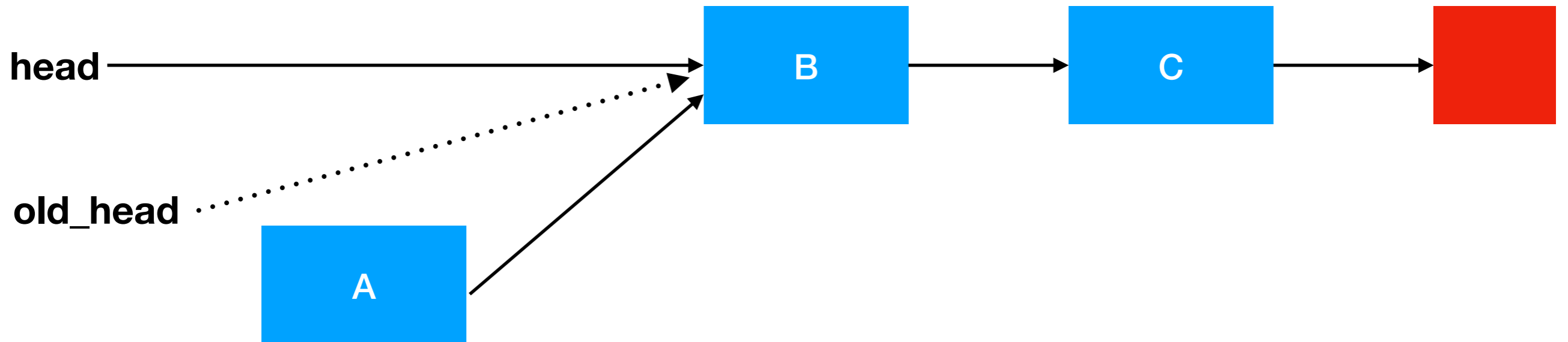
# CAS push

Keep track of old head



# CAS push

set new elements next pointer to old head

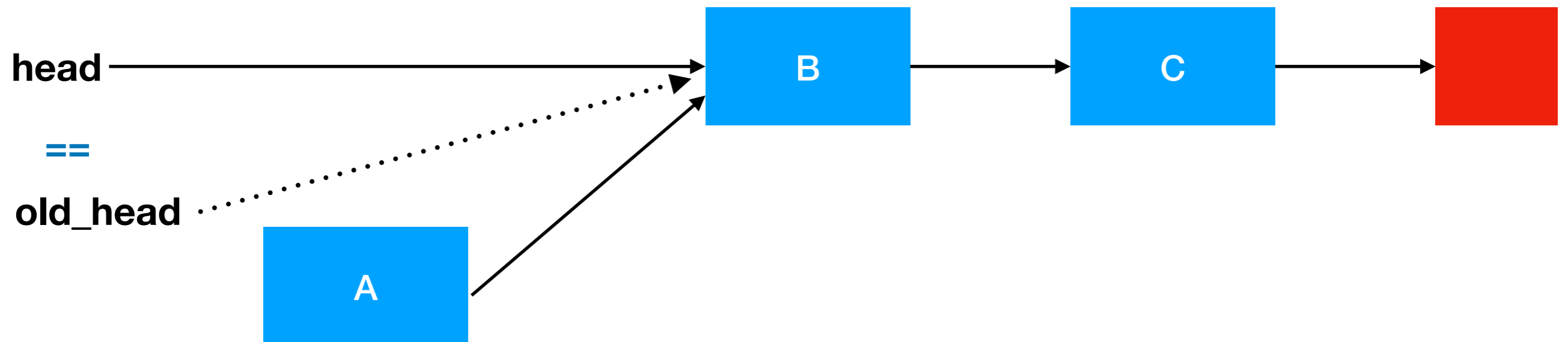


# CAS push, success

start atomic operation

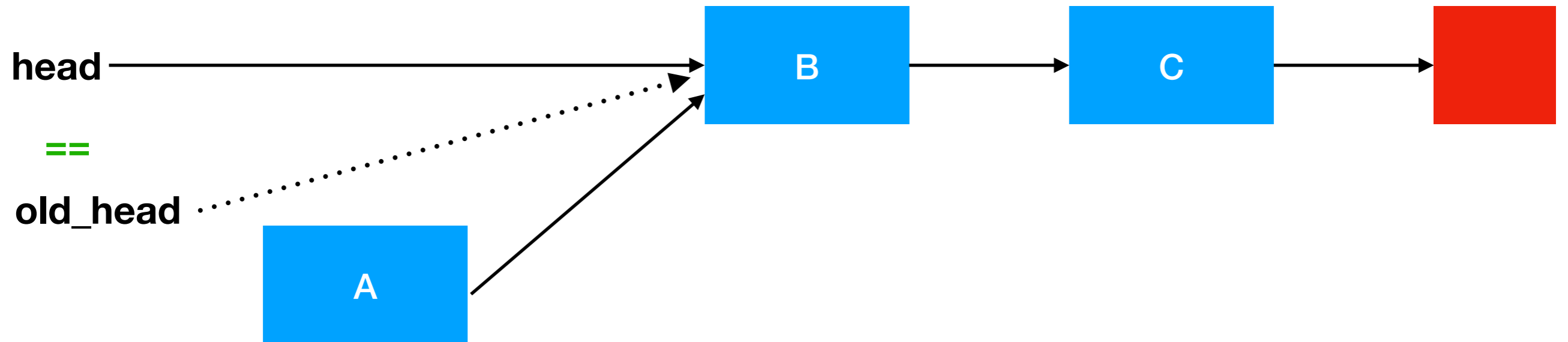
---

still equal?



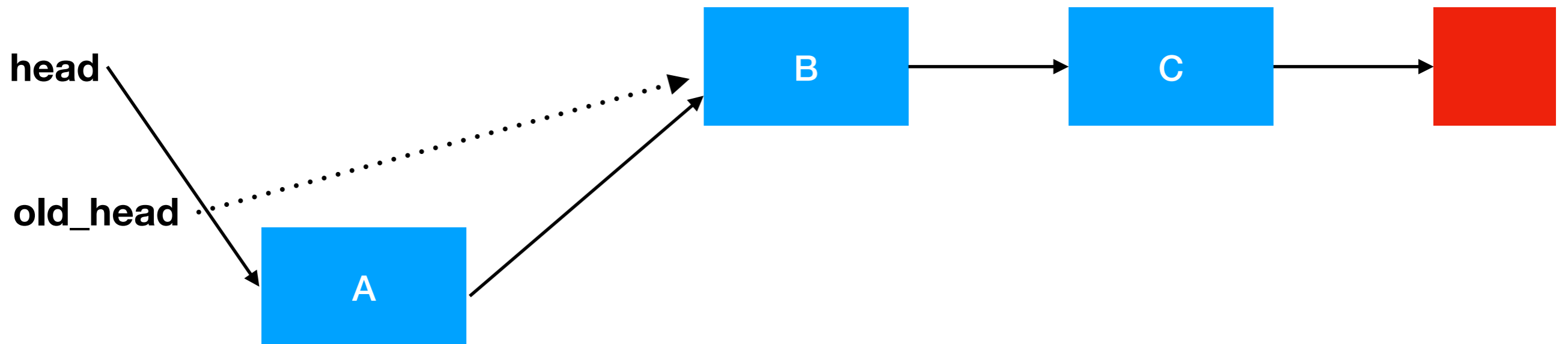
# CAS push, success

still equal? **YES**



# CAS push, success

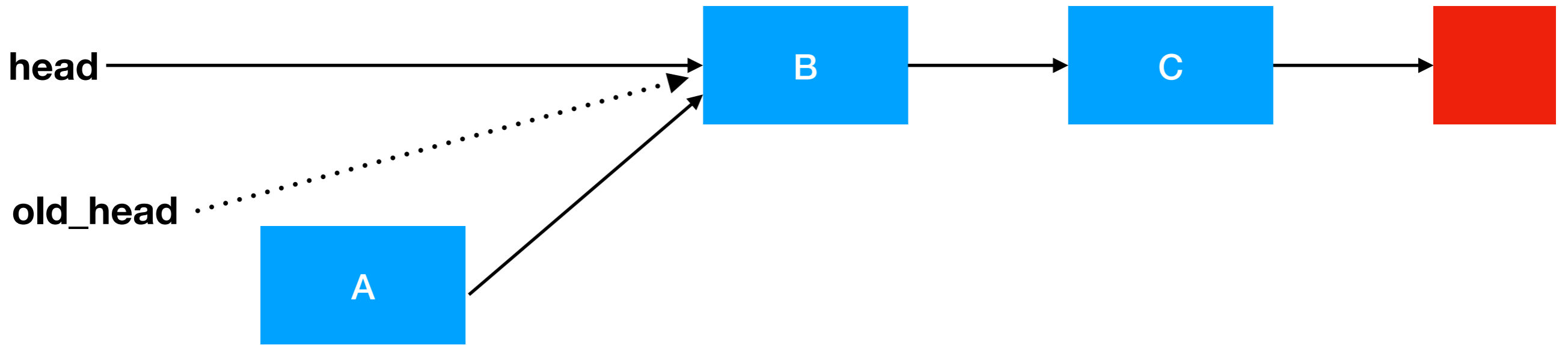
set list head to new element



---

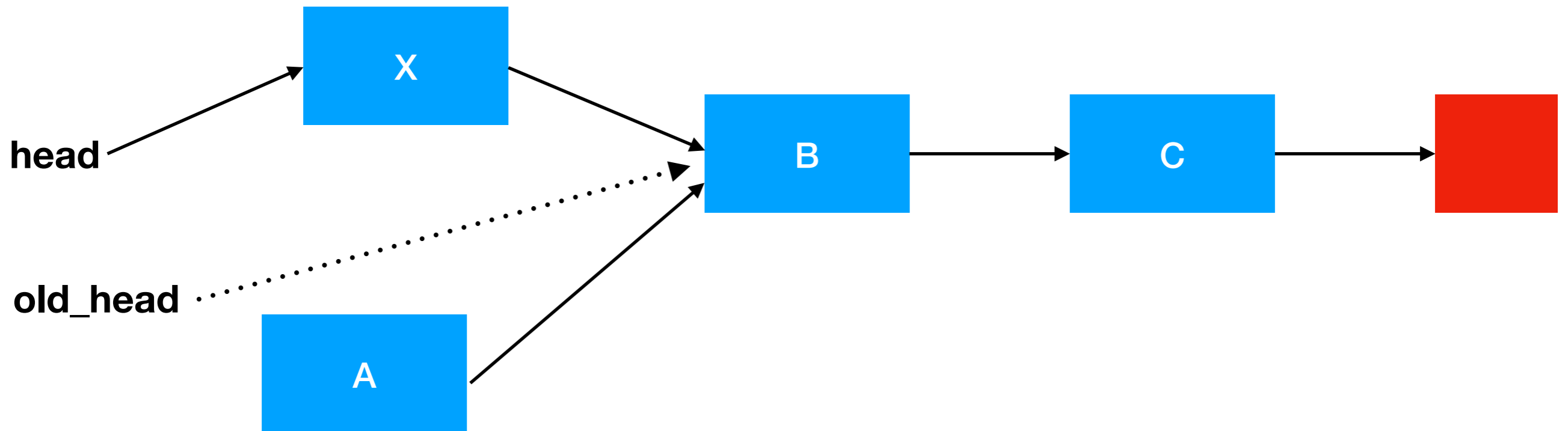
end atomic operation

# CAS push



# CAS push

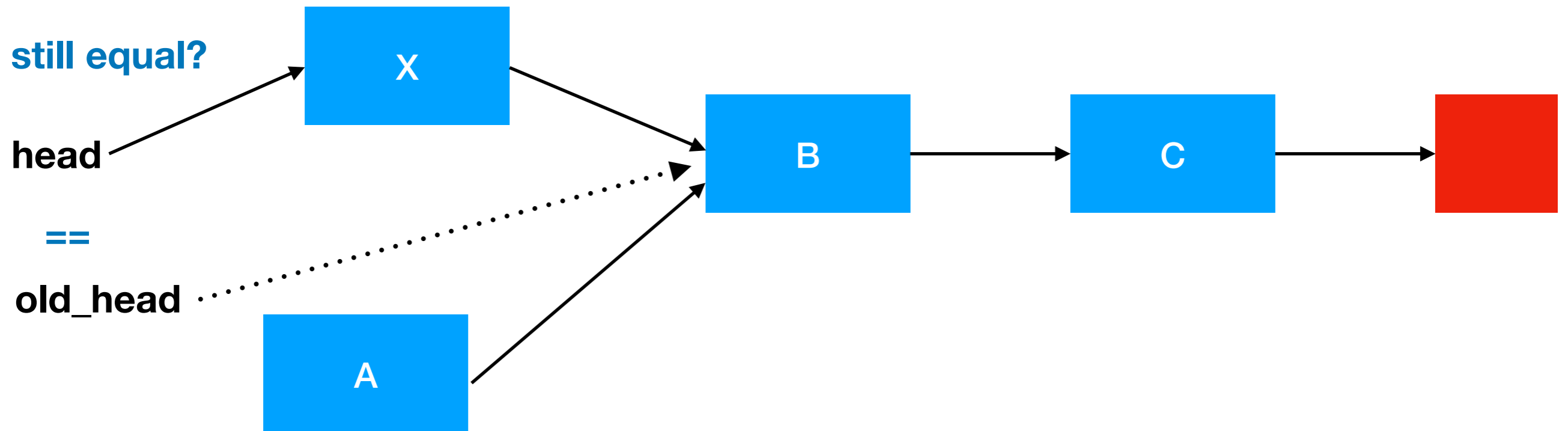
Another thread pushed X!



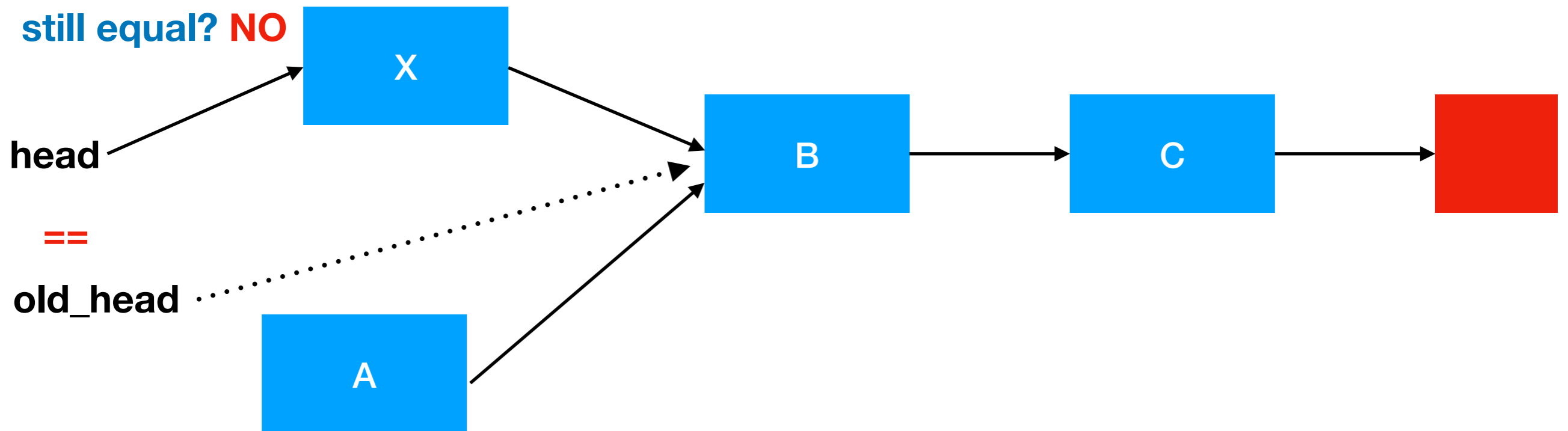


# CAS push, failure

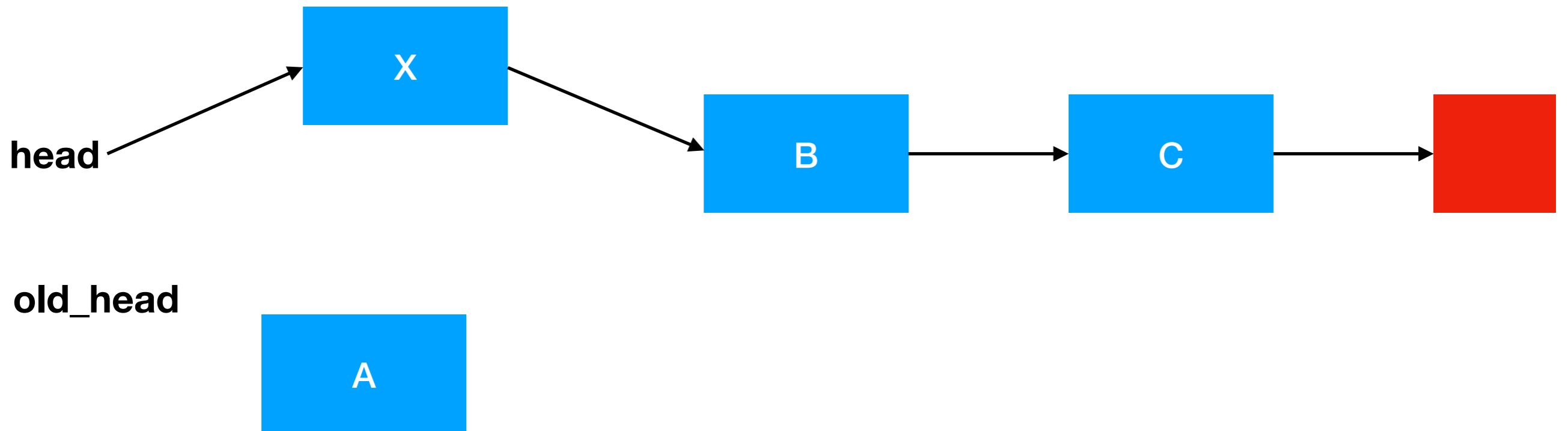
start atomic operation



# CAS push, failure



# CAS push, failure



---

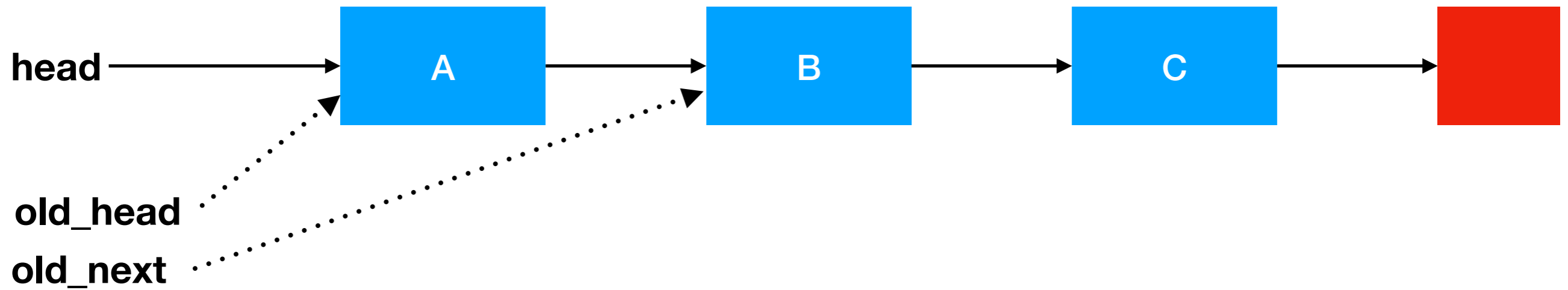
end atomic operation

# ABA problem

- List elements can be re-used
  - Memory is limited, pointers can reappear => still low risk
  - Improve performance by keeping a pool of unused list elements => much greater risk of re-use!
- What if a list element is
  - popped,
  - pushed (with new content),
  - during the non-atomic part of a Pop?

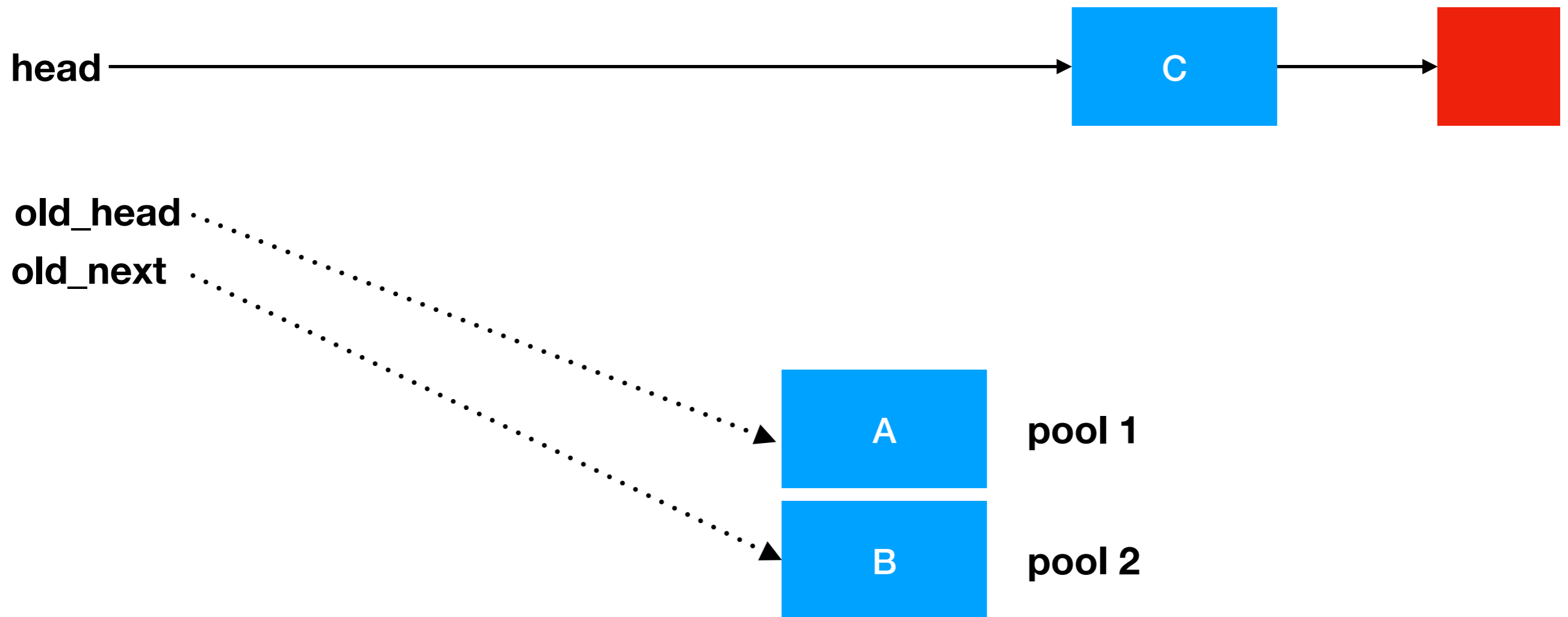
# ABA problem

thread 0 starting pop



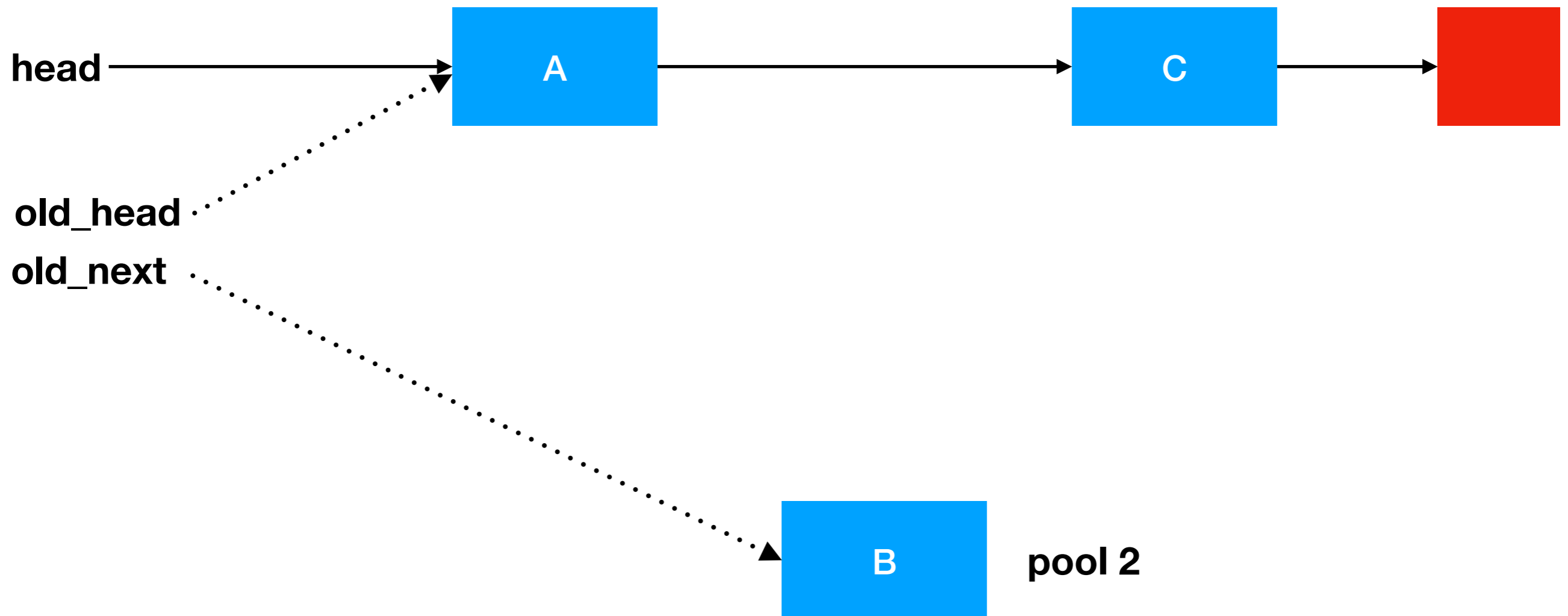
# ABA problem

thread 1 pops A, thread 2 pops B



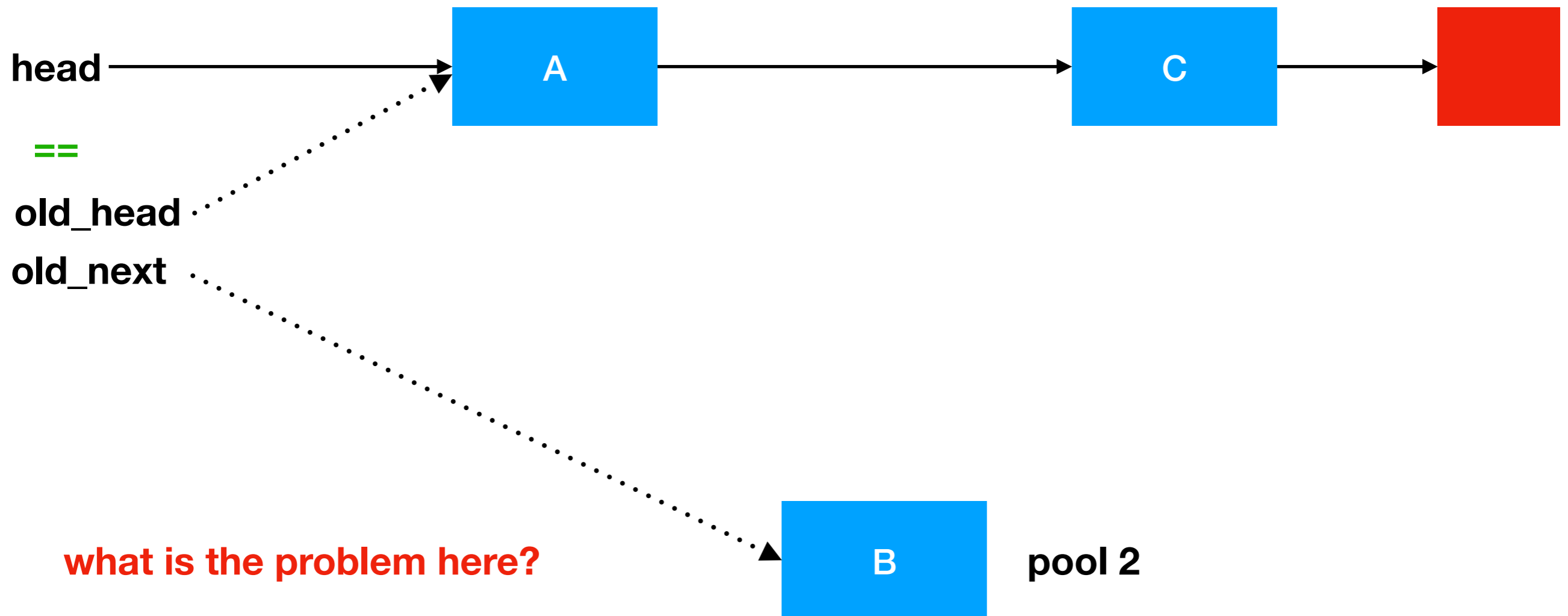
# ABA problem

thread 1 pushes A



# ABA problem

thread 0 resumes pop; enters atomic region:  
compares head and old\_head

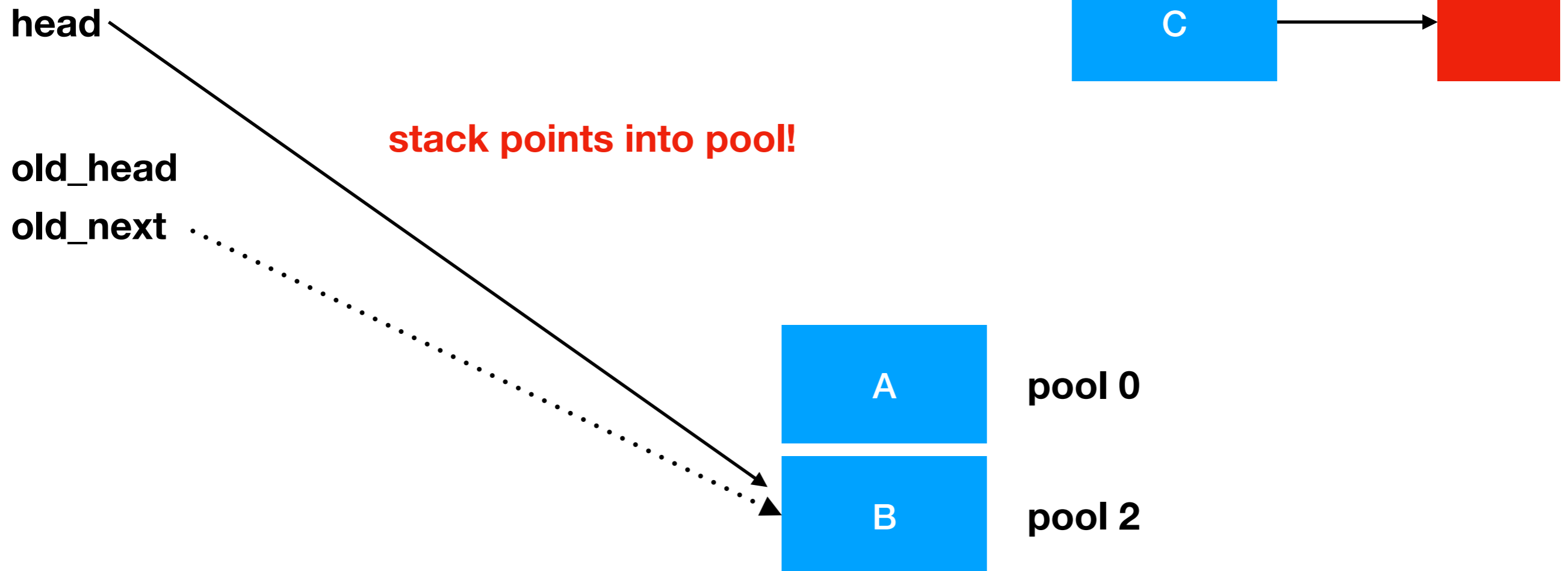




# ABA problem

A is popped, setting head to old\_next (B)

elements have leaked!



# Lab 2

- Goal for the lab:
  - Implement non-blocking unbounded stack
  - Use atomic operations
  - Study the ABA problem
    - Detect it or force it to occur
    - Can it be avoided?



**This is the  
main goal of  
the lab!**

# Questions?