Speed-reading for Paygo

Resolve Linux Buffered I/O bottleneck and accelerate the reclamation of page

이현민 (2017030182)
박건욱 (2017029607)

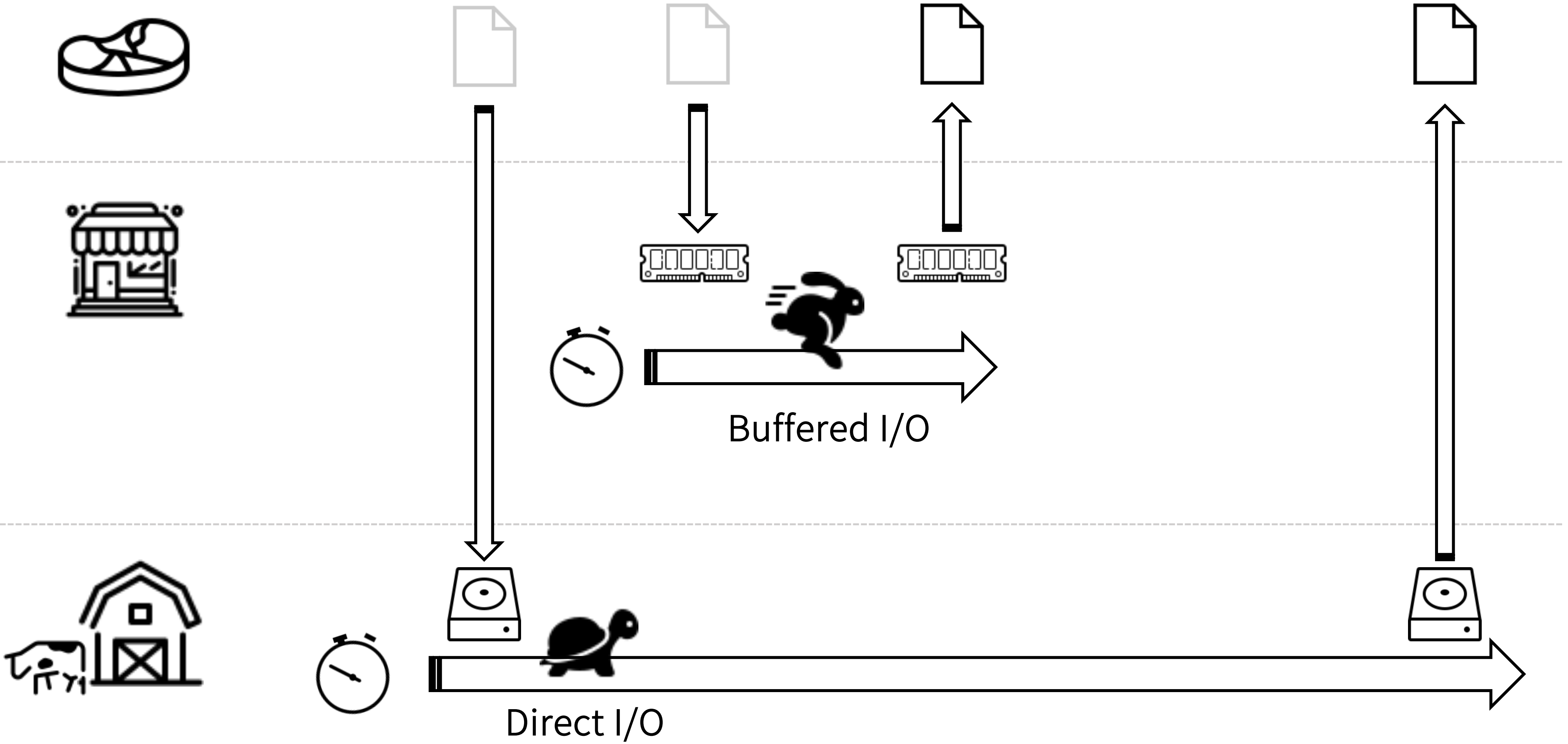
Index

1. Lockless page-cache
2. Distributed reference count
3. Distribution manager
4. Speed-reading for paygo
5. Evaluation
6. Limitation

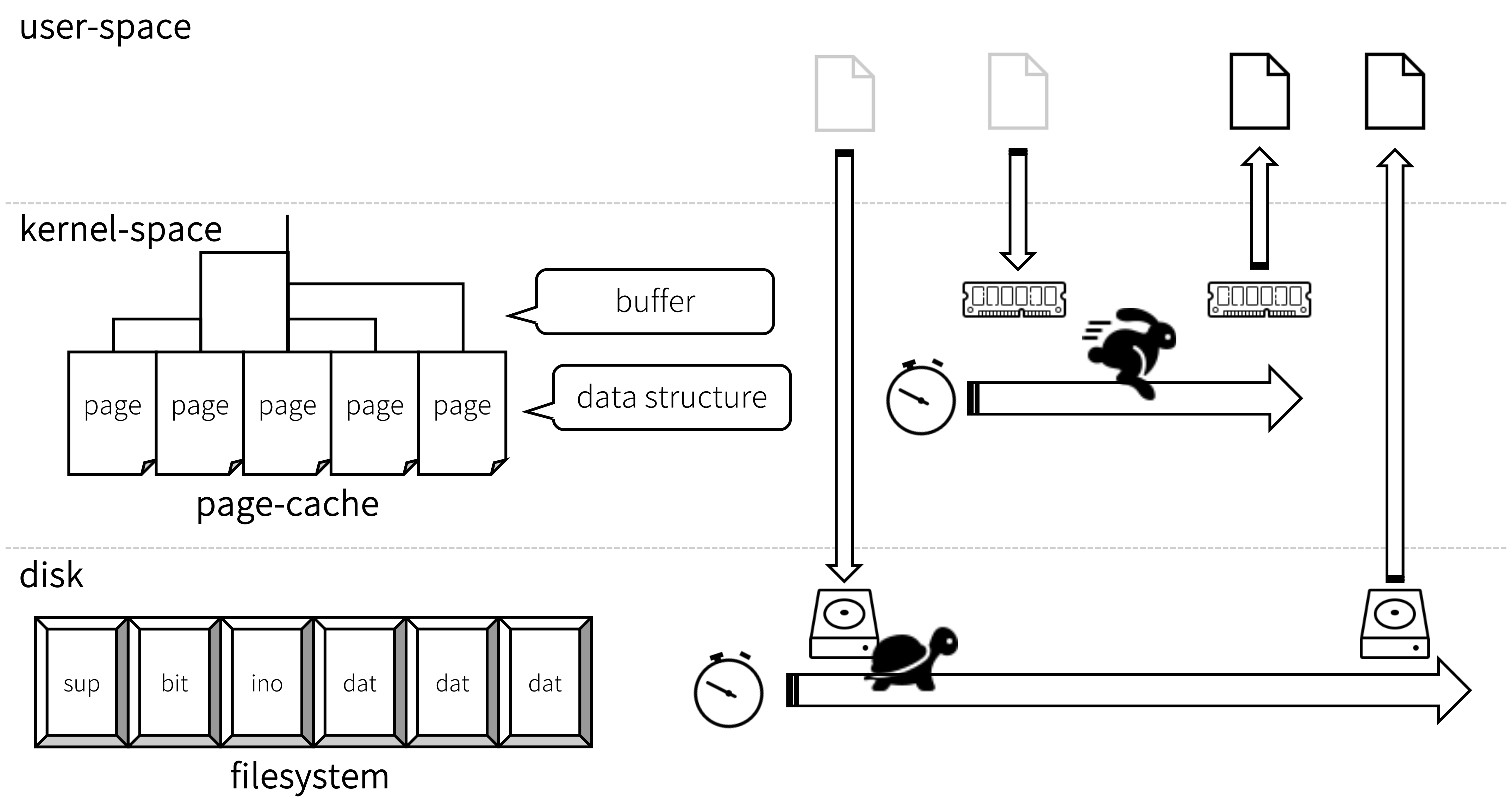
Introduction & background

Lockless page-cache

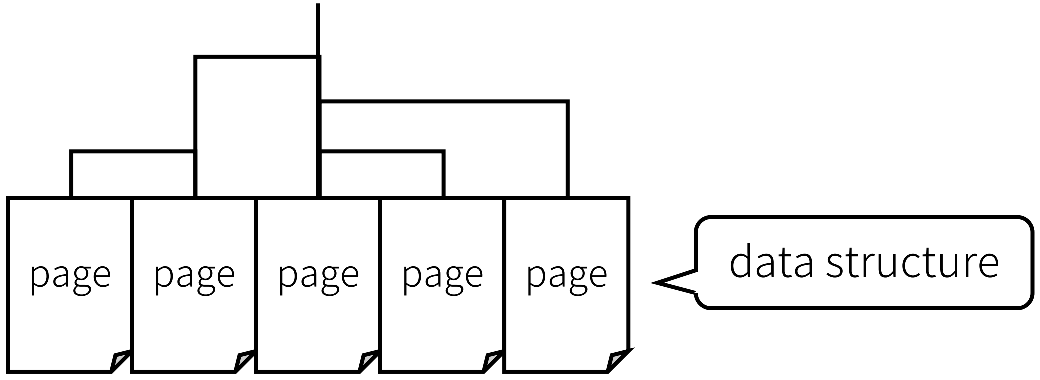
Buffered I/O is faster than direct I/O



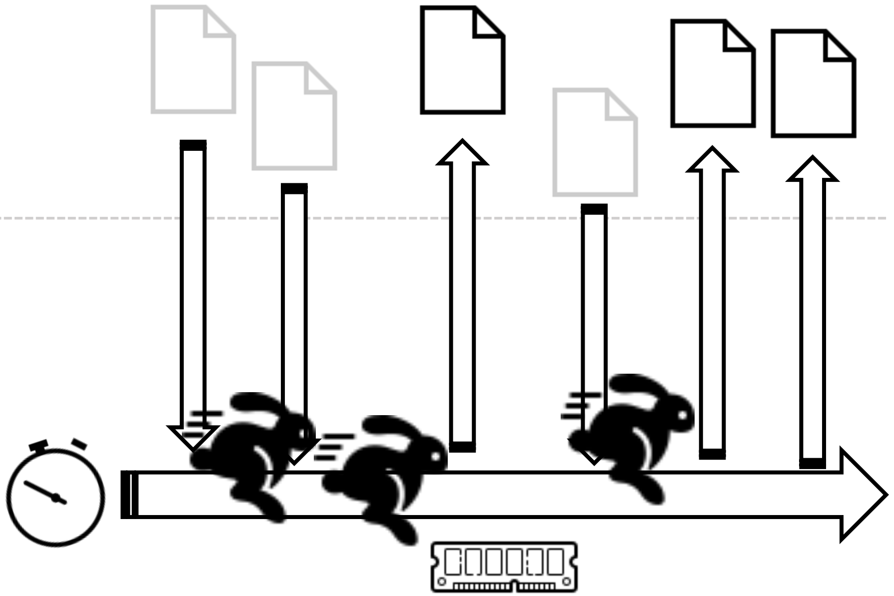
Linux calls the buffer page-cache



Design principles of page-cache



page-cache

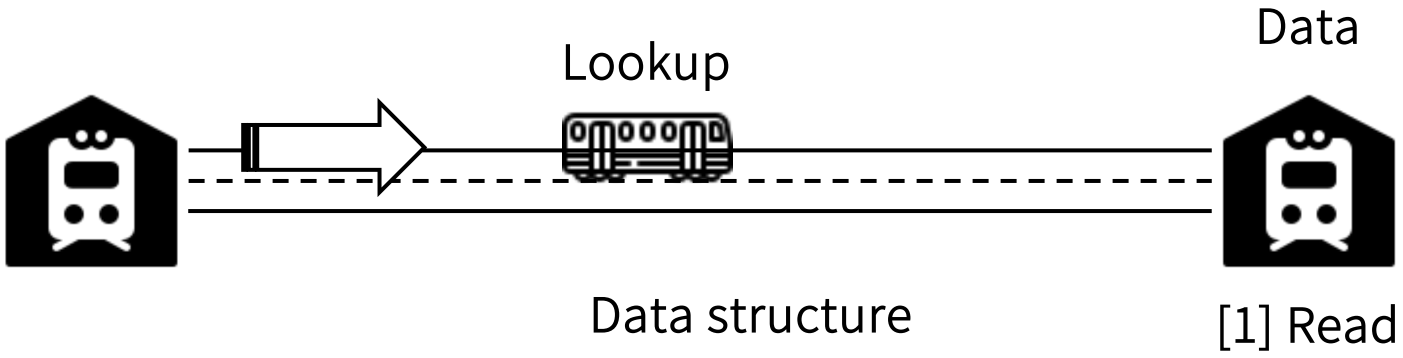


- Page-cache should be concurrent.
- Lookups for page-cache should be fast. no waiting during lookup

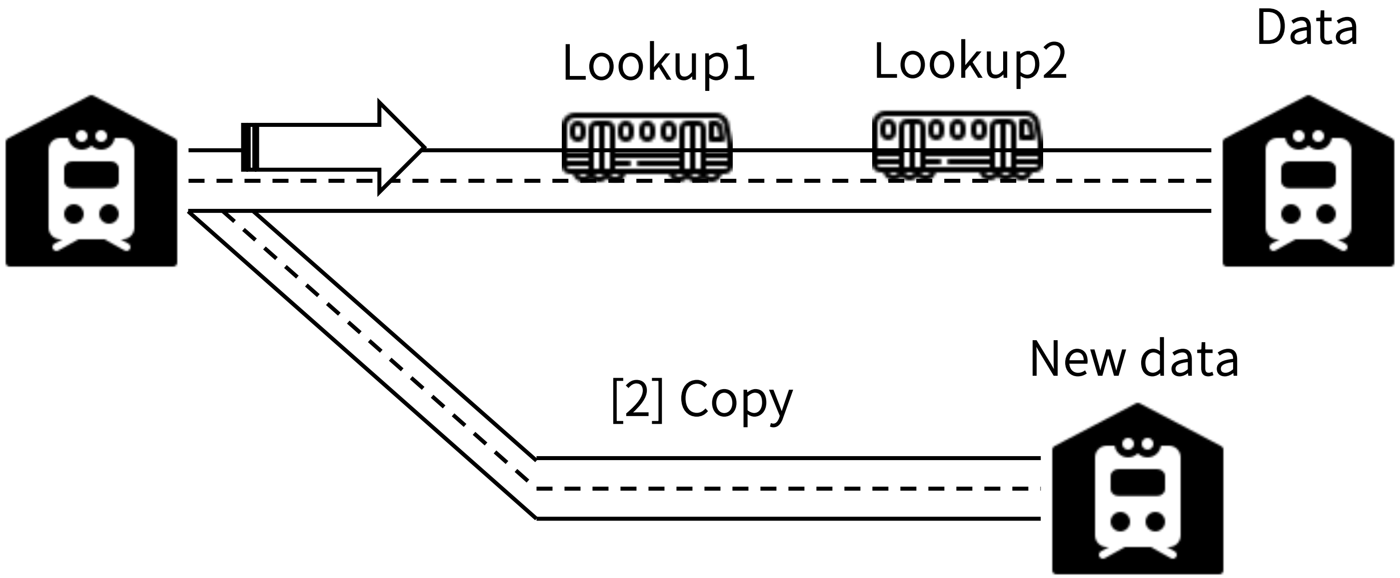
Read-optimized techniques

Technique	Overhead of read	assume a single writer Overhead of write
• Sequential lock	retry during other's write	-
• Readers-writer lock	cache-line contention	-
• <u>Read-copy-update</u>	calculating the end of grace period	copy and update

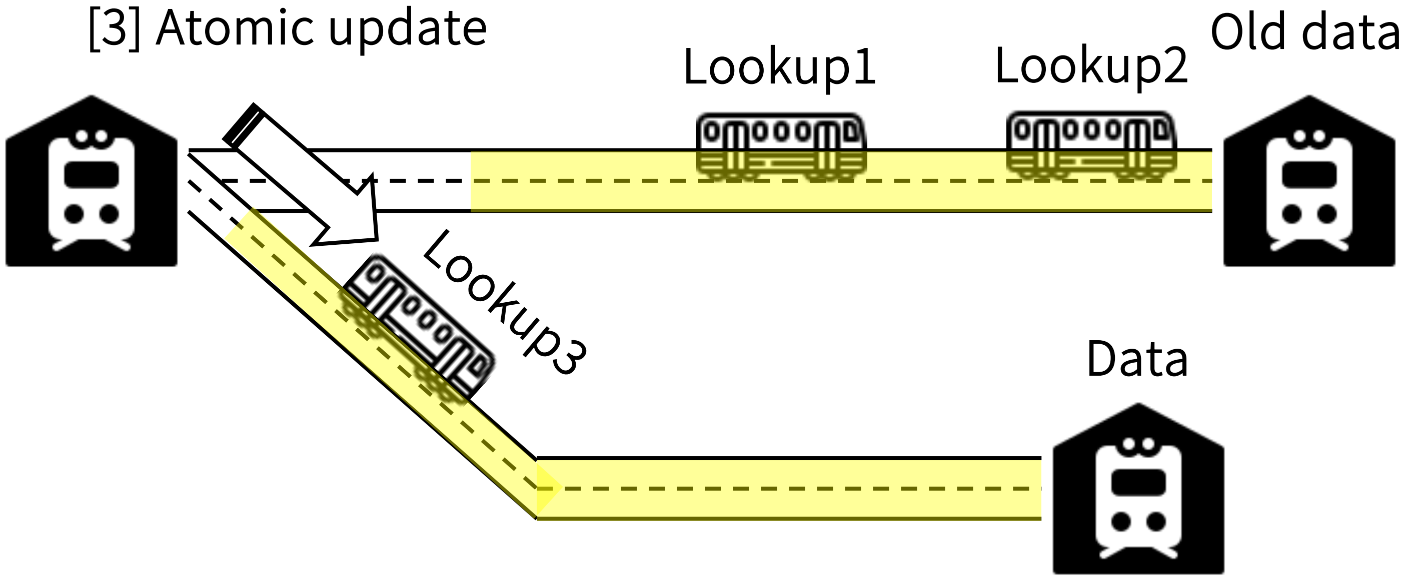
RCU: Read



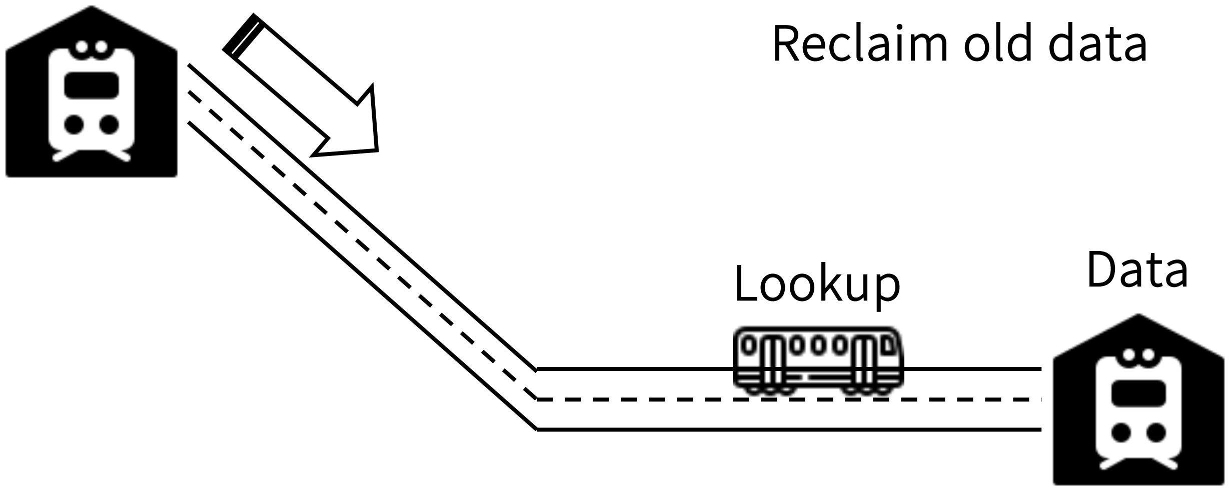
RCU: Copy



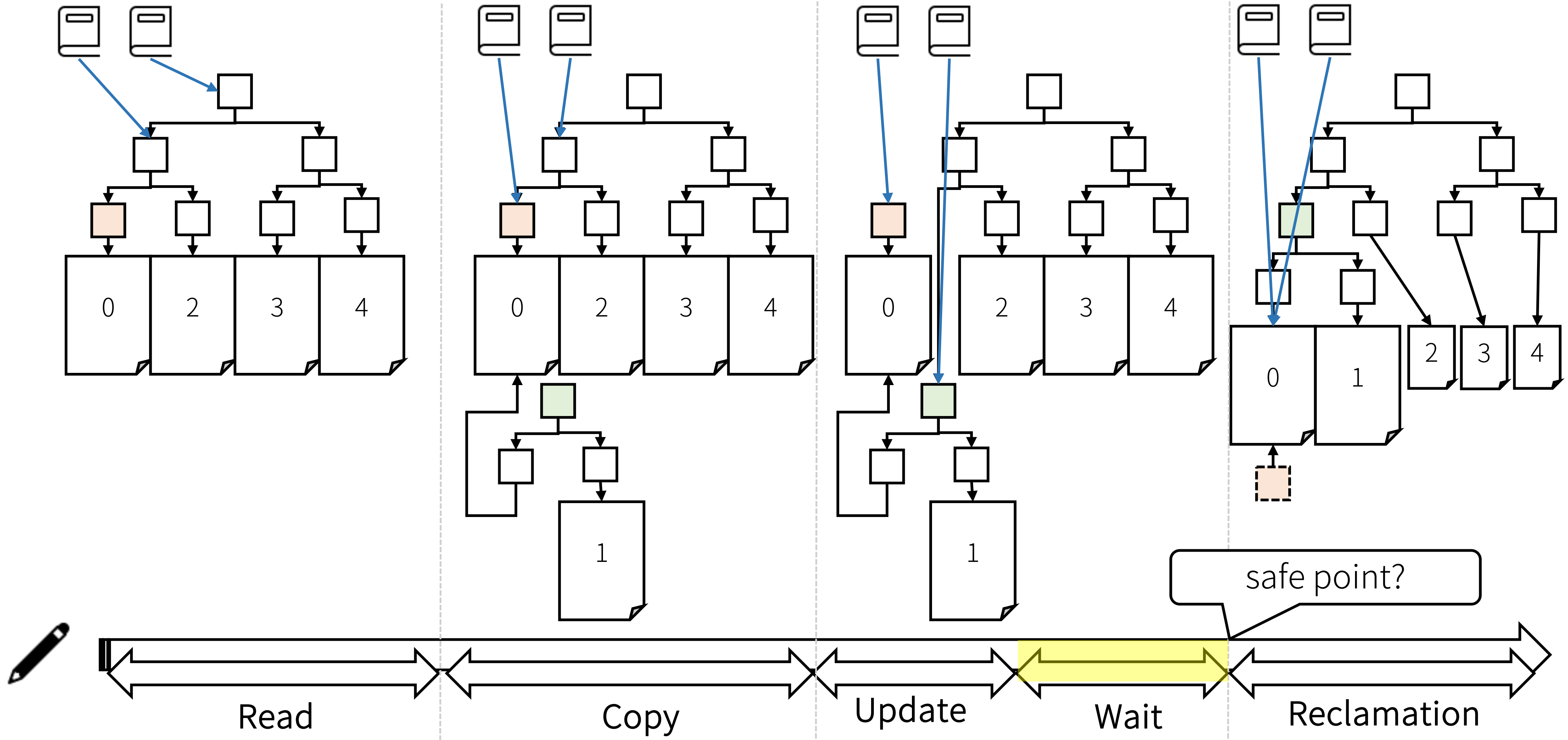
RCU: Update & grace period



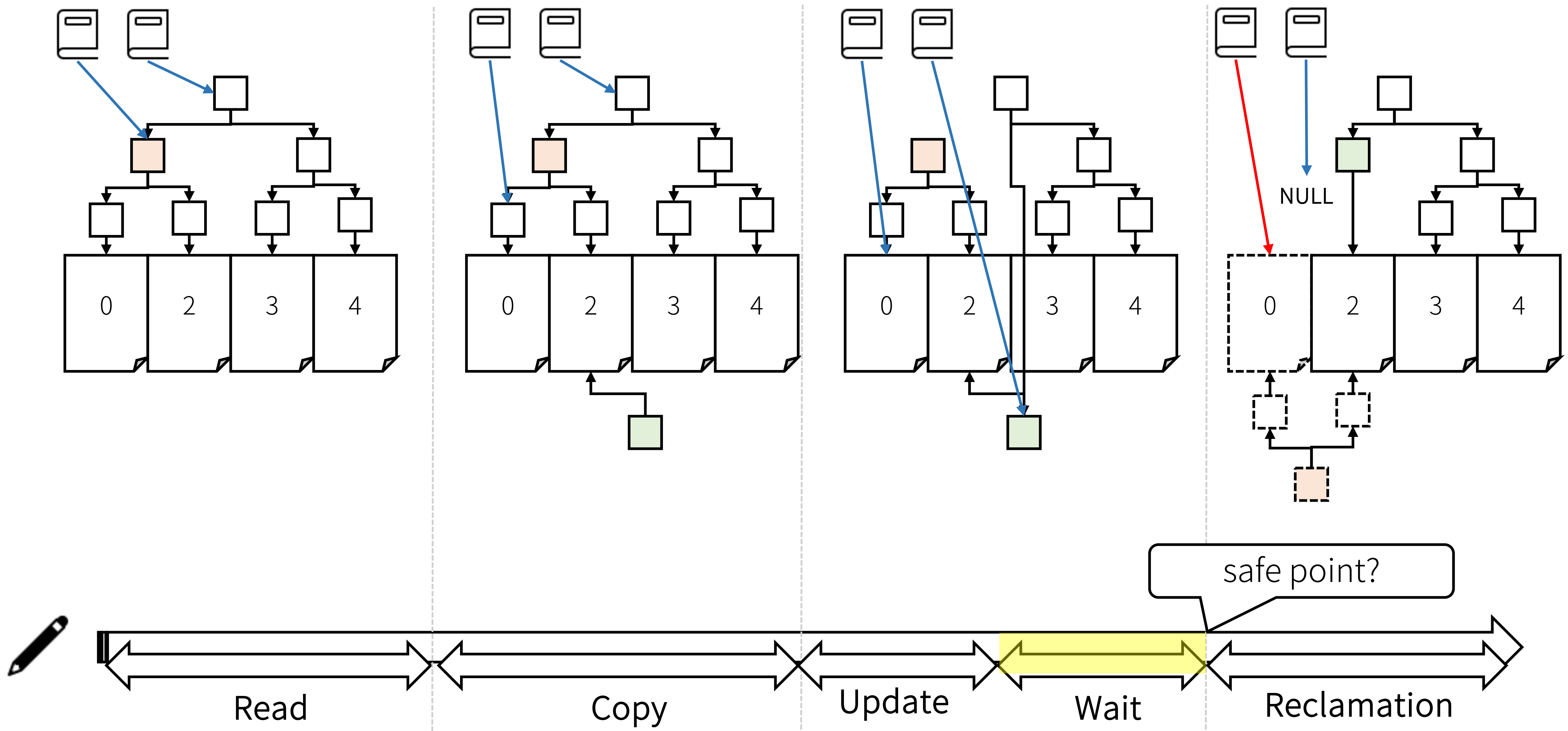
RCU: reclamation



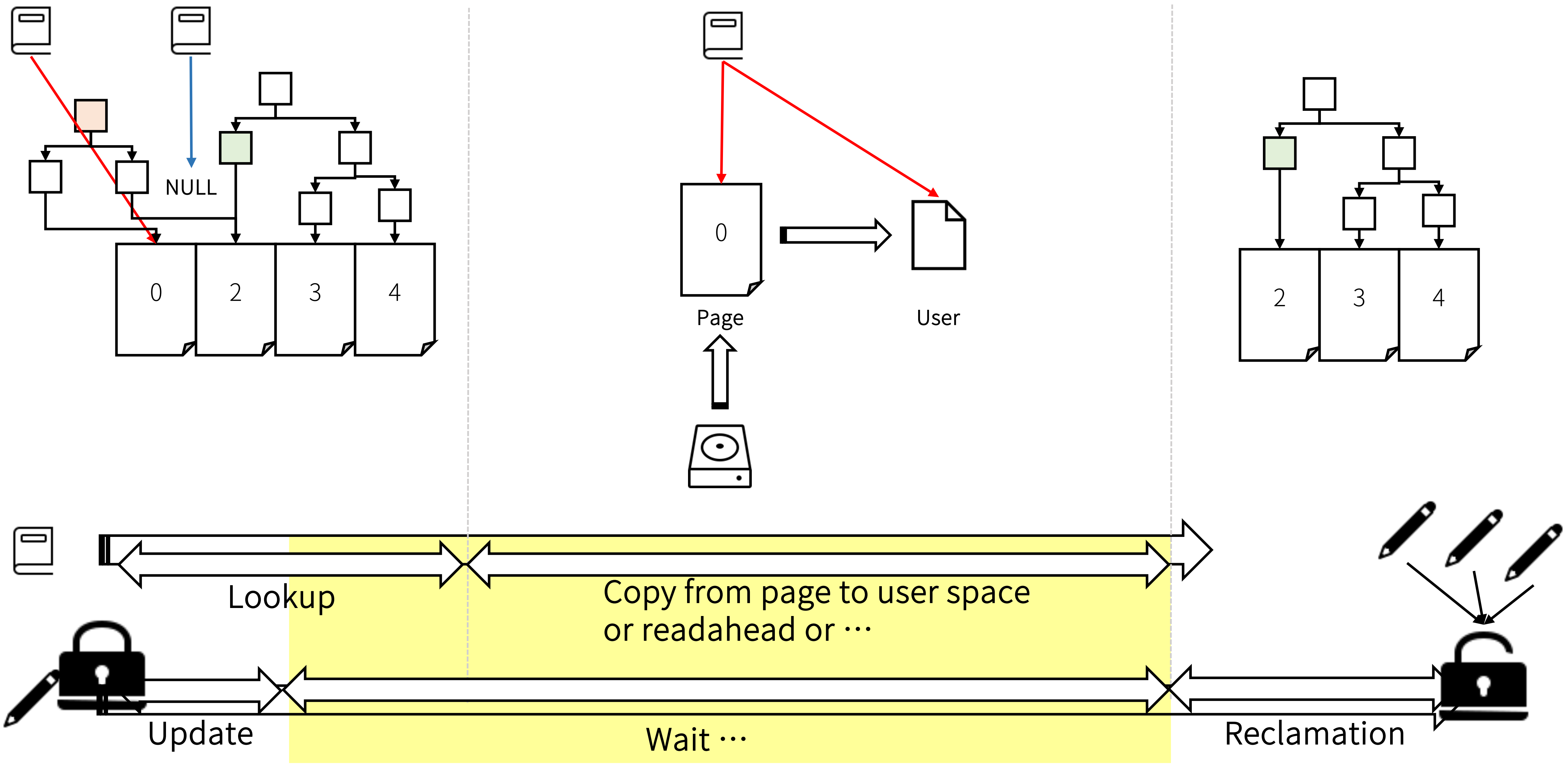
Lookups during create



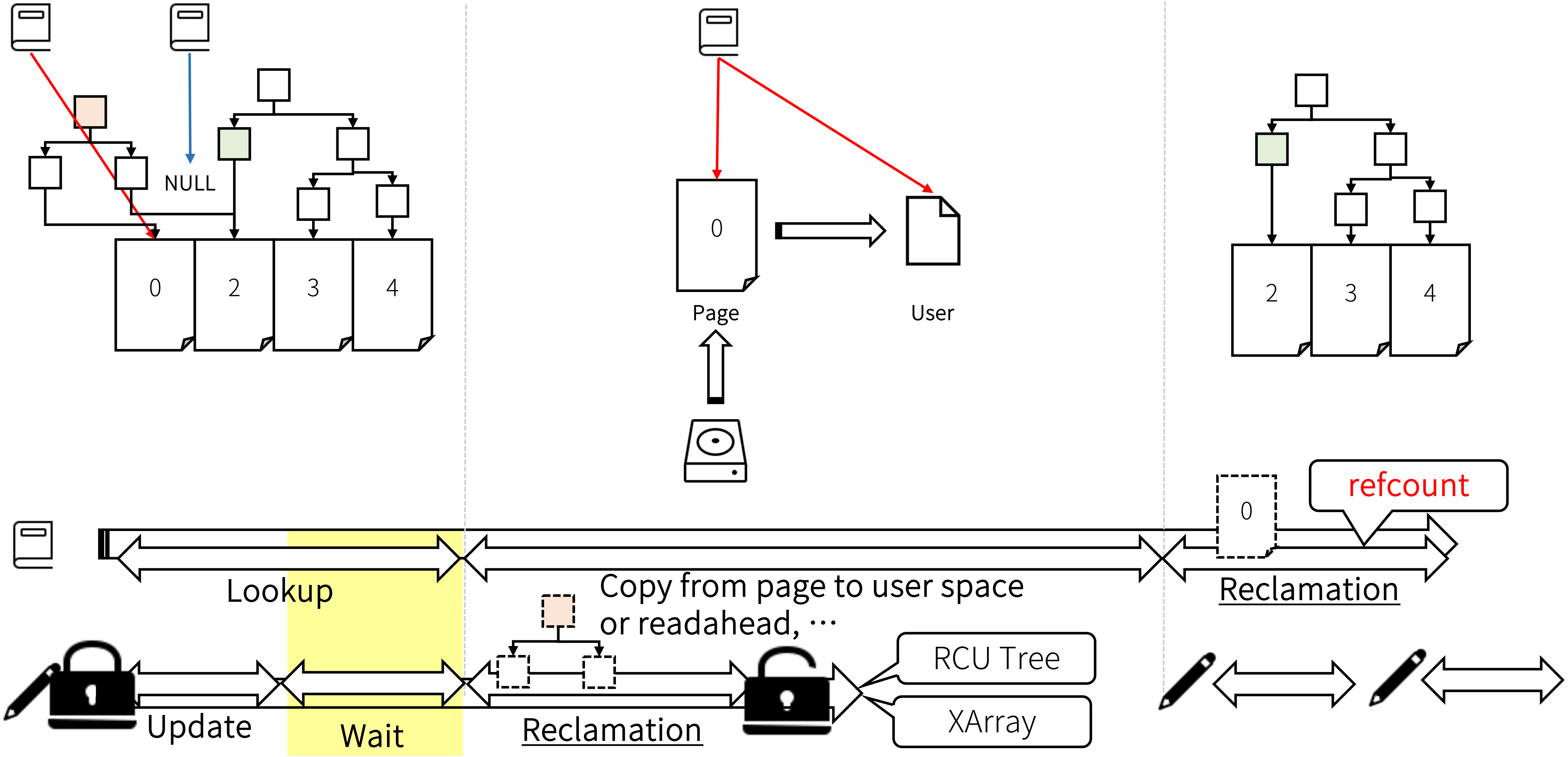
Lookups during removal



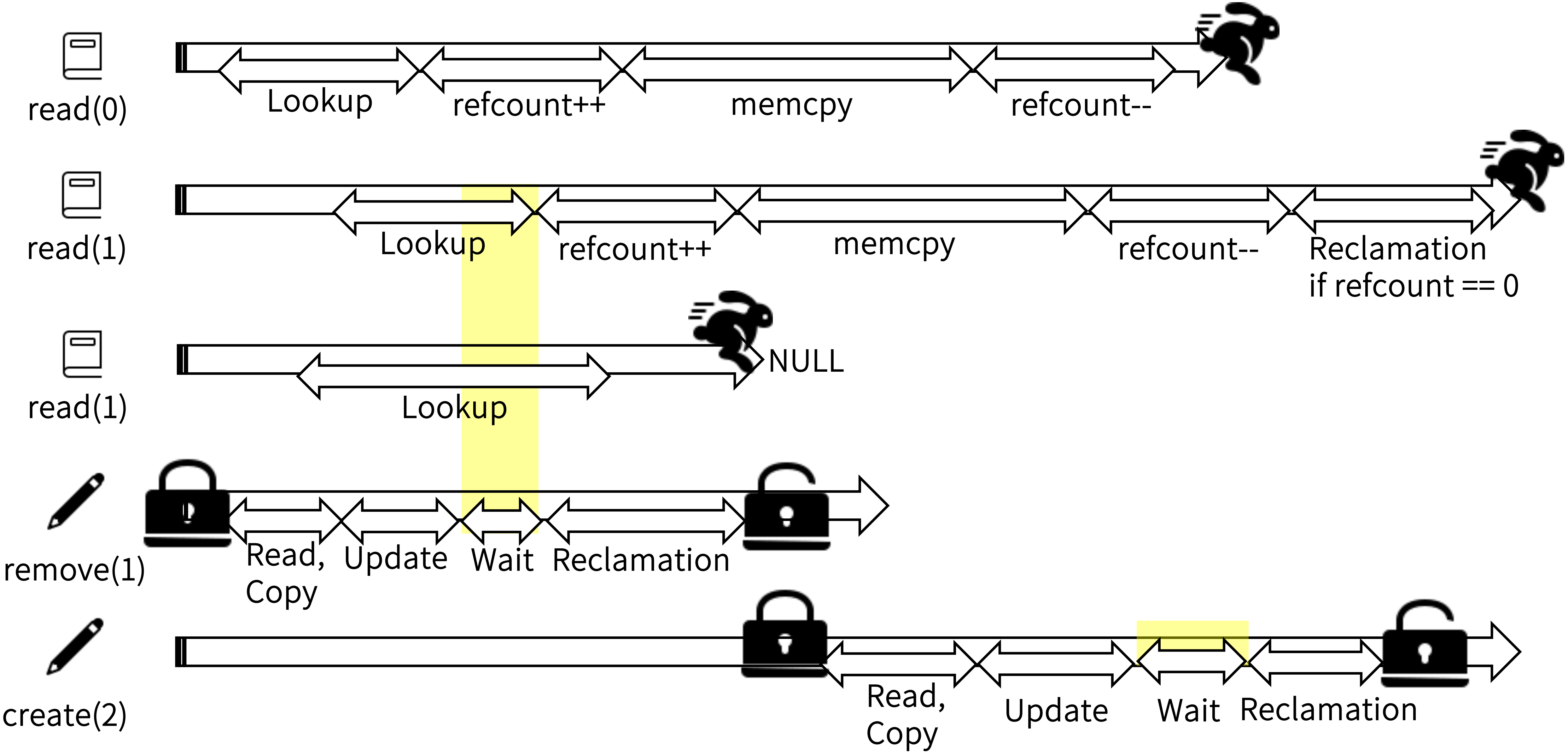
Grace period can be extended



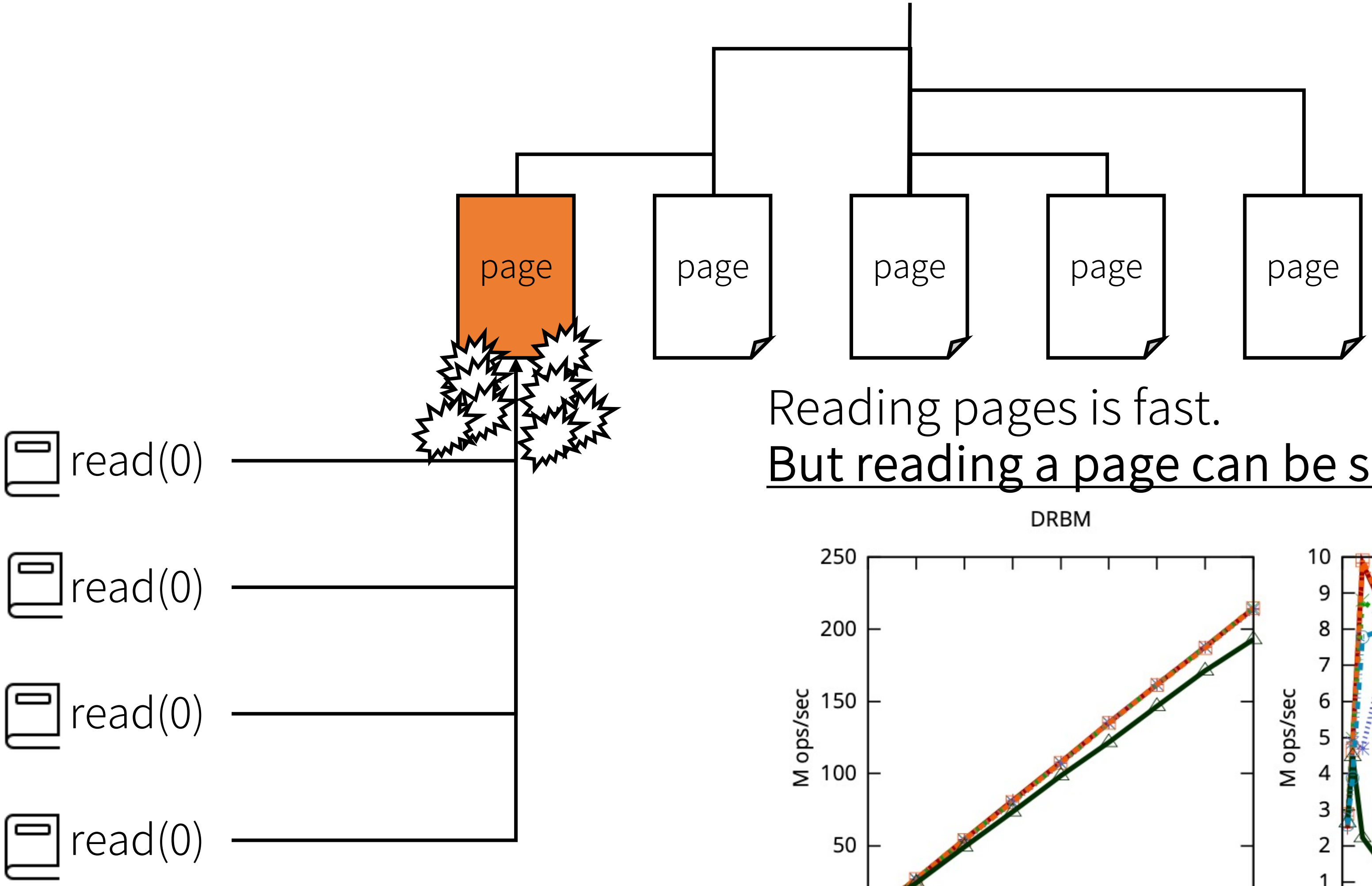
Separated reclamation



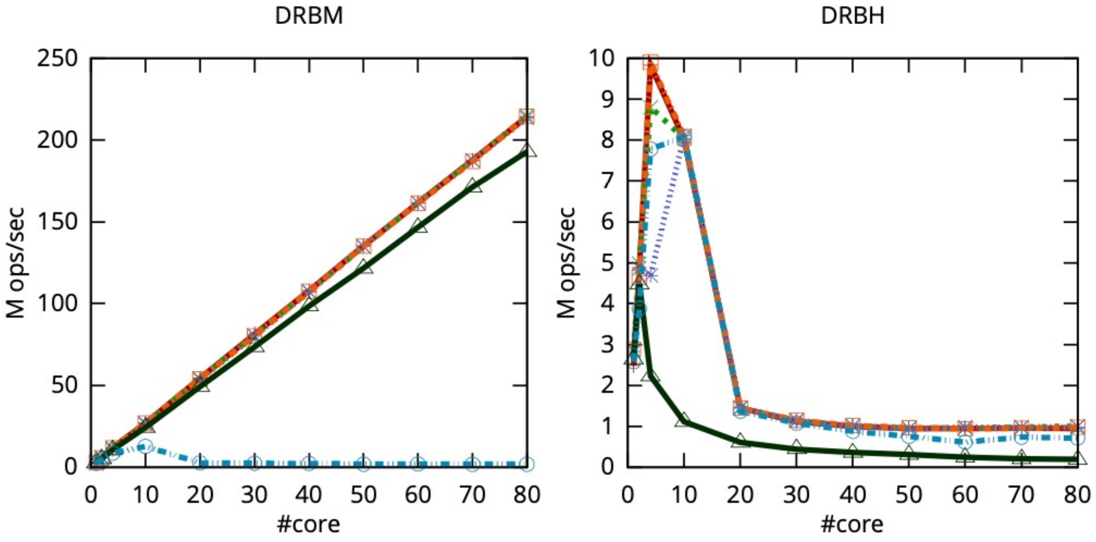
Lockless page-cache with refcount



Bottleneck point



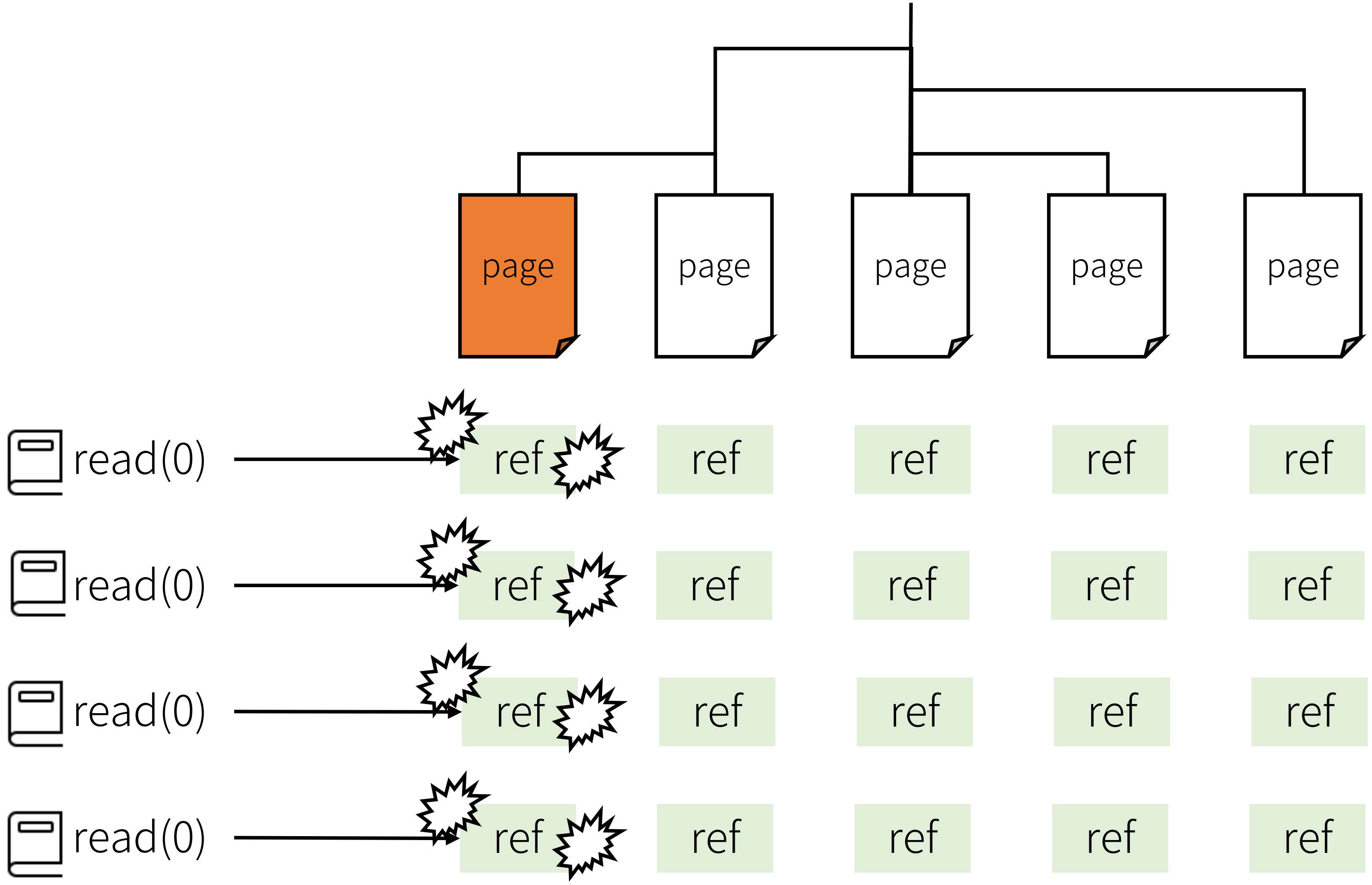
Reading pages is fast.
<u>But reading a page can be slow.</u>



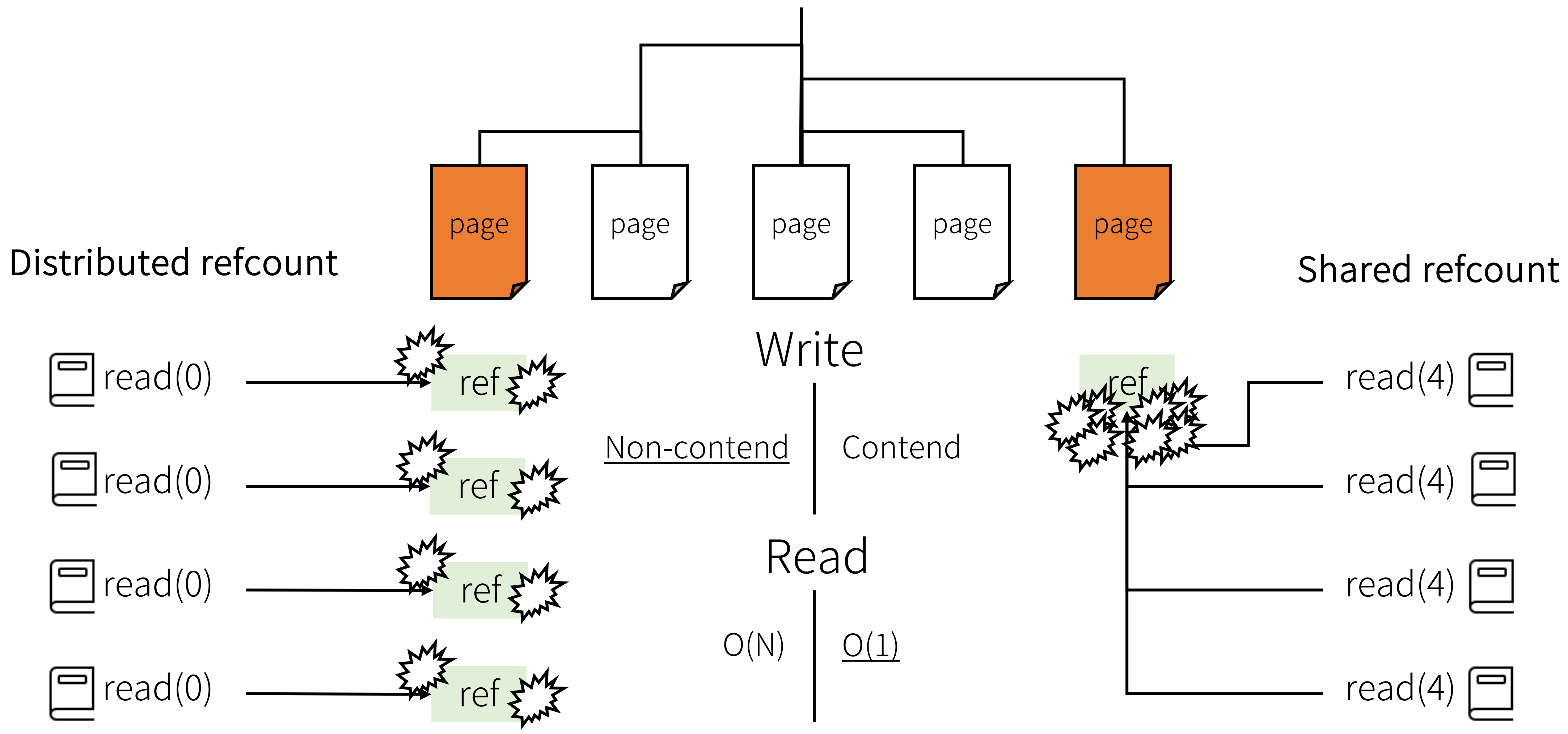
How to resolve the bottleneck

Distributed reference count

Distributed reference count



Read/write refcount trade-off



Related works

An Analysis of Linux Scalability to Many Cores

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev,
M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores

Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, Hyungsoo Jung*
Hanyang University
{syjung, jongbinkim, msryu, sykang, hyungsoo.jung}@hanyang.ac.kr

The search for fast, scalable counters

[Posted February 1, 2006 by corbet]

LODIC: Logical Distributed Counting for Scalable File Access

Jeoungahn Park* Taeho Hwang† Jongmoo Choi‡
Changwoo Min § Youjip Won *
**KAIST, Korea †Hanyang University, Korea ‡Dankook University, Korea §Virginia Tech, USA*

RadixVM: Scalable address spaces for multithreaded applications (revised 2014-08-05)

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

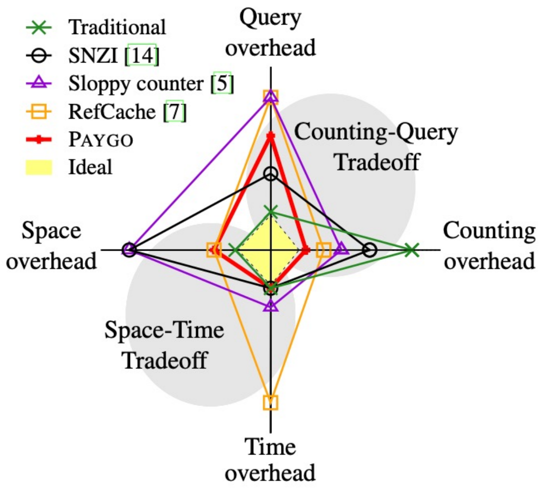
Paygo

Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores

Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, Hyungsoo Jung*

Hanyang University

{*syjung, jongbinkim, msryu, sykang, hyungsoo.jung*}@*hanyang.ac.kr*



	Traditional*	SNZI	Sloppy counter	RefCache	PAYGO
Counting overhead	atomic ops.	atomic ops.‡	global lock	–	–
Space overhead†	$O(N)$	$O(M \cdot N)$	$O(M \cdot N)$	$O(M \cdot C + N)$	$O(M \cdot C + N)$
Query overhead§	$O(1)$	$O(1)$	$O(M)$	$O(1) + 2 \cdot epoch$	$O(M)^{\S\S}$
Time overhead	–	–	every threshold	every epoch and collision	–

* A single atomic reference counter

† N: # of objects, M: # of local counters per object, C: # of hash entries

‡ SNZI recursively updates the counter of the parent node whenever the counter of the child node changes from 0 to 1 and vice versa.

§ Time to determine if the reference counter of a *single* object is zero or not

§§ PAYGO has practically less query overhead than Sloppy counter (§3.4).

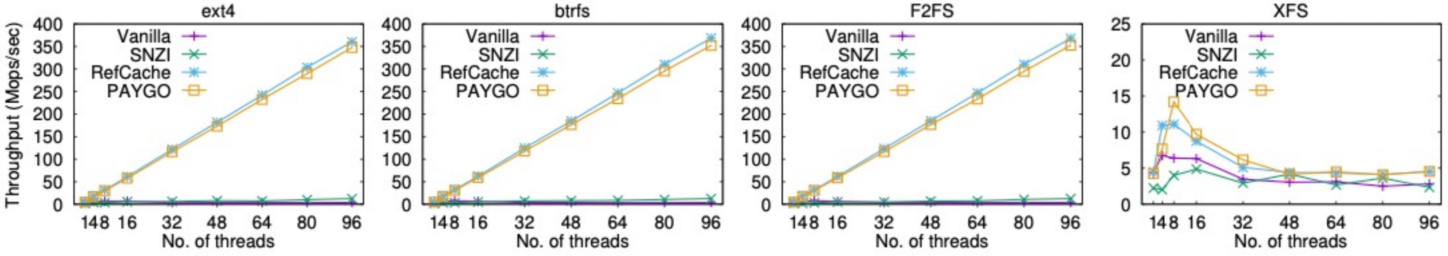


Figure 7: Scalability comparison under strongly contending workloads: the Linux page cache.

Lodic

LODIC: Logical Distributed Counting for Scalable File Access

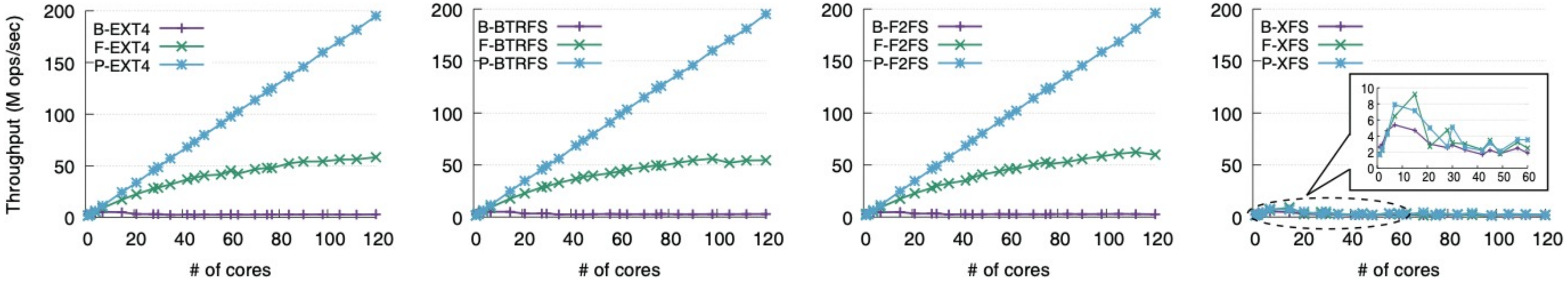
Jeoungahn Park* Taeho Hwang† Jongmoo Choi‡
Changwoo Min § Youjip Won *

*KAIST, Korea †Hanyang University, Korea ‡Dankook University, Korea §Virginia Tech, USA

	Atomic Counter	SNZI [31]	Sloppy Counter [17]	RefCache [23]	PayGo [40]	LODIC
Counting Overhead	Contending atomic ops	Non-contending atomic ops	Global lock	Non-atomic ops	Mostly non-atomic ops	Mostly non-contending atomic ops
Space Overhead	$O(N)$	$O(N \cdot C)$	$O(N \cdot C)$	$O(C \cdot H + N)$	$O(C \cdot H + N)$	$O(N)$
Query Overhead	$O(1)$	$O(1)$	$O(C)$	$O(1) + 2 \cdot epoch$	$O(C)$	$O(S)$
Time Overhead	None	None	Every threshold	Every epoch and collision	Every hash collision	None

N: # of objects C: # of CPUs H: size of hash table S: degree of sharing

Table 1: Comparison of reference counting techniques.



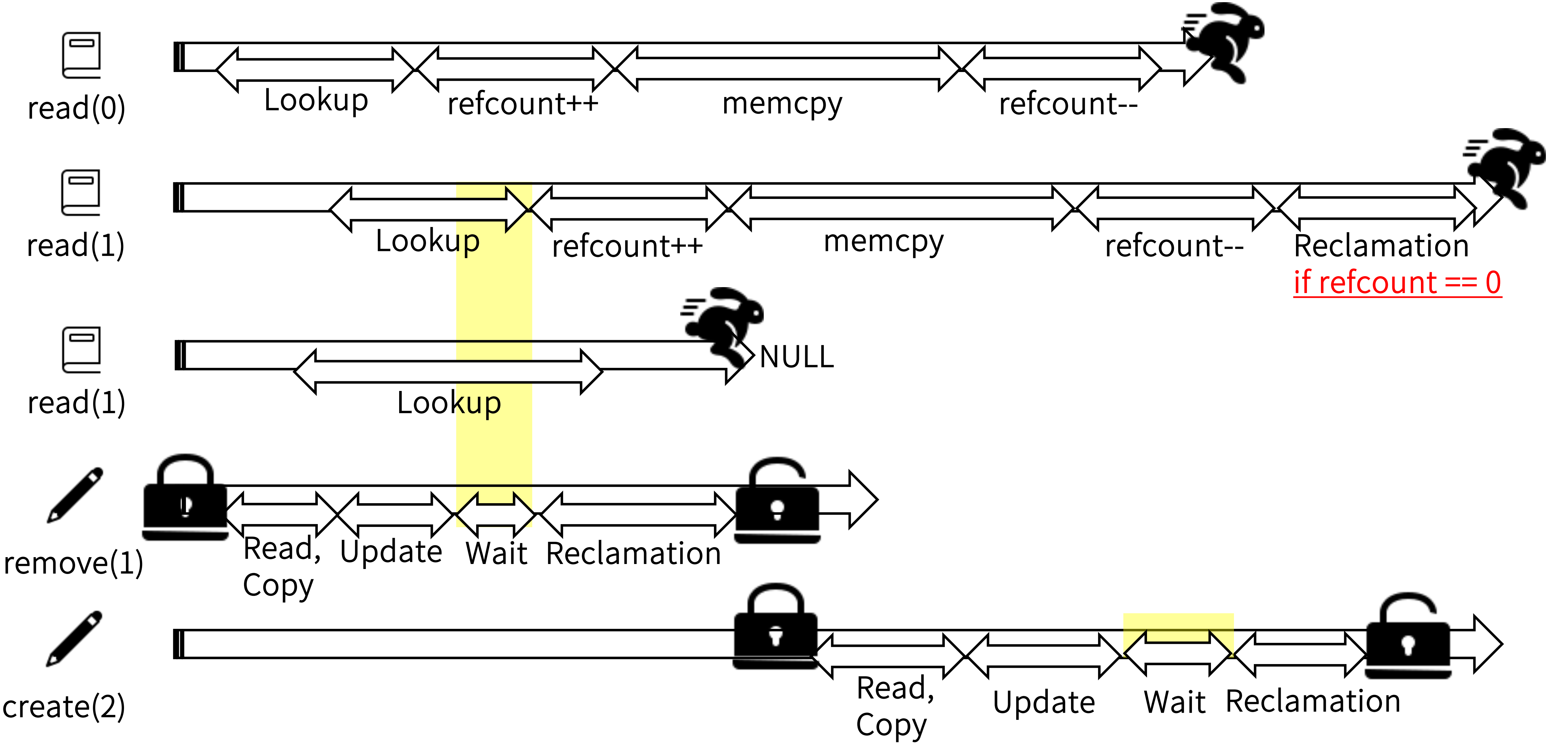
(a) EXT4 **(b)** BTRFS **(c)** F2FS **(d)** XFS

Figure 14: FxMark (DRBH): Baseline ('B'), File-based reverse mapping ('F'), Process-based reverse mapping ('P').

Refcount overheads of page-cache

refcount \ page-cache	Lookup-side rcu_read(page)	Removal-side rcu_write(page)
- Counting overhead write(refcount)	○	✕
- Query overhead read(refcount)	✕	○

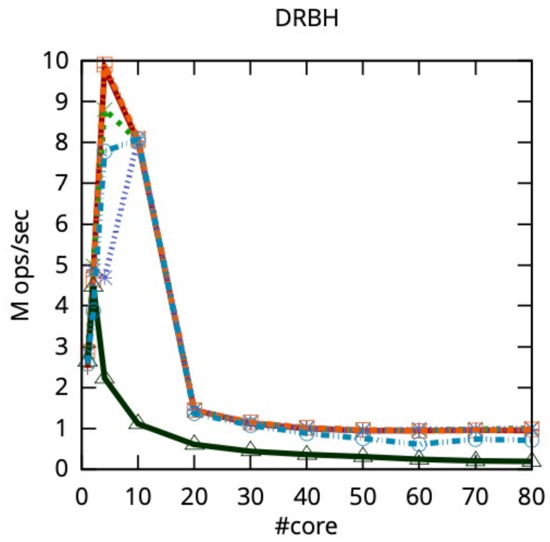
Lookup-side generates query overhead!



Refcount overheads of page-cache

page-cache / refcount	Lookup-side rcu_read(page)	Removal-side rcu_write(page)
- Counting overhead write(refcount)	○	✕
- Query overhead read(refcount)	○	○

Same result for distributed refcount?



Lookup-side: counting overhead + query overhead

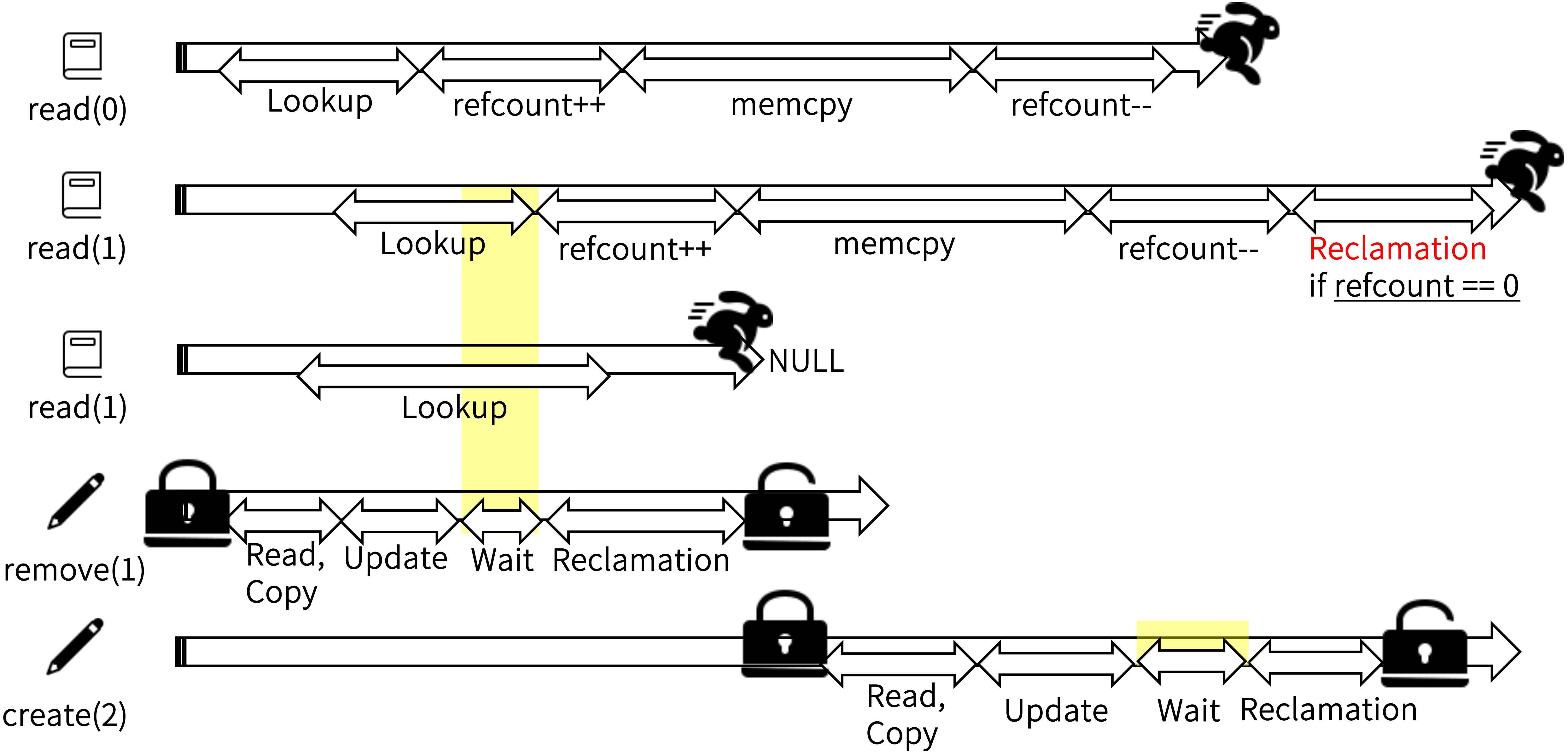
Non-contend

O(N) + cache-coherence

new PIC of reclamation

Distribution manager

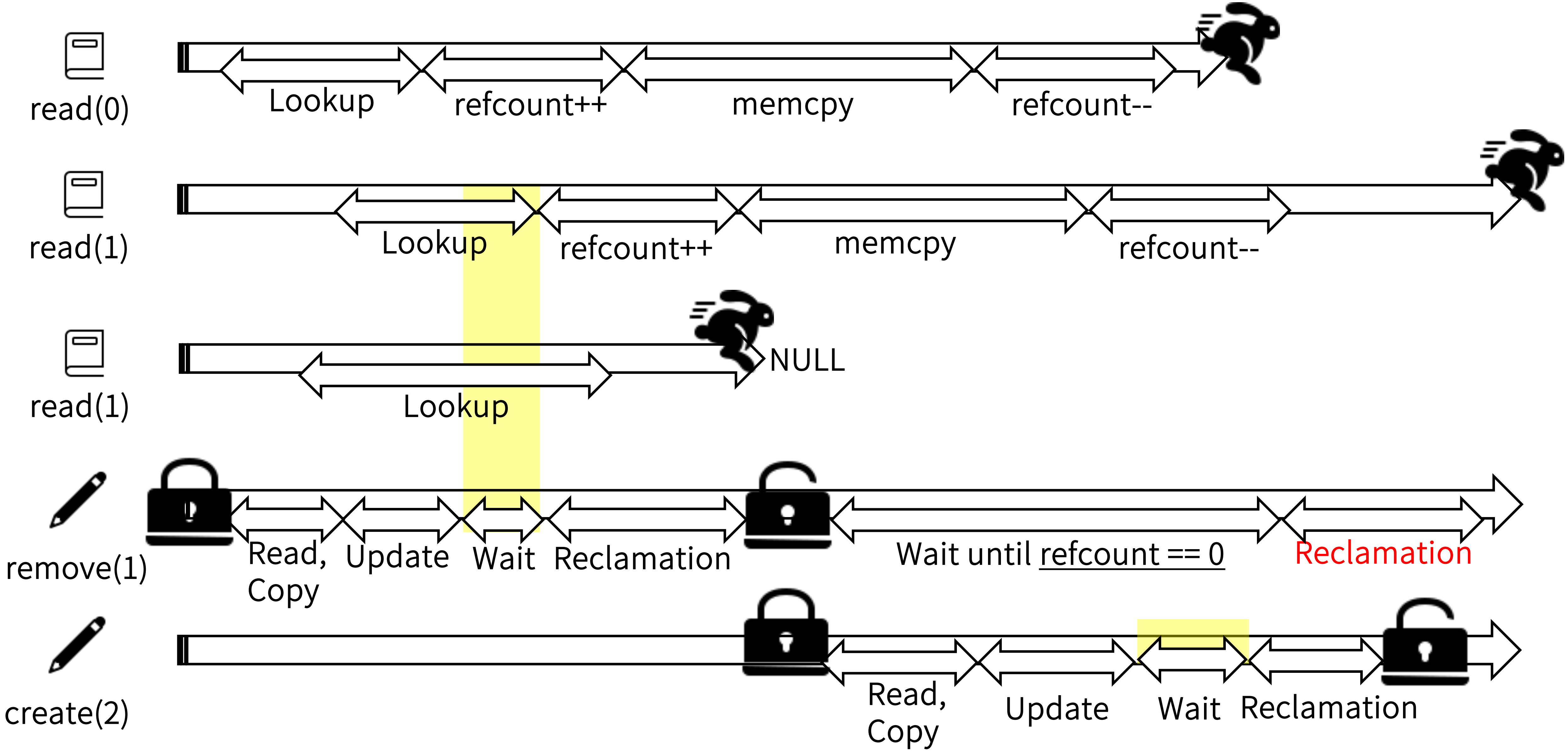
PIC 1: lookup-side



PIC 1: refcount overheads

refcount \ page-cache	Lookup-side rcu_read(page)	Remove-side rcu_write(page)
- Counting overhead write(refcount)	○	✕
- Query overhead read(refcount)	○	○

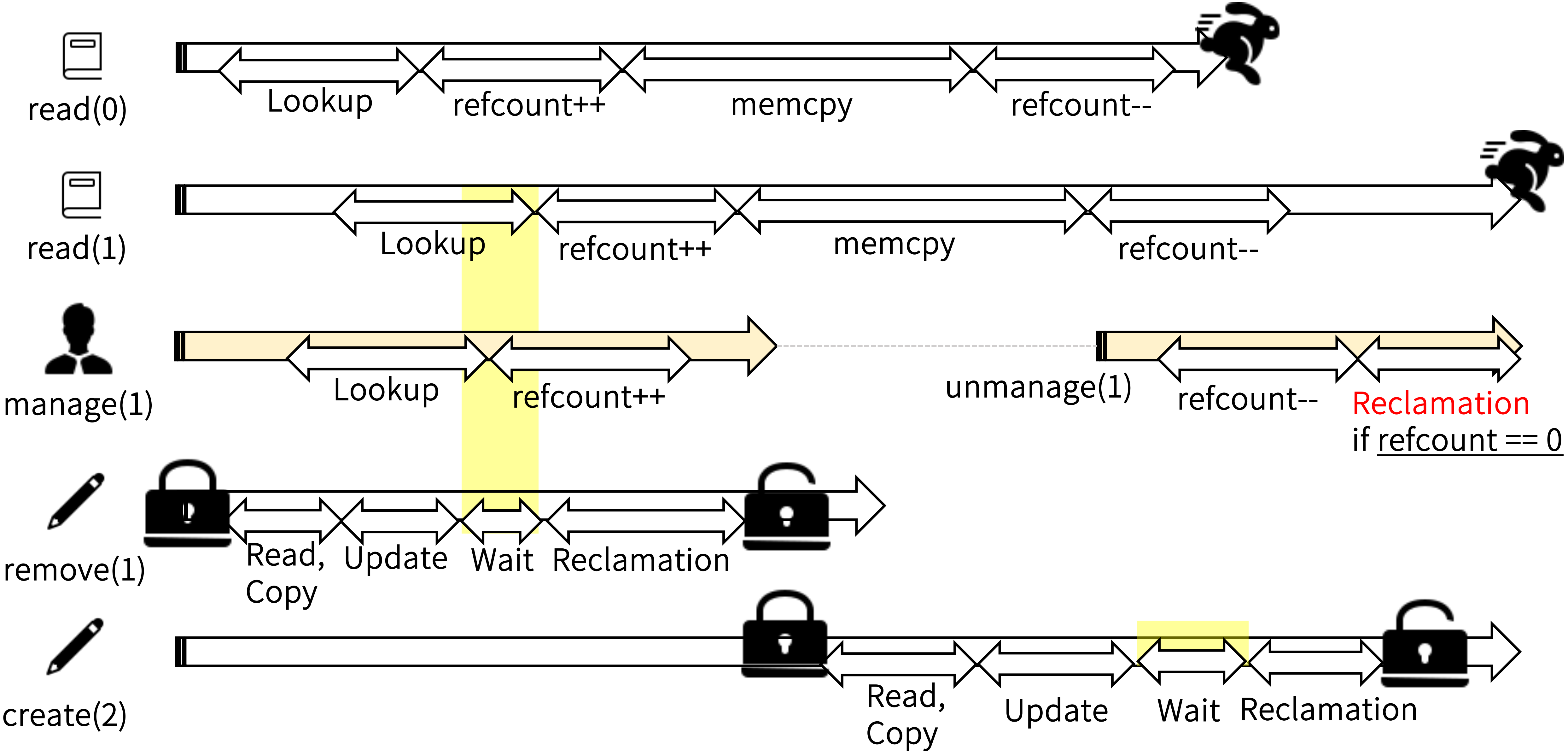
PIC 2: removal-side



PIC 2: refcount overheads

refcount \ page-cache	Lookup-side rcu_read(page)	Remove-side rcu_write(page)
- Counting overhead write(refcount)	O Retry	✗
- Query overhead read(refcount)	✗	O Blocking

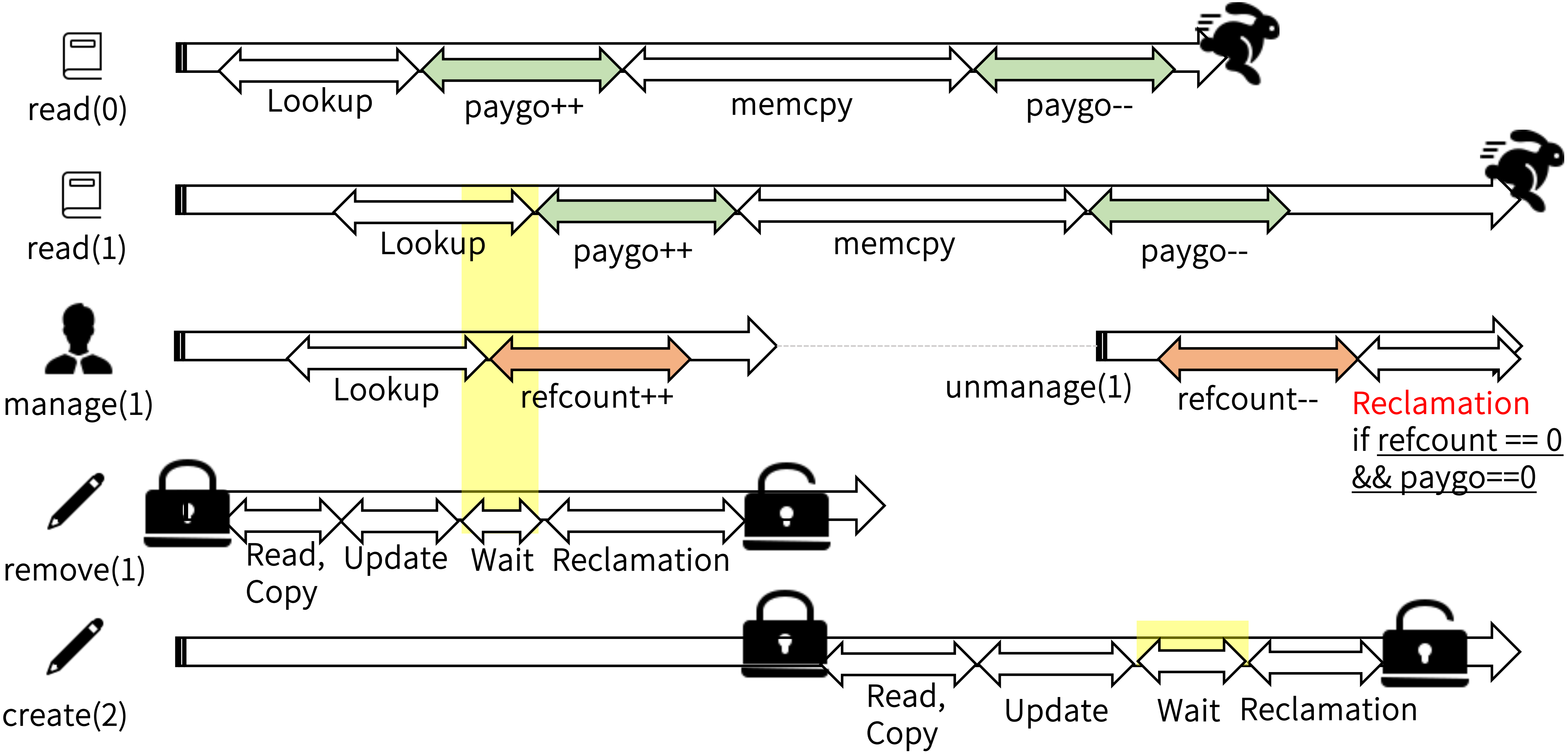
PIC 3: new manager



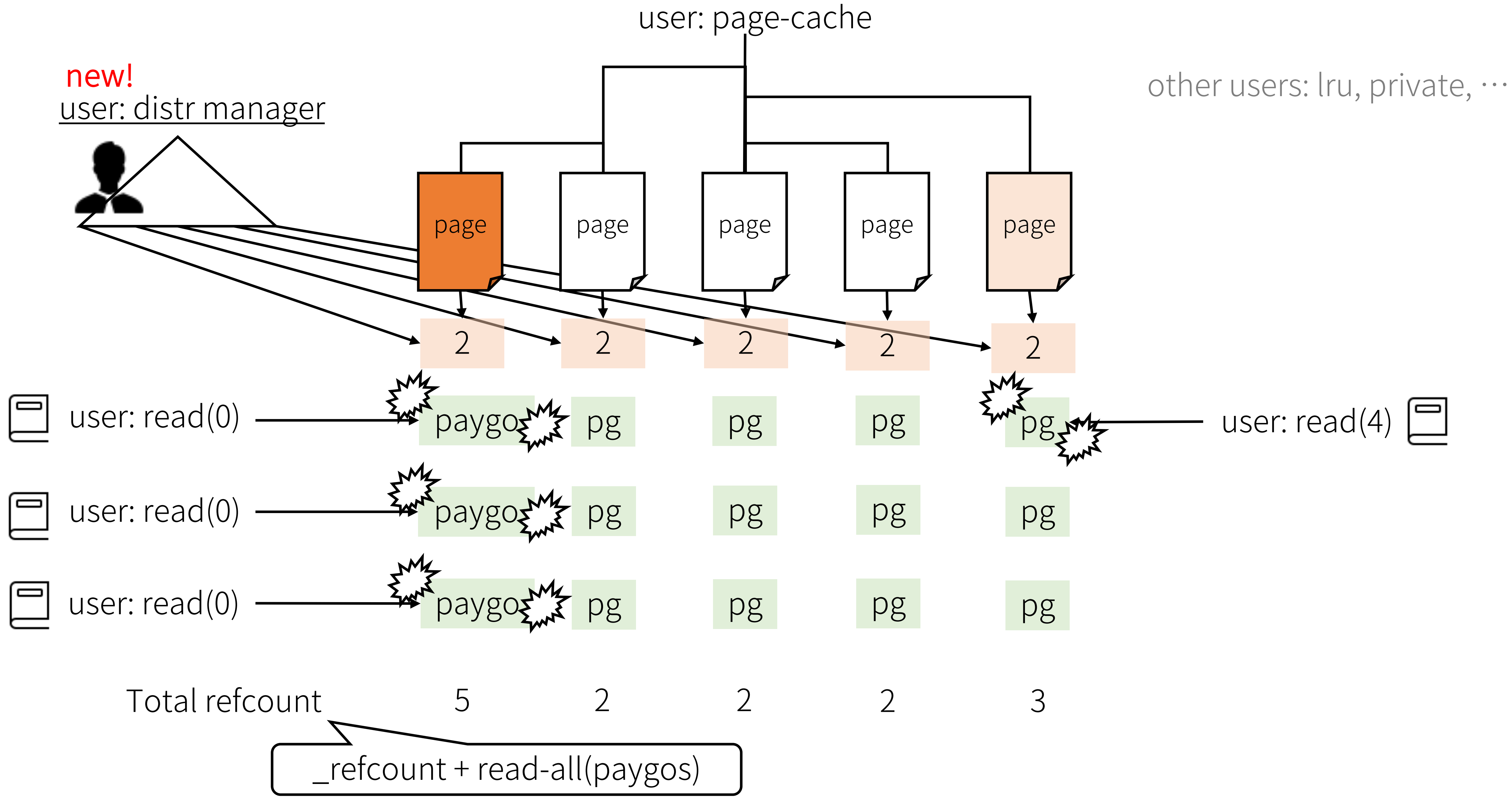
PIC 3: refcount overheads

page-cache refcount	Lookup-side rcu_read(page)	Removal-side rcu_write(page)
- Counting overhead write(refcount)	O	X
- Query overhead read(refcount)	X	O

Distribution manager uses 2 types of refcount!



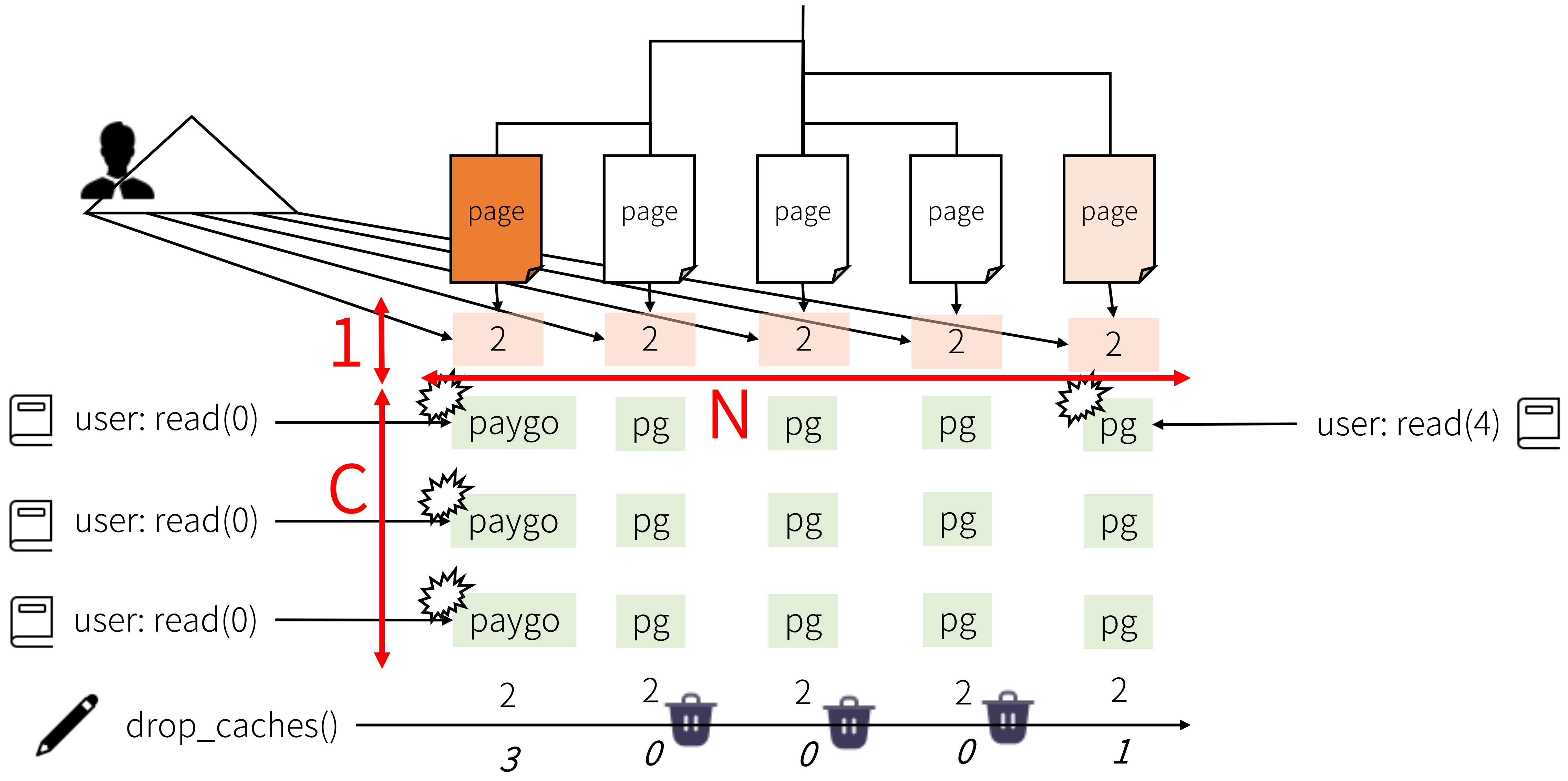
Distribution is possible!



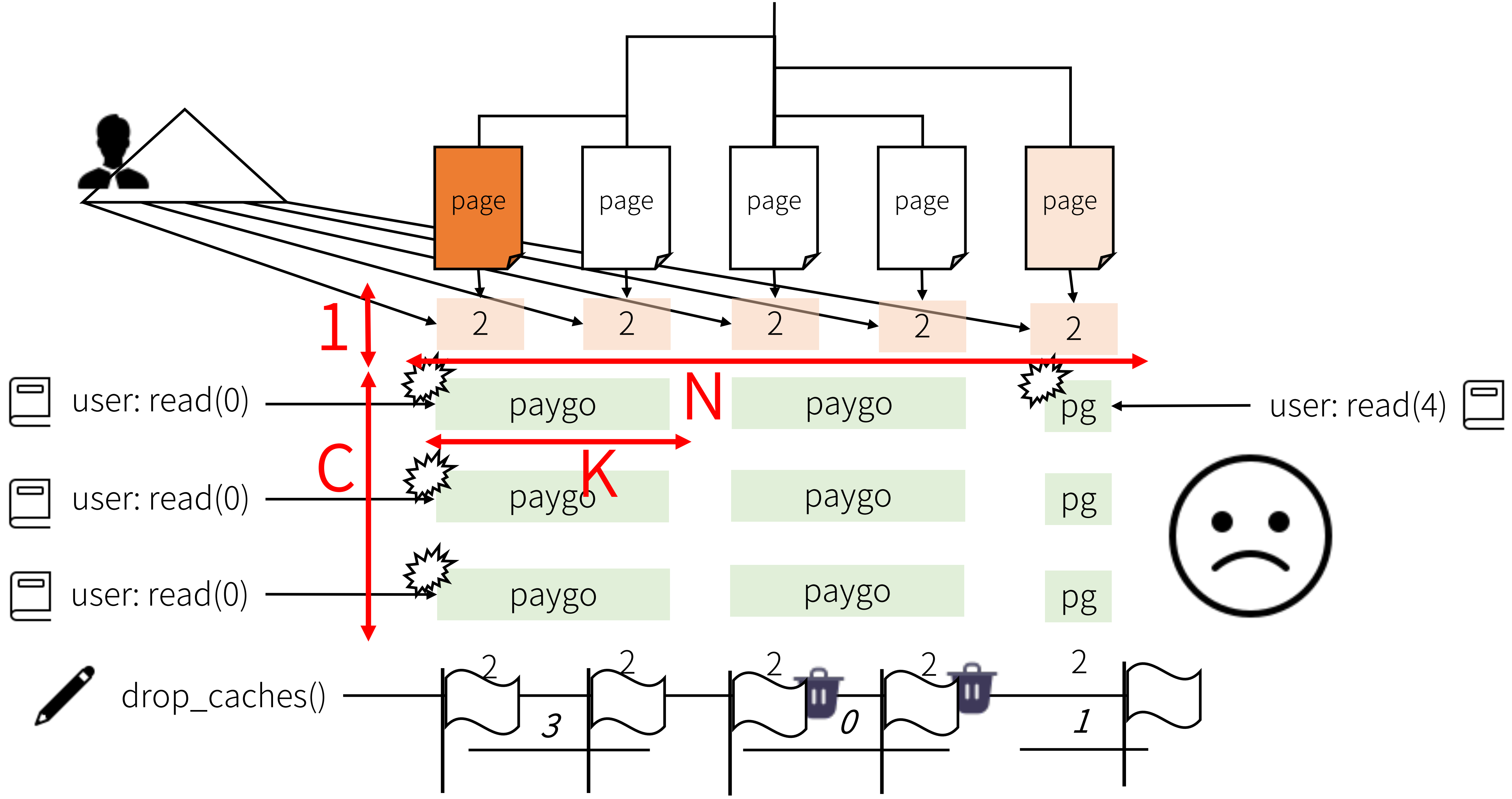
Improved query overhead

Speed-reading for paygo

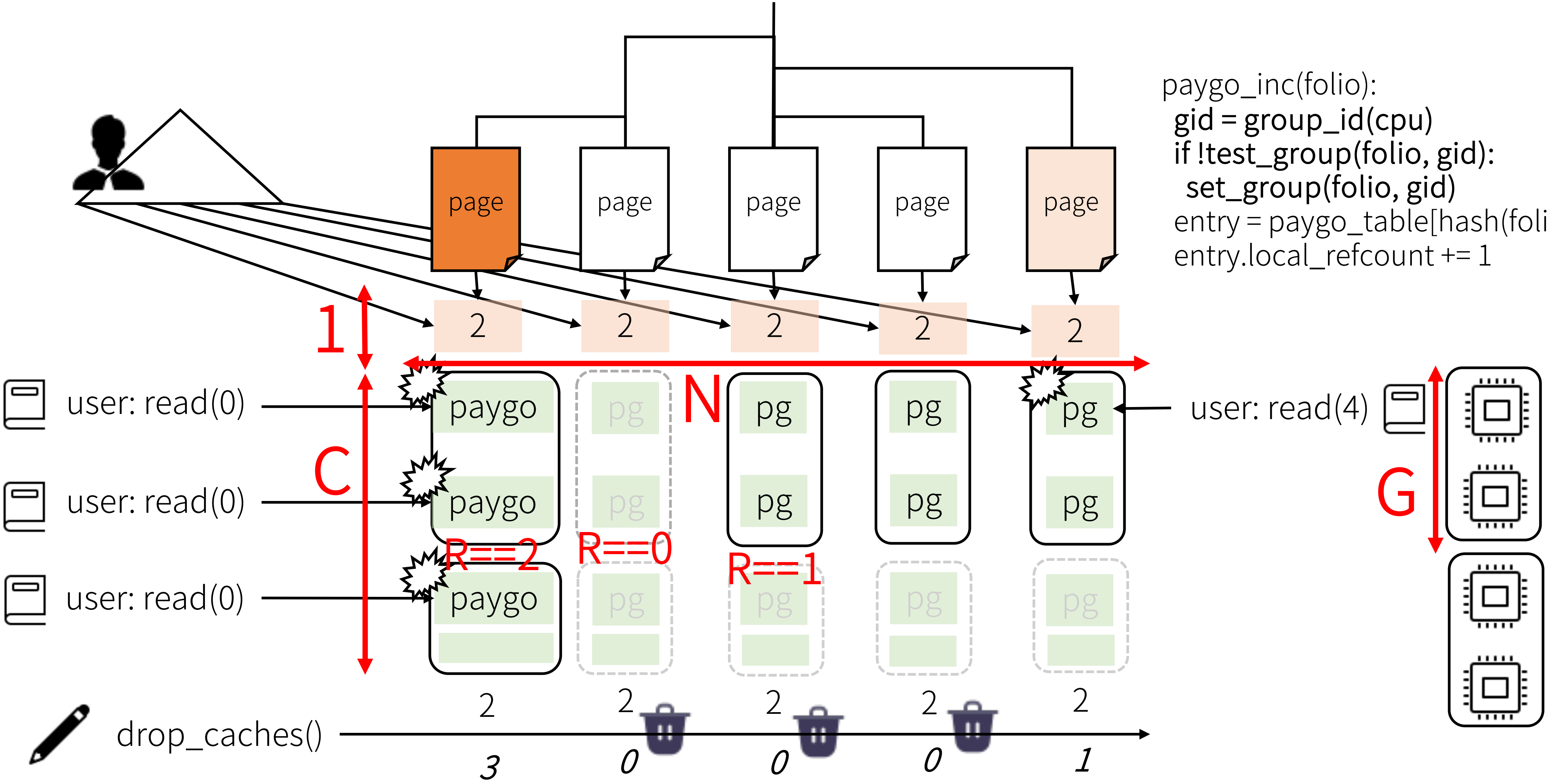
Original paygo: O(N * (C+1))



Compounded refcount: O(N * (C+1) / K)

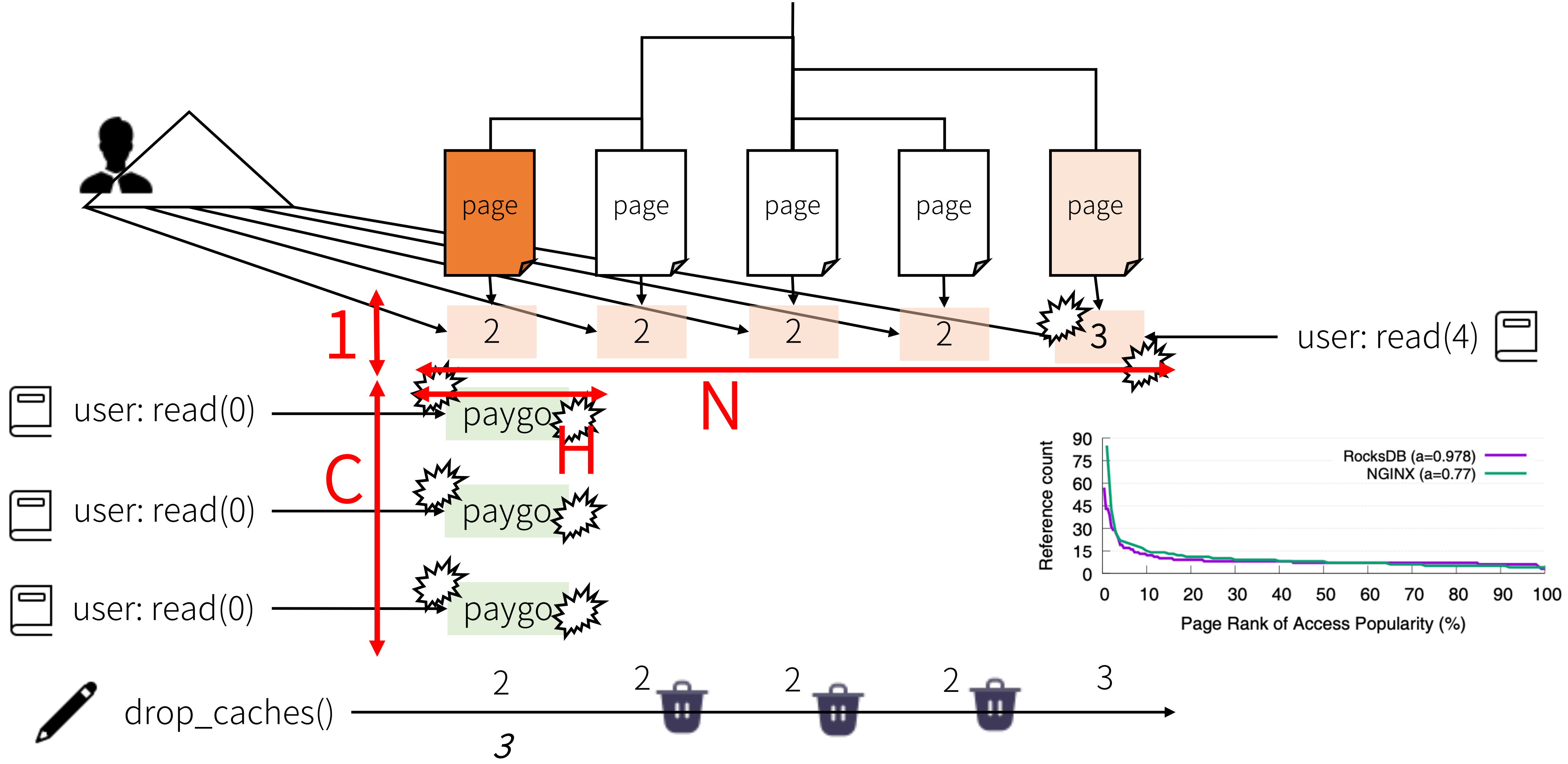


Grouping: $O(N * (C/G*\mu_R+1))$

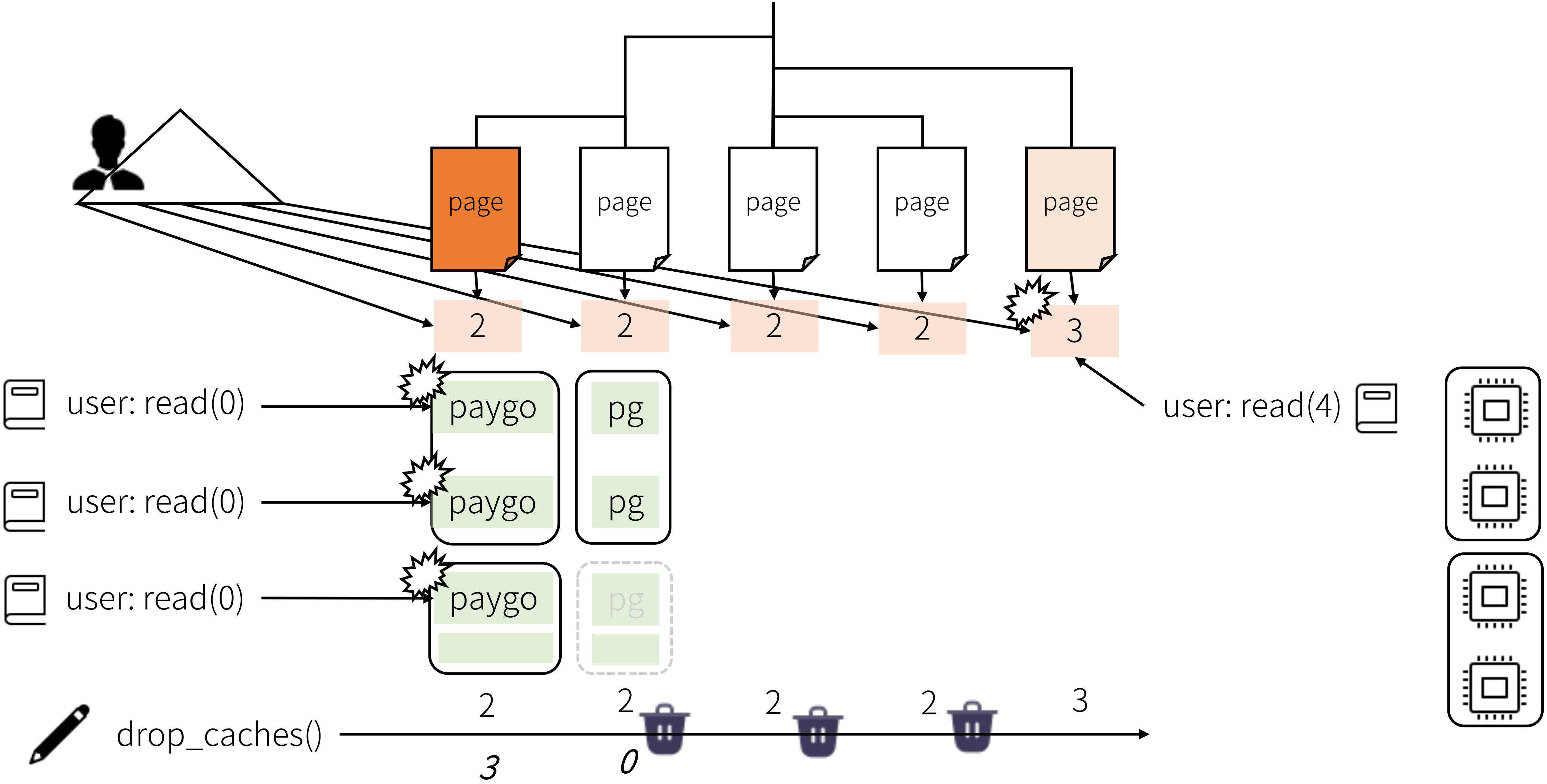



```
paygo_inc(folio):
  gid = group_id(cpu)
  if !test_group(folio, gid):
    set_group(folio, gid)
  entry = paygo_table[hash(folio)]
  entry.local_refcount += 1
```

Hot section: O(N + H * C)



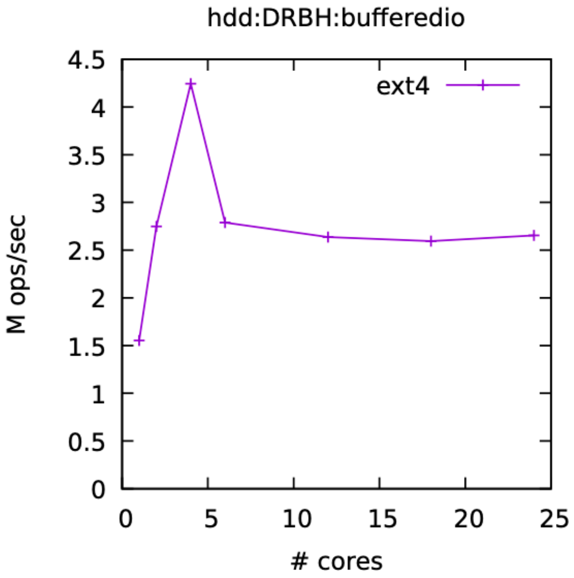
Combine grouping and hot section



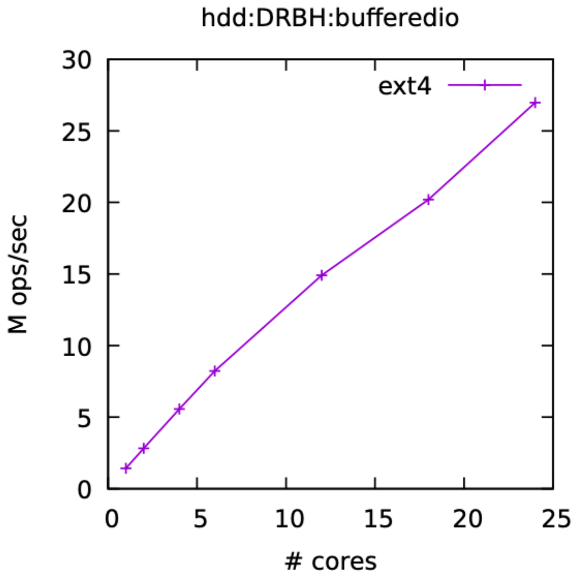
Evaluation

Counting overhead

- 24 cores (12 cores/CPU, 2 sockets Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz), 48GB DRAM
- Linux v6.2 on QEMU v6.2.0
- DRBH Workload on FxMark



[vanilla]

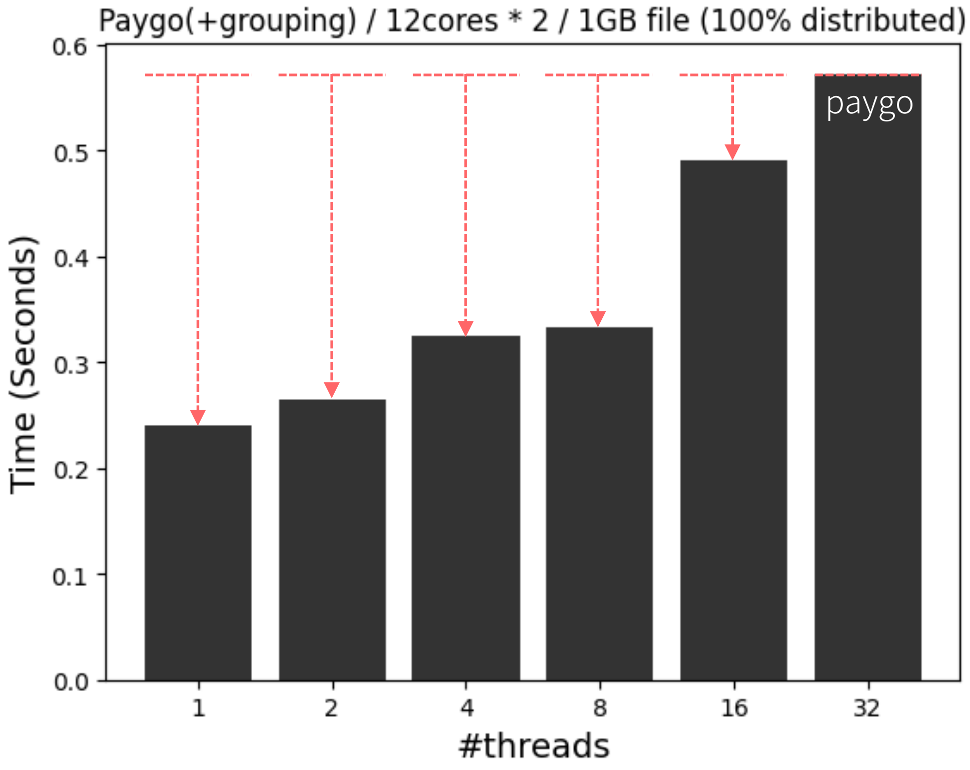


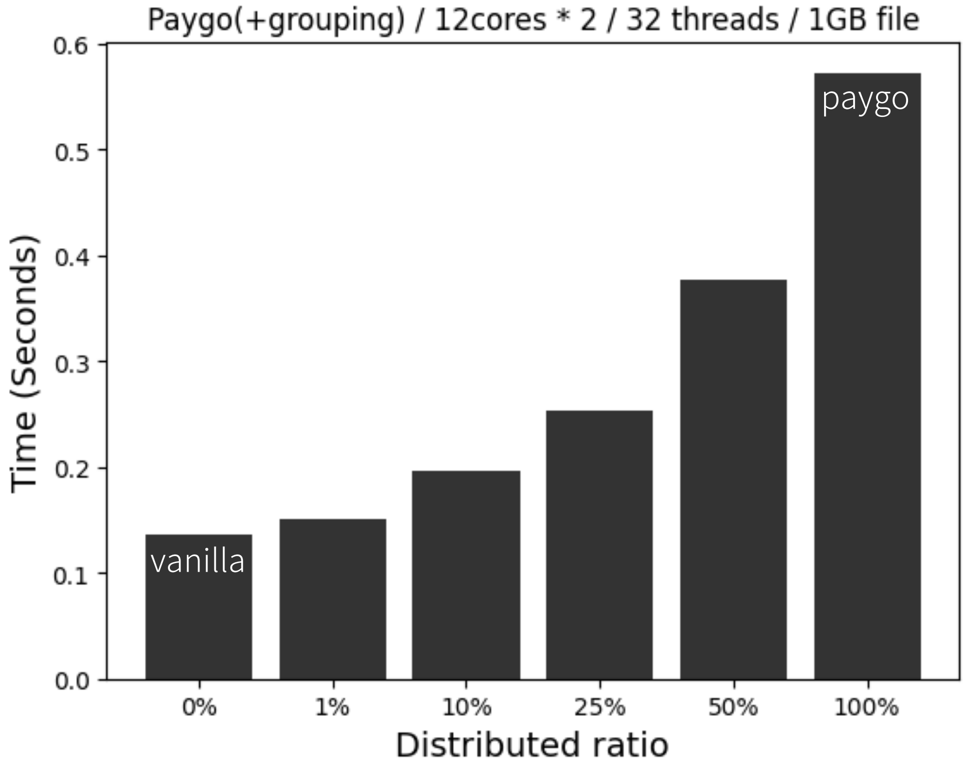
[paygo(+grouping)]

https://github.com/HM4725/refcount/tree/master/krefcount/ucounting

Query overhead

- 24 cores (12 cores/CPU, 2 sockets Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz), 48GB DRAM
- Linux v6.2 on QEMU v6.2.0
- $ time echo 1 > /proc/sys/vm/drop_caches





https://github.com/HM4725/refcount/tree/master/krefcount/uquery

Limitations

Grouping

- Without clearing groups, query overhead is bound to increase.
- Even though there isn't any contention.

Hot section

- A user should specify the area manually.

The free lunch **is over.**