Memory Reclamation

Concurrent Algorithms
Fall 2020
Igor Zablotchi

EPFL

Introduction

- So far in the course, we have assumed that memory is infinite
- This assumption needs not be true
  - In practice, memory is **finite**
  - Memory reclamation
- Topic of ongoing research

What is Memory Reclamation (MR)?

- Applications need memory
- Most realistic applications grow and shrink in memory
- Grow = allocate memory
- Shrink = free no-longer-useful memory

What is Memory Reclamation (MR)?

```
ds = new_data_structure(…);
node n = new_node(…);
insert(ds, n);
// use n in some way
remove(ds,n);
```

Need to free n!

Freeing Memory is Necessary

- Otherwise, applications might run out of memory or use too much memory

Automatic Garbage Collection

- Some languages (e.g., Java) have automatic memory management
- Memory is allocated & freed without explicit programmer intervention
- Garbage collector decides automatically when a pointer should be freed

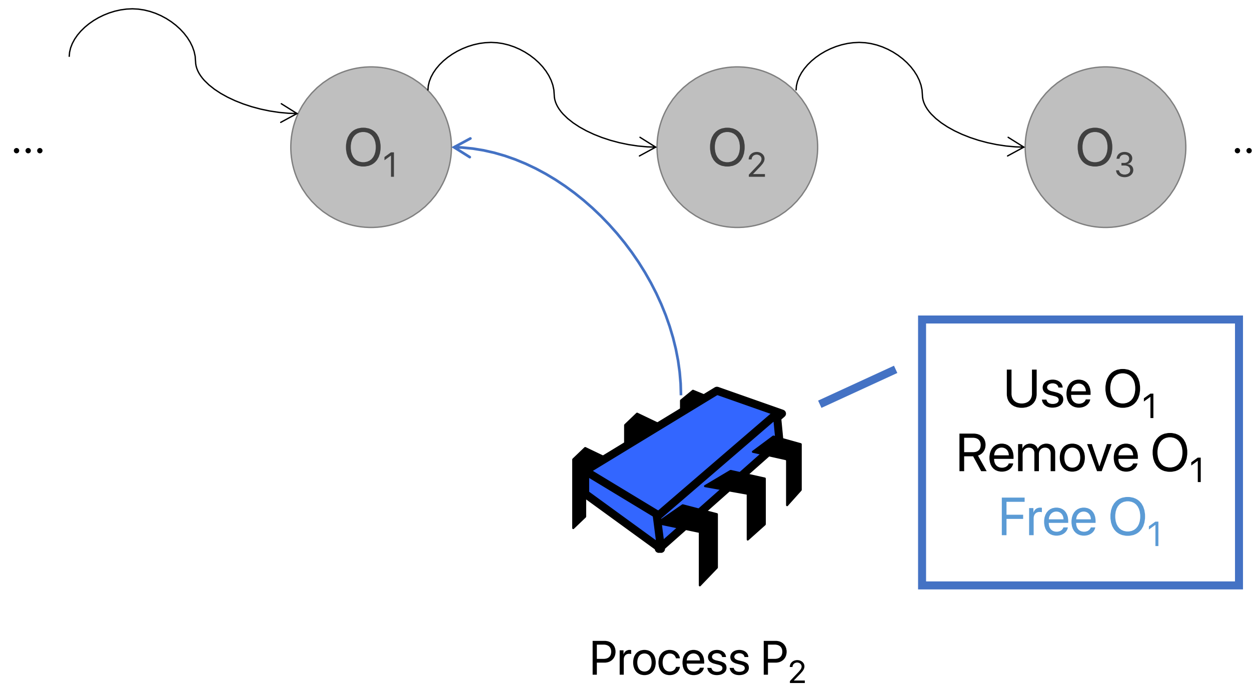
Explicit Memory Management

- Other languages (e.g., C, C++) require the programmer to allocate & free memory explicitly
- Programmer needs to determine when to free some memory location
- This is our focus for this class

1-process MR is Easy

- Allocate some memory
- Use it
- Free after last use

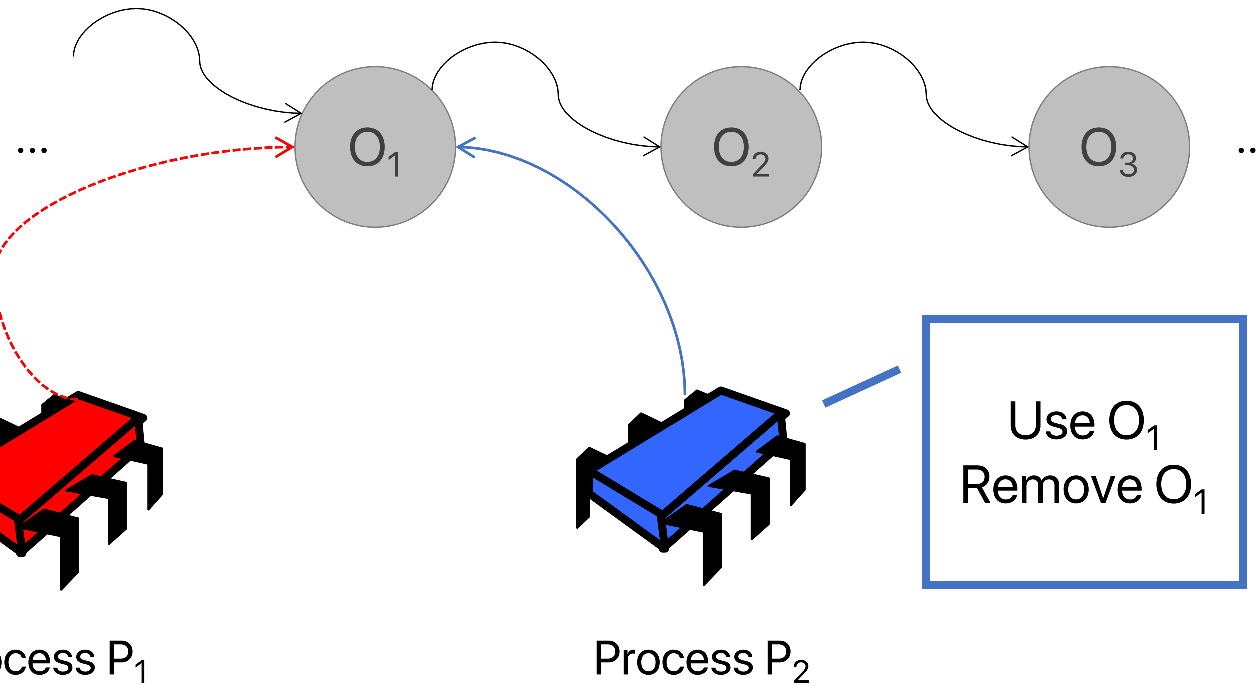
1-process MR is Easy



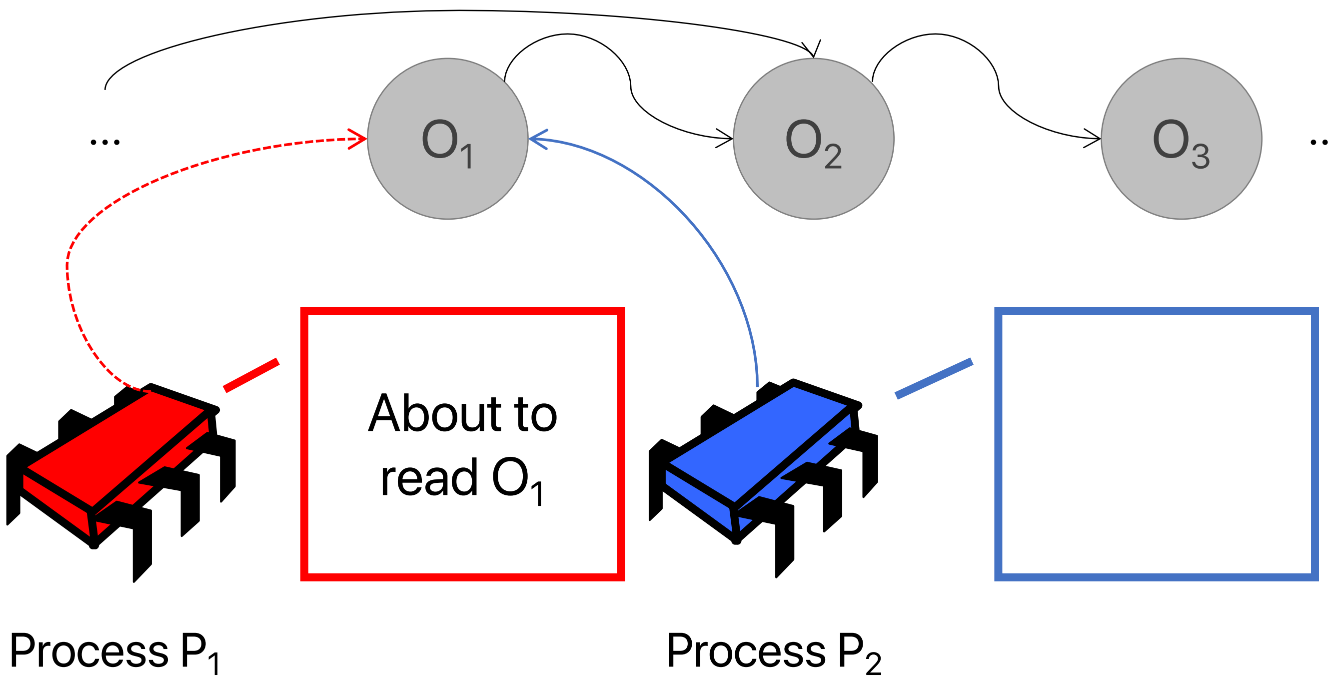
Concurrent MR is Difficult

- No easy way for a process to determine if a memory location will be used later by a different process

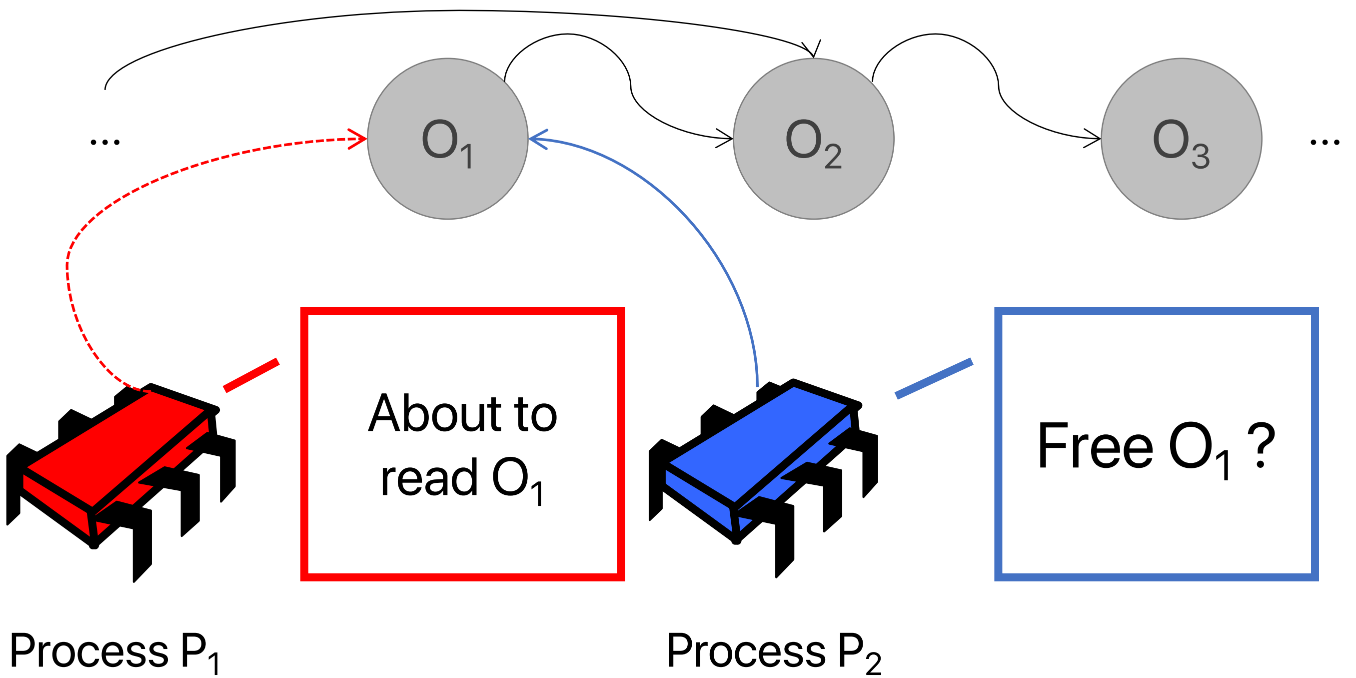
Concurrent MR is Difficult



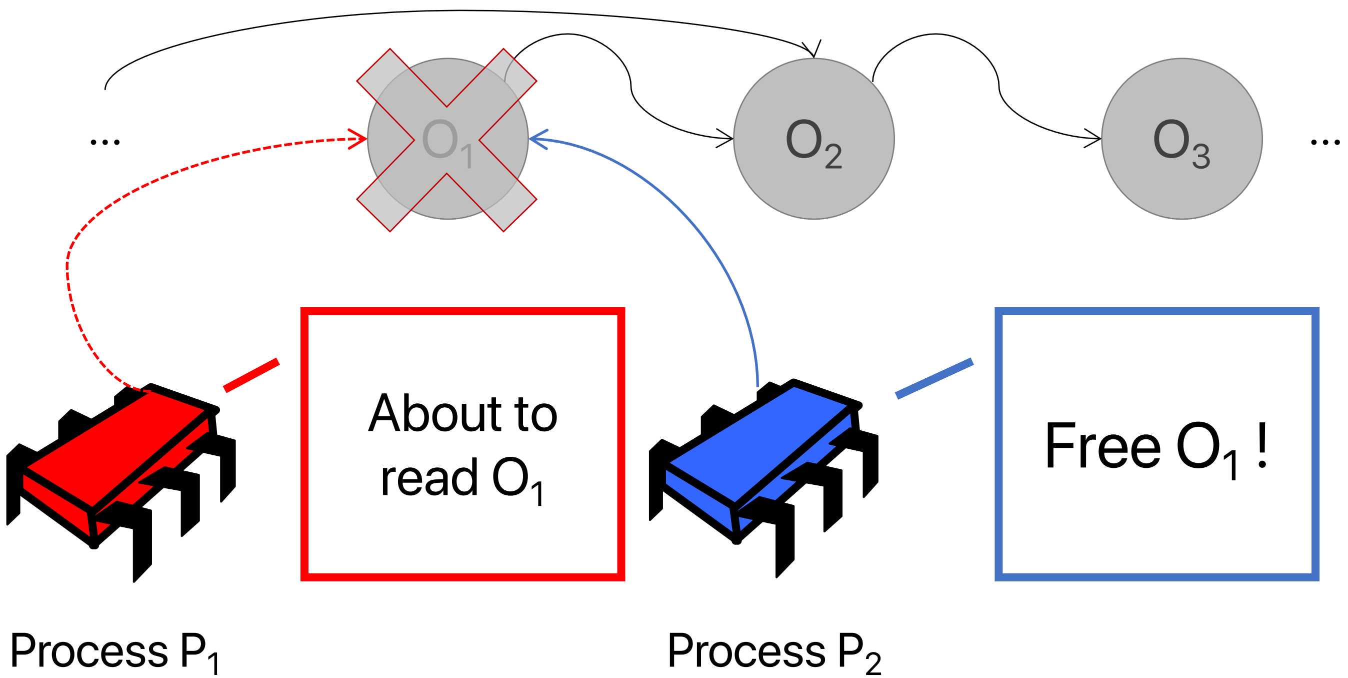
Concurrent MR is Difficult



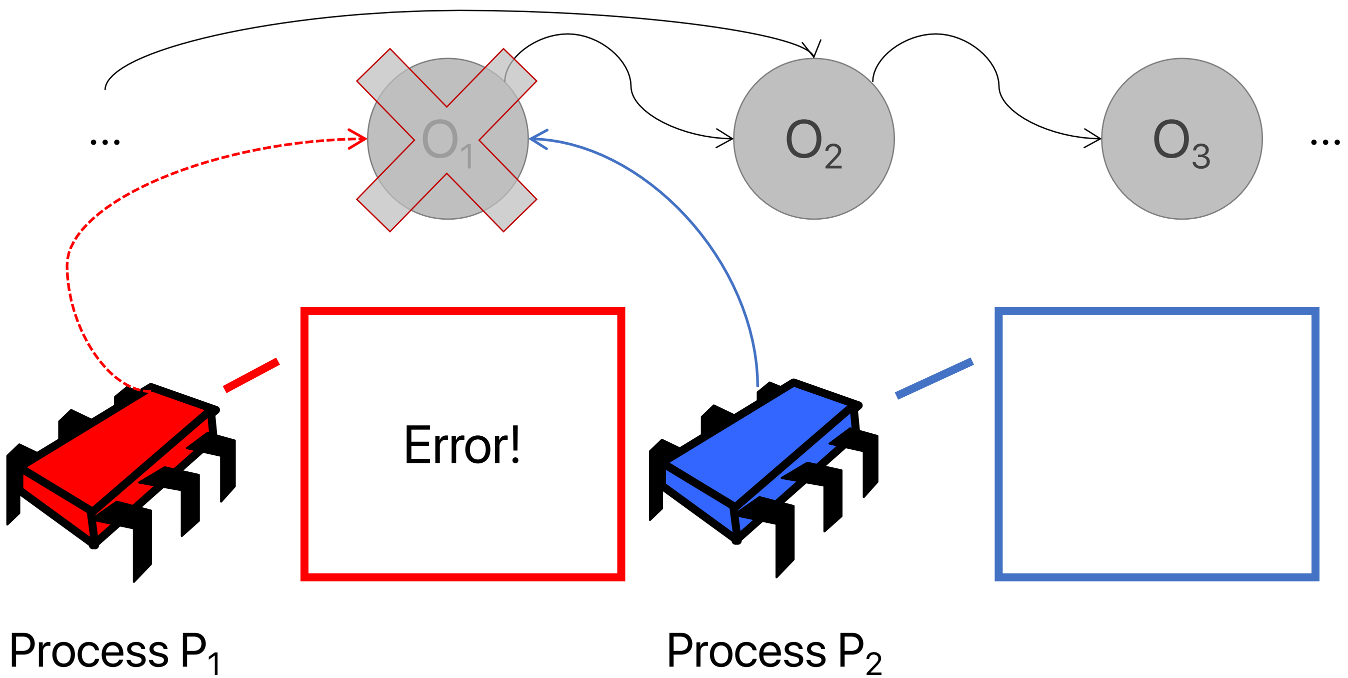
Concurrent MR is Difficult



Concurrent MR is Difficult



Concurrent MR is Difficult



Take-away So Far

- Memory reclamation = deciding when to free memory
- Necessary:
  - Most applications need to allocate + free
  - C, C++ are here to stay
  - No MR → excessive memory use
- Challenging (concurrent case):
  - Need a way to determine when all processes are done with some memory location

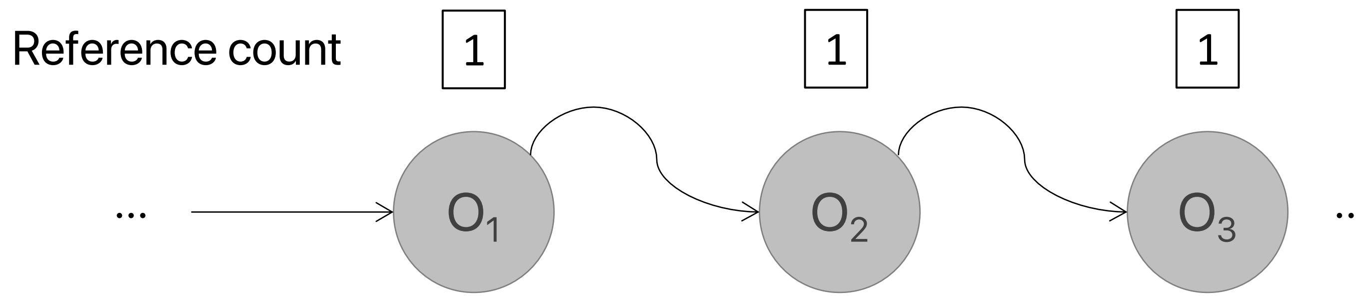
Outline

- Introduction
- Traditional MR Algorithms
  - Lock-free Reference Counting
  - Hazard Pointers
  - Epoch-based Reclamation
- QSense: A Hybrid MR Algorithm
- Conclusion

Lock-free Reference Counting

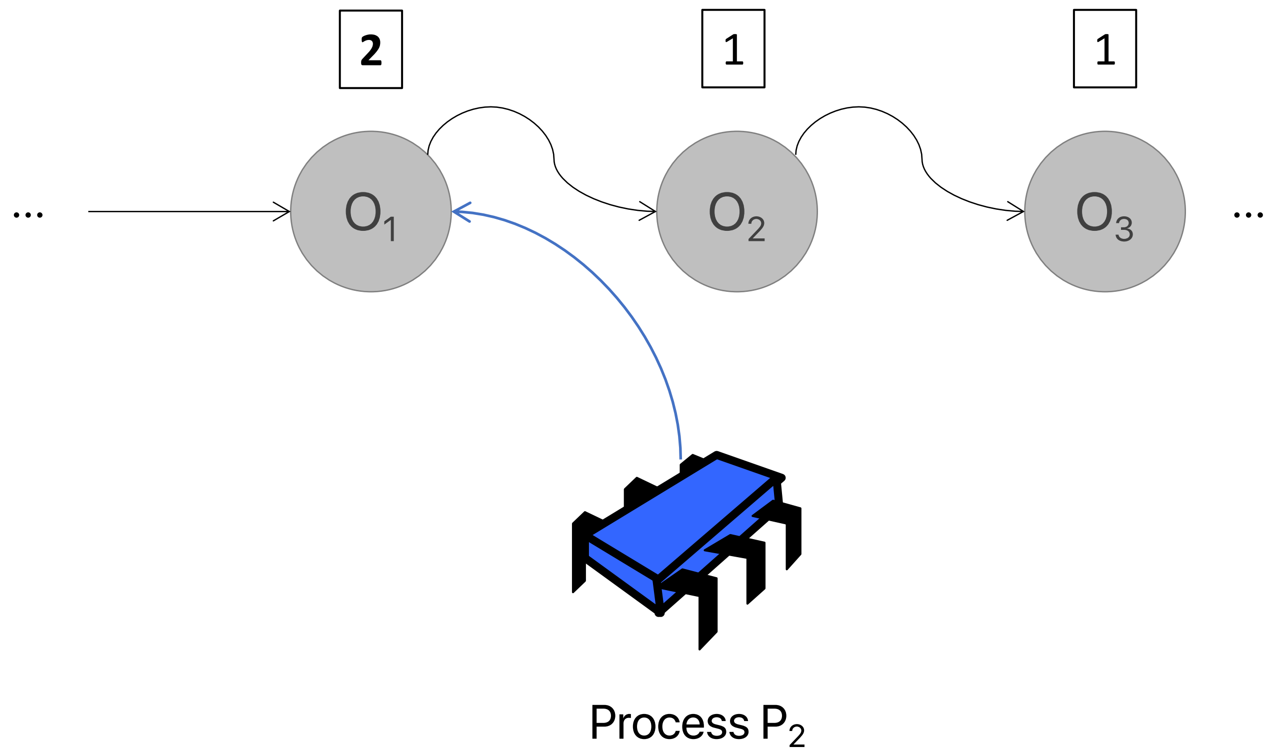
- Main idea:
  - For each memory location, keep track of how many references are held to it.
  - When there are 0 references, safe to reclaim.

LFRC Example



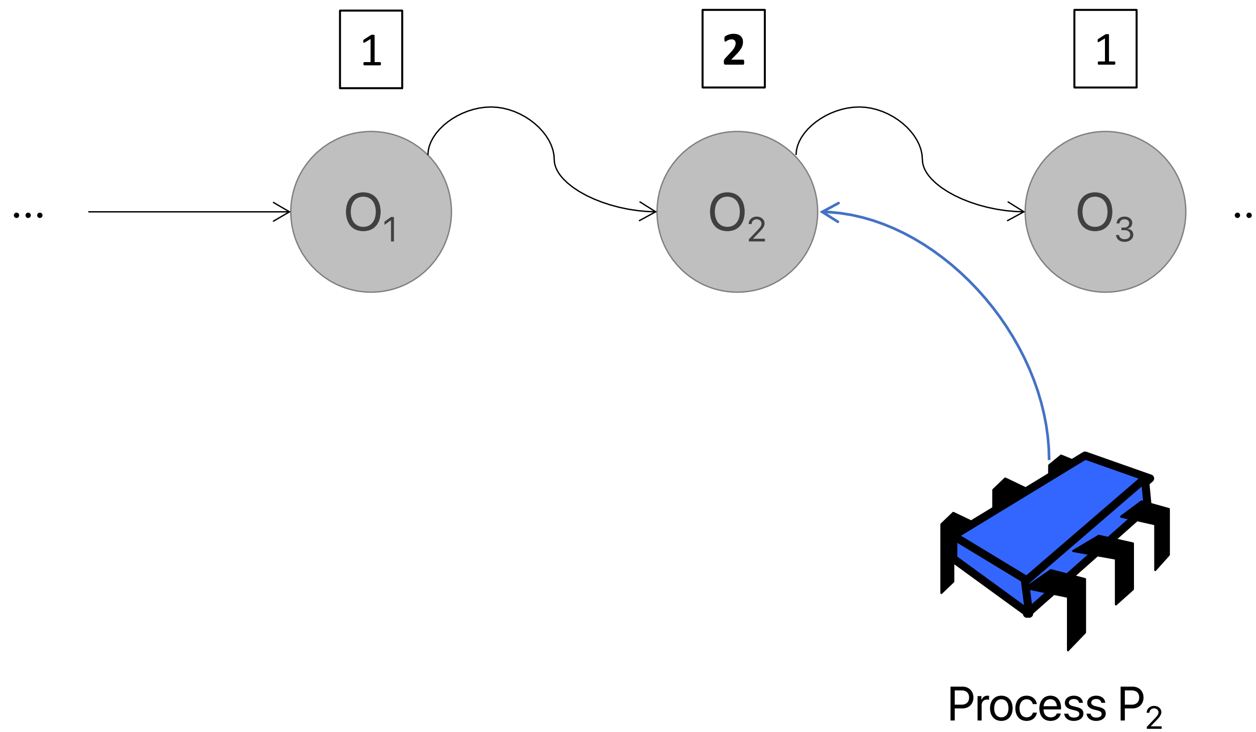
A linked list. No process has references. Each node has reference count = 1 (the reference from the previous node in the list).

LFRC Example



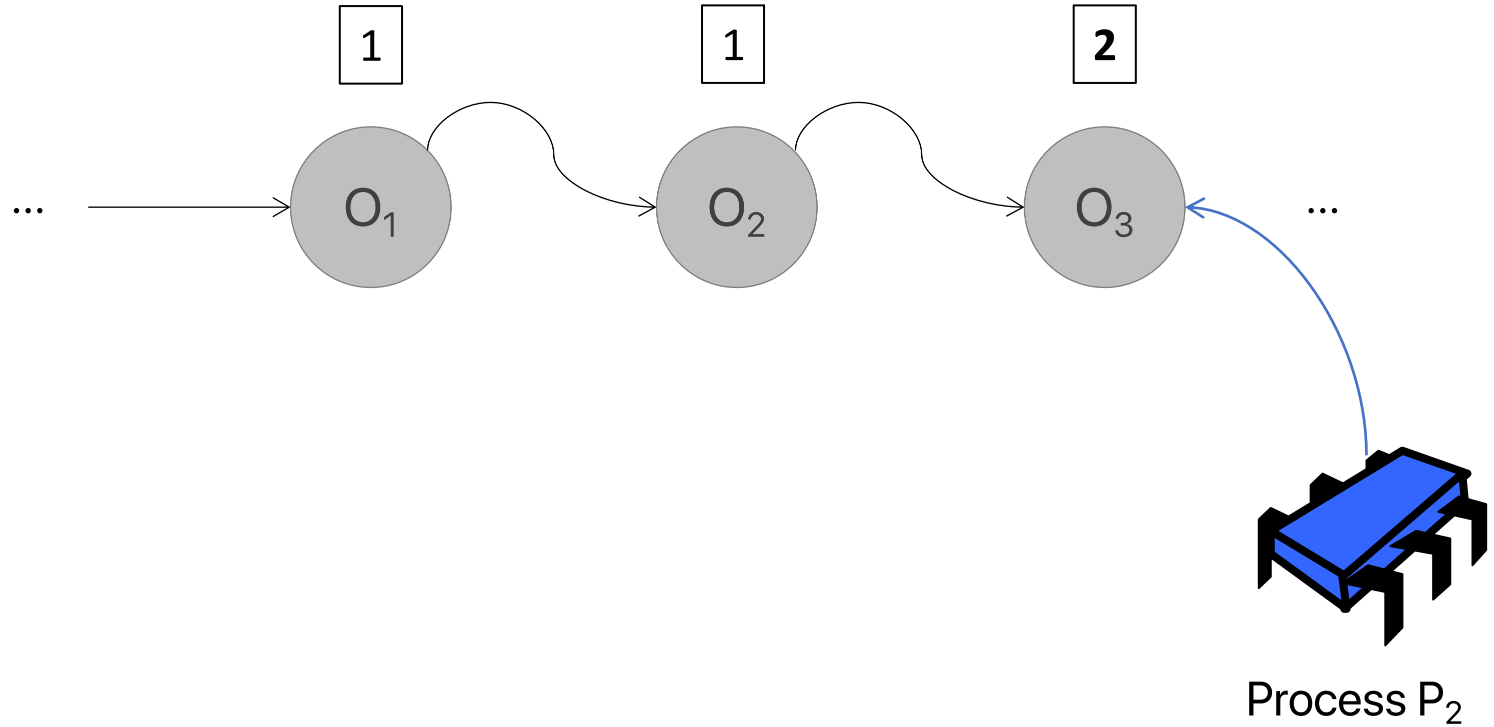
A thread is reading. The node that the thread is currently looking at has reference count = 2.

LFRC Example



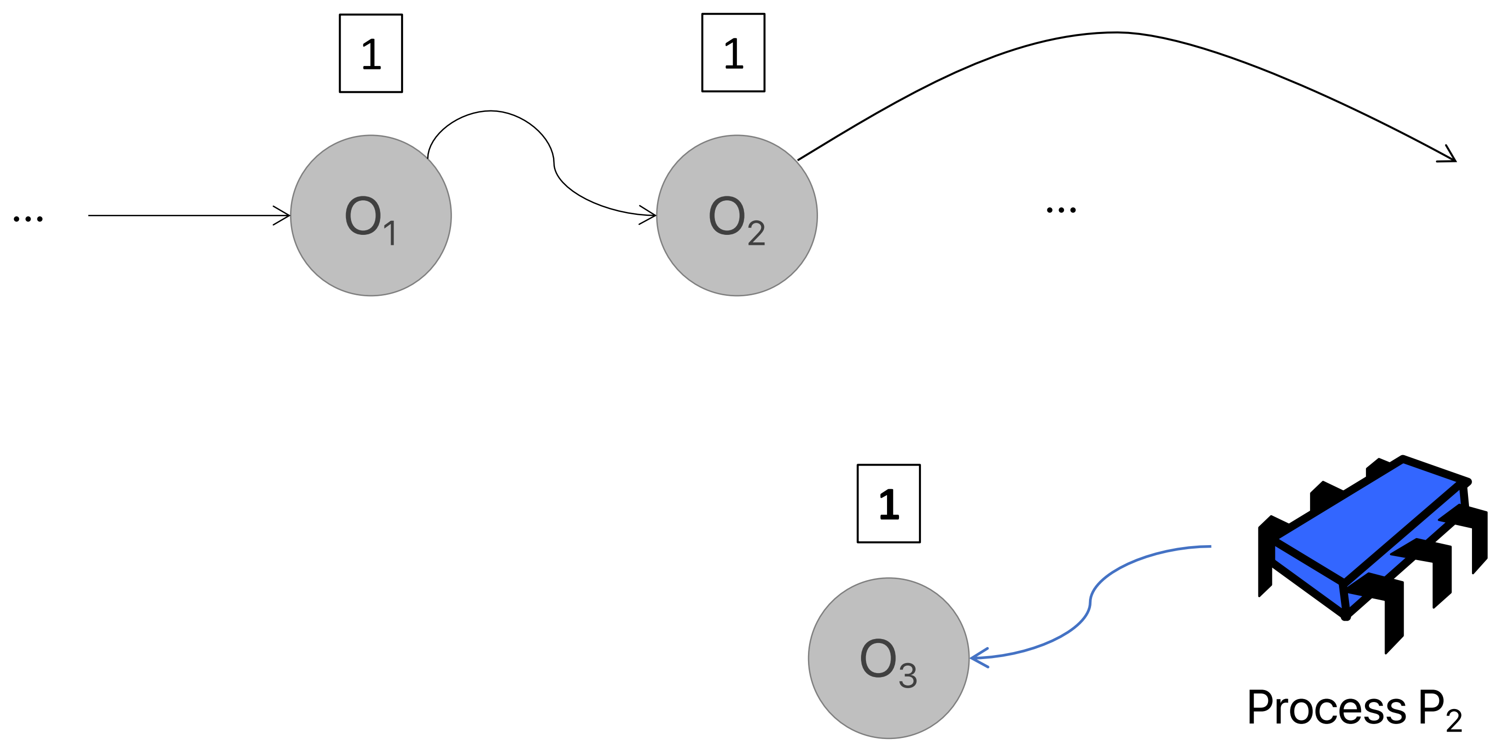
A thread is reading. The node that the thread is currently looking at has reference count = 2.

LFRC Example



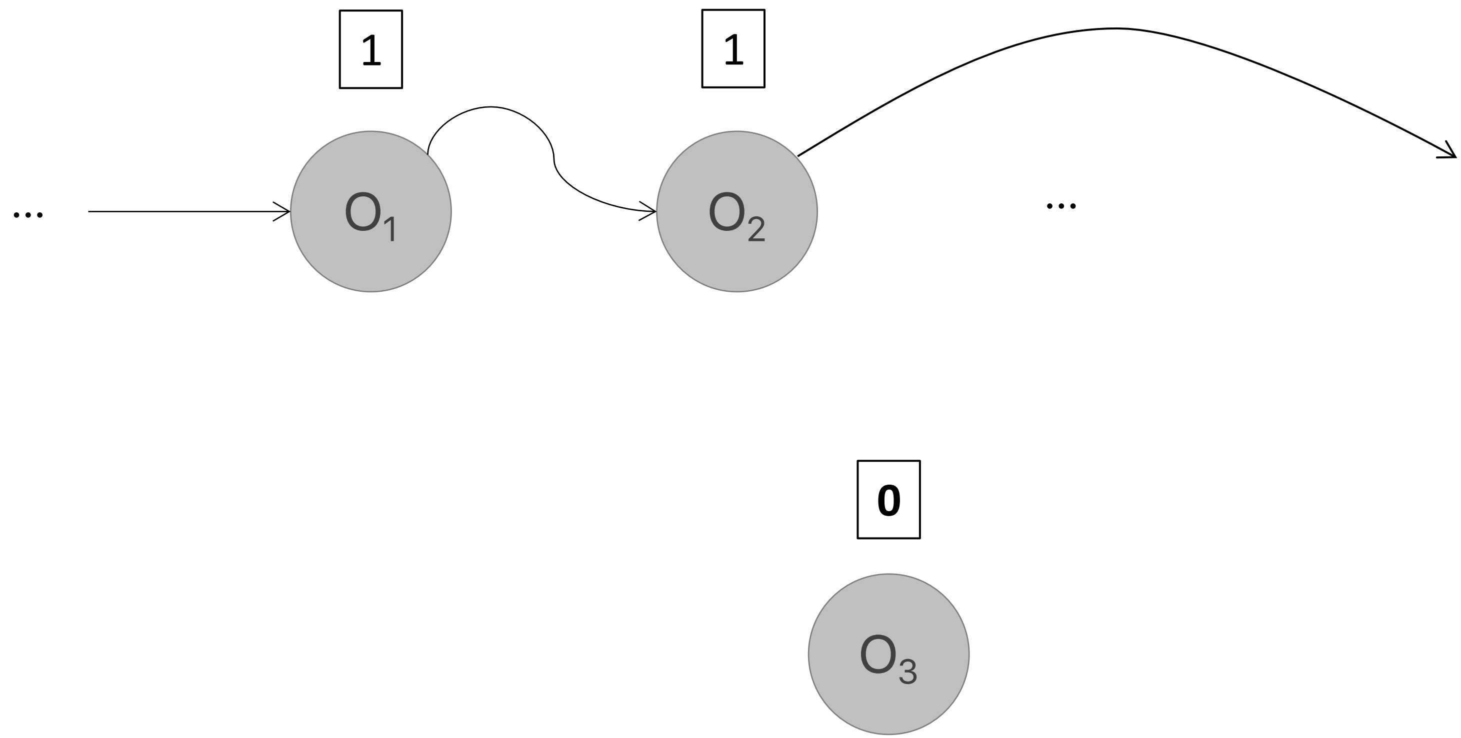
A thread is reading. The node that the thread is currently looking at has reference count = 2.

LFRC Example



A thread has removed node O_3 from the list. O_3 now has reference count = 1 (the reference from the thread).

LFRC Example



The thread has released its reference to O_3. O_3 now has 0 references. Its memory can be freed.

Pros and cons of LFRC

- ✓ Lock-free (wait-free version exists)
- ✓ Easy to understand & implement

- ✗ Need to update reference counter on every access, even if read-only → bad performance
- ✗ Update of reference counter requires expensive atomic instructions → extremely bad performance!

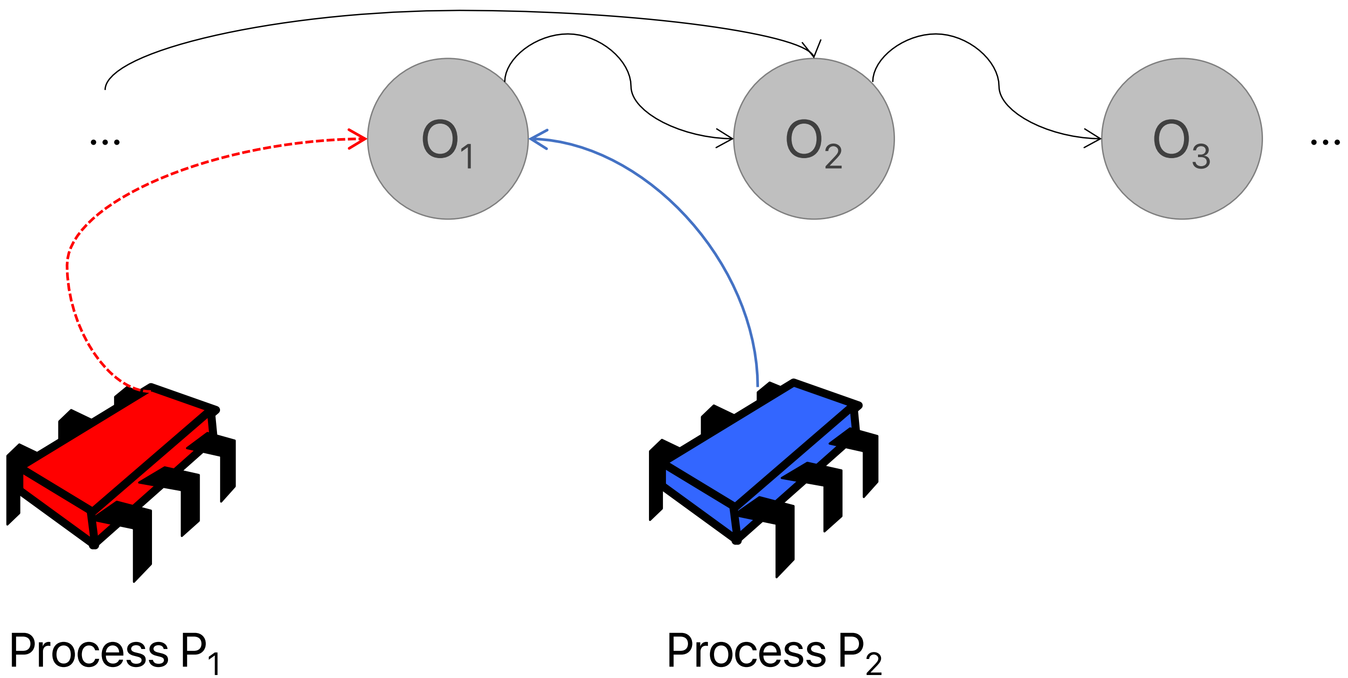
Outline

- Introduction
- Traditional MR Algorithms
  - Lock-free Reference Counting
  - Hazard Pointers
  - Epoch-based Reclamation
- QSense: A Hybrid MR Algorithm
- Conclusion

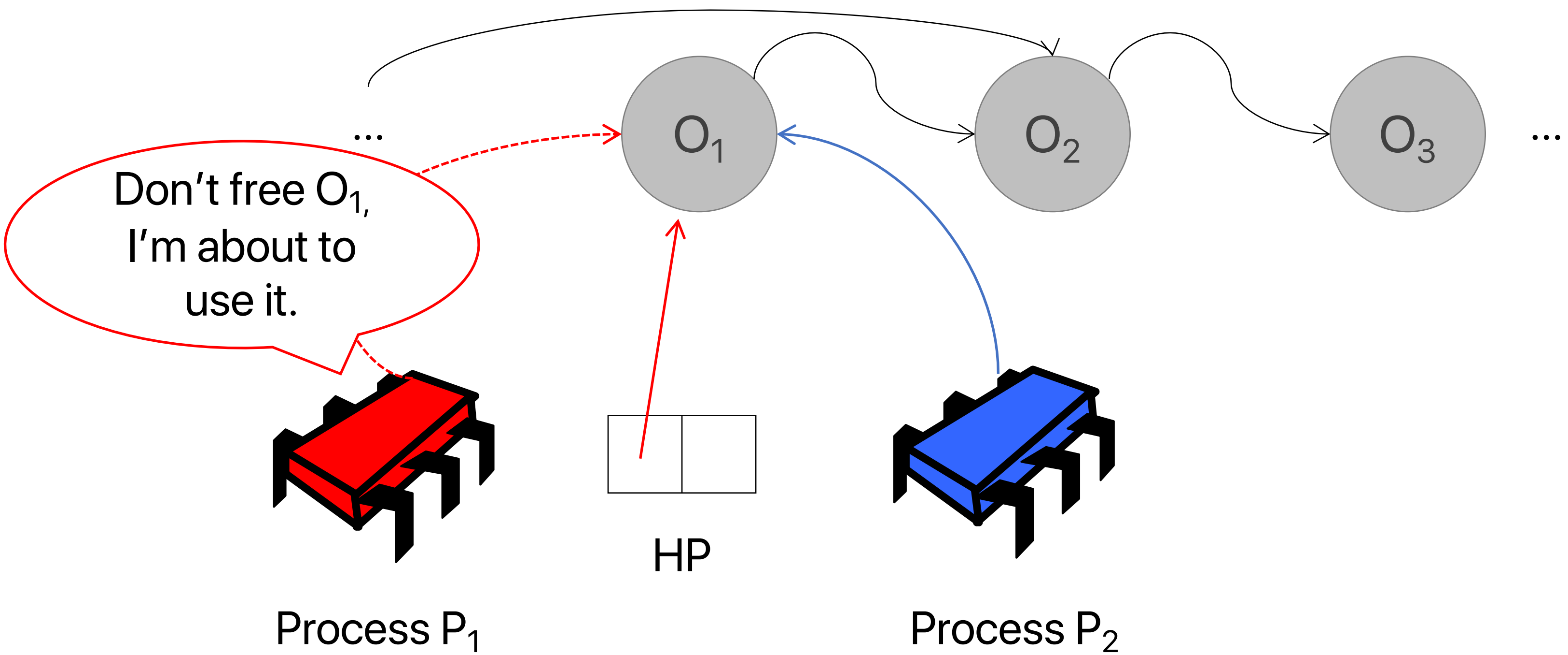
Hazard Pointers (HP)

- Main idea:
  - Each process announces memory locations it plans to access: hazard pointers
  - Processes only free memory that is not protected by hazard pointers

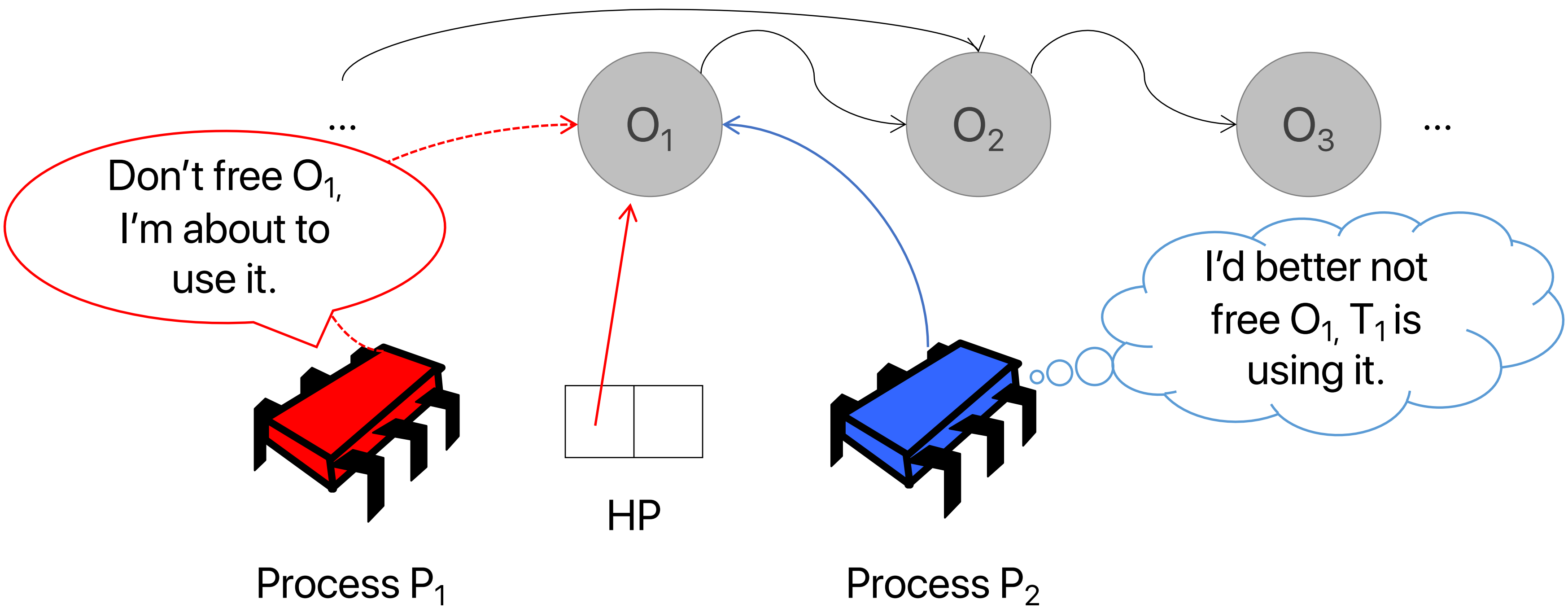
Hazard Pointers (HP)



Hazard Pointers (HP)



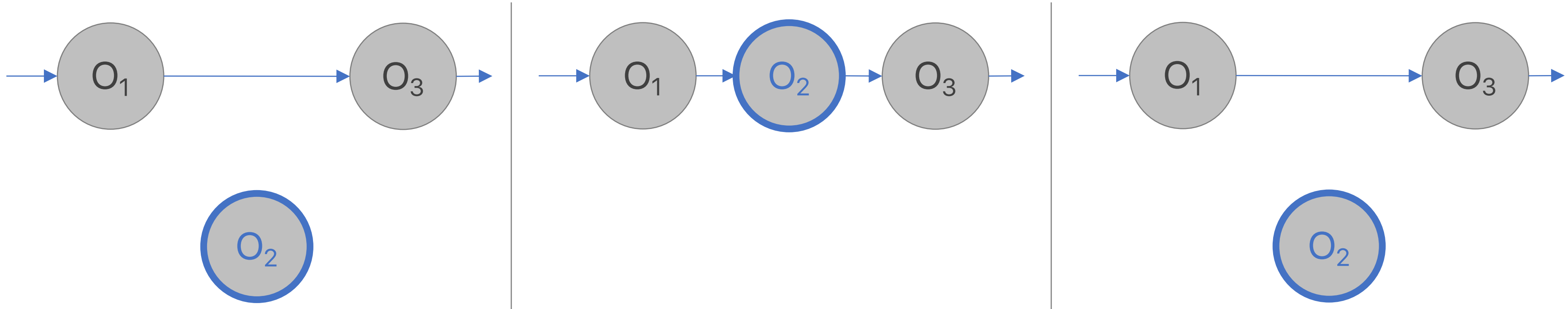
Hazard Pointers (HP)



HP – More Details

0. Reachability

- Reachable node = can be found by following pointers from data structure root(s)



Before inserting
→ O_2 not yet reachable

In the data structure
⇔ O_2 reachable

After deletion
→ O_2 no longer reachable

HP – More Details

1. Announcing hazard pointers

Without hazard pointers

```
1. Read a reference p
2. Do something with p
3. (Release reference to p)
```

With hazard pointers

```
1. Read a reference p
2. HP = p // protect p
3. Check if p is still
   reachable. If yes,
   continue, otherwise
   restart operation.
4. Do something with p
5. (Release reference to p)
```

HP – More Details

2. Deleting elements

- Each process has a “limbo list” containing nodes that have been deleted but not yet freed
- After process p_i deletes a node *n* from the data structure, it adds *n* to p_i’s limbo list

HP – More Details

3. Reclaiming memory

- When the limbo list grows to a certain size *R*, p_i initiates a **scan**:
  - For each node *n* in the limbo list:
    - Look at HPs of all processes. Is any of them pointing to *n*?
    - If not, free *n*'s memory
    - (If yes, do nothing)

HP Guarantees

Constant time per node reclaimed

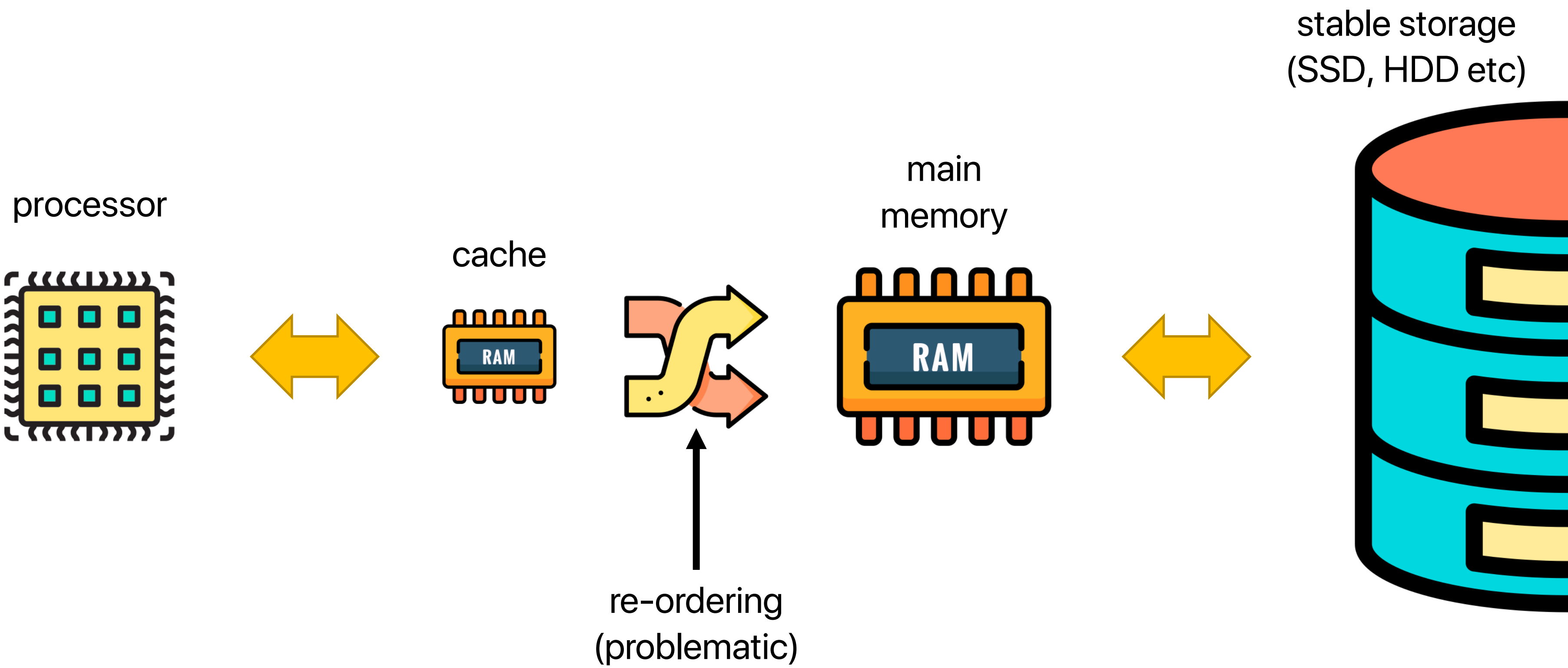
+

Bounded memory overhead

→ Great performance and reliability
(in theory)

The Re-ordering Problem

Modern architectures reorder instructions

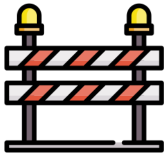


The Re-ordering Problem

Modern architectures reorder instructions

```
// read reference to n
Announce_HP(n);

Check(n);
// continue using n
```

Memory Barriers 

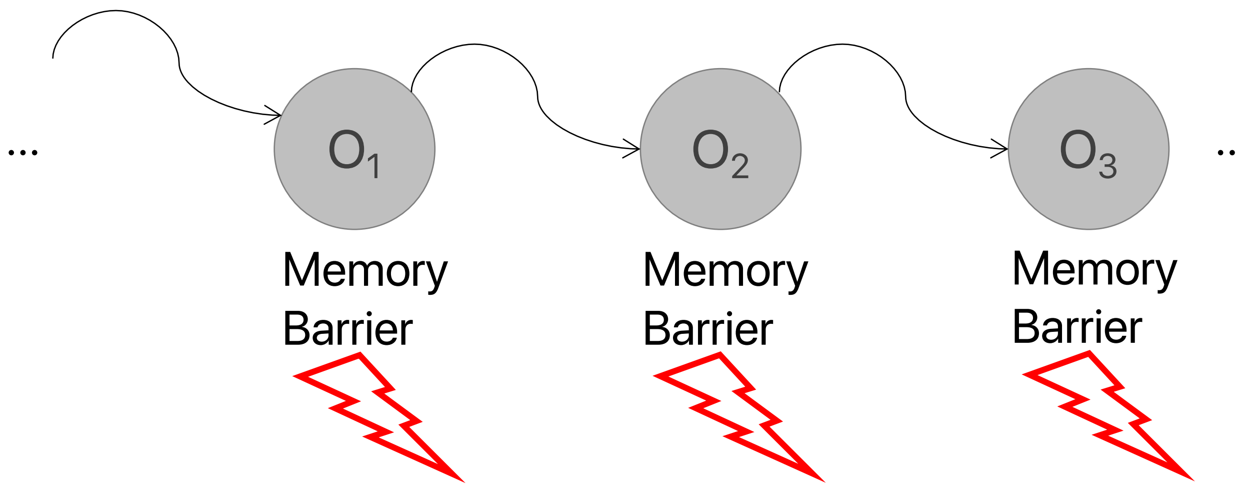
- Memory barriers prevent re-ordering
- But they are expensive (slow)

HPs Need Barriers

Modern architectures reorder instructions

```
// read reference to n
Announce_HP(n);
Memory_barrier();
Check(n);
// continue using n
```

Barriers – Bad for Performance



→ HP good in theory, slow in practice

Pros and Cons of HP

✓ Limits memory use

✓ Lock-free

✗ Need to update HP on every access, even if read-only → bad performance

✗ Need memory barriers → bad performance

✗ Complex to implement & use → prone to errors

Outline

- Introduction
- Traditional MR Algorithms
  - Lock-free Reference Counting
  - Hazard Pointers
  - Epoch-based Reclamation
- QSense: A Hybrid MR Algorithm
- Conclusion

Epoch-based Reclamation (EBR)

- Main idea:
  - Processes keep track of each other's progress
  - After deleting an object, when all processes have made enough progress, memory can be freed

EBR, Step by Step

- Step 1: processes declare when they enter & exit **critical sections**

```
// code
enter_critical_section();
// more code
exit_critical_section();
// even more code
```

Here, we may access “dangerous” memory (memory that can be freed)

Here, only safe memory accesses are allowed (memory that is never freed)

EBR, Step by Step

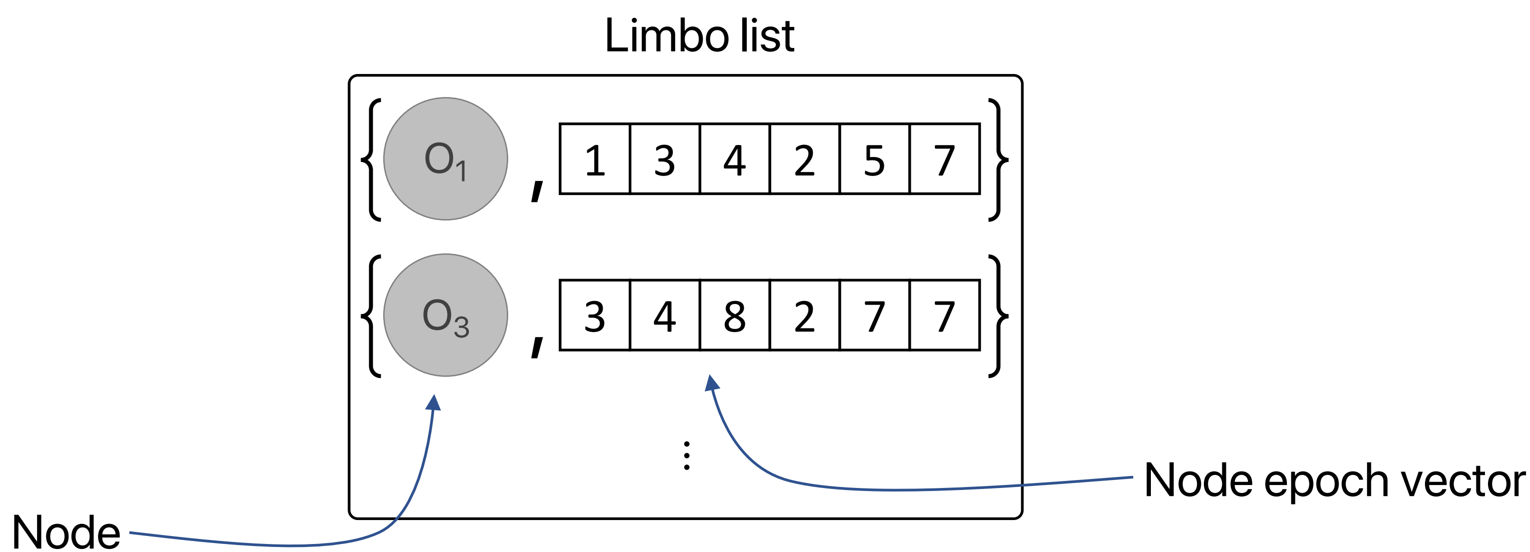
- Step 2: each process has an *epoch* (an integer, initially 0). The epoch is incremented by 1 when entering and exiting a critical section.

```
// code                          ← epoch = 0
enter_critical_section();
// more code                     ← epoch = 1
exit_critical_section();
// even more code                ← epoch = 2
```

→ epoch is **odd** if inside critical section and **even** otherwise

EBR, Step by Step

- Step 3: After deleting an element, add it to a per-process limbo list, together with current epochs of all processes



EBR, Step by Step

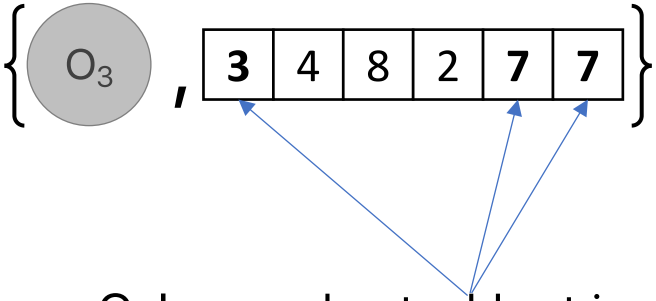
- Step 4: Periodically scan limbo list

Scan:
- cur_vec = current epoch vector
- For each node *n* in the limbo list:
  - node_vec = n's epoch vector
  - For each process i:
    - if node_vec[i] is odd
      - if node_vec[i] >= cur_vec[i]
        - Continue to next node
  - Free node

EBR, Step by Step

- Step 4: Periodically scan limbo list

Scan:
- cur_vec = current epoch vector
- For each node *n* in the limbo list:
  - node_vec = n's epoch vector
  - For each process i:
    - if node_vec[i] is odd
      - if node_vec[i] >= cur_vec[i]
        - Continue to next node
  - Free node

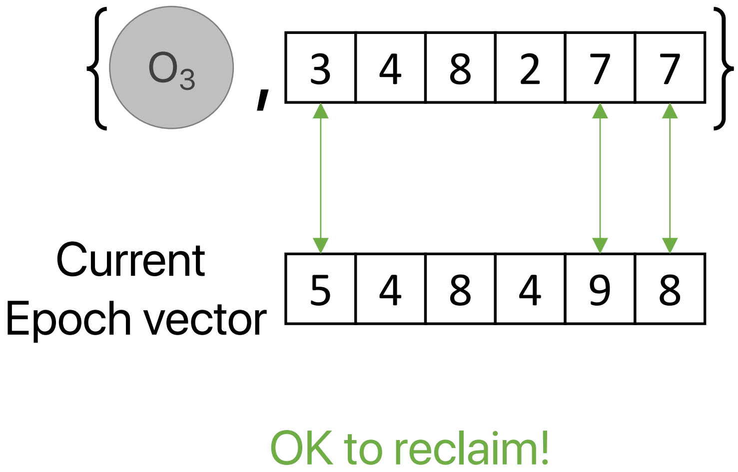


Only care about odd entries (processes inside crit. sec.)! Processes outside crit. sec. cannot access this node.

EBR, Step by Step

- Step 4: Periodically scan limbo list

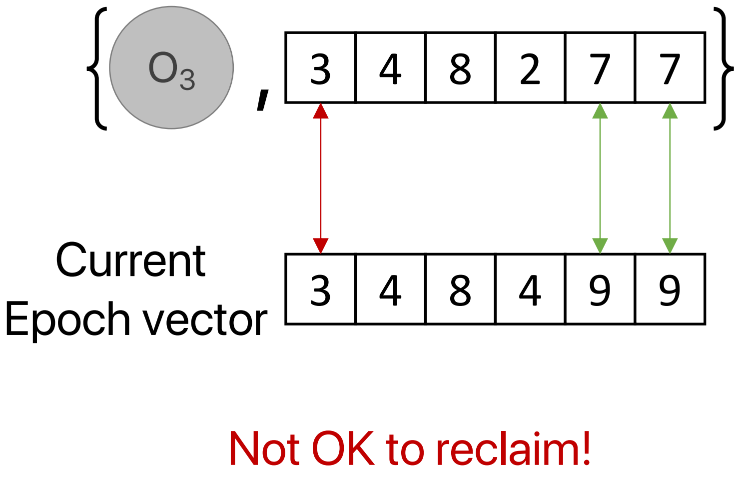
Scan:
- cur_vec = current epoch vector
- For each node *n* in the limbo list:
    - node_vec = n's epoch vector
    - For each process i:
        - if node_vec[i] is odd
            - if node_vec[i] >= cur_vec[i]
                - Continue to next node
    - Free node



EBR, Step by Step

- Step 4: Periodically scan limbo list

Scan:
- cur_vec = current epoch vector
- For each node *n* in the limbo list:
  - node_vec = n's epoch vector
  - For each process i:
    - if node_vec[i] is odd
      - if node_vec[i] >= cur_vec[i]
        - Continue to next node
  - Free node



Pros and Cons of EBR

✓ Small overhead → very good performance

✓ Easy to use

✗ Blocking (not lock-free)

→ can invalidate lock- or wait-freedom of data structure

→ if some process is delayed inside a critical section, memory cannot be reclaimed any more

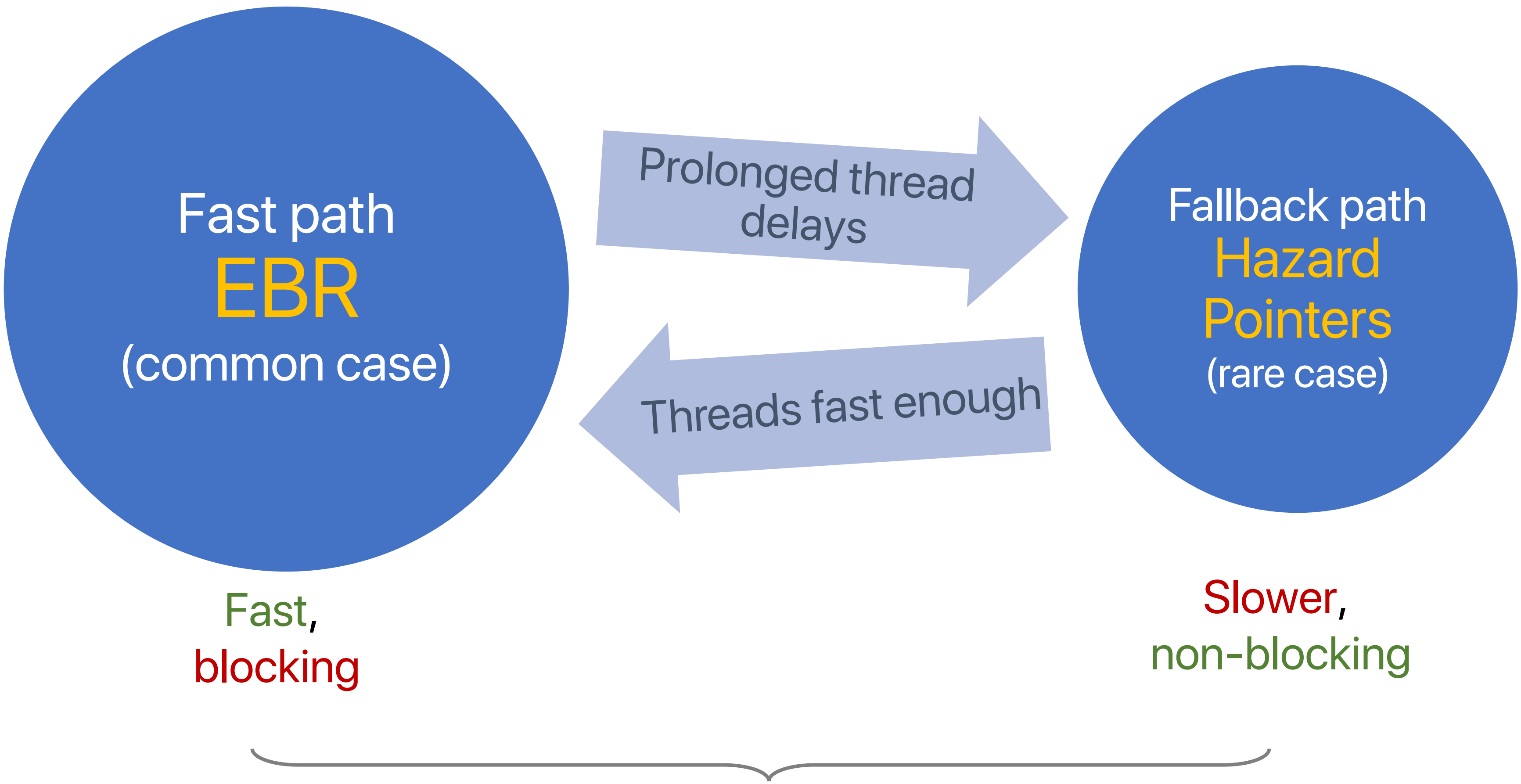
Outline

- Introduction
- Traditional MR Algorithms
  - Lock-free Reference Counting
  - Hazard Pointers
  - Epoch-based Reclamation
- QSense: A Hybrid MR Algorithm
- Conclusion

HP and QSBR – Complementary

	Non-blocking	Small Overhead
EBR	✗	✓
HP	✓	✗

A Hybrid Approach



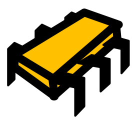
Fast in the common case, resilient when necessary

A Hybrid Approach

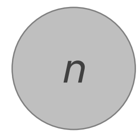
- Keep track of both HPs and epochs
- When scanning:
  - If on fast path, use EBR-style scan
  - If on slow path, use HP-style scan

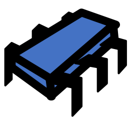
Ideally, we should only use memory barriers in the fallback path.

The Barrier Strikes Back



R is reading *n*



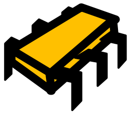


D is deleting *n*

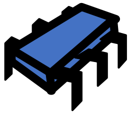
- Read a pointer to a node *n* (Load)
- Assign HP to *n* (Store)
- If fallback mode is active (Load), then
    - Execute a memory barrier
- Recheck *n* (Load)
- Use *n* (Loads and Stores)

- Remove *n*
- If on fallback path
    - Scan hazard pointers
    - If no HPs for *n*, then
        - Free *n*
- Else [...]

The Barrier Strikes Back



n



R is reading *n*

D is deleting *n*

- Read a pointer to a node *n* (Load)
- Assign HP to *n* (Store)
- If fallback mode is active (Load), then
  - ~~Execute a memory barrier~~
- Recheck *n* (Load)

Here, we are on the fast path (fallback mode off)

Some process P activates fallback mode here

- Remove *n*
- If on fallback path
  - Scan hazard pointers
  - If no HPs for *n*, then
    - Free *n*

- Use *n* (Loads and Stores)

The Barrier Strikes Back

☹️ It seems that we cannot turn memory barriers on and off.

🤔 But what if we could eliminate them altogether?

→ Cadence: HPs without Memory Barriers

Cadence – Main Insight

context switch = memory barrier

for process being switched out

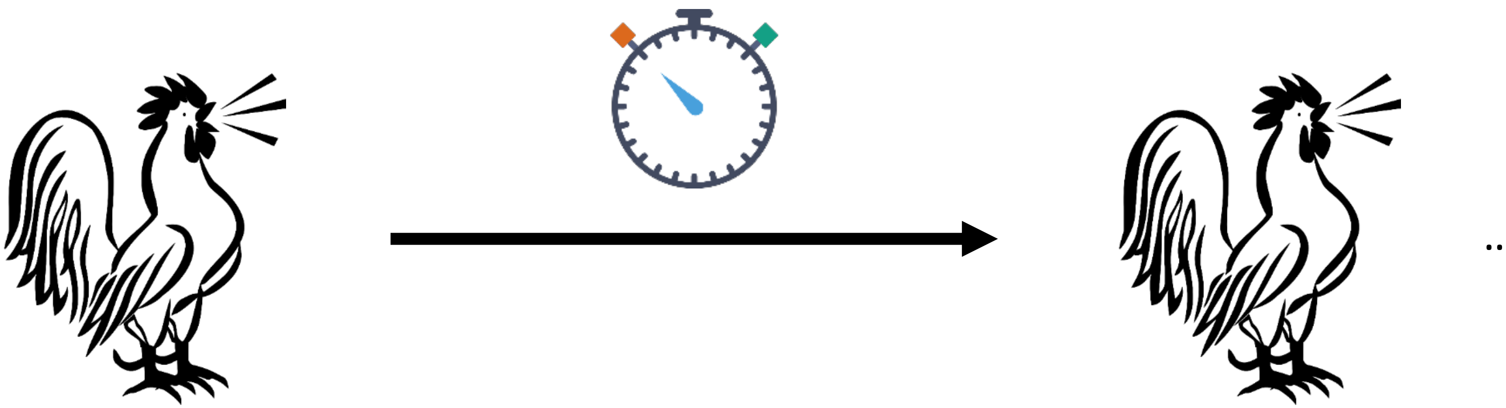
Can we use this to replace memory barriers in the HP algorithm?

Cadence

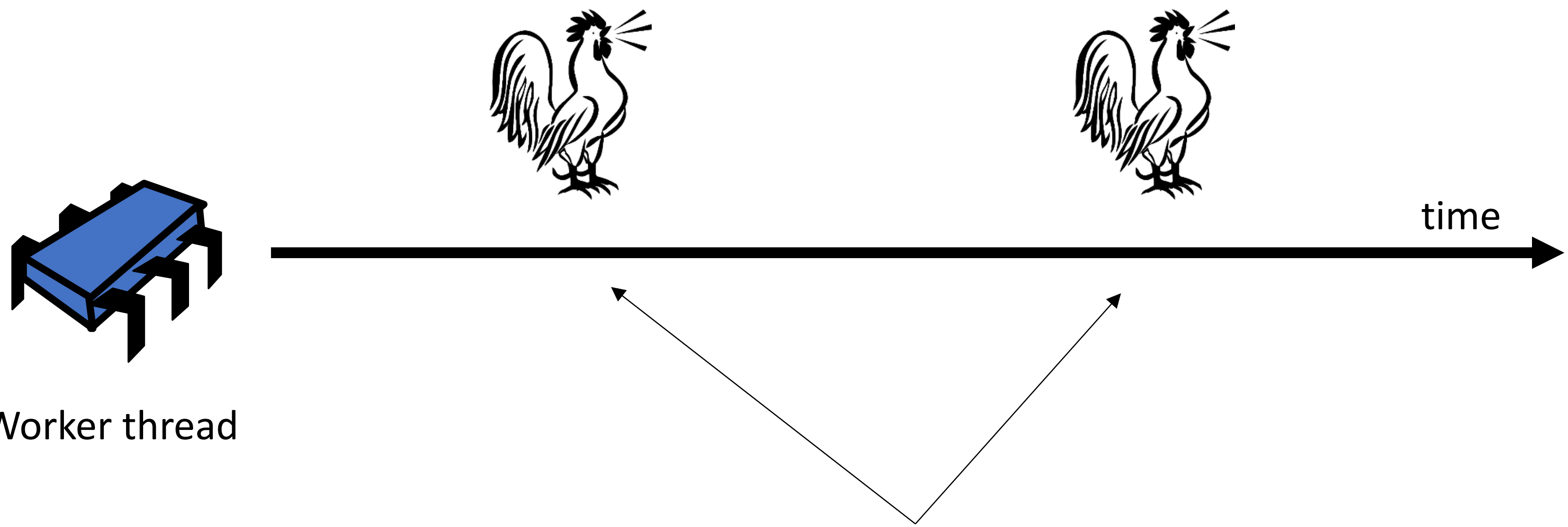
Two main concepts:

rooster processes and deferred reclamation

Rooster Processes



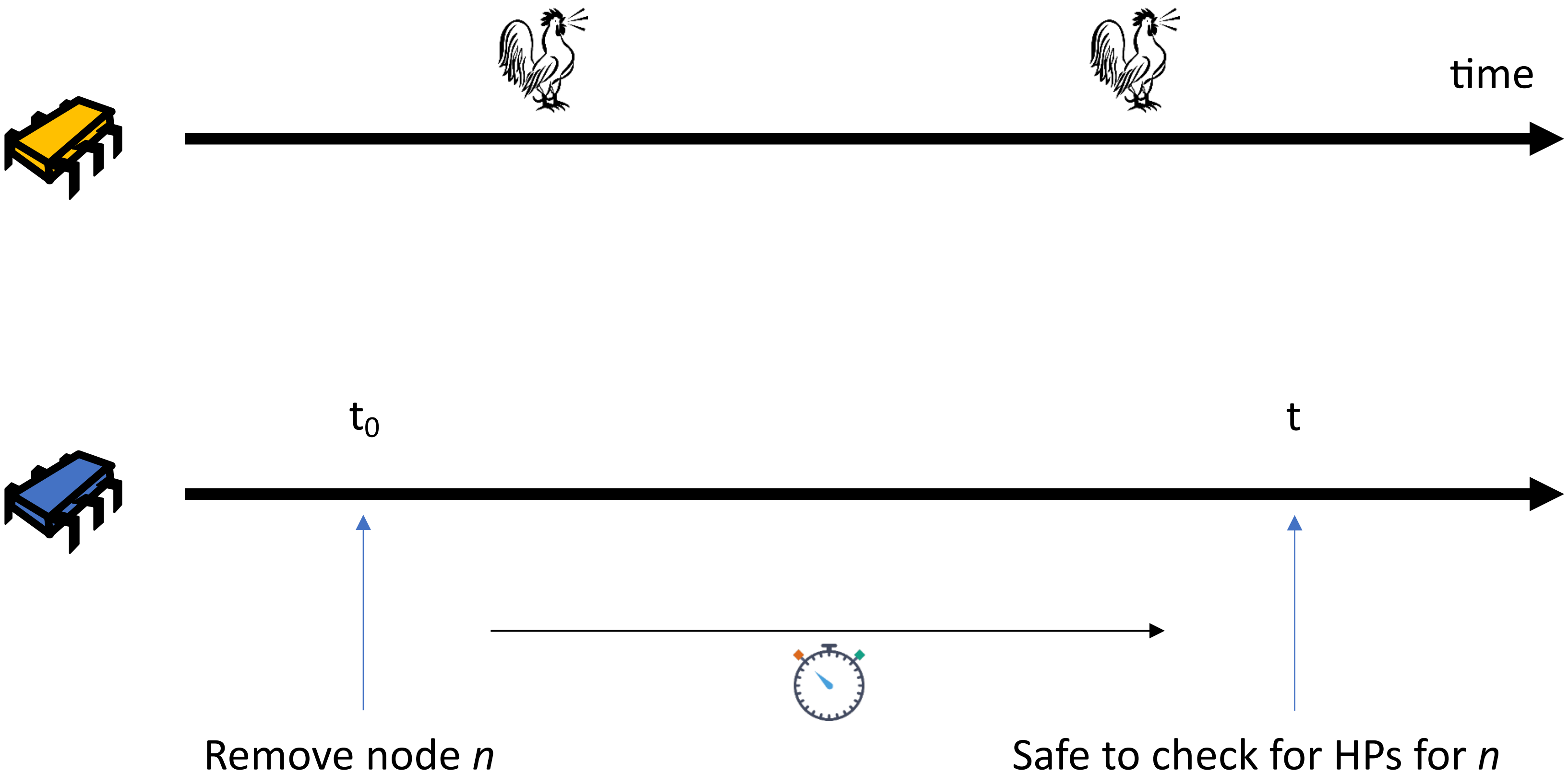
Rooster Processes



Context switch ≈ Memory Barrier

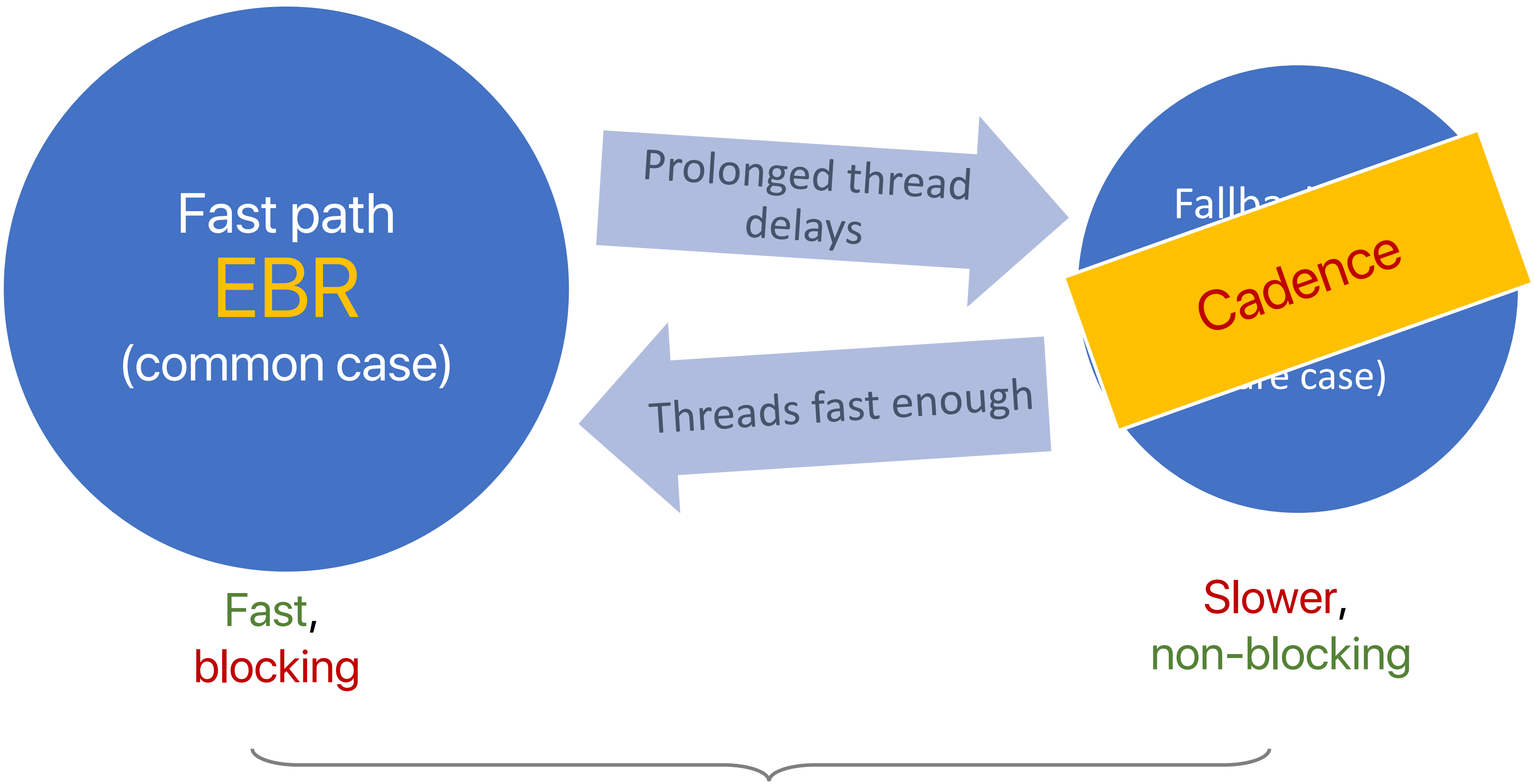
→ HP writes become visible

Deferred Reclamation



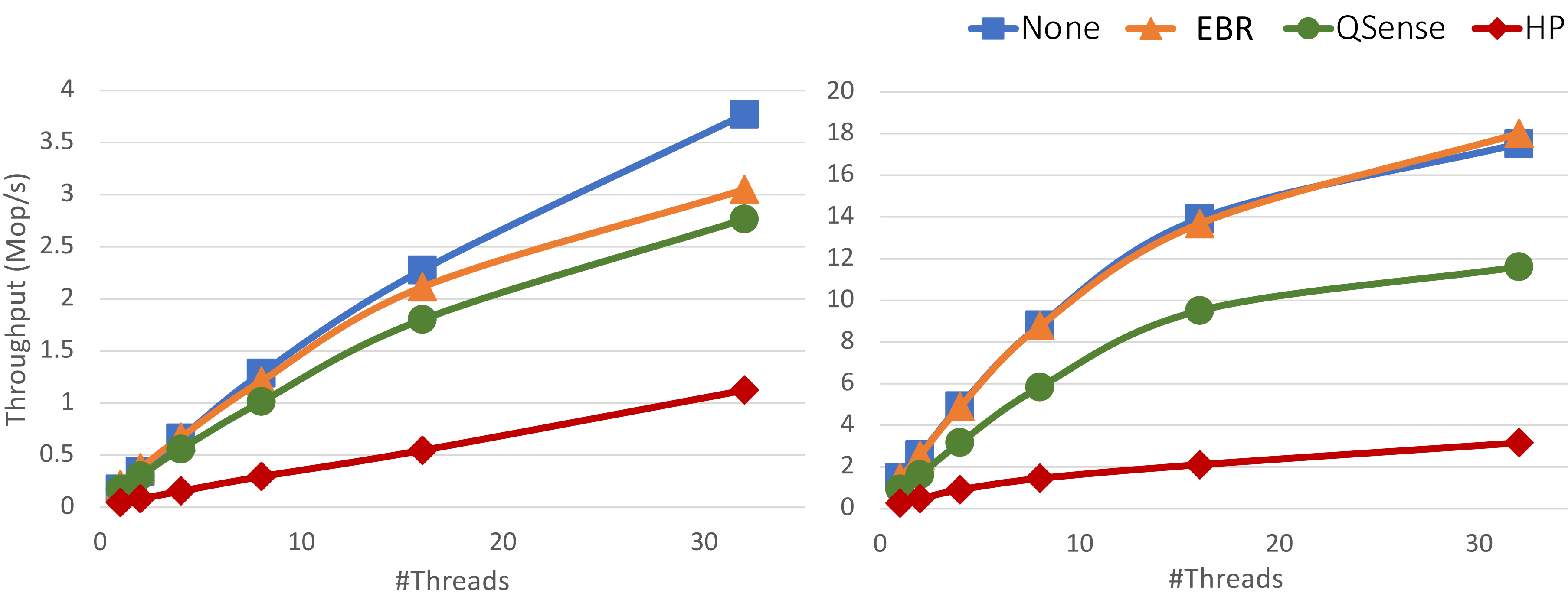
We no longer need memory barriers when using HPs.

QSense: Hybrid MR



Fast in the common case, resilient when necessary

QSense Performance – Common Case

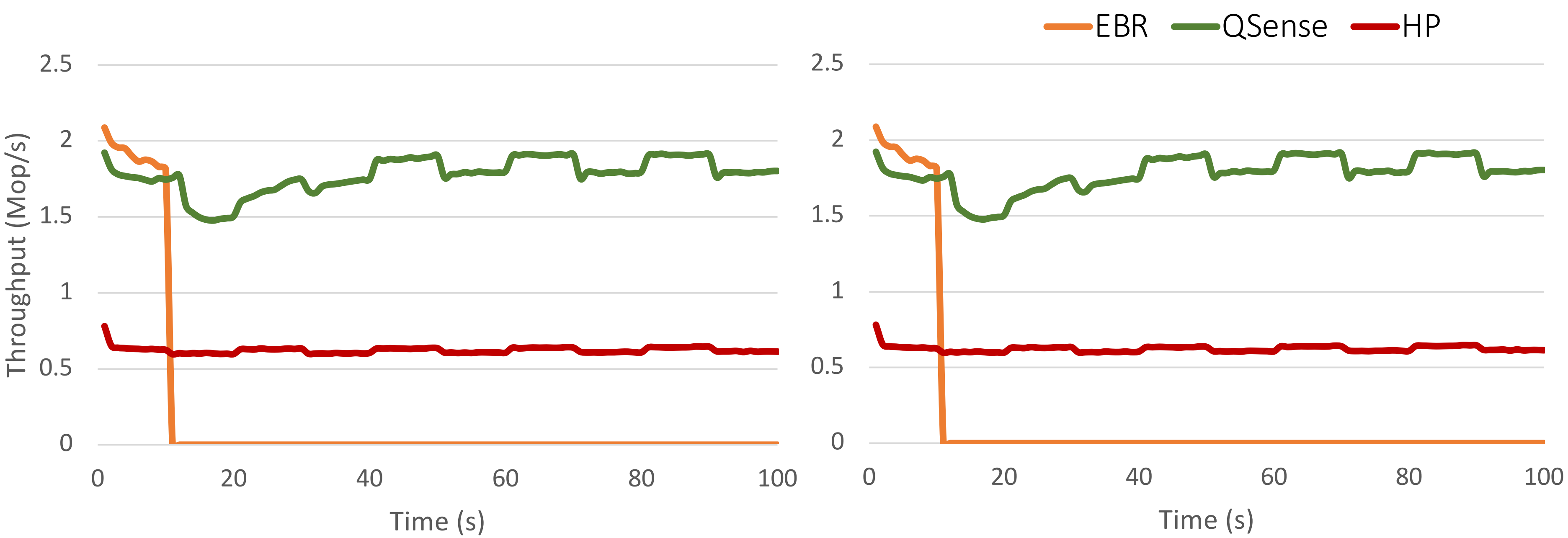


Linked list
[2K elements]

Skiplist
[32K elements]

[50% reads, 50% updates]

QSense Behavior with Delays



Linked list
[2k elements]

Skiplist
[32k elements]

[8 threads, 50% reads, 50% updates]

Recap

- What is memory reclamation?
- Traditional MR Techniques: LFRC, HP, EBR
- Cadence: HPs without memory barriers
- QSense: a hybrid of Cadence and EBR
  - Fast in the common case
  - Robust when necessary

Further Reading

- T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing, 67(12), 2007.
- J. D. Valois. Lock-free linked lists using compare-and-swap. PODC 1995.
- M.M. Michael, M.L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, Computer Science Department, University of Rochester. 1995.
- D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. PODC 2001.
- M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst., 15(6), 2004.
- **O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi. Fast and Robust Memory Reclamation for Concurrent Data Structures. SPAA 2016.**
- T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi. Log-Free Concurrent Data Structures. USENIX ATC 2018
- N. Cohen, R. Guerraoui, and I. Zablotchi. The Inherent Cost of Remembering Consistently. SPAA 2018