

Concurrent Memory Reclamation

Concurrent Algorithms

Fall 2017

Igor Zablotchi



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

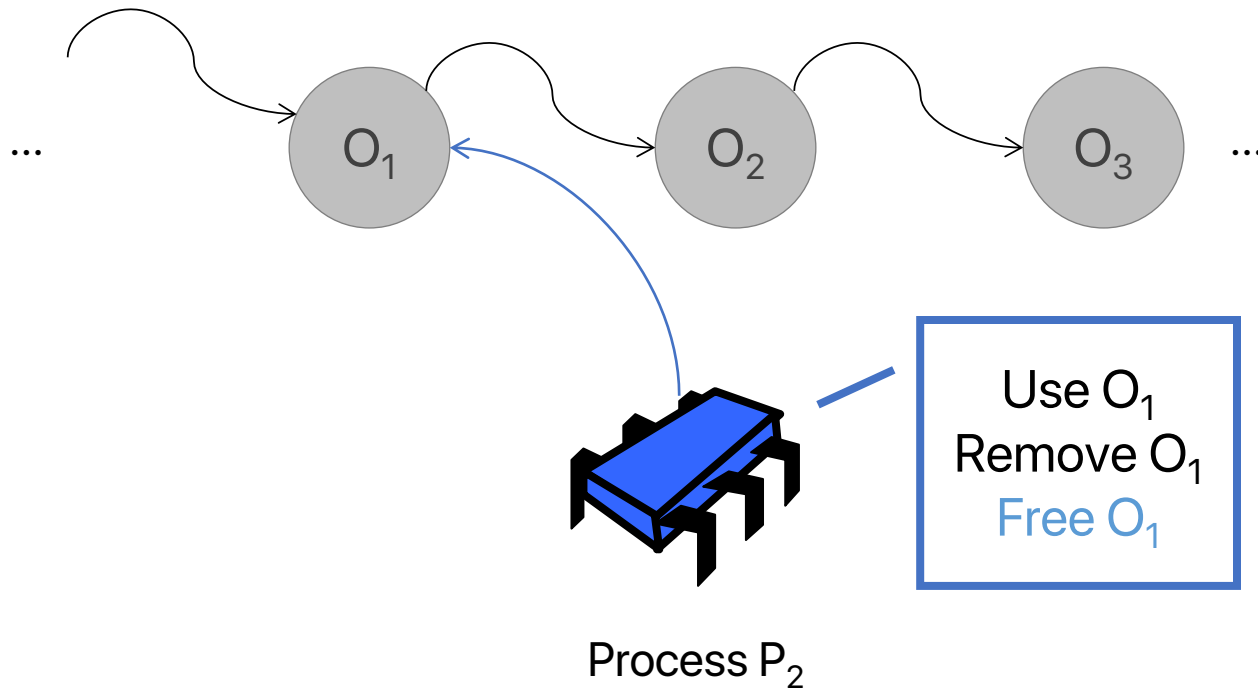
Explicit Memory Management

- 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

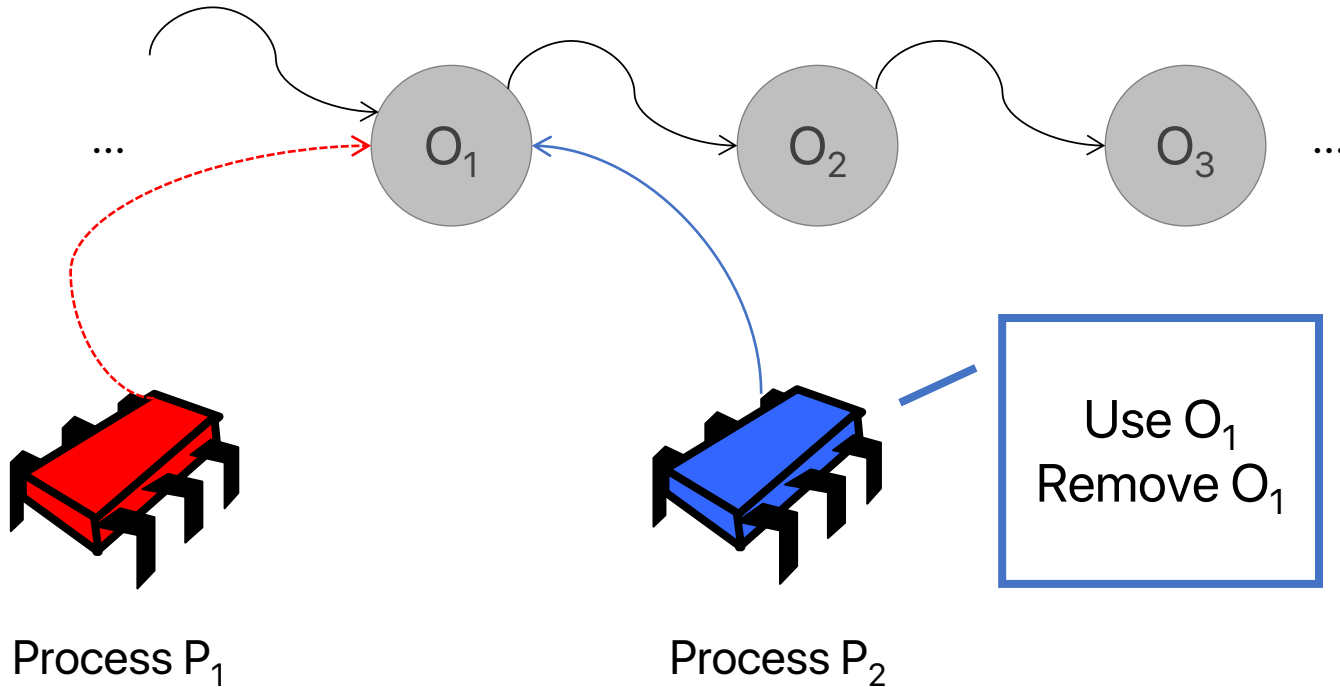
1-process MR is Easy



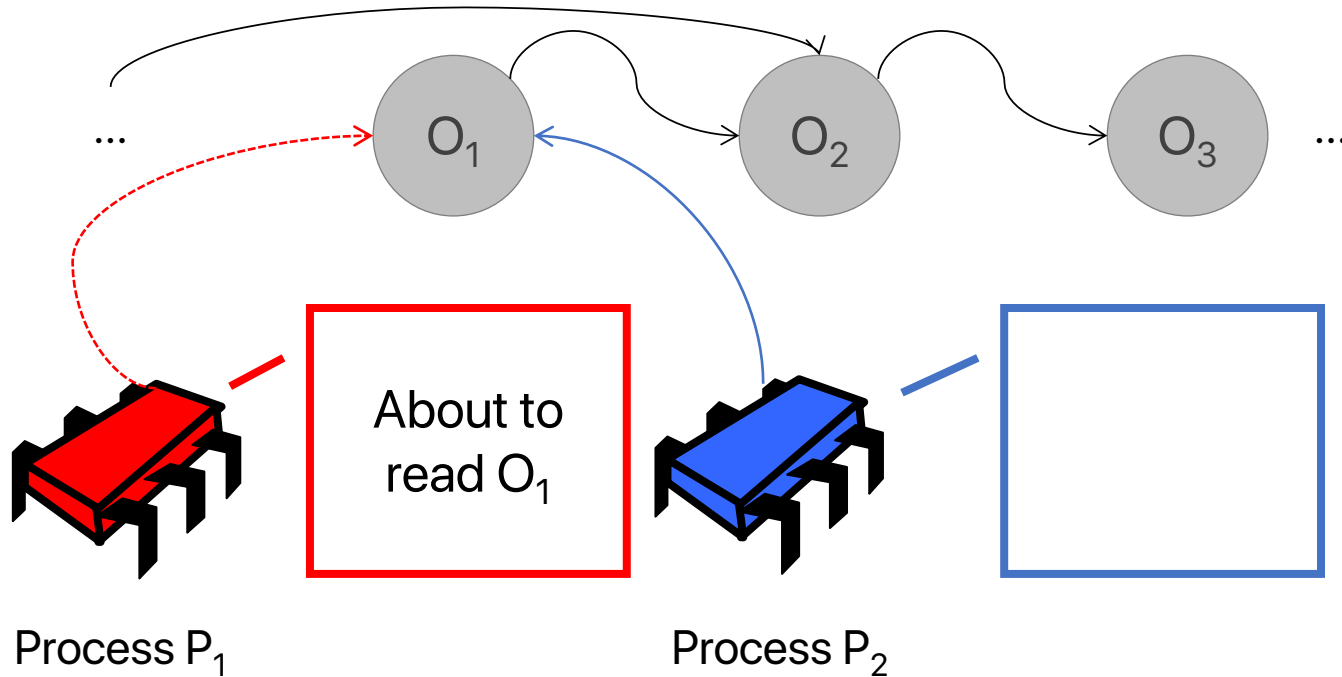
Concurrent MR is Difficult

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

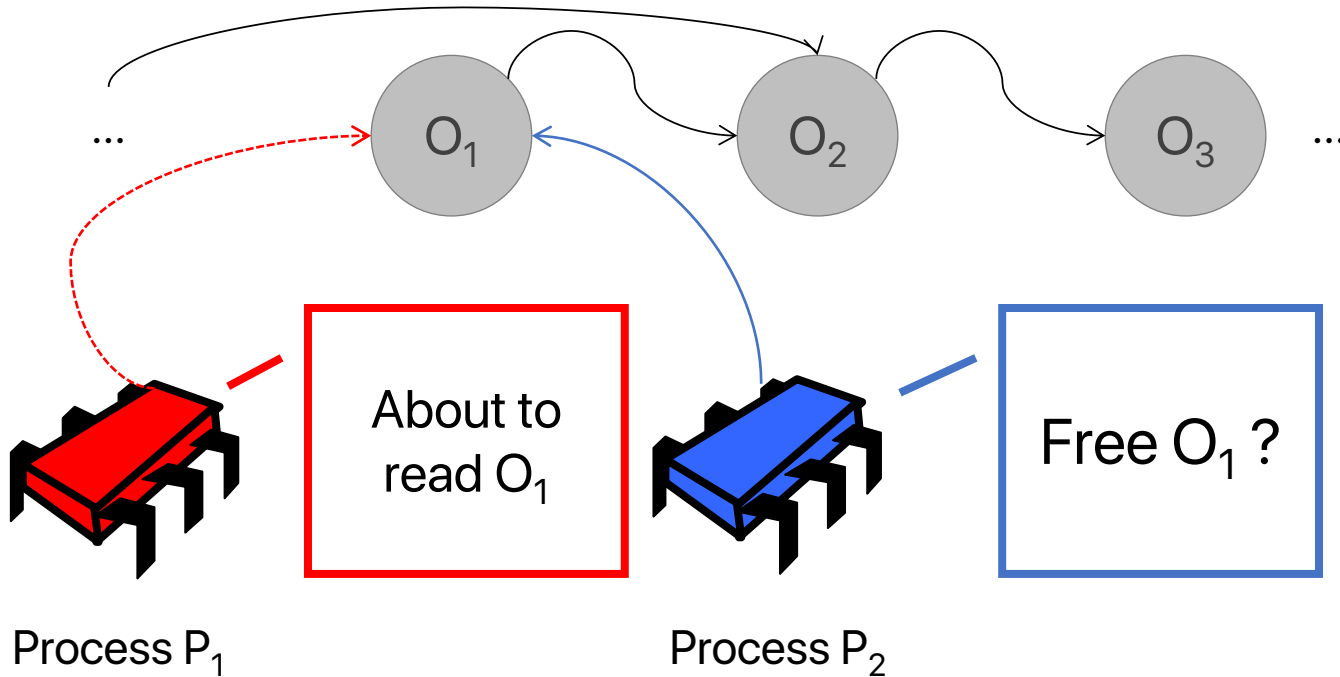
Concurrent MR is Difficult



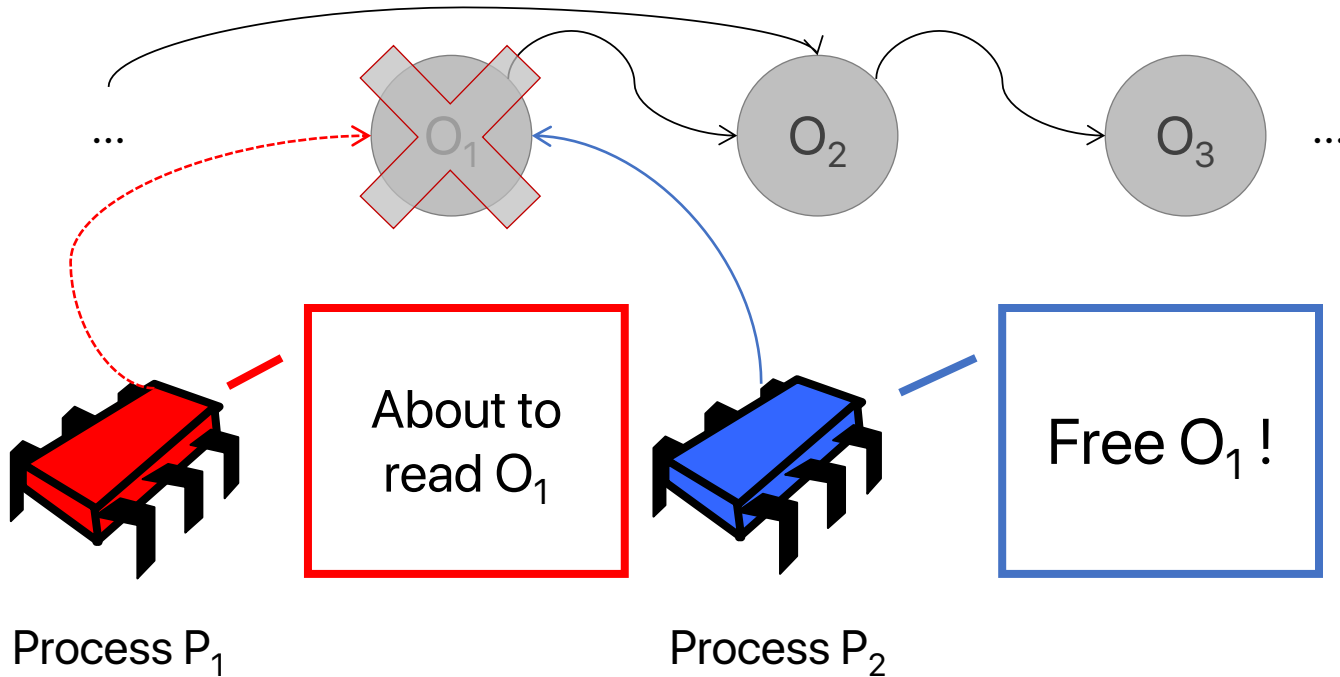
Concurrent MR is Difficult



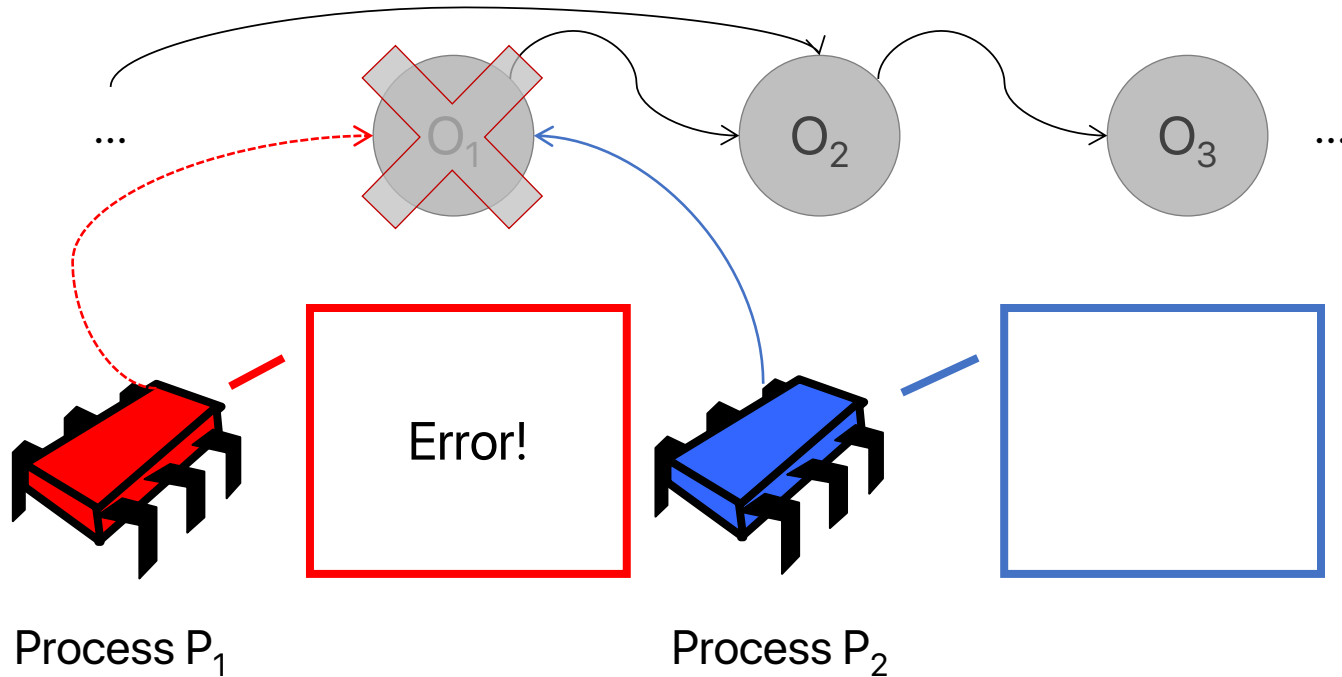
Concurrent MR is Difficult



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

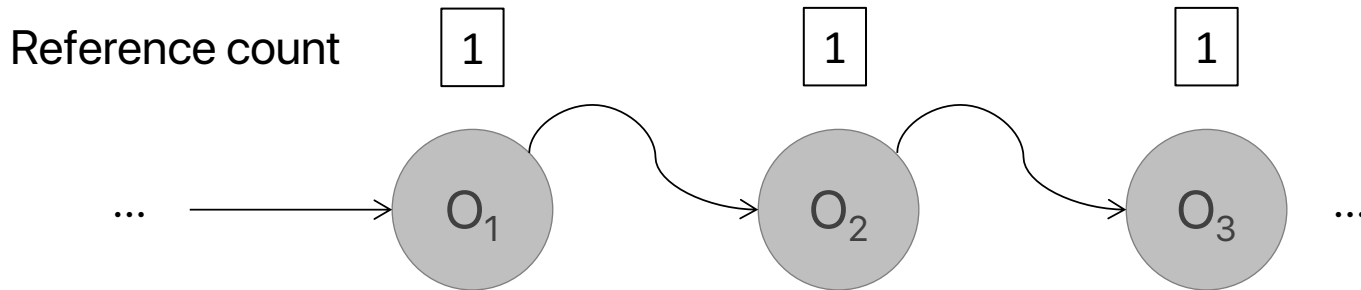
A Few MR Techniques

- Lock-free Reference Counting
- Hazard Pointers
- Epoch-Based Reclamation

Lock-free Reference Counting

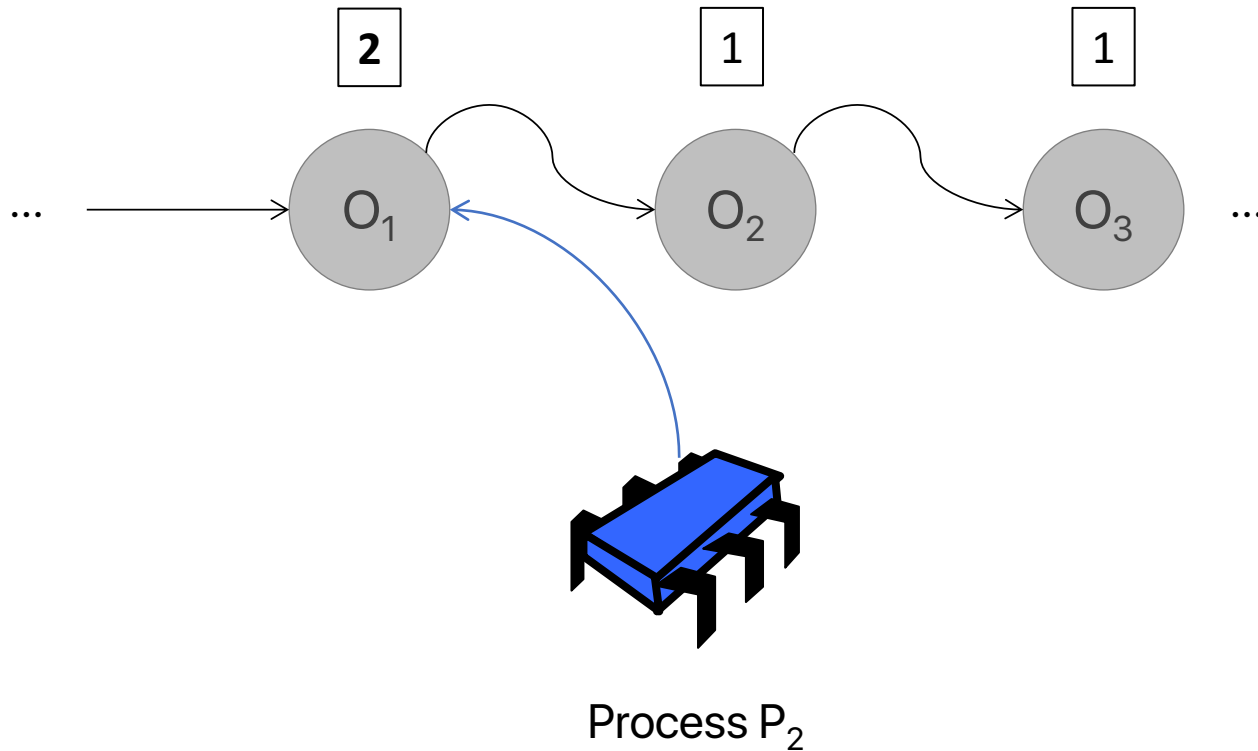
- Main idea:
 - For each memory location, keep track of how many references are held to it.
 - When there are 0 references, safe to reclaim.

LFRC Example



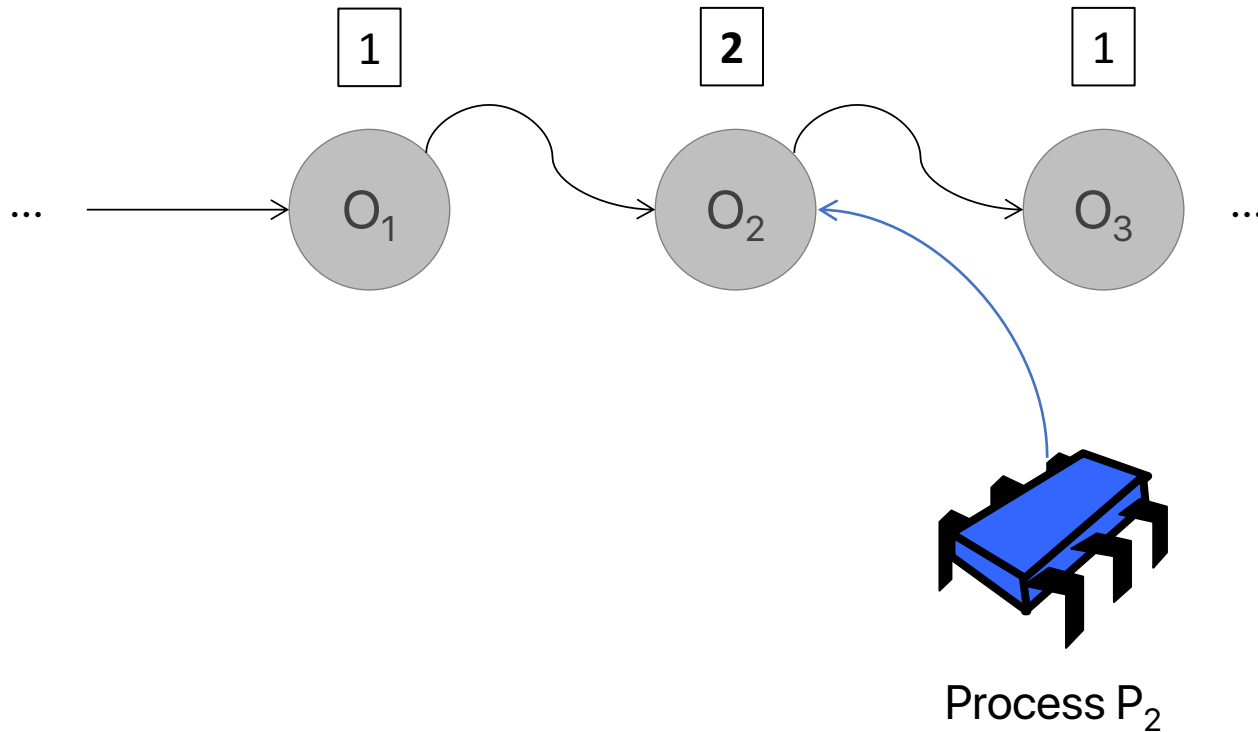
A linked list. No process has references. Each node has reference count = 1 (the reference from the previous node in the list).

LFRC Example



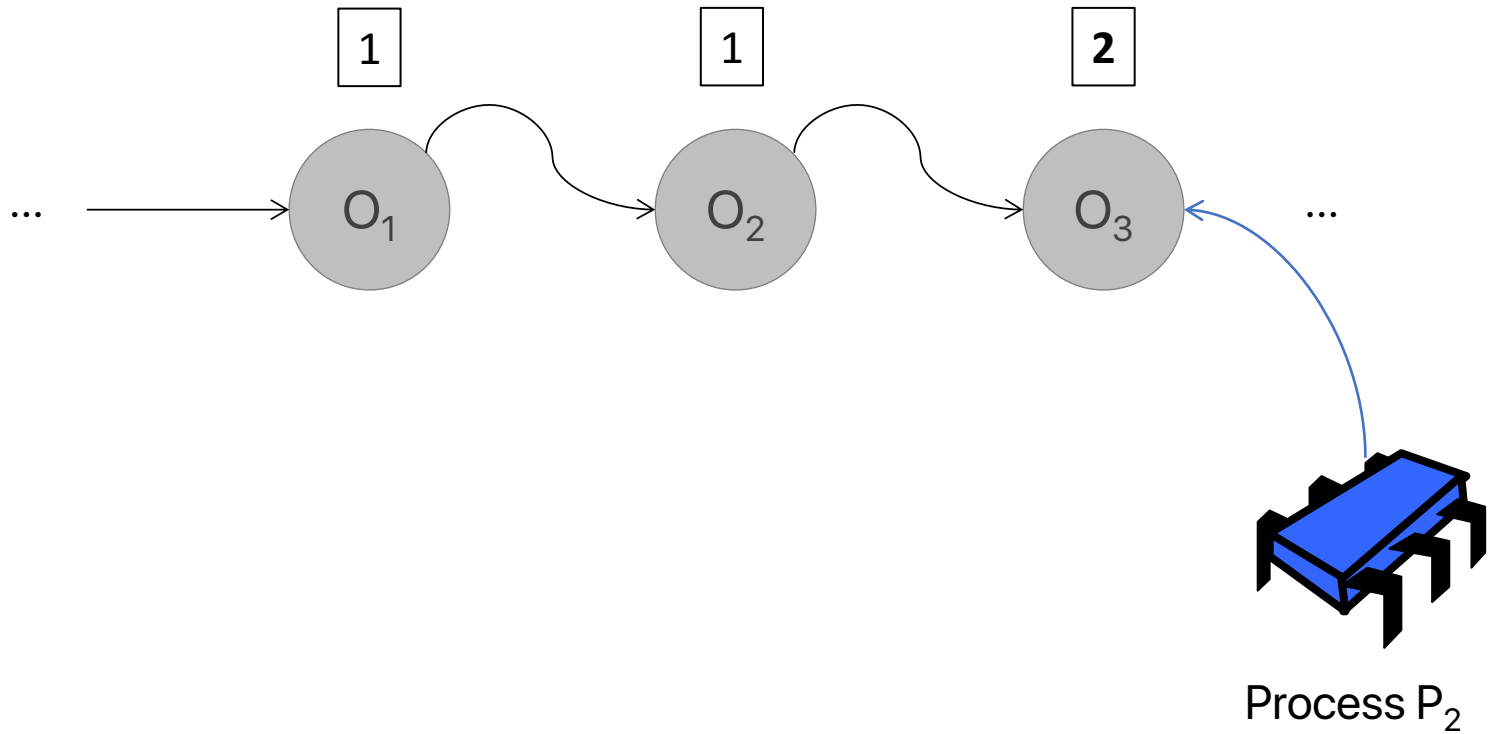
A thread is reading. The node that the thread is currently looking at has reference count = 2.

LFRC Example



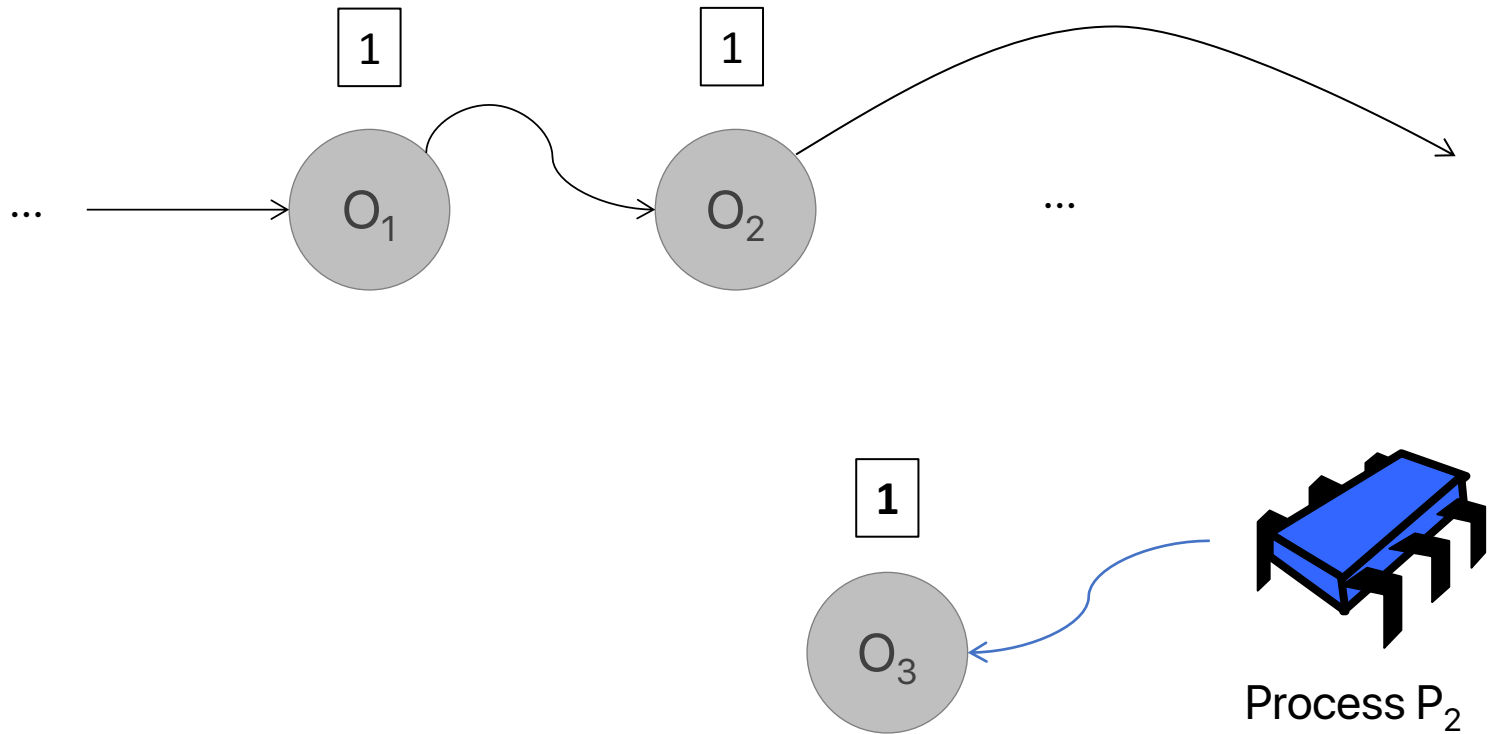
A thread is reading. The node that the thread is currently looking at has reference count = 2.

LFRC Example



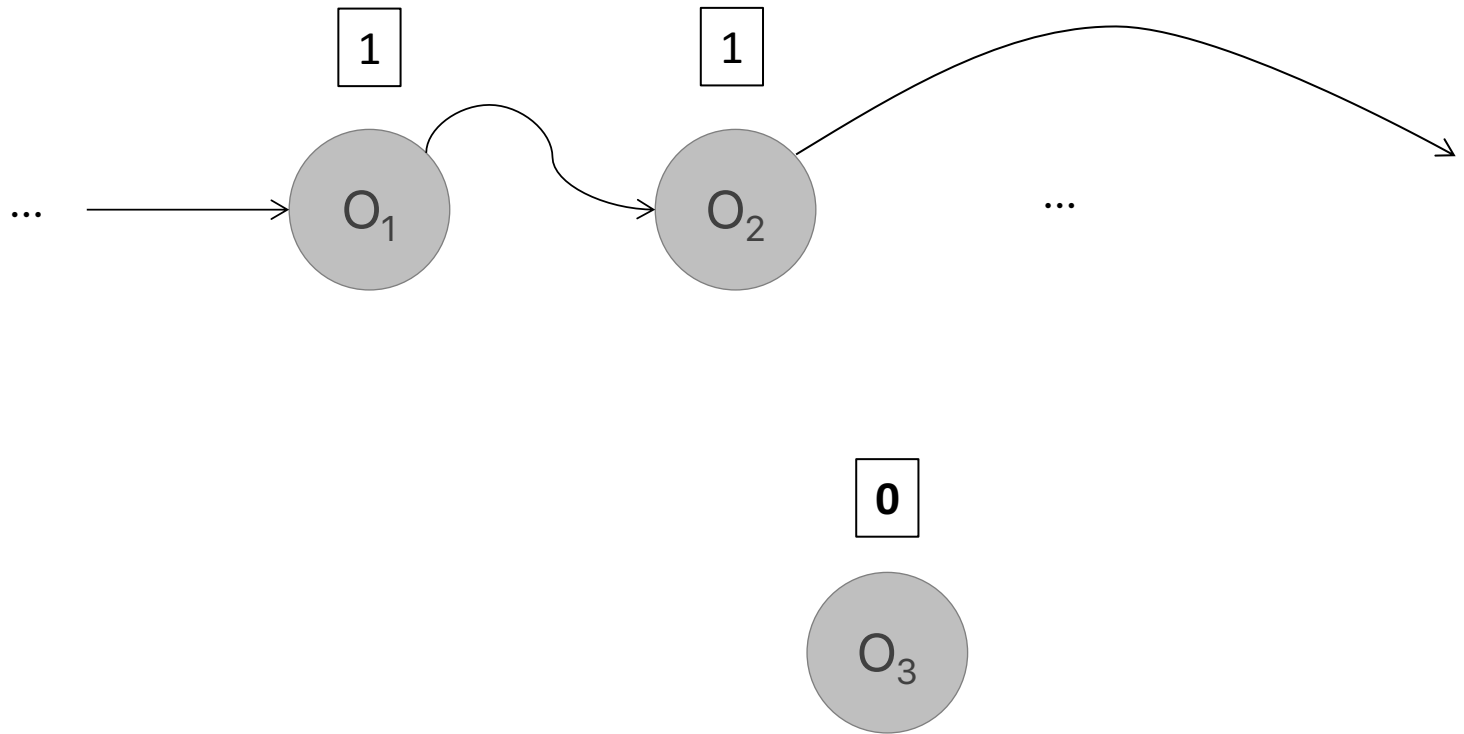
A thread is reading. The node that the thread is currently looking at has reference count = 2.

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.

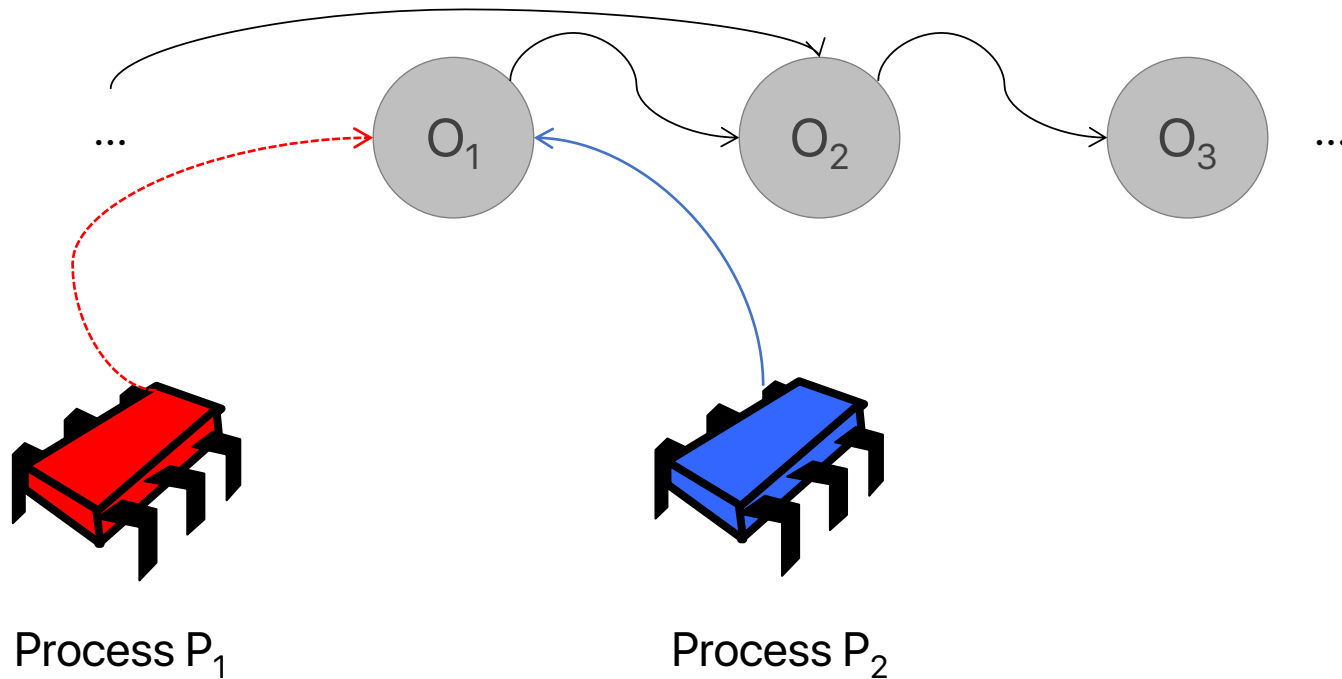
Pros and cons of LFRC

- ✓ 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!

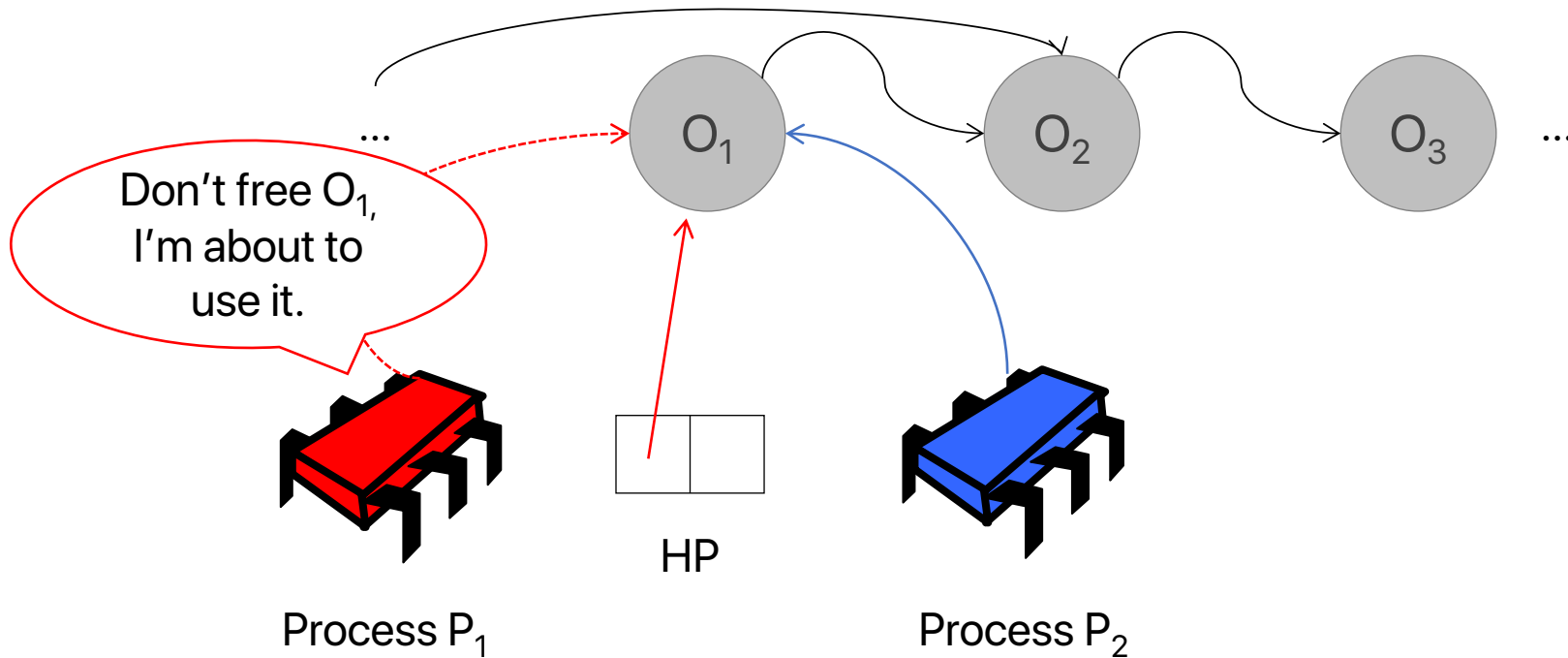
Hazard Pointers (HP)

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

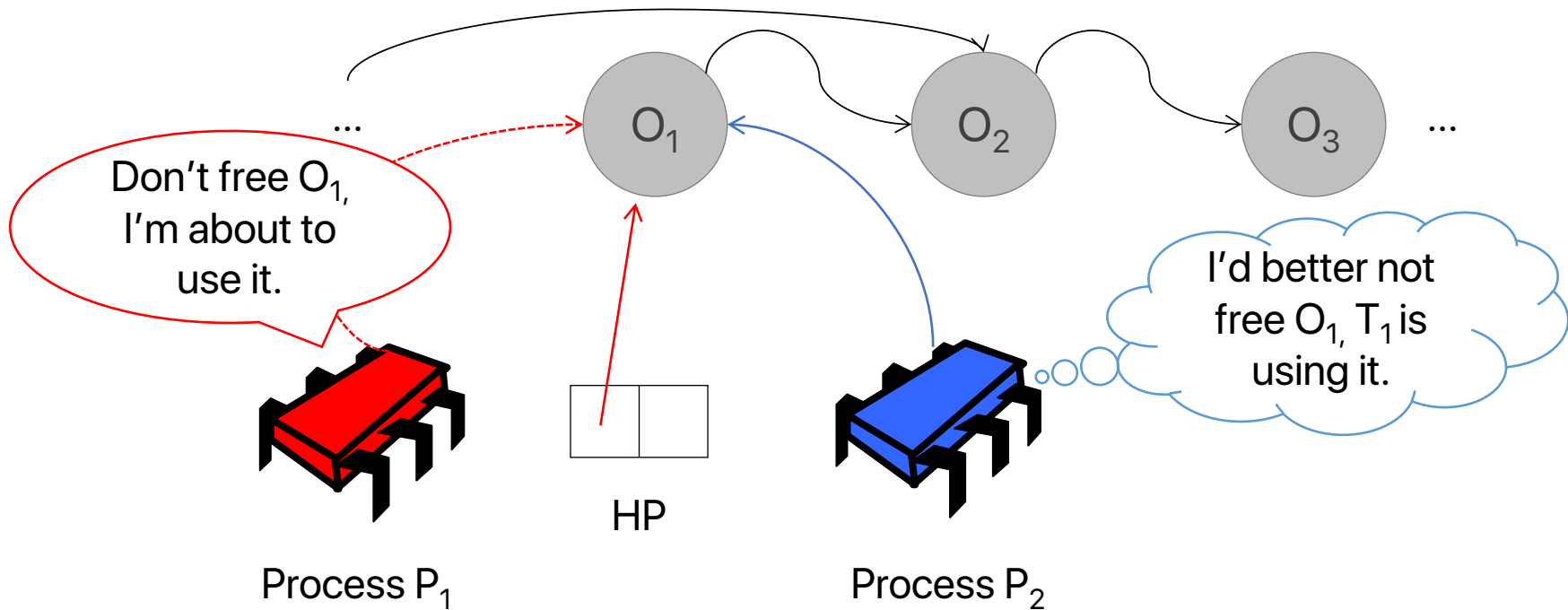
Hazard Pointers (HP)



Hazard Pointers (HP)



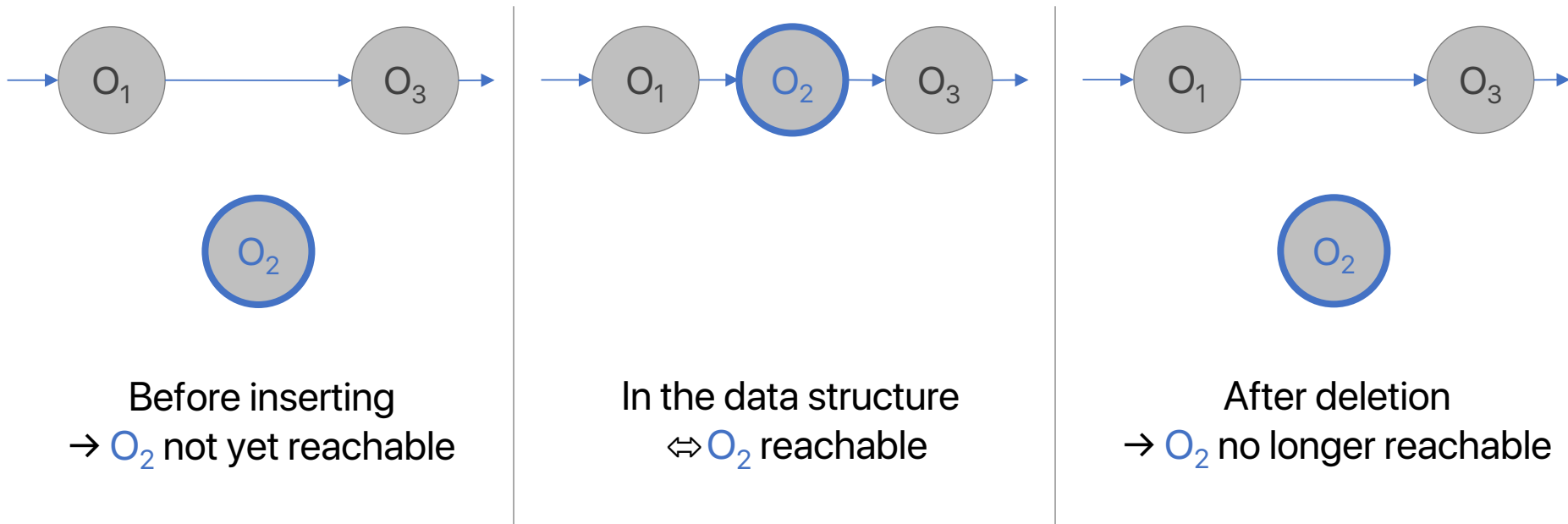
Hazard Pointers (HP)



HP – More Details

0. Reachability

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



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)

Pros and Cons of HP

- ✓ Limits memory use
- ✓ Lock-free
- ✗ Need to update HP on every access, even if read-only → bad performance
- ✗ Complex to implement & use → prone to errors

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

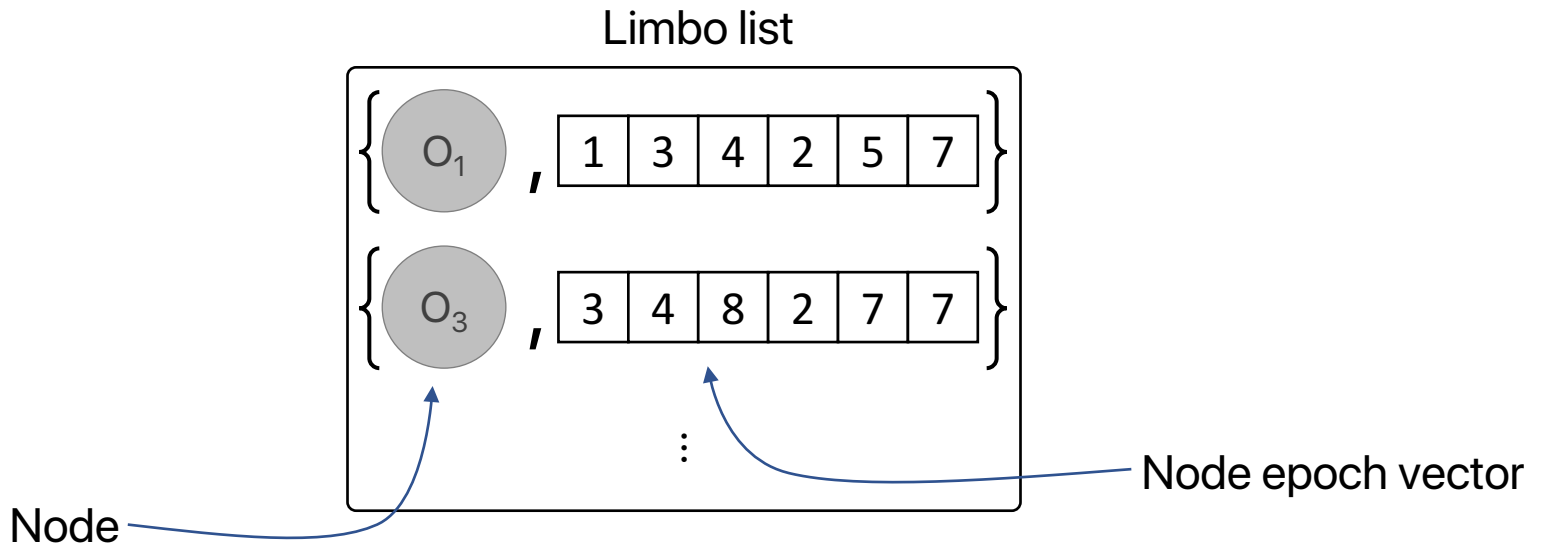
- Step 2: each process has an *epoch* (an integer, initially 0). The epoch is incremented by 1 when entering and exiting a critical section.

```
// 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:

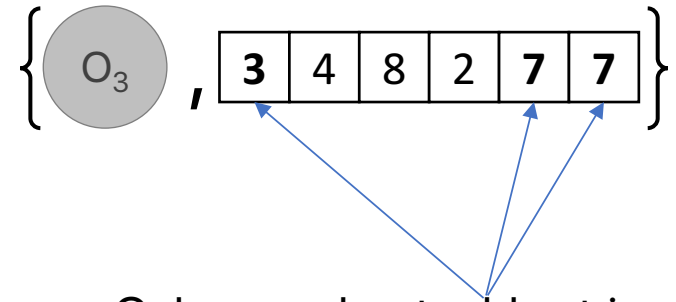
- `cur_vec` = current epoch vector
- For each node n in the limbo list:
 - `node_vec` = n 's epoch vector
 - For each process i :
 - if `node_vec[i]` is odd
 - if `node_vec[i] >= cur_vec[i]`
 - Continue to next node
 - Free node

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 $node_vec[i] \geq cur_vec[i]$
 - Continue to next node
 - Free node



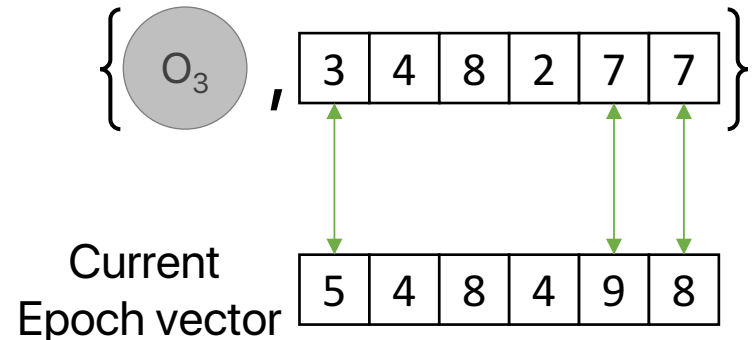
Only care about odd entries
(processes inside crit. sec.)!
Processes outside crit. sec.
cannot access this node.

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 $node_vec[i] \geq cur_vec[i]$
 - Continue to next node
 - Free node



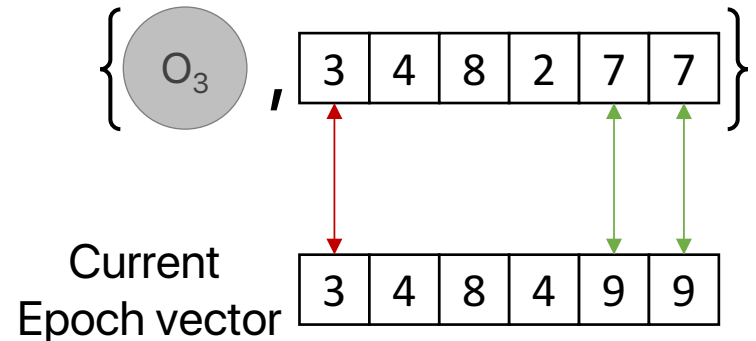
OK to reclaim!

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 $node_vec[i] \geq cur_vec[i]$
 - Continue to next node
 - Free node



Not OK to reclaim!

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

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. Zabolotchi. Fast and Robust Memory Reclamation for Concurrent Data Structures. *SPAA 2016*.