TDDD56
Lesson 1: Lab Series Intro

Sehrish Qummar

sehrish.qummar@liu.se

Staff

- **Sehrish Qummar**, course/lab assistant, lessons
  Contact person for CPU labs
  sehrish.qummar@liu.se

- **Ingemar Ragnemalm**, lab assistant, GPU lectures
  Contact person for GPU labs
  ingemar.ragnemalm@liu.se

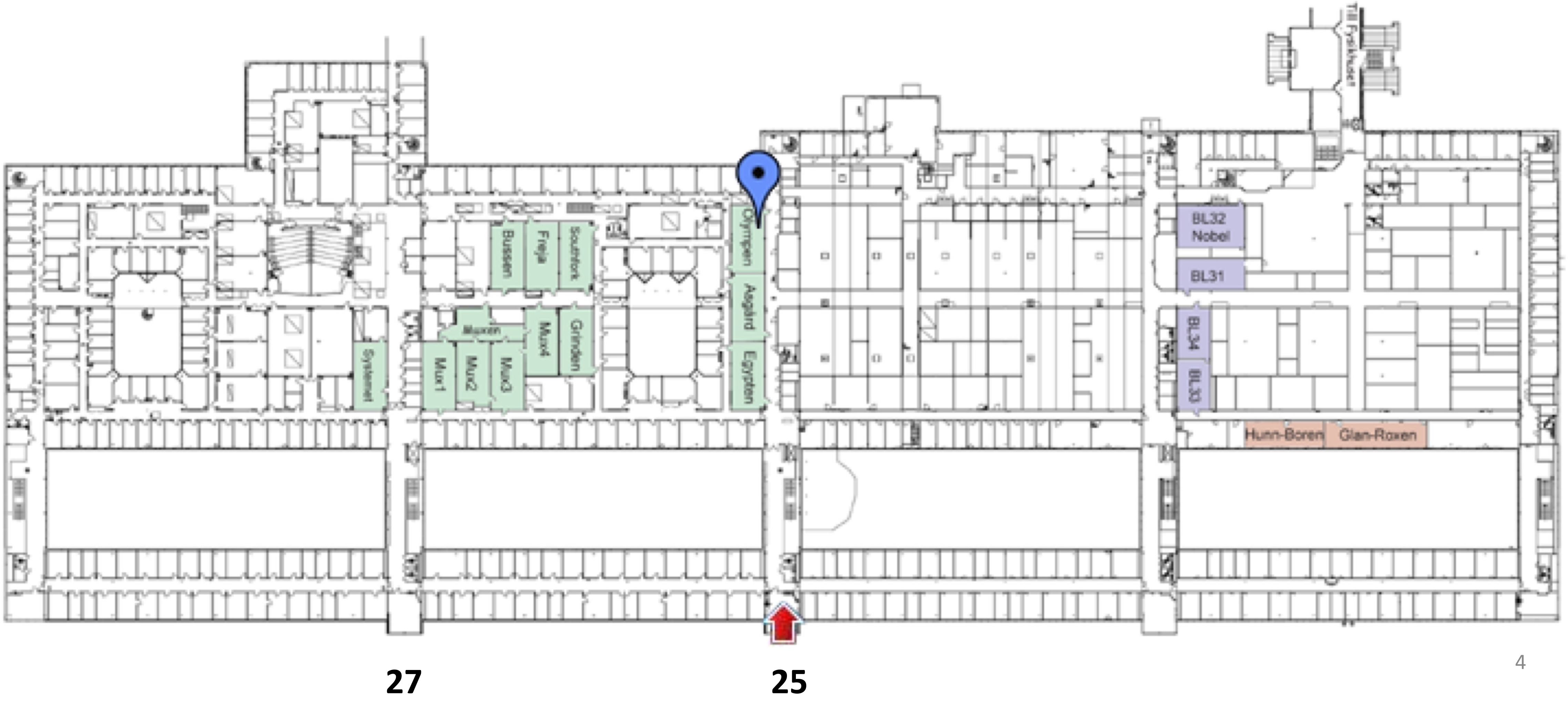
- **Sajad Khosravi**, lab assistant
  sajad.khosravi@liu.se

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
    - Intel Xeon CPU W-2145
    - 8 cores, 3.70 GHz
    - 16 GiB memory
    - 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

Lab Schedule

	WebReg	Week		
CPU	Lab 1	v45	Load Balancing	Lesson 1
	Lab 2	v46	Non-Blocking Data Structures	Lesson 1
	Lab 3	v47	High level parallel programming	Lesson 2
GPU	Lab 4	v48	CUDA 1	
	Lab 5	v49	CUDA 2	
	Lab 6	v50	OpenCL	

General Information

- **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!
    - Time out **15 min**
- **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!

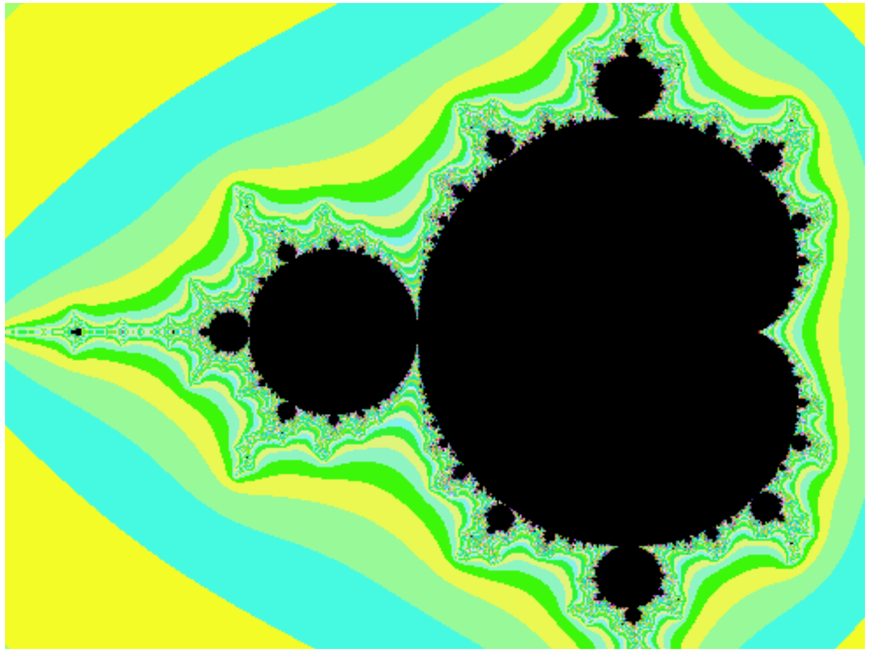
Information Resource

- Lab instruction
- Source files
- TDDD56 lectures, lesson slides

Lab 1

Lab 1 – Load Balancing

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

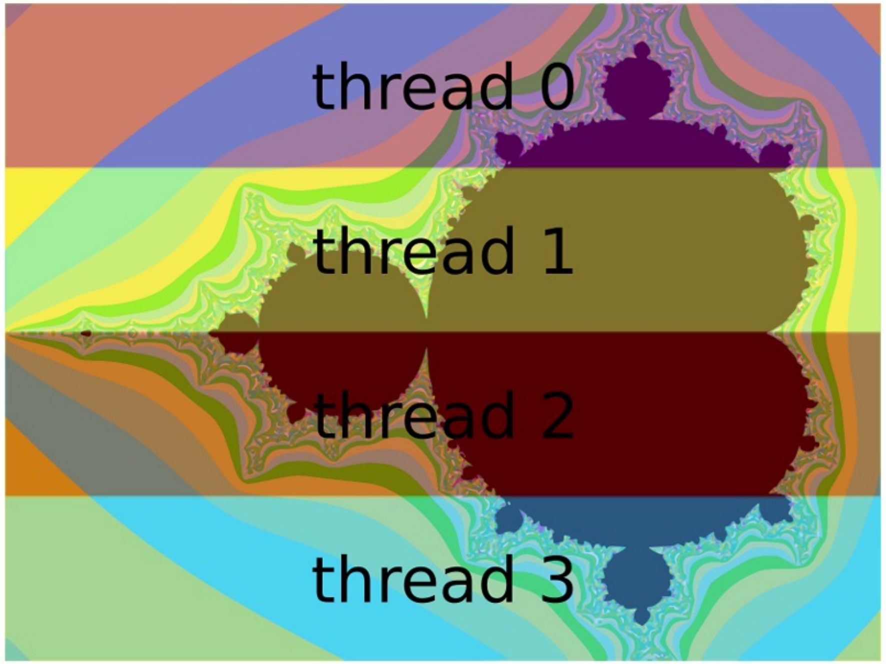


$$f_c(z) = z^2 + c$$

https://en.wikipedia.org/wiki/Mandelbrot_set
https://www.youtube.com/watch?v=NGMRB4O922I

Lab 1 – Load Balancing

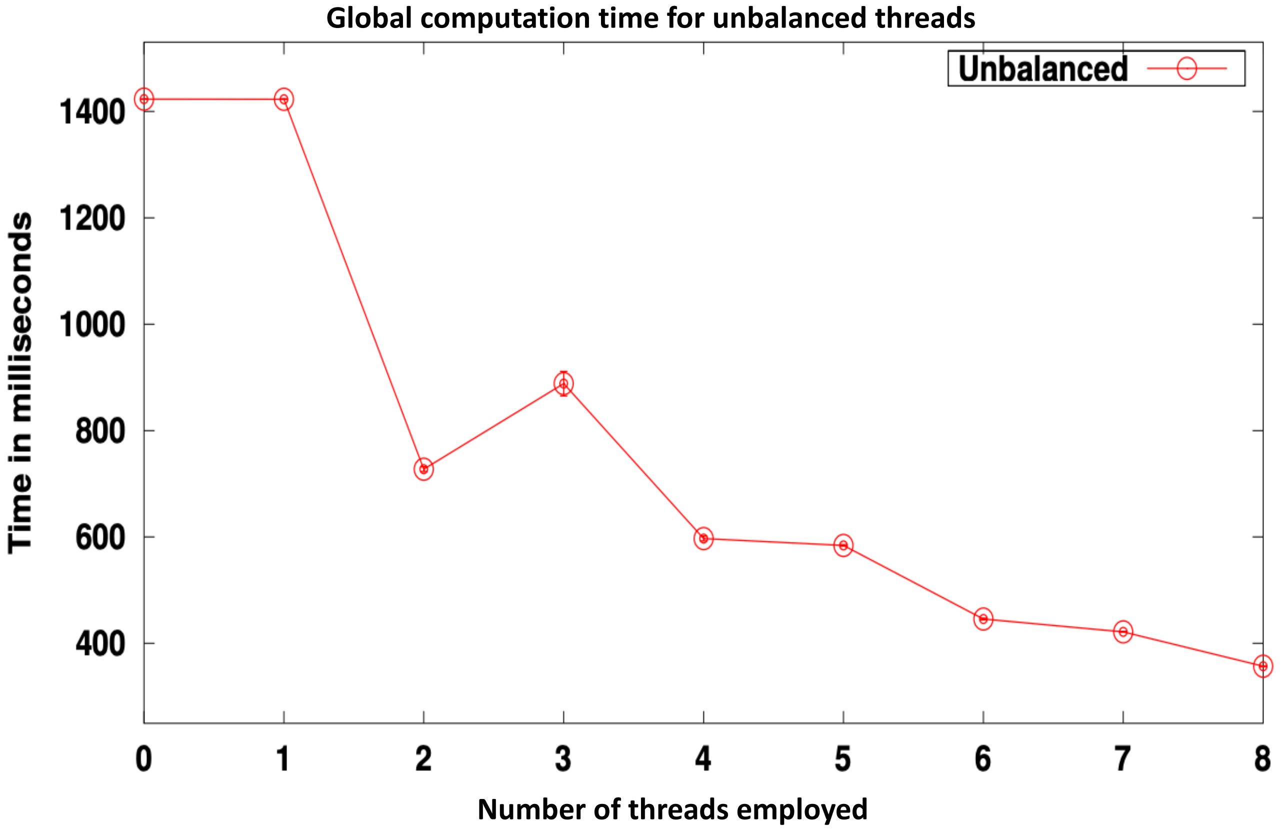
- Goals for the lab:
  - Implement a solution with **near-equal load**
  - Try different approaches
  - Utilize properties of the domain
    - How well will your solution work in a general case?
- Three implementations need to be done:
  - LOADBALANCE=0 (Naïve approach)
  - LOADBALANCE=1
  - LOADBALANCE=2



$$f_c(z) = z^2 + c$$

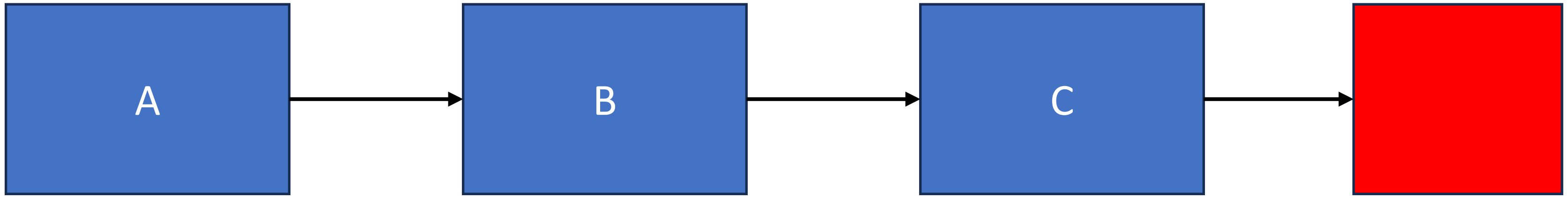
Lab 1 – Load Balancing

- Test your code
  - With maximum 16 threads
  - Compare balanced and unbalanced results



Lab 2

Lab 2 Non-blocking Stack



- Working with Pthreads on multicore CPU
- Using atomic operations (CAS)
- Implementing efficient parallel data structures
- Stacks implemented as **linked lists**
- Non-blocking: **NO LOCKS**!
- **Push** and **Pop** operations with atomic instructions

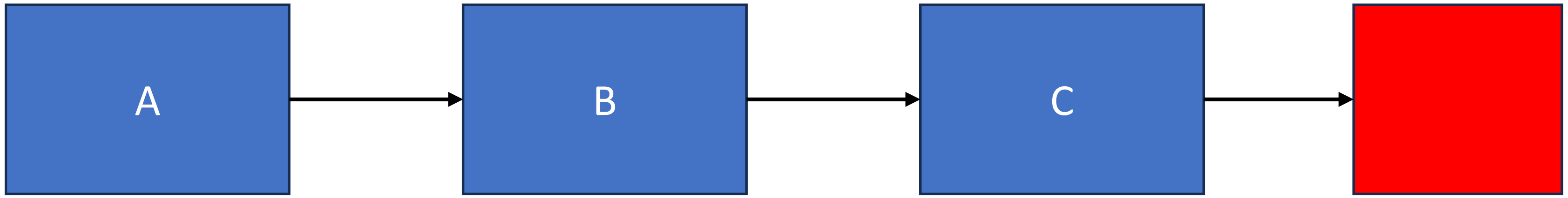
Compare and Swap

- Do atomically:
  - If *pointer*!= *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;
  }
```

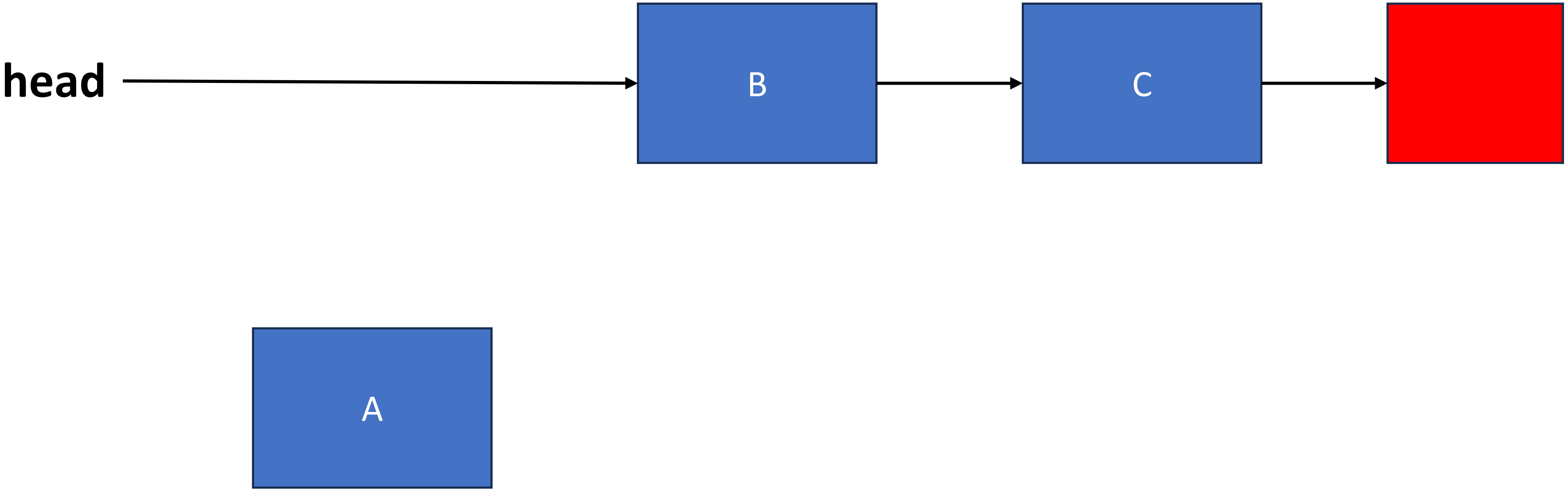
CAS for Stack



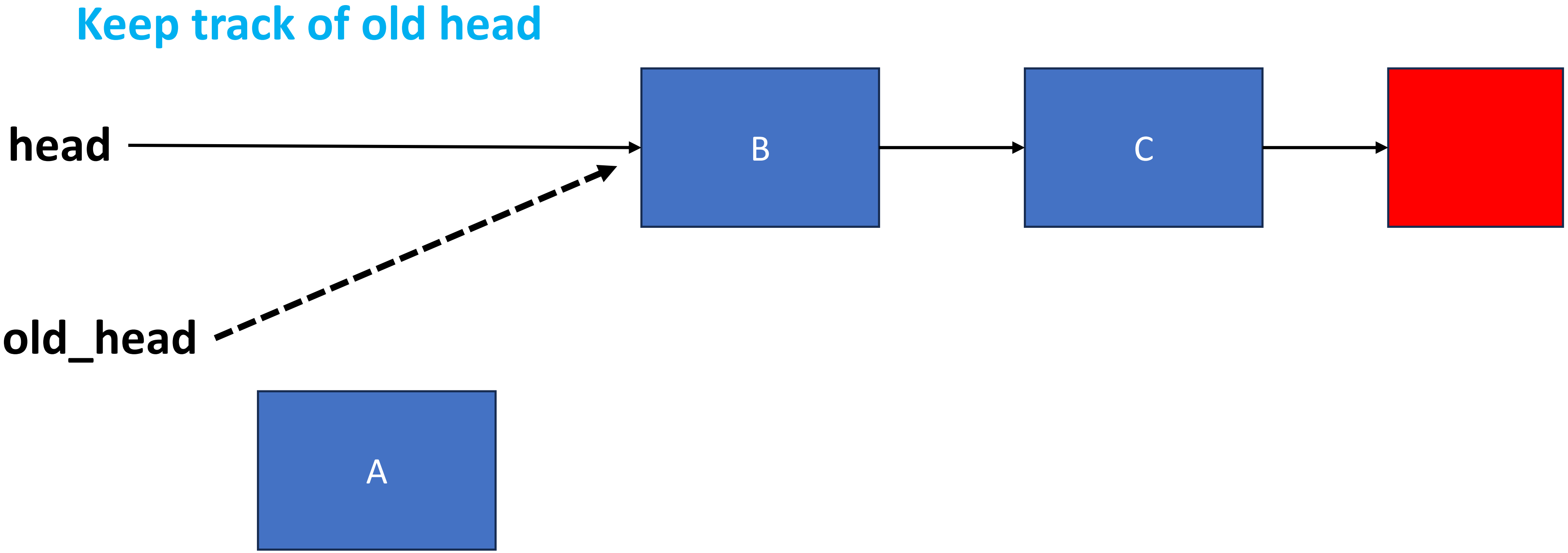
- Push
  - Keep track of old head
  - Set new elements next pointer to old head
  - **Atomically**:
    - Compare current head with saved old head
    - If still equal, set list head to new element

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

CAS push

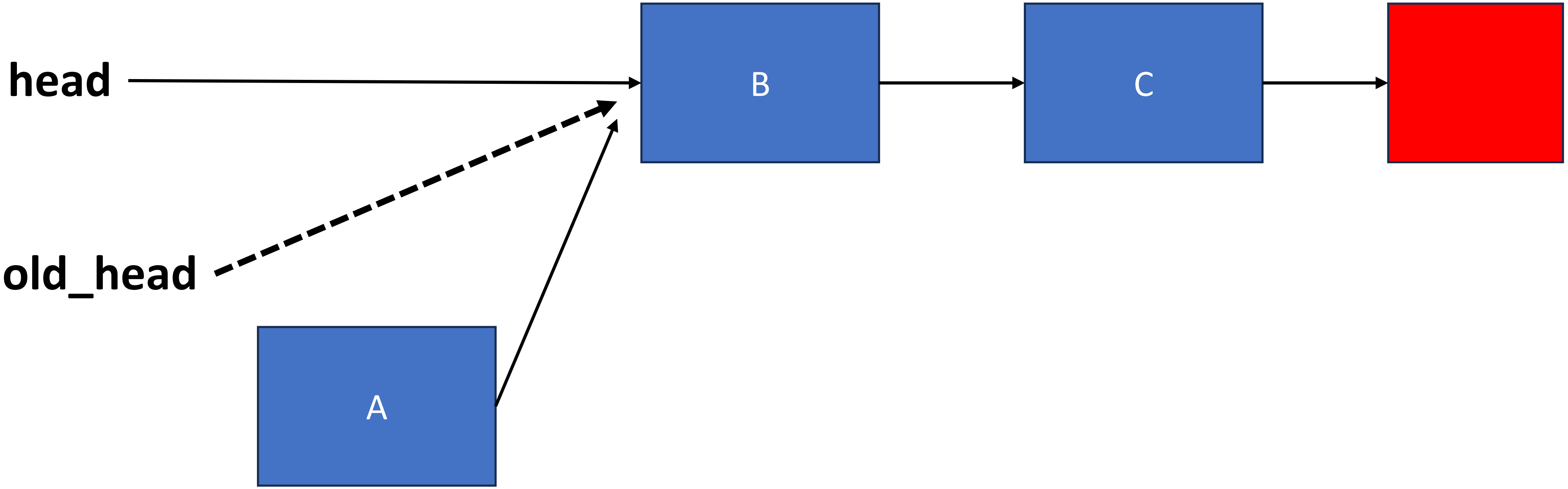


CAS push



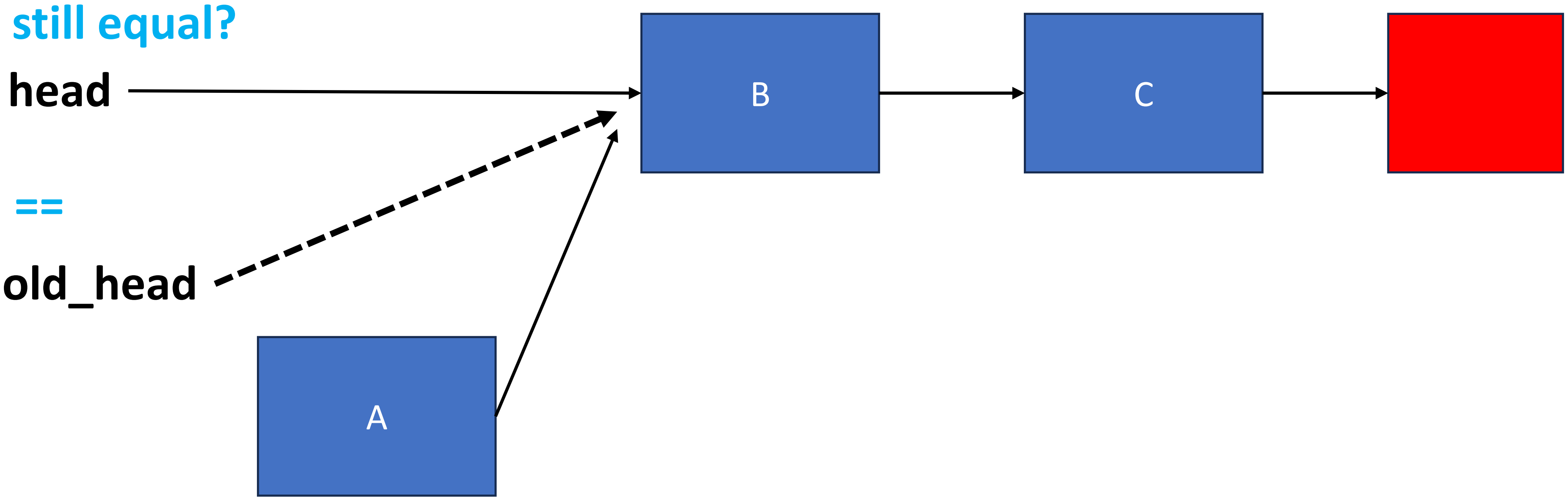
CAS push

set new elements next pointer to old head

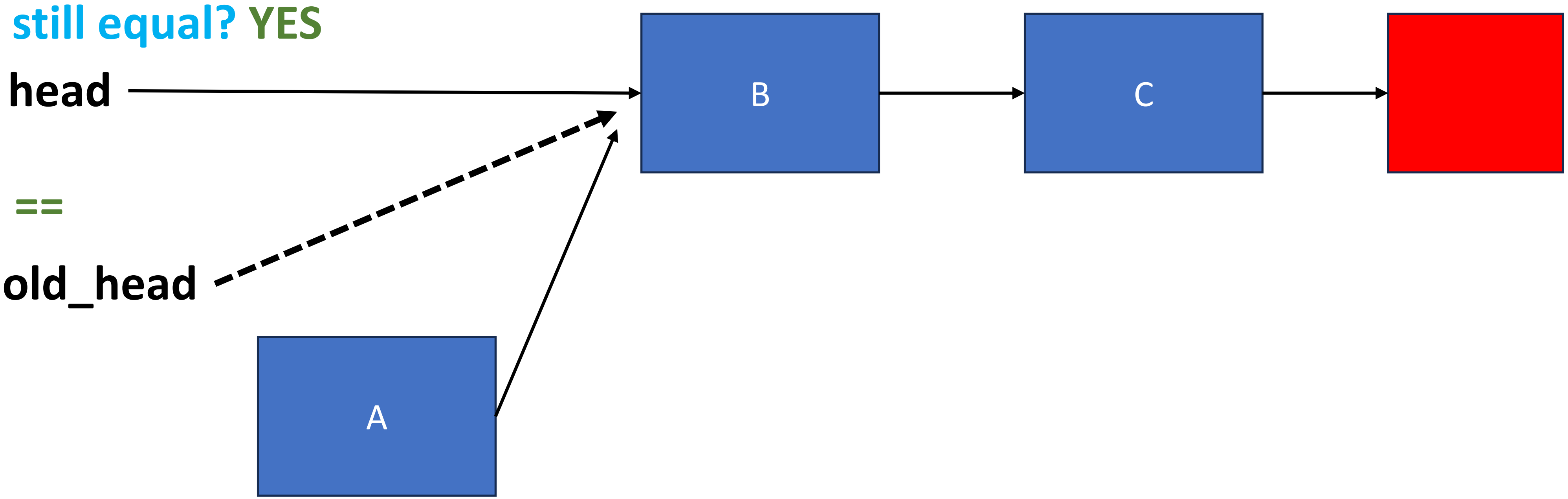


CAS push, success

start atomic operation

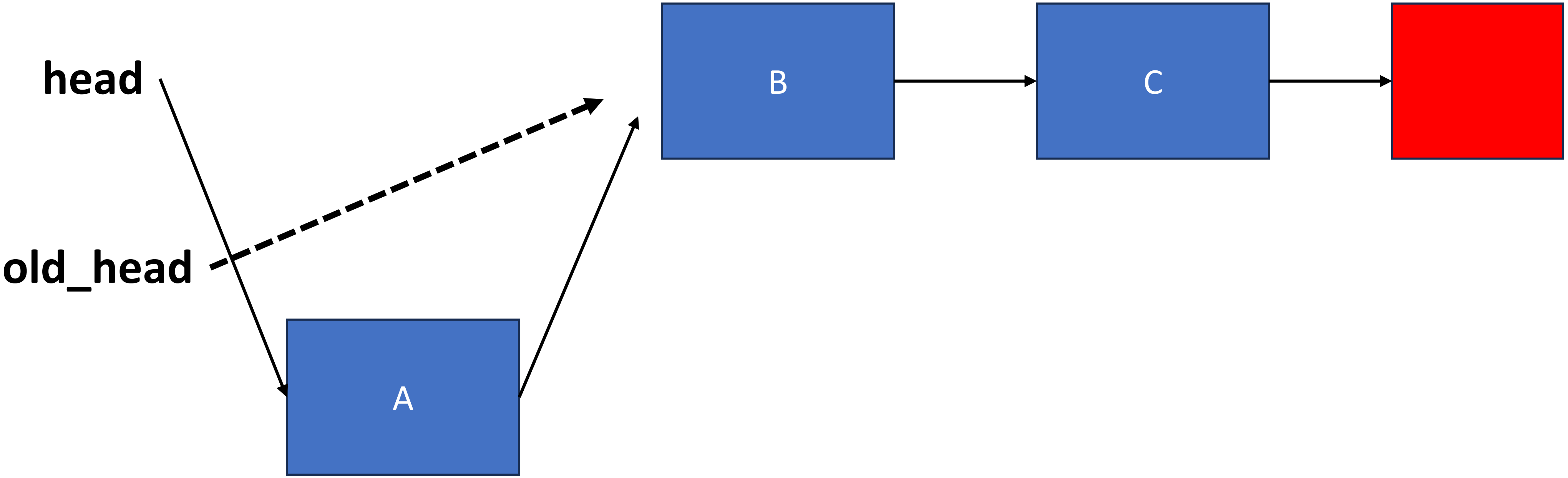


CAS push, success



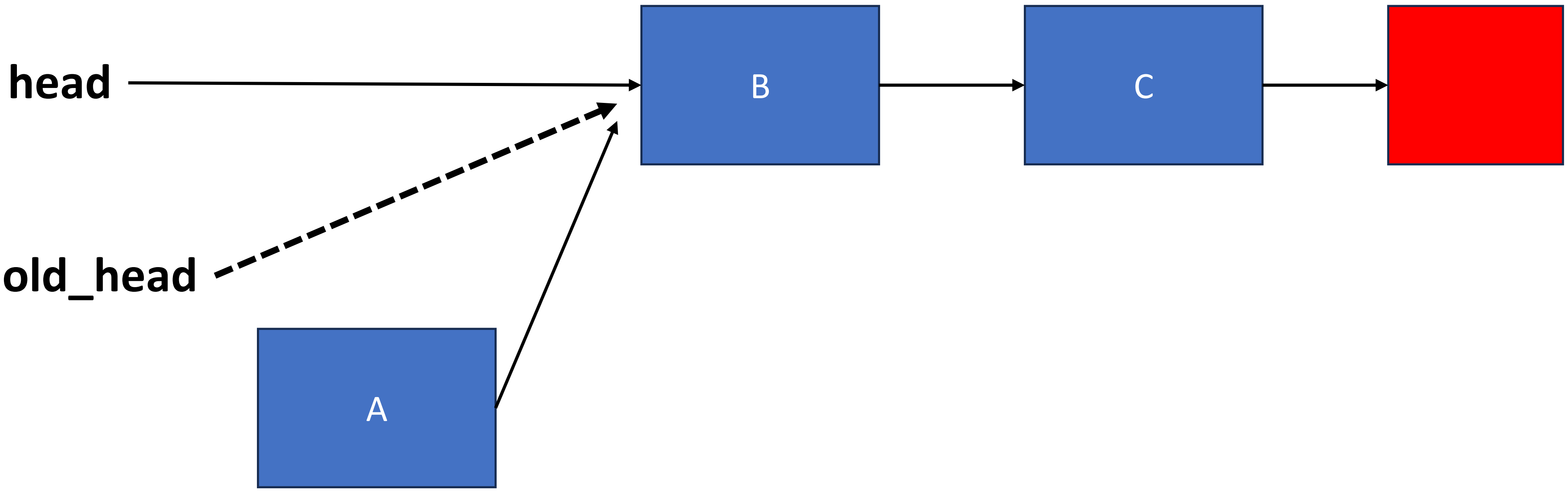
CAS push, success

Set list head to new element

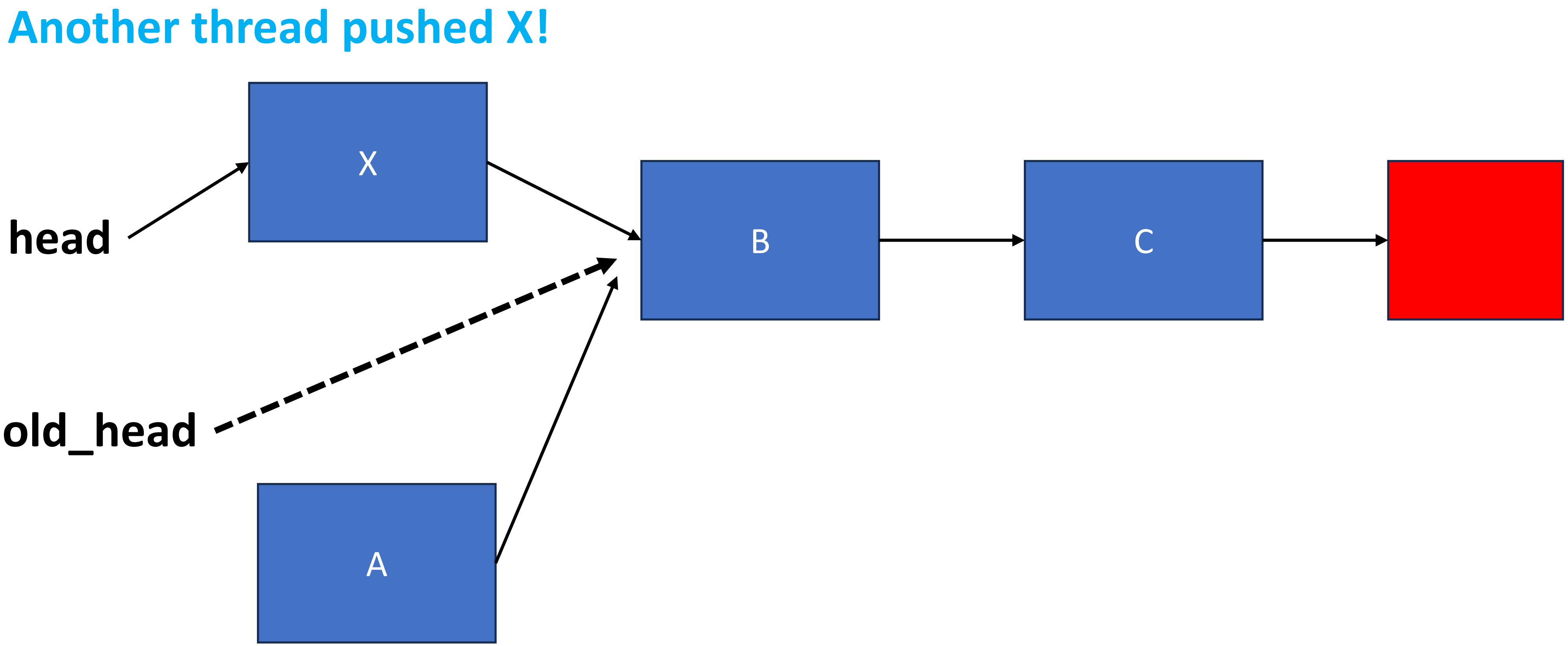


end atomic operation

CAS push

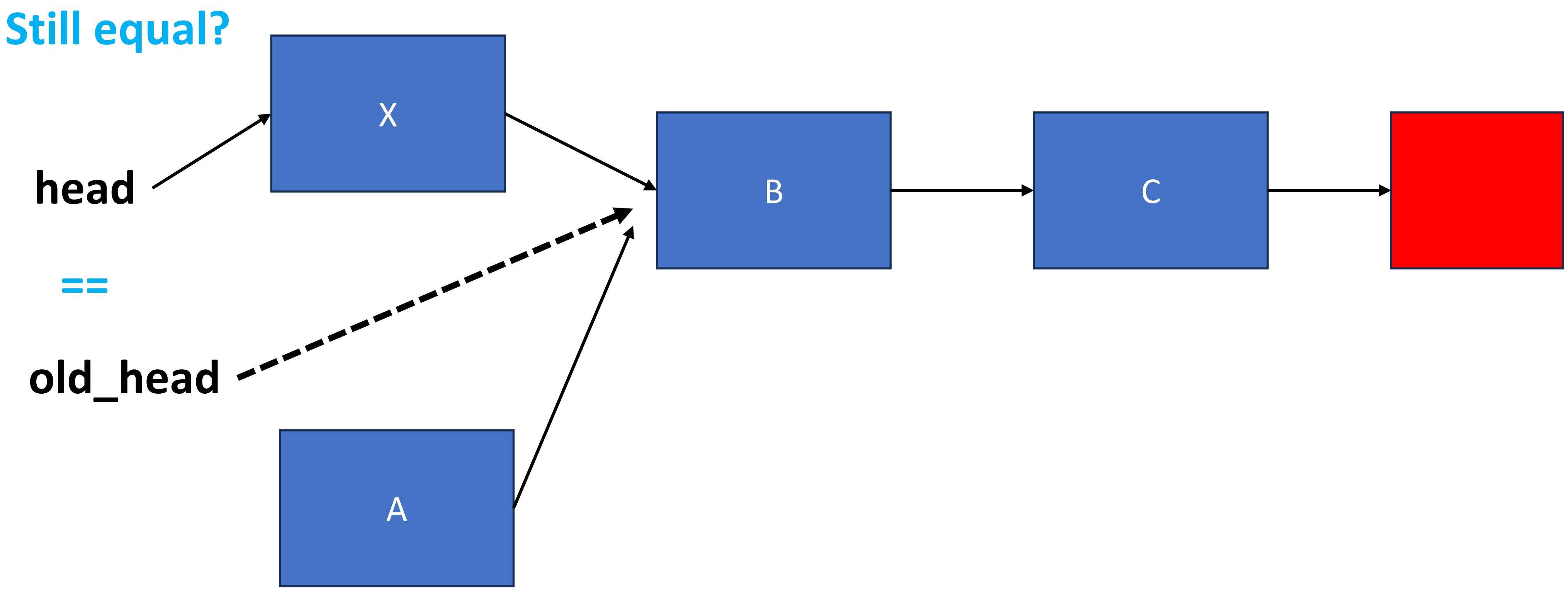


CAS push

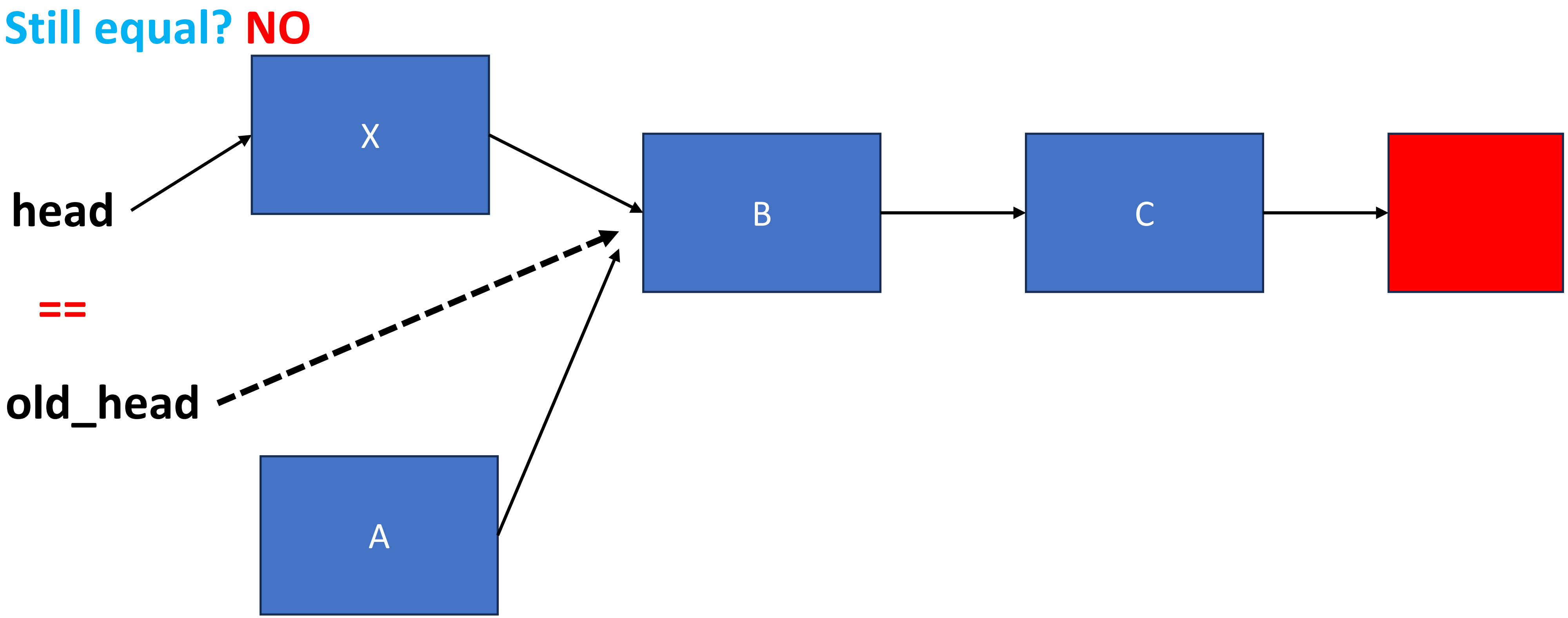


CAS push, failure

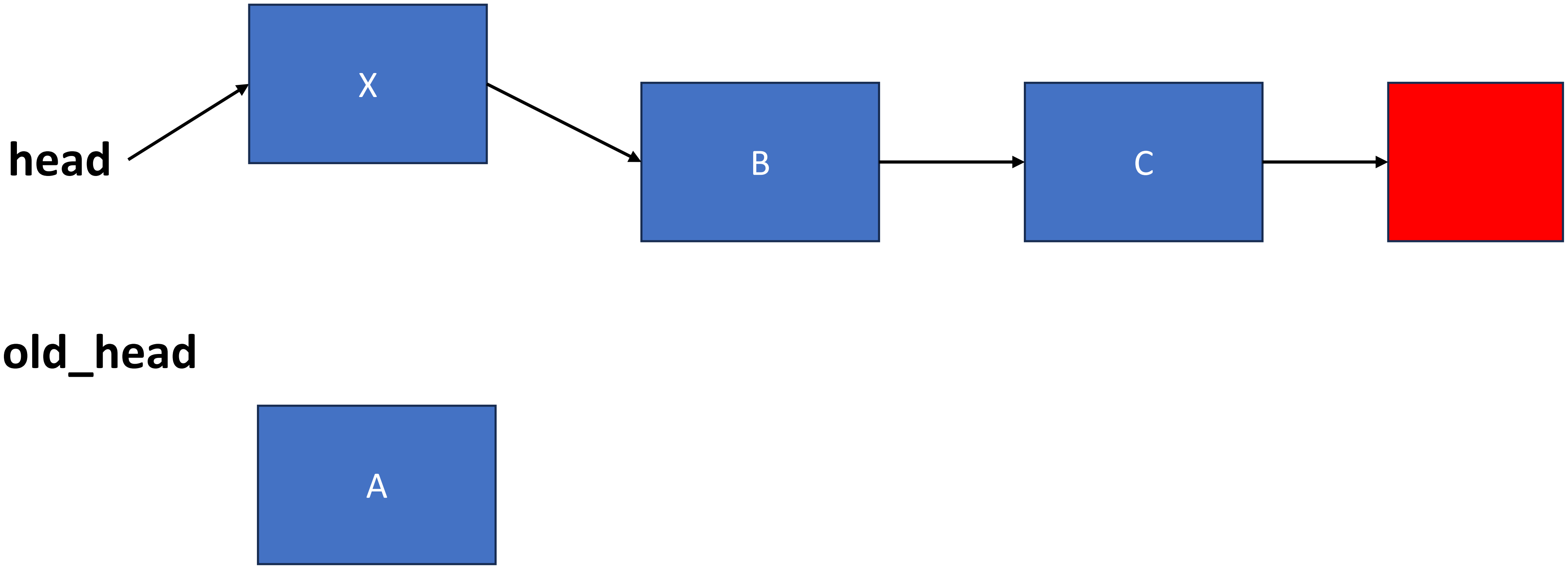
start atomic operation



CAS push, failure



CAS push, failure



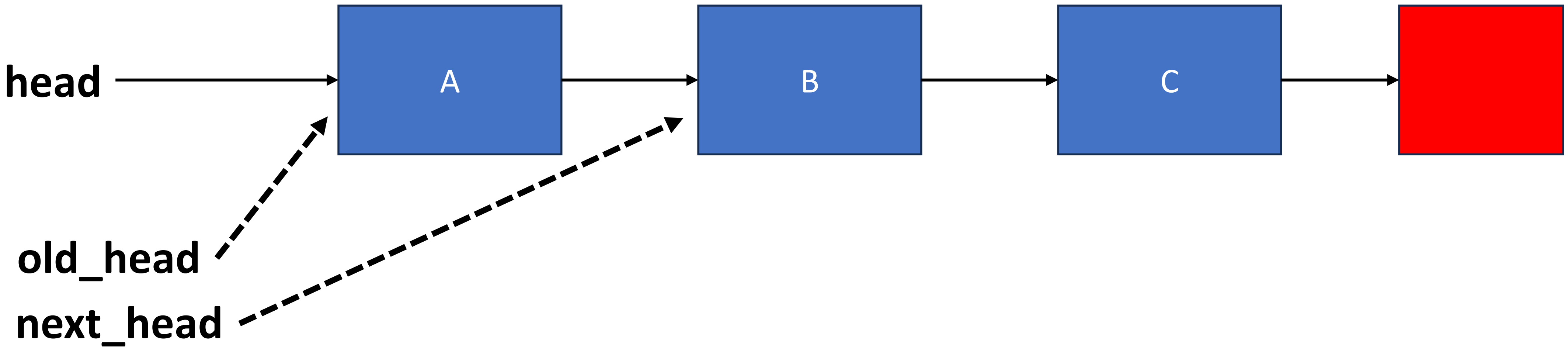
end atomic operation

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?

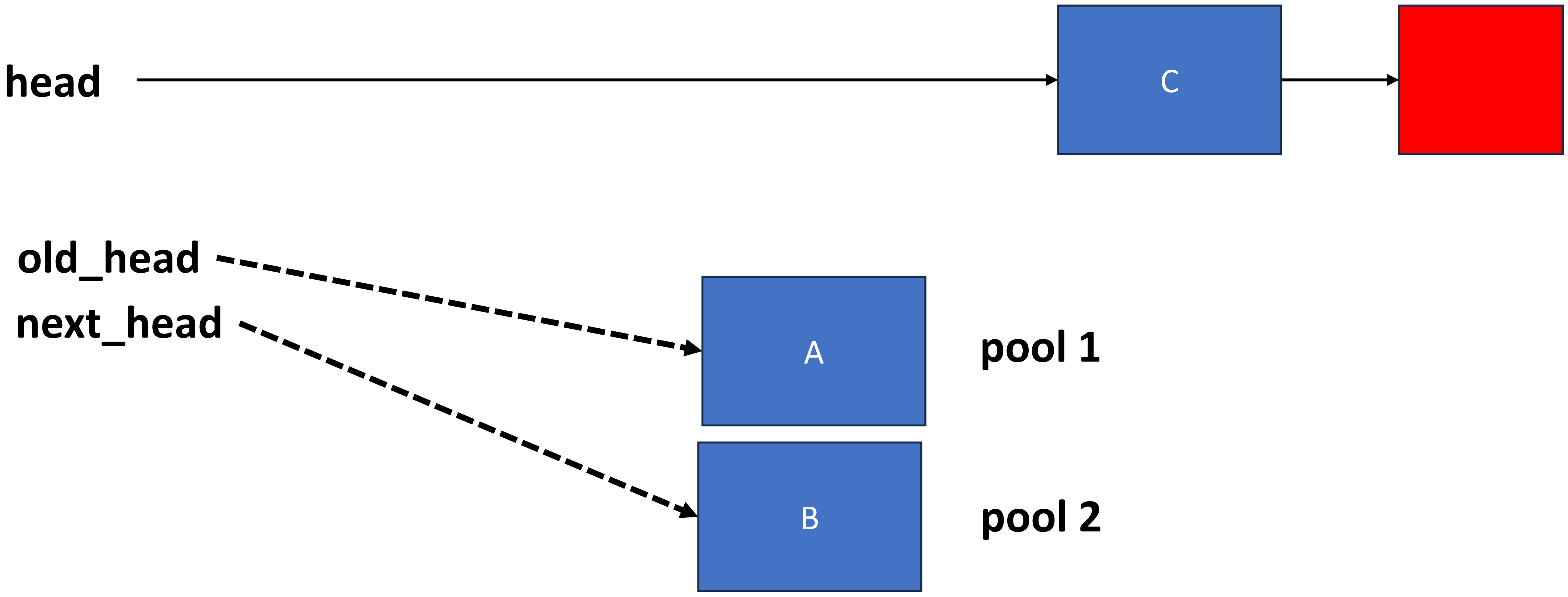
ABA problem

thread 0 start pop



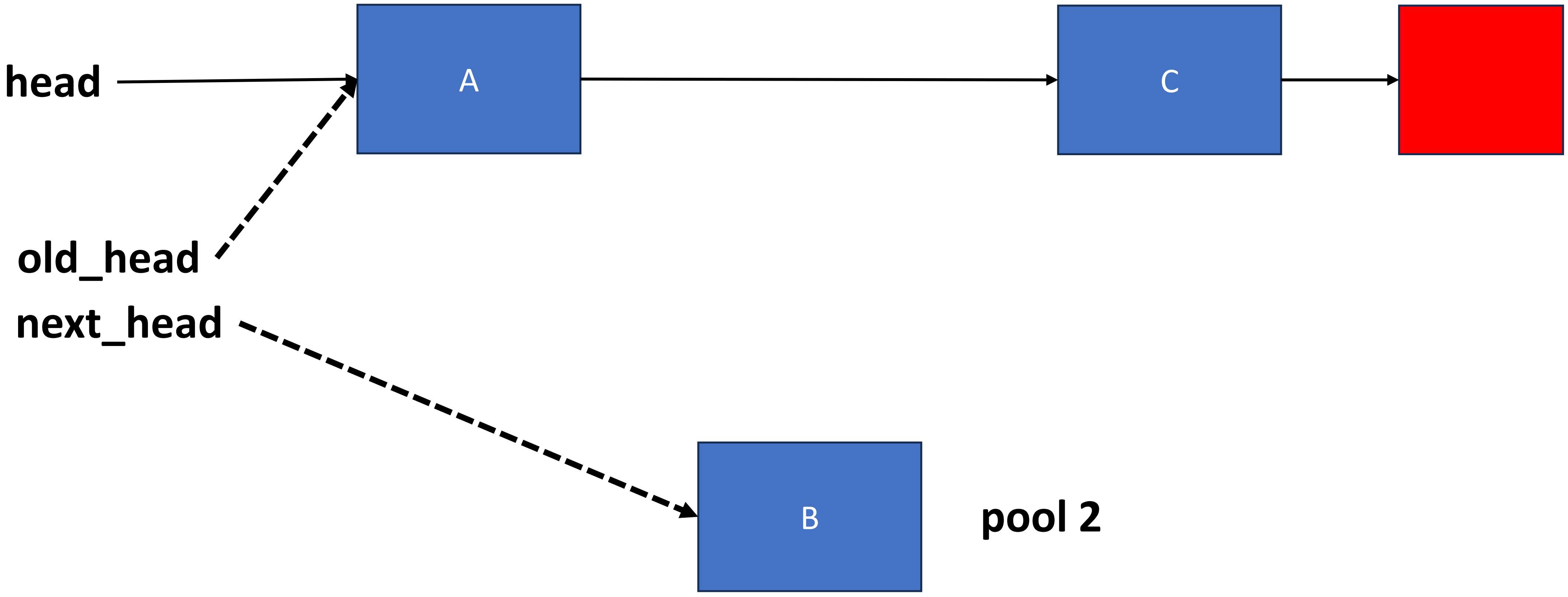
ABA problem

thread 1 pops A, thread 2 pops B



ABA problem

thread 1 pushes A



head

A

C

old_head

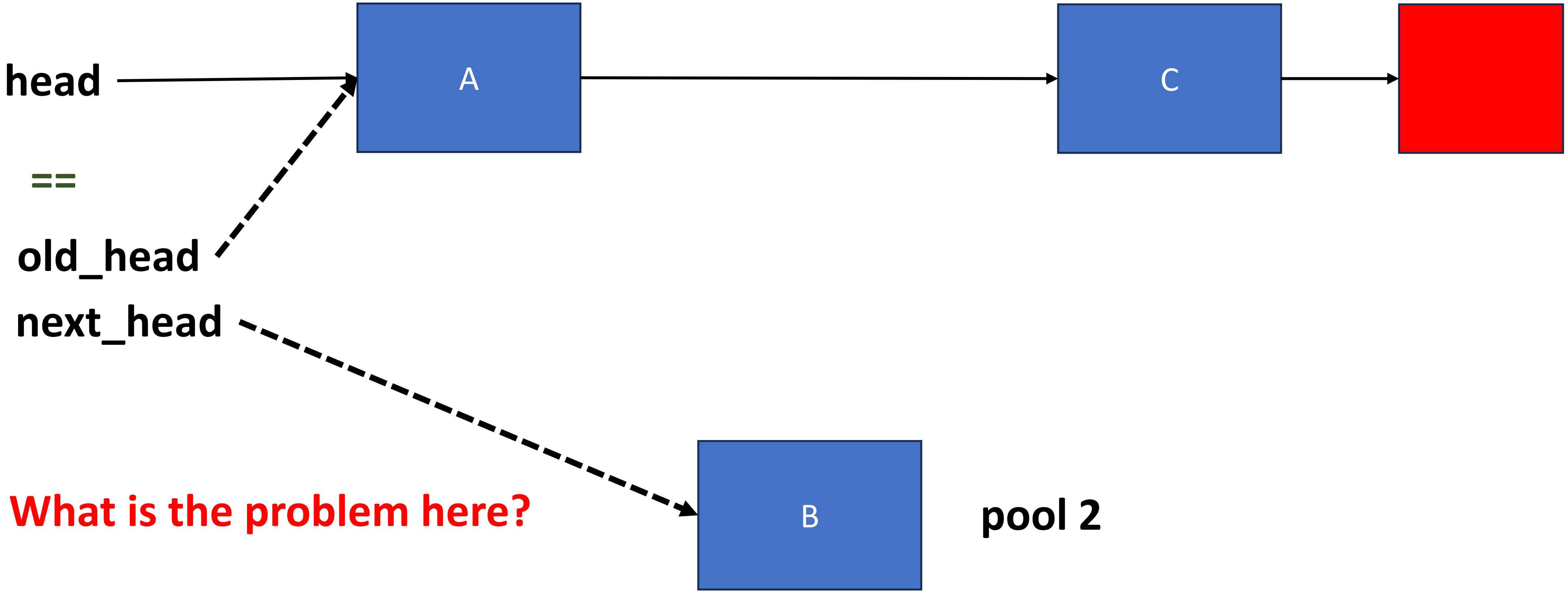
next_head

B

pool 2

ABA problem

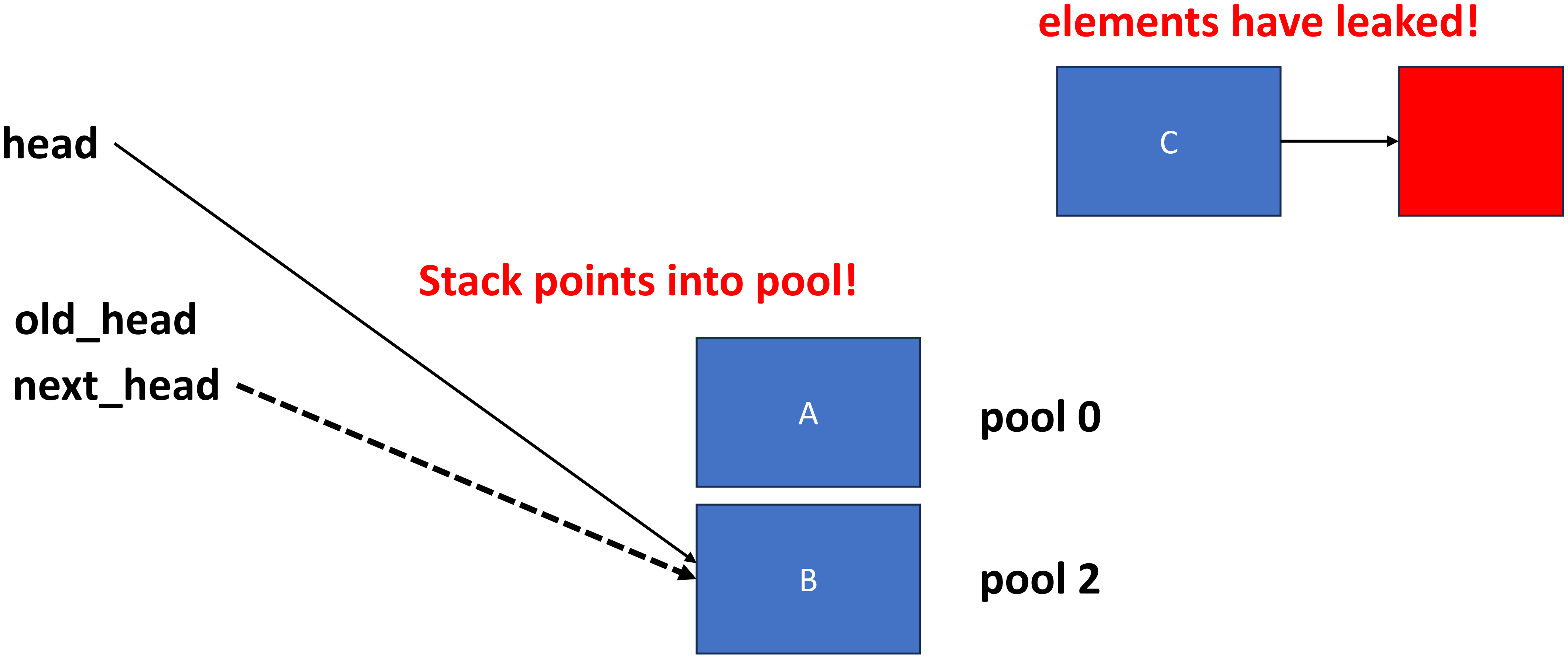
thread 0 resumes pop; enters atomic region:
Compares head and old_head



What is the problem here?

ABA problem

A is popped, setting head to old_next (B)



elements have leaked!

Stack points into pool!

Lab 2 Non-blocking Stack

- Goal for the lab:
  - Implement non-blocking unbounded stack with custom memory allocator
    - Reimplement push and pop operations
    - Use atomic operations
  - Study the ABA problem
    - Detect it or force it to occur
    - Can it be avoided?

Questions ?