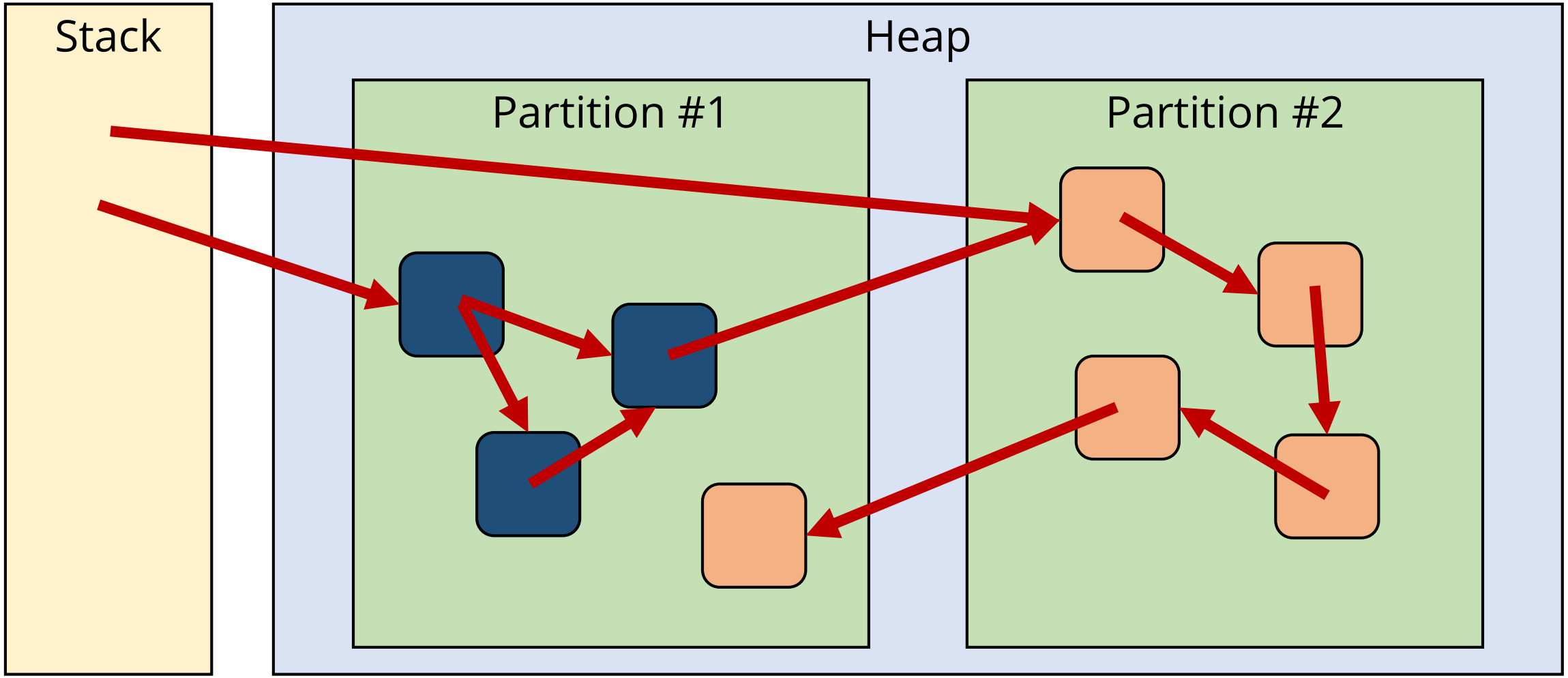
Generational GC

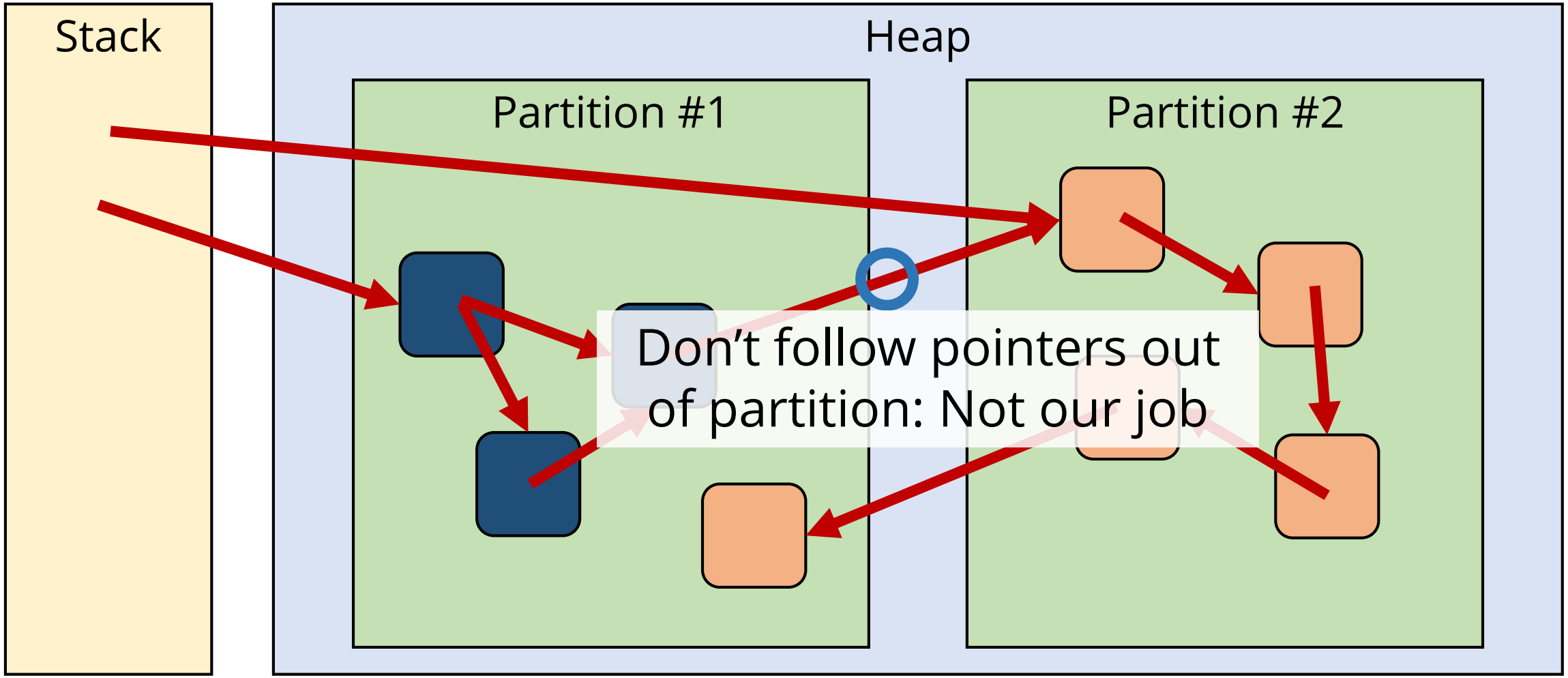
Review

- Partitioning: Divide heap, use different strategies per heap
- Generational GC: Partition by age
- Most objects die young

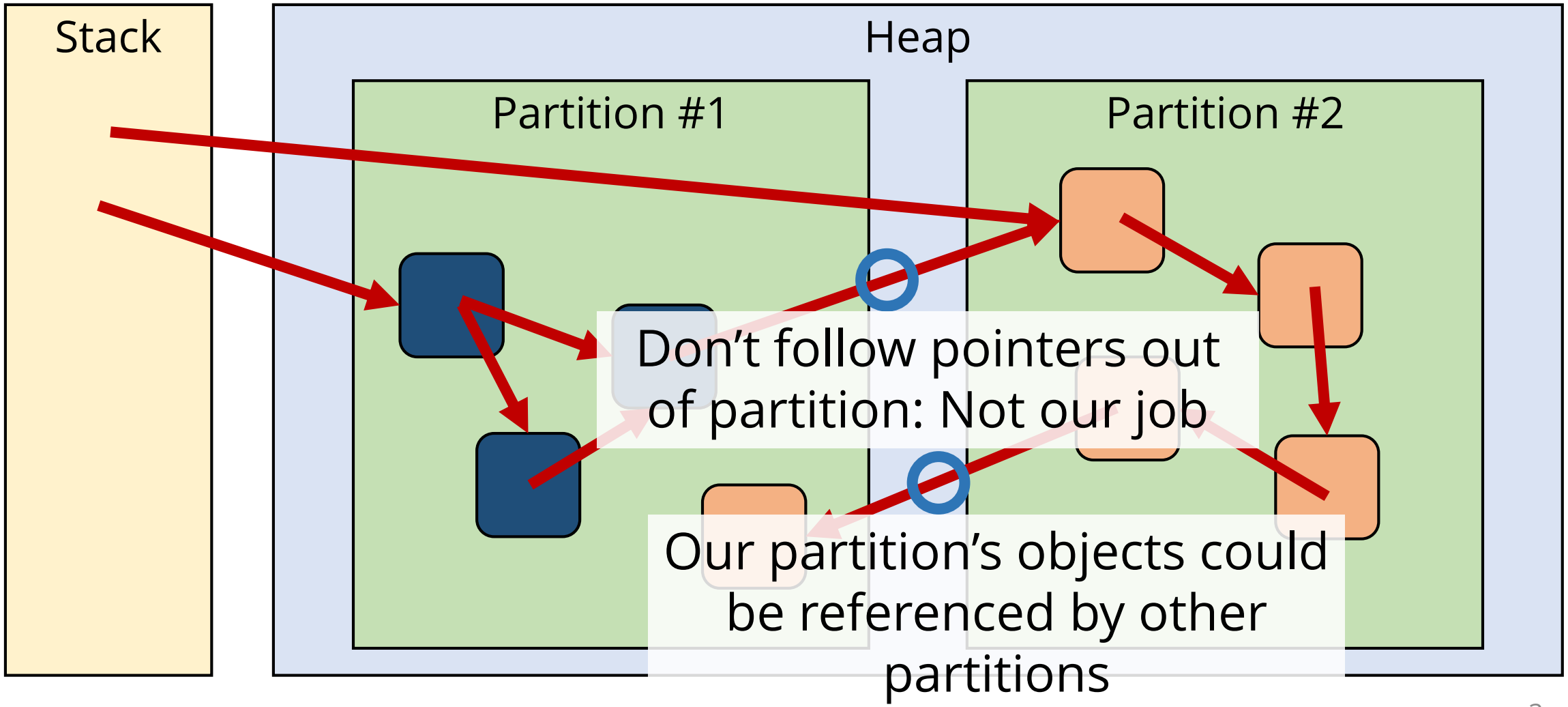
Single-partition scanning



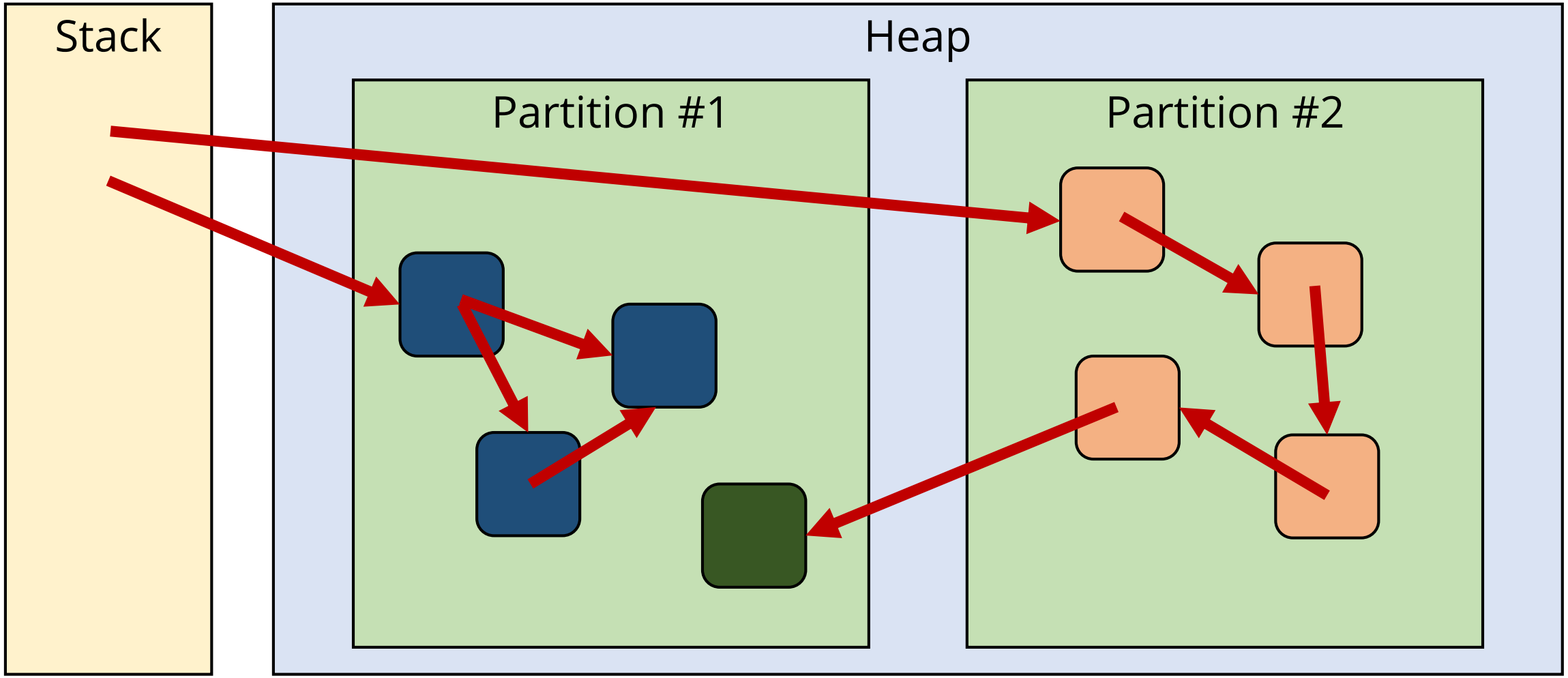
Single-partition scanning



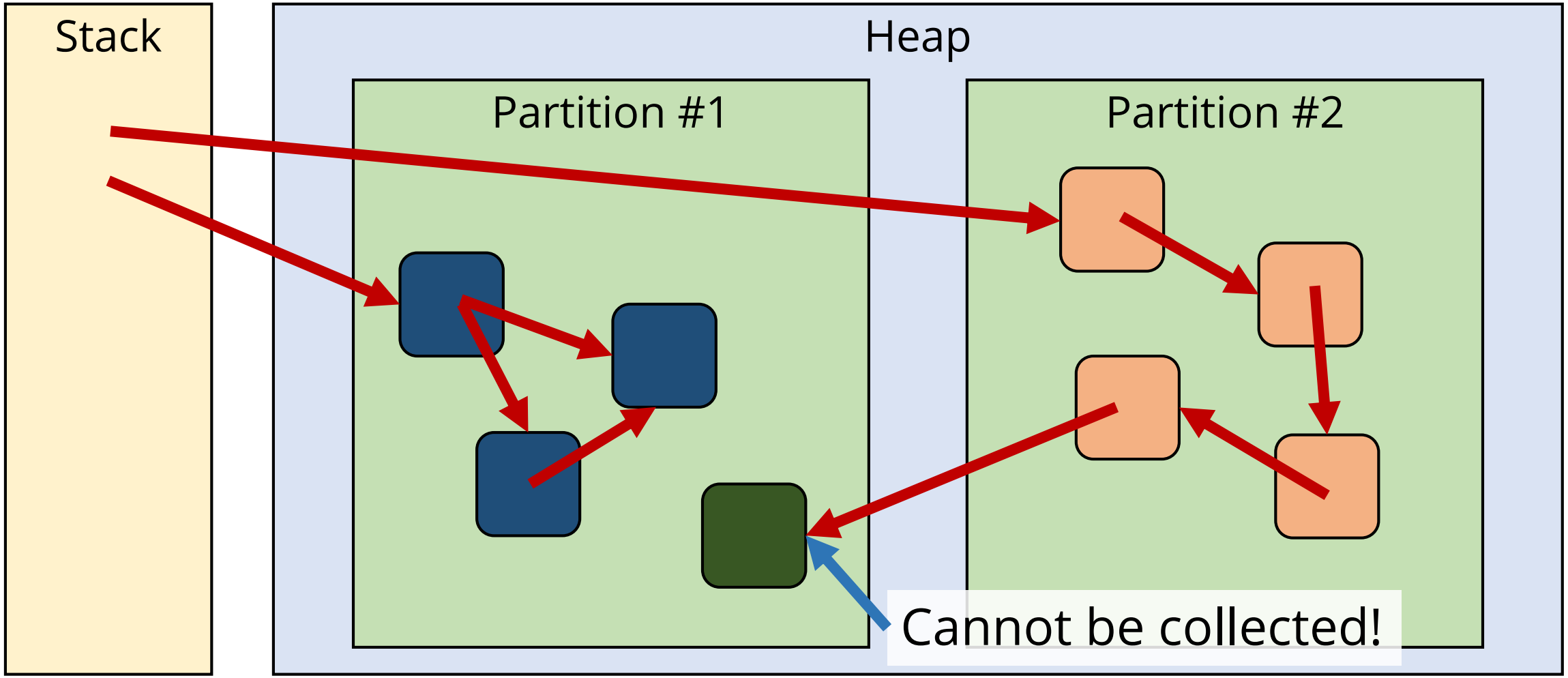
Single-partition scanning



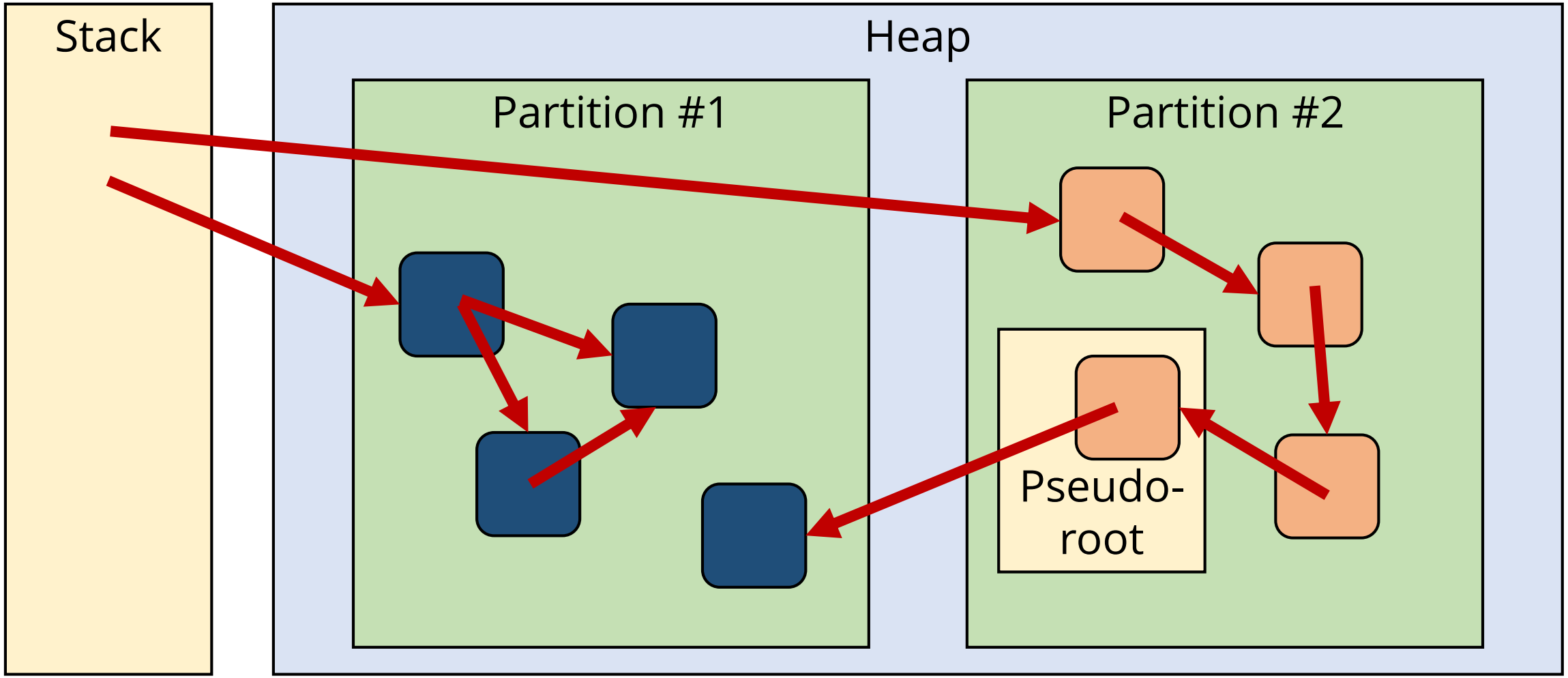
Single-partition scanning



Single-partition scanning



Alternate approach



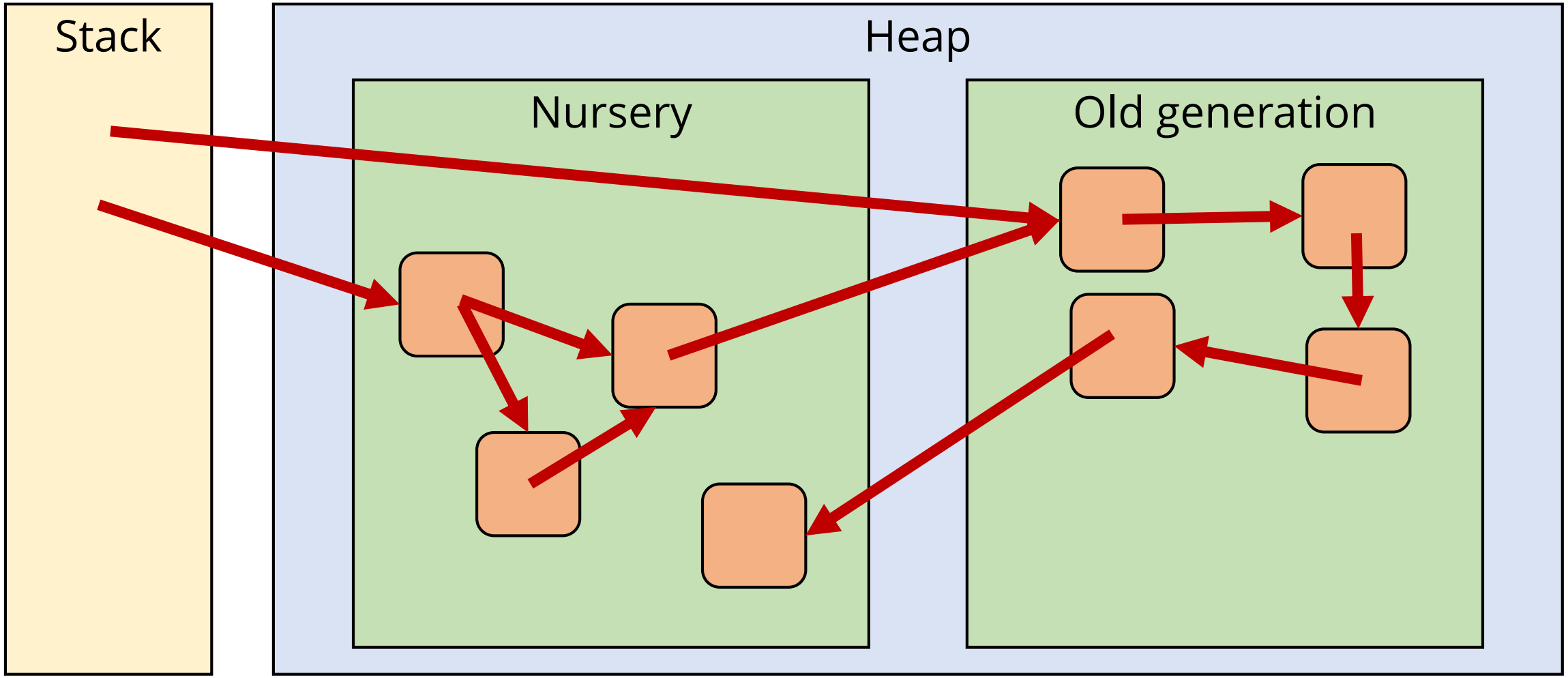
The big idea

- Most objects die young
- Partition heap by age
- “Nursery” partition collected frequently
- “Old” partition only collected during full-heap GC

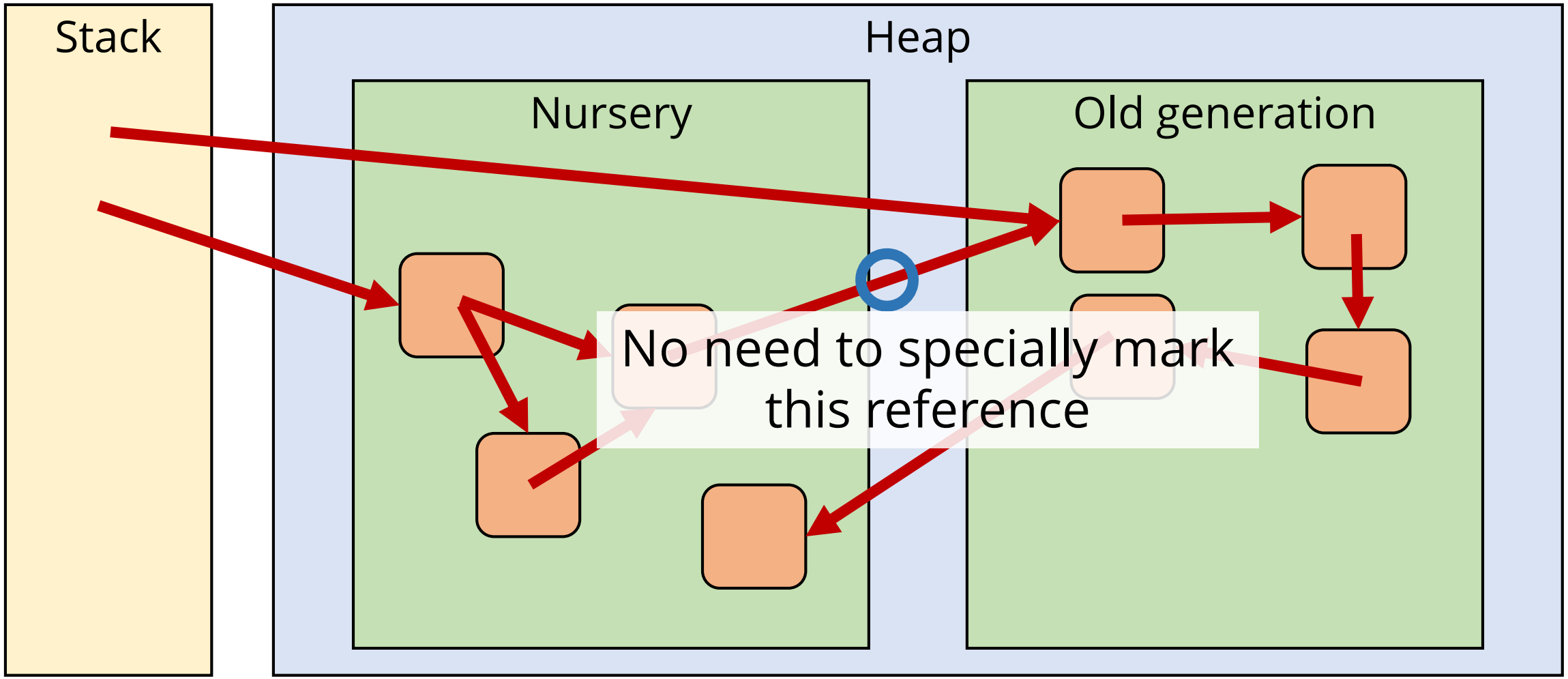
Inter-partition references

- Never collect old partition without young
- Only need to remember old→young inter-partition references
- Write barrier is back! (Details later)
- Treat old objects w/ young refs as roots

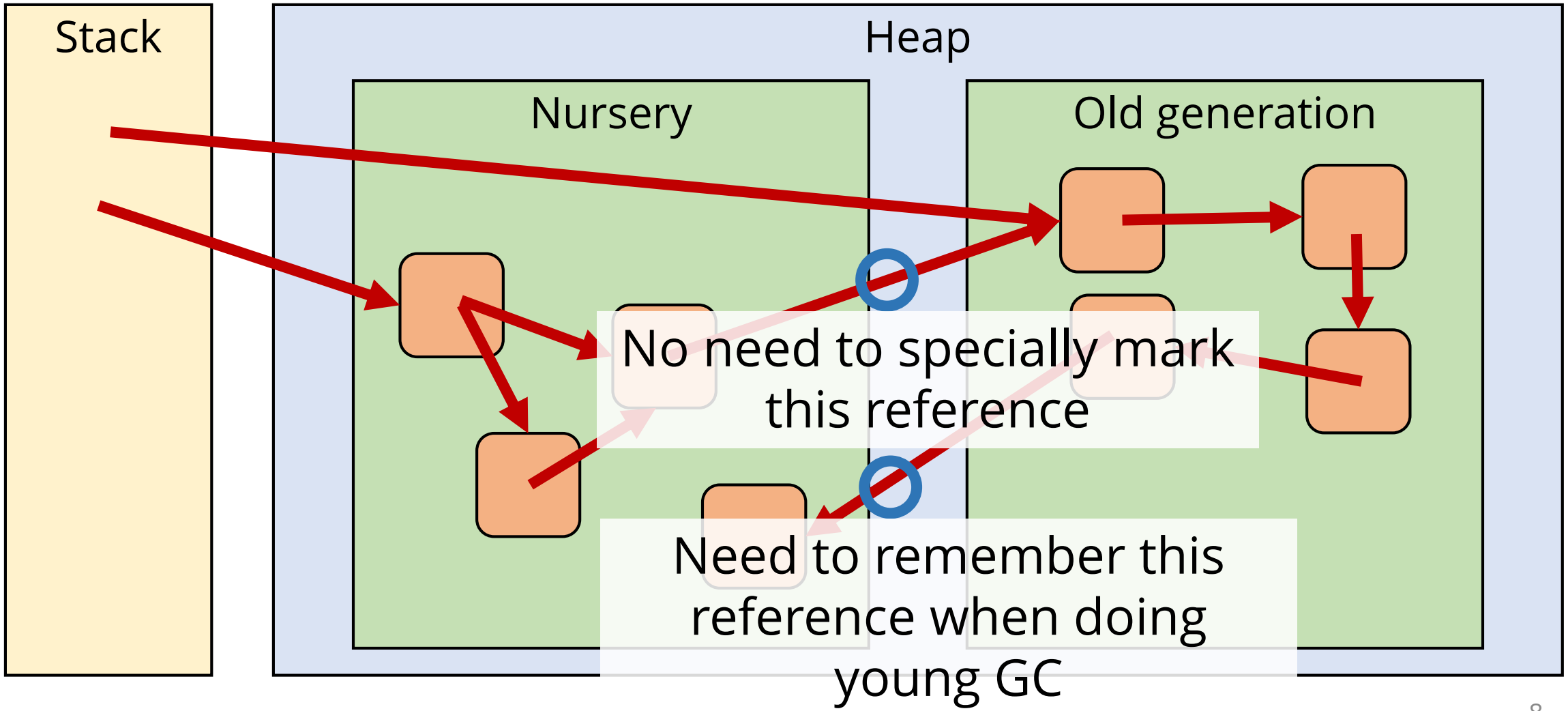
Generational GC



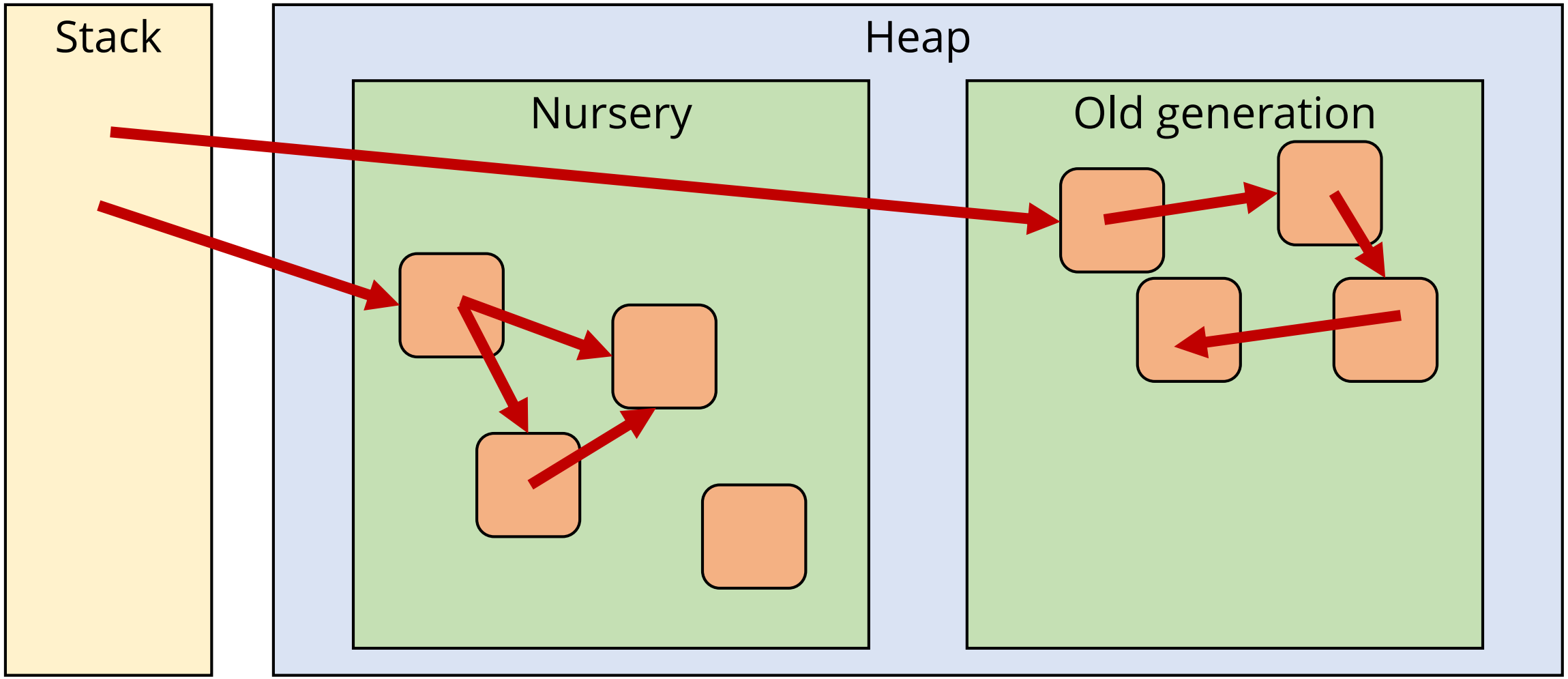
Generational GC



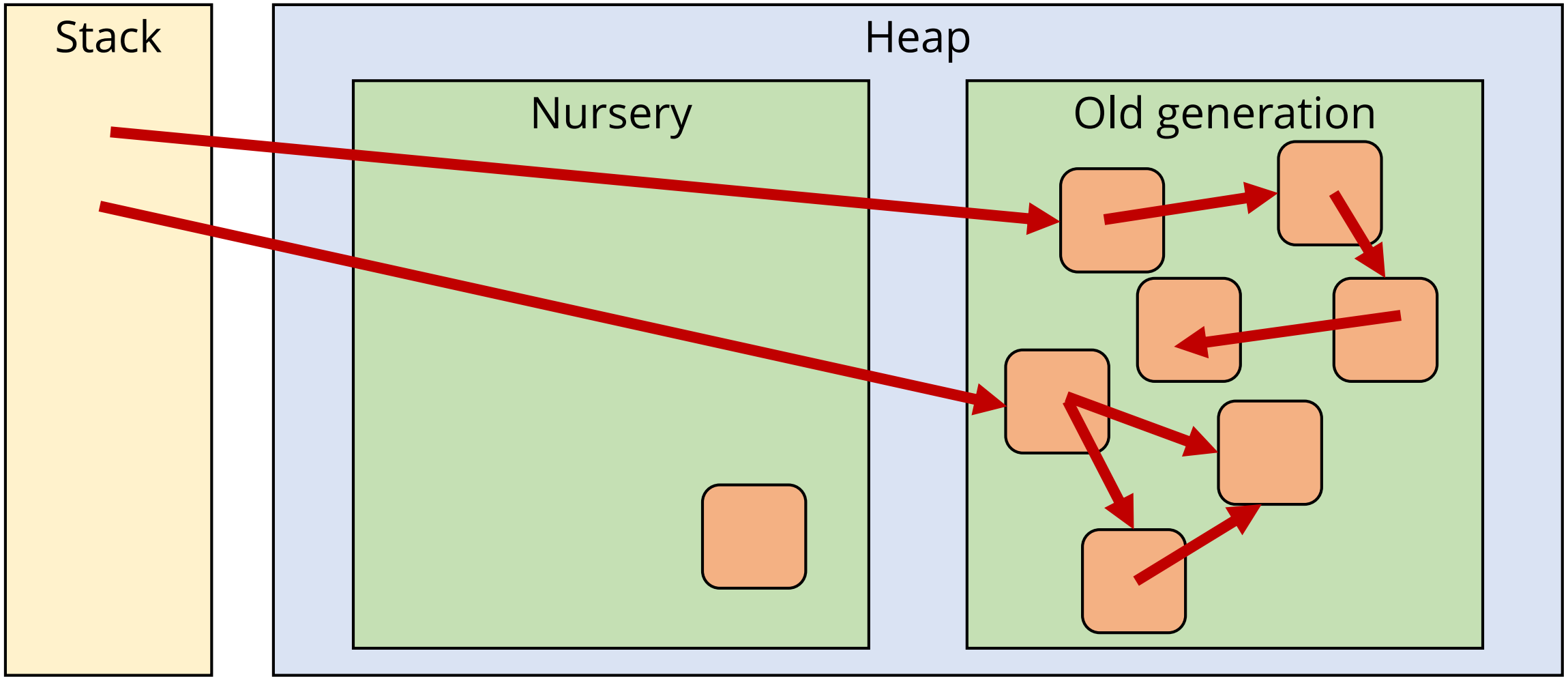
Generational GC



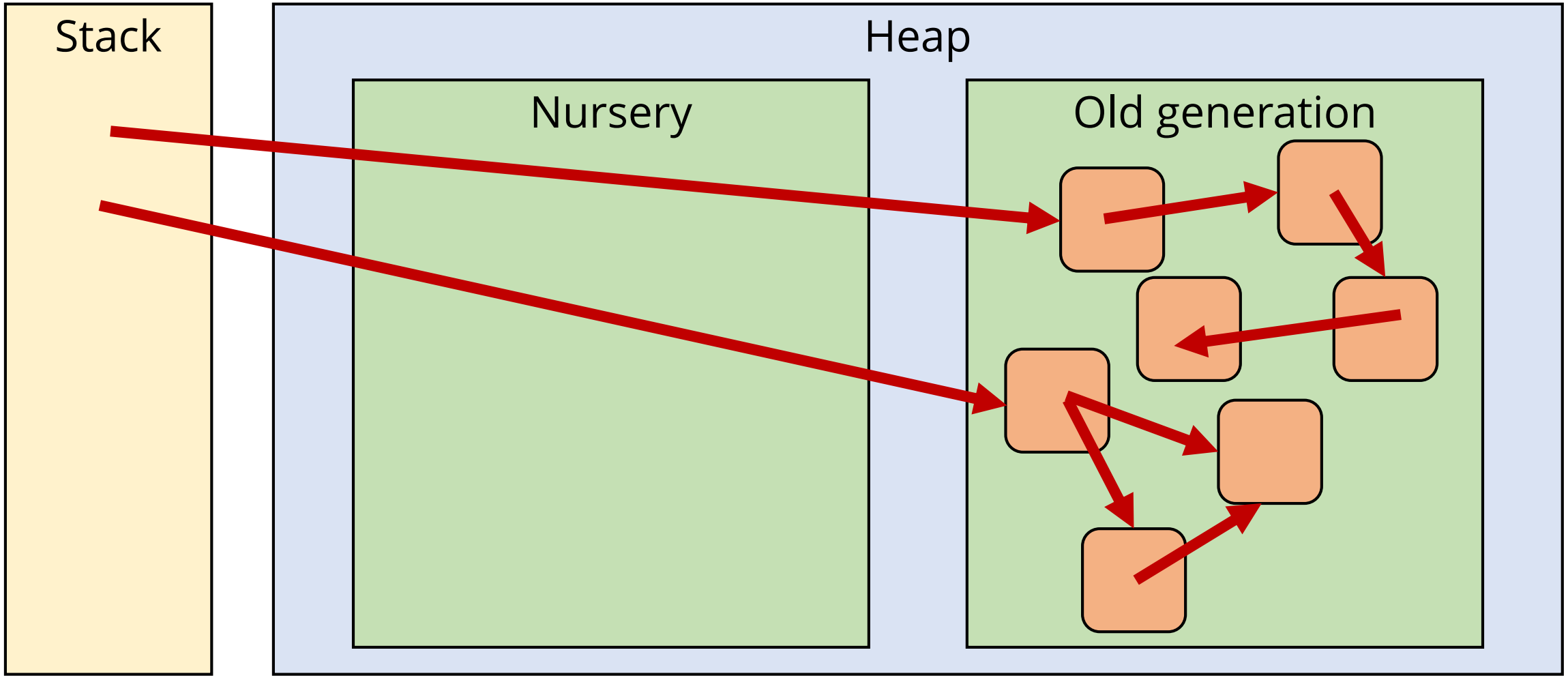
Promotion



Promotion



Promotion



En masse generational

- When nursery is full, collect nursery
- Copying GC, treat old generation as tospace
  - Old gen can have any allocator
- After GC, nursery is empty by definition
- When old generation is full, full collect

When to GC

- Allocation in old generation only done by young GC
- GC of old generation only done due to allocation in old generation
- Therefore: Full collection caused by partial young collection

Weird promotions

- Old generation is full, GC everything…
- Cannot now promote: Old gen is full!
- Nursery in half-collected state, but can’t continue copying

Solution: No worries!

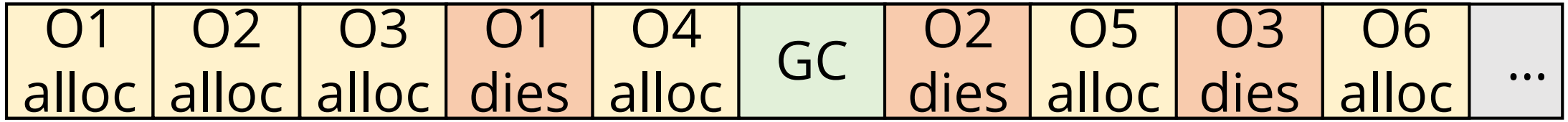
- Update references to already-copied objects
- Scan but don’t copy newly-found young objects
- When finished with full collection, collect young again
- Careful for collection loops!

En-masse advantages

- Copying without semispaces: No wasted heap
- Young collection very fast
- Young-death objects don’t impact performance

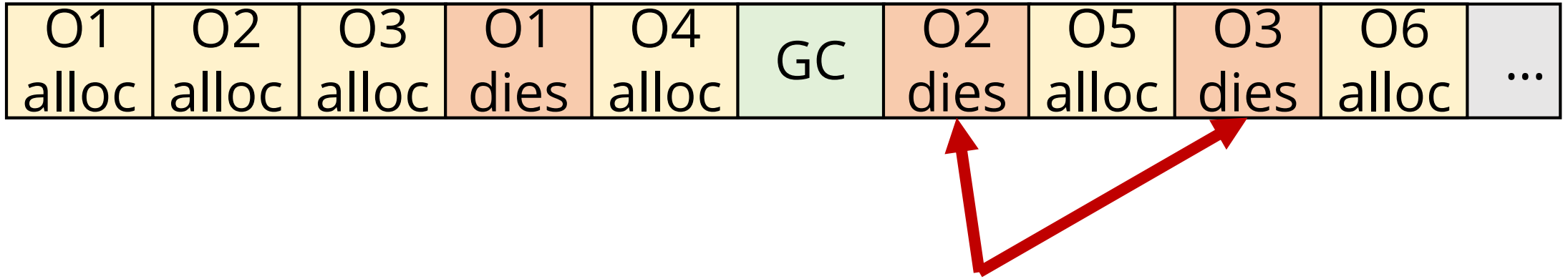
GC timeline

Time →



GC timeline

Time →



Wrongly promoted, will take a long time to collect!

Tempered generational GC

- Don’t promote objects on first GC
- Decide whether to promote from age
- Leave very young objects in nursery

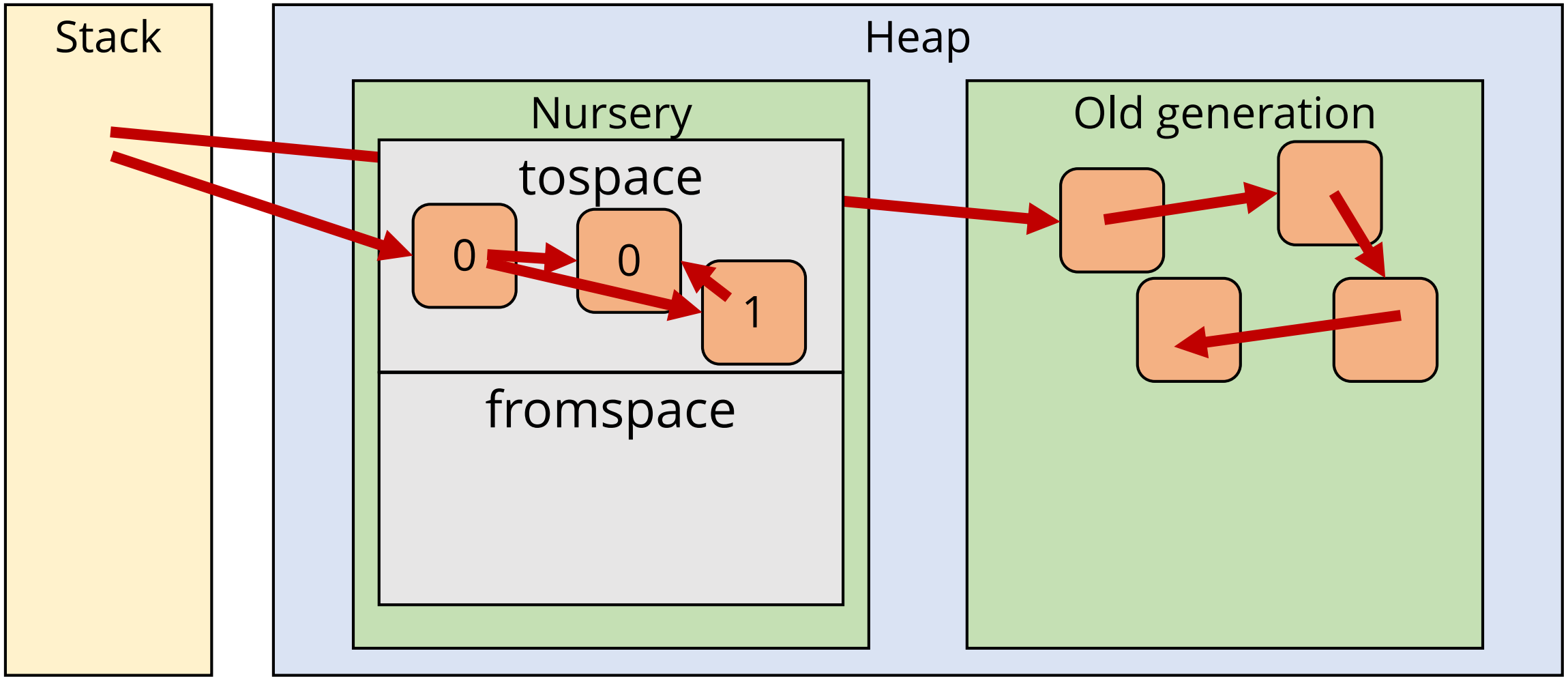
Determining age

- Wall clock:
  - Easy: Put time in header, check it
  - Problem: Machine- and program-dependent, “young” for one app may be “old” for other
- Memory age (words allocated since):
  - Robust to app differences
  - Difficult/impossible to implement

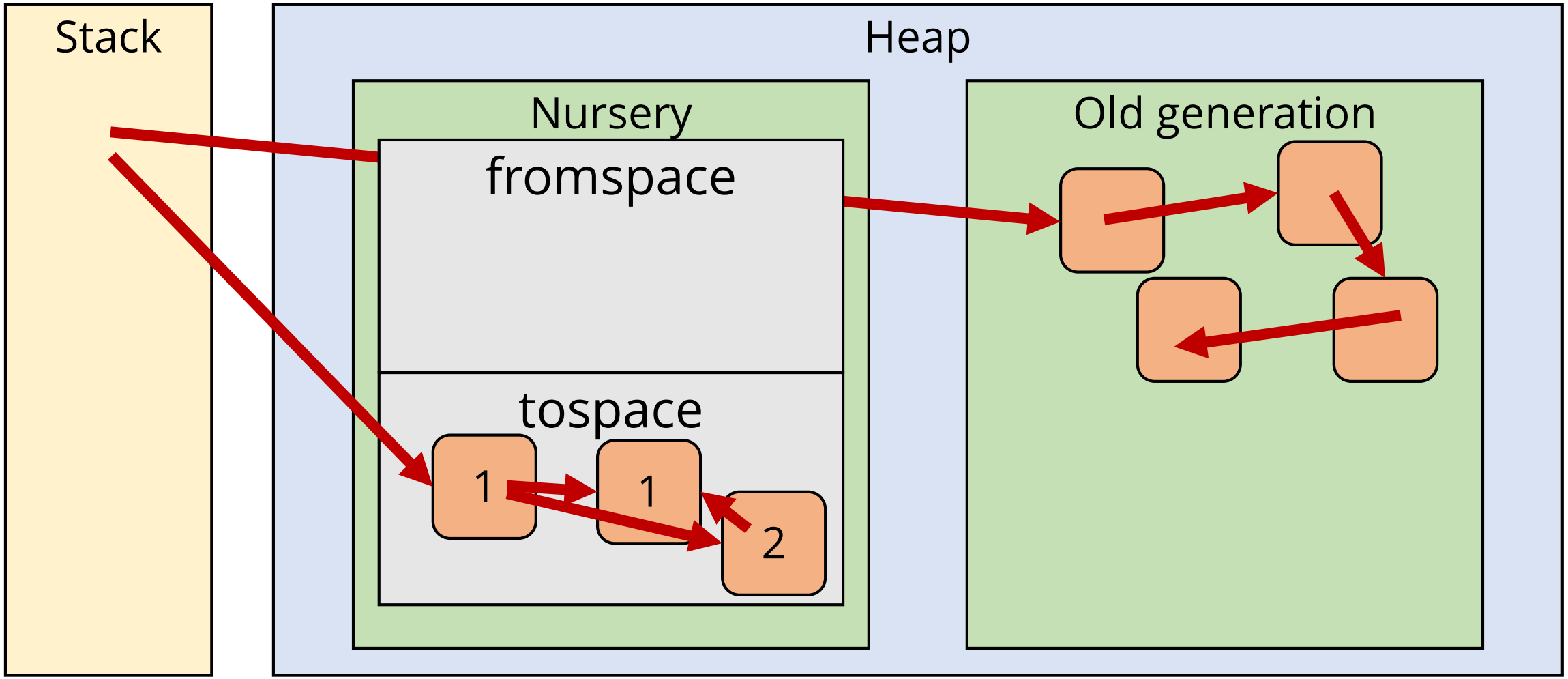
Determining age

- Collection count:
  - Every time object survives collection, collection count +1
  - Approximates memory age
  - Easy to track: Add counter to header

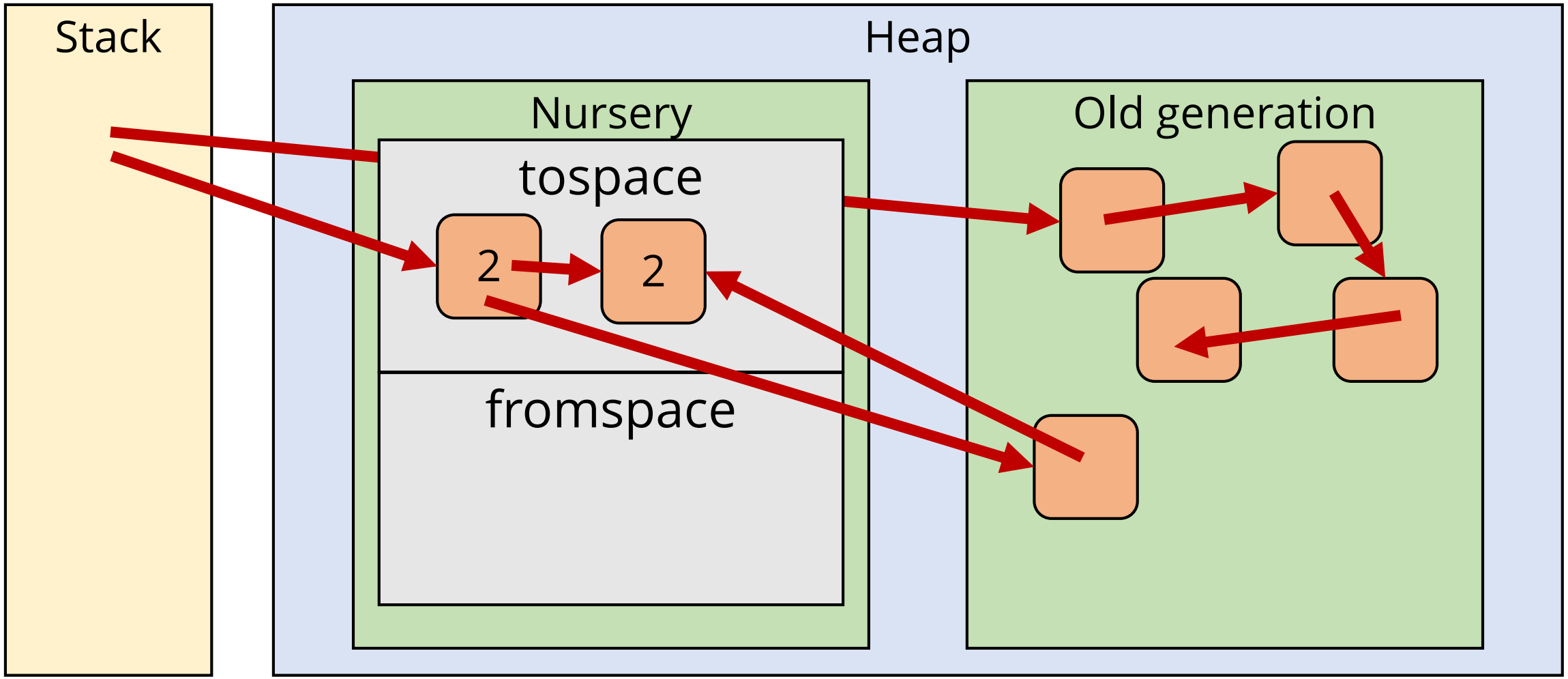
Collection counter



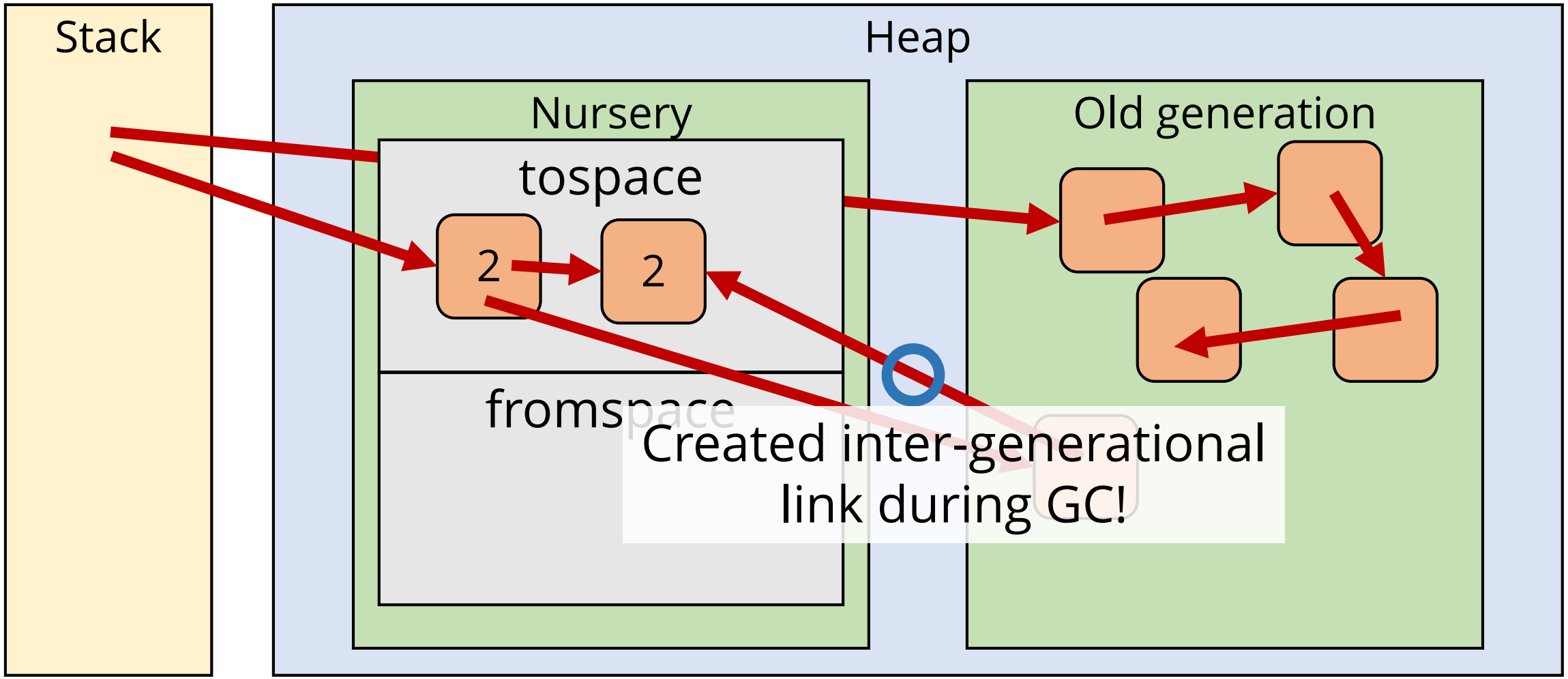
Collection counter



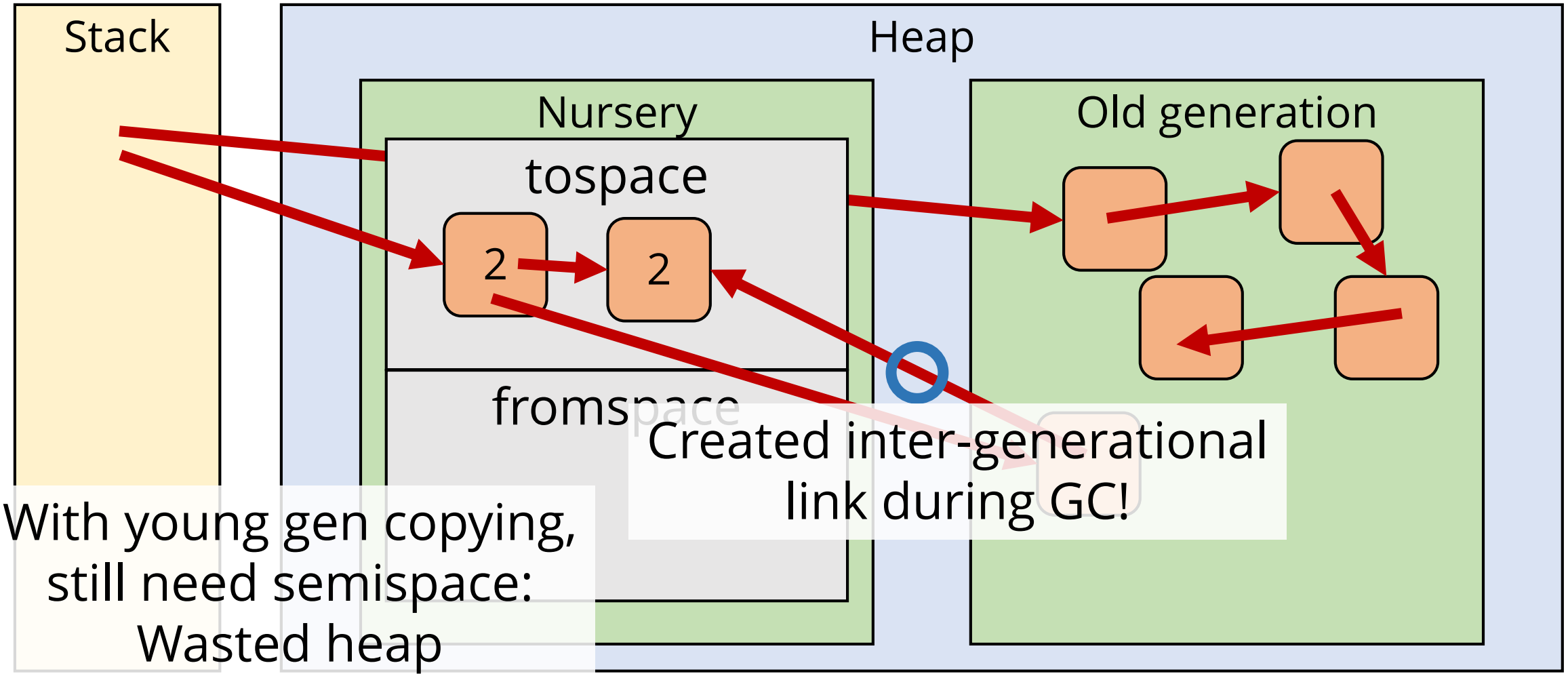
Collection counter



Collection counter



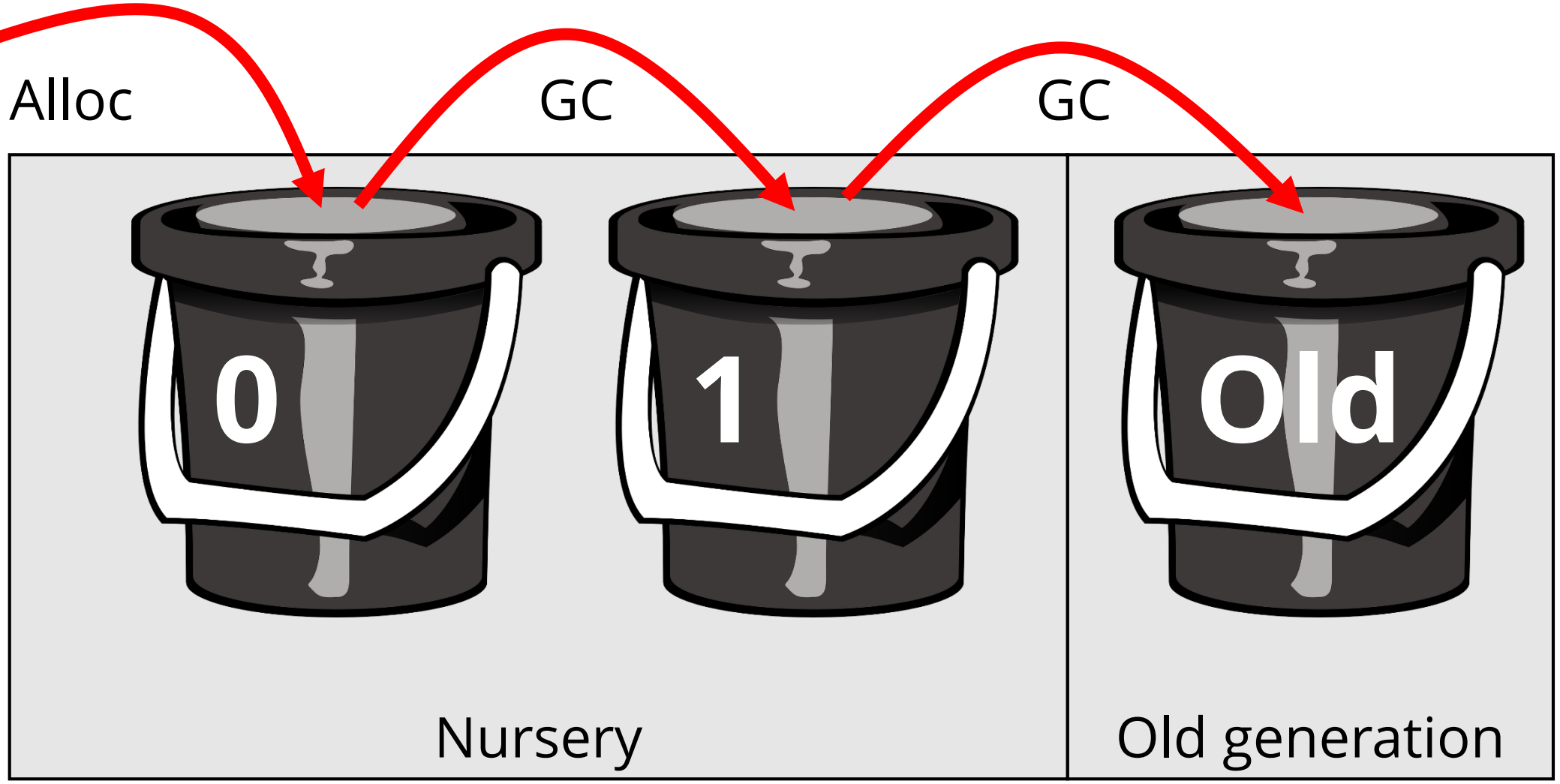
Collection counter



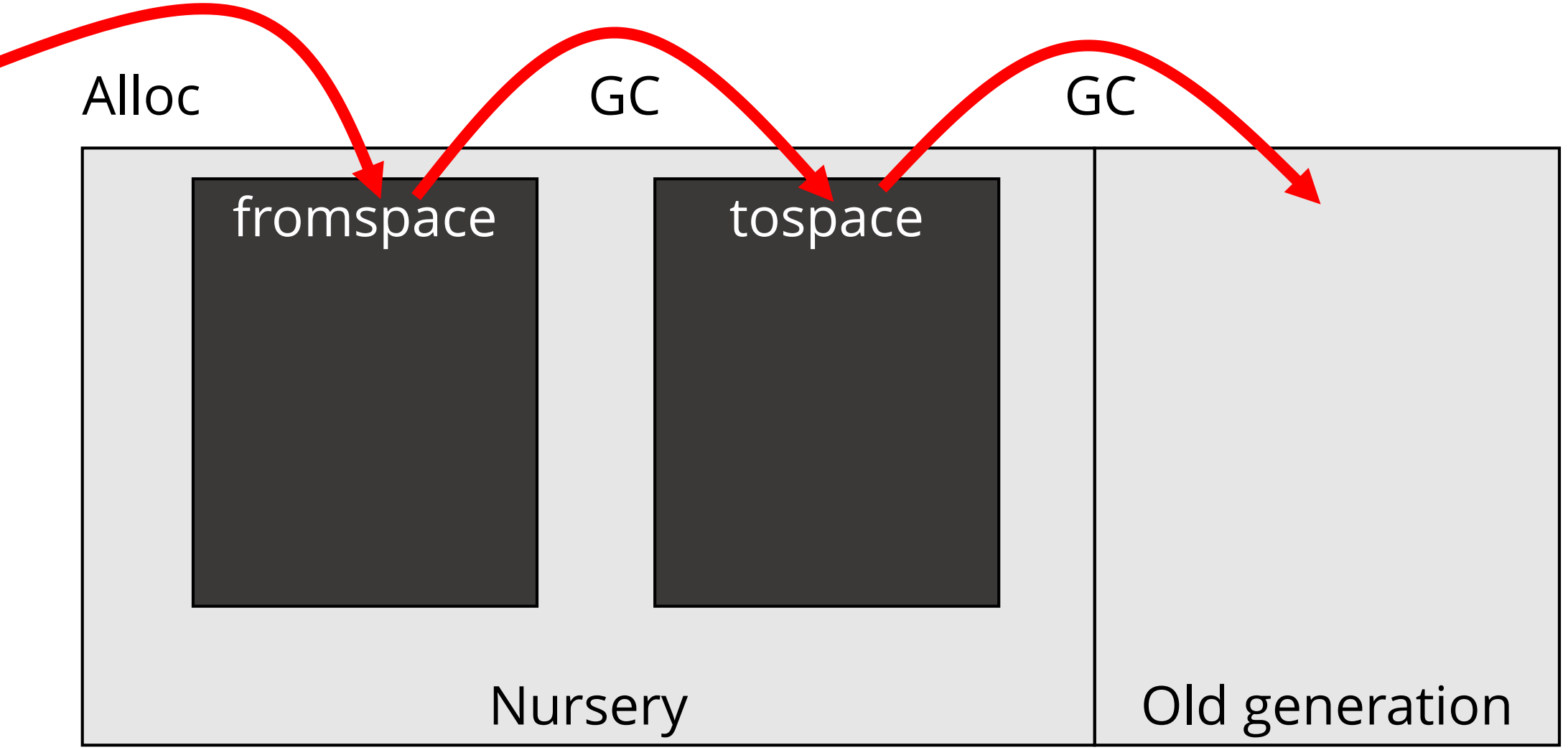
Collection counter

- Semispace problems are back: Waste half of young space
- Not all objects promoted: Can create inter-generational links

The bucket brigade



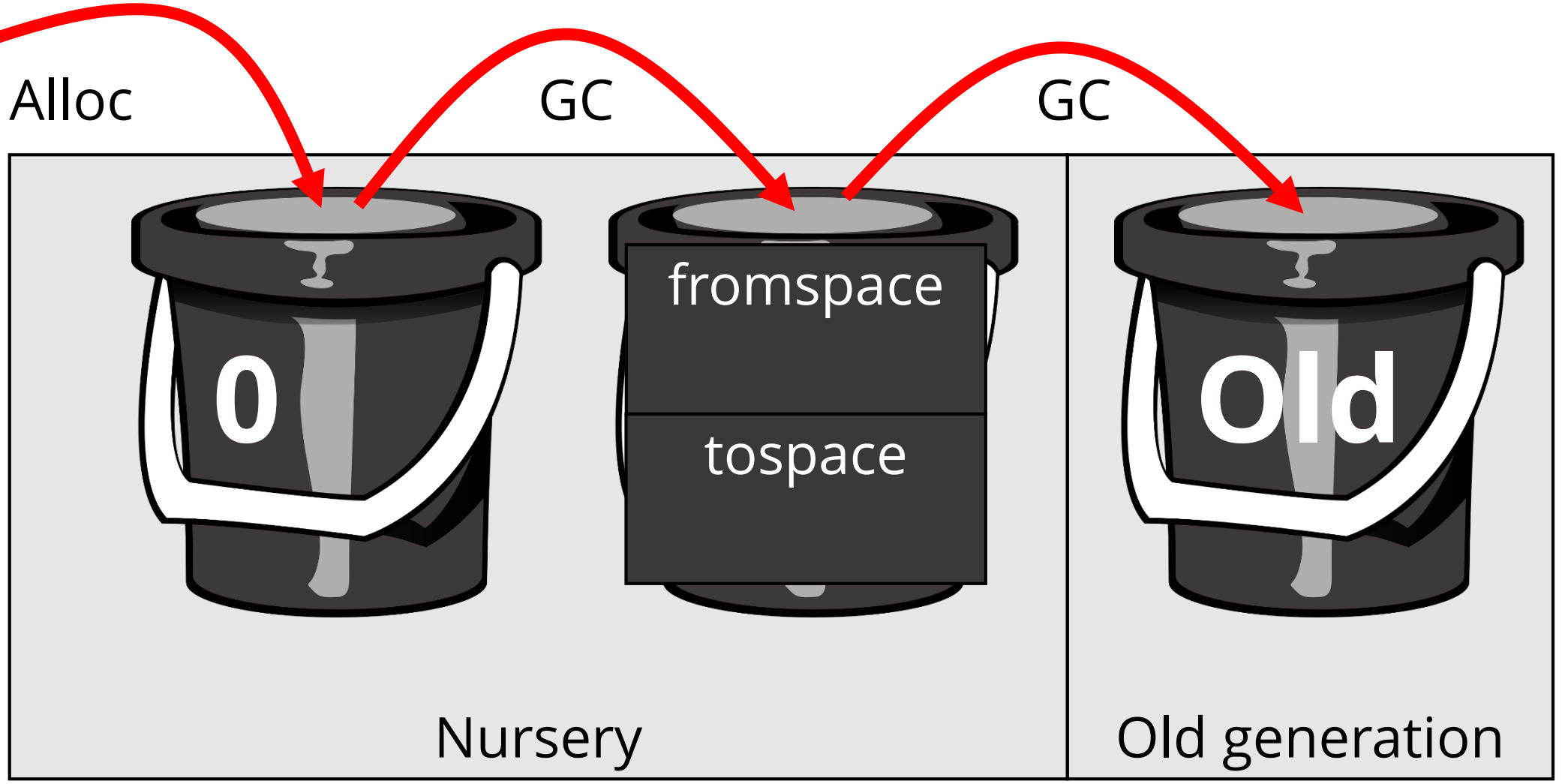
The broken bucket brigade



Bucket brigade

- Copy objects from bucket n to bucket n+1
- Cannot just use semispace copying with from=0, to=1:
  - tospace already partially used
  - Emptied tospace prefix not available

The Java bucket brigade



Java [HotSpot] collection

- Nursery:
  - Bucket 0 (“Eden”) flat space
  - Bucket 1 (“Survivor”) semi-space copying
- Old generation mark-and-compact

Young collection algorithm

```
youngCollect():
  (scan roots)
  b1fromspace, b1tospace := b1tospace, b1fromspace
  while loc := worklist.pop():
    obj := *loc
    if !obj->header.forward:
      if obj in bucket 0:
        obj->header.forward := (copy to b1tospace)
      else if obj in bucket 1:
        obj->header.forward := (copy obj to old gen)
      (add references in newly-copied object)
    *loc := obj->header.forward
```

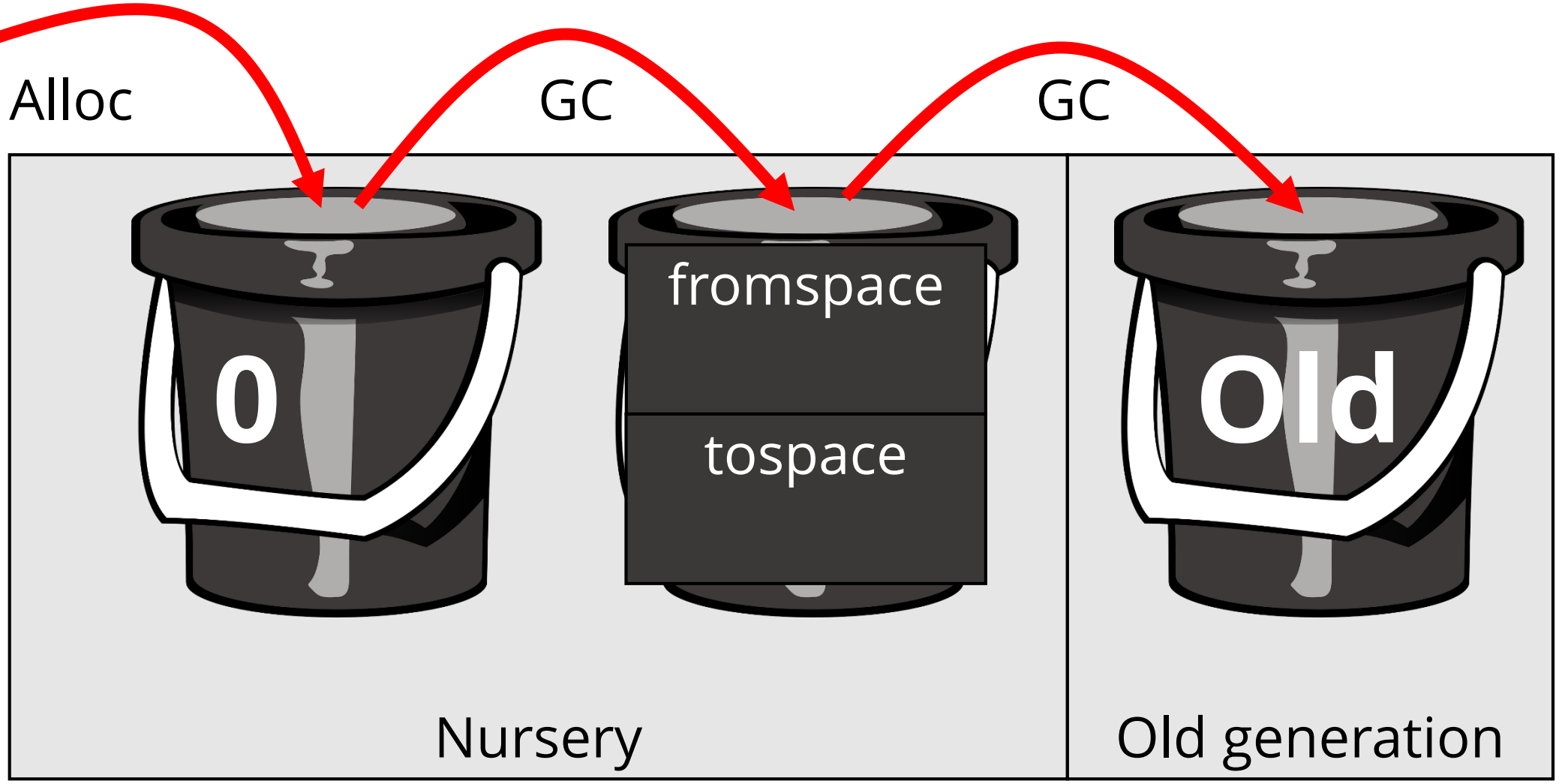
Overhead

- But still wasting heap for semispaces!
- Most objects die young: Even by bucket 1, most objects dead
- Remember: You allocate pools, so you choose heap, generation, bucket sizes!
- Typically, bucket 0 32x larger than one space in bucket 1

Caveat

- Of course, bucket 0 could fill b1tospace…
- Expand b1 spaces during collection: Want H=n*L (all objects copied in are in L)
- Can’t expand more? Fallback to copying to old gen

Breather



Write barrier

- To collect just nursery, remember objects in old generation which have inter-generation references
- In practice: Over-approximate
- Treat portions of old generation as roots for nursery collection

Remembered set

- “Remember” a portion of old generation as “interesting”
- During young GC, scan objects in remembered set
- During full GC, clear remembered set

Cards

- Typical (read: Java) GCs use “cards”
- Divide pools into smaller chunks called cards of power-of-2 size
- Remembered set is bit-array of size #-of-cards-per-pool
- To remember a card, mark its bit

Card barrier

```
write(obj, loc, val):
  if genOf(obj) == old and
     genOf(val) == young:
    poolOf(obj)->remember[cardOf(obj)] := 1
  *loc = val

poolOf(ptr):
  ptr & POOL_MASK

cardOf(ptr):
  (ptr & ~POOL_MASK) >> CARD_SIZE_POW2
```

Card use

- During young GC, scan each old pool's remembered set
- For each '1', scan objects in corresponding card
- Note: *Cards* must be parsable!

Parsable cards

- Objects may span cards
- To parse card, need to know offset of first object in card
- Additional element in pool: Offsets of first objects within cards

Parsable cards

```
oldAllocate():
  … bump-pointer …:
    ret := end
    end += size
    if cardOf(obj) != cardOf(end):
      pool->firstObjs[cardOf(end)] :=
        offsetInCard(end)

  … split free-list …:
    if cardOf(ret) != cardOf(split):
      pool->firstObjs[cardOf(split)] :=
        offsetInCard(split)
```

Summary

- Young gen in buckets, promoted to old gen from oldest bucket
- Oldest bucket semispace
- Pointers from old gen remembered by cards
- Cards treated like roots (but scanned as objects)