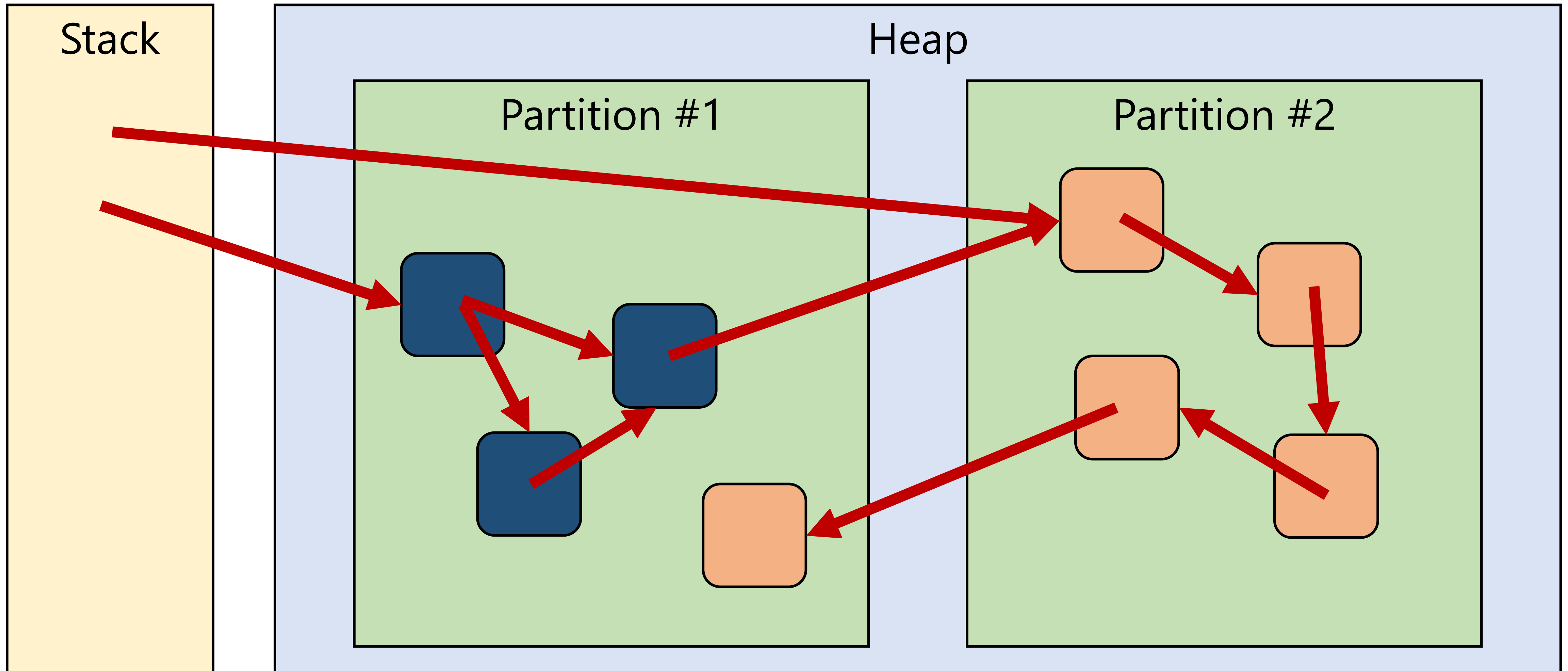
Generational GC

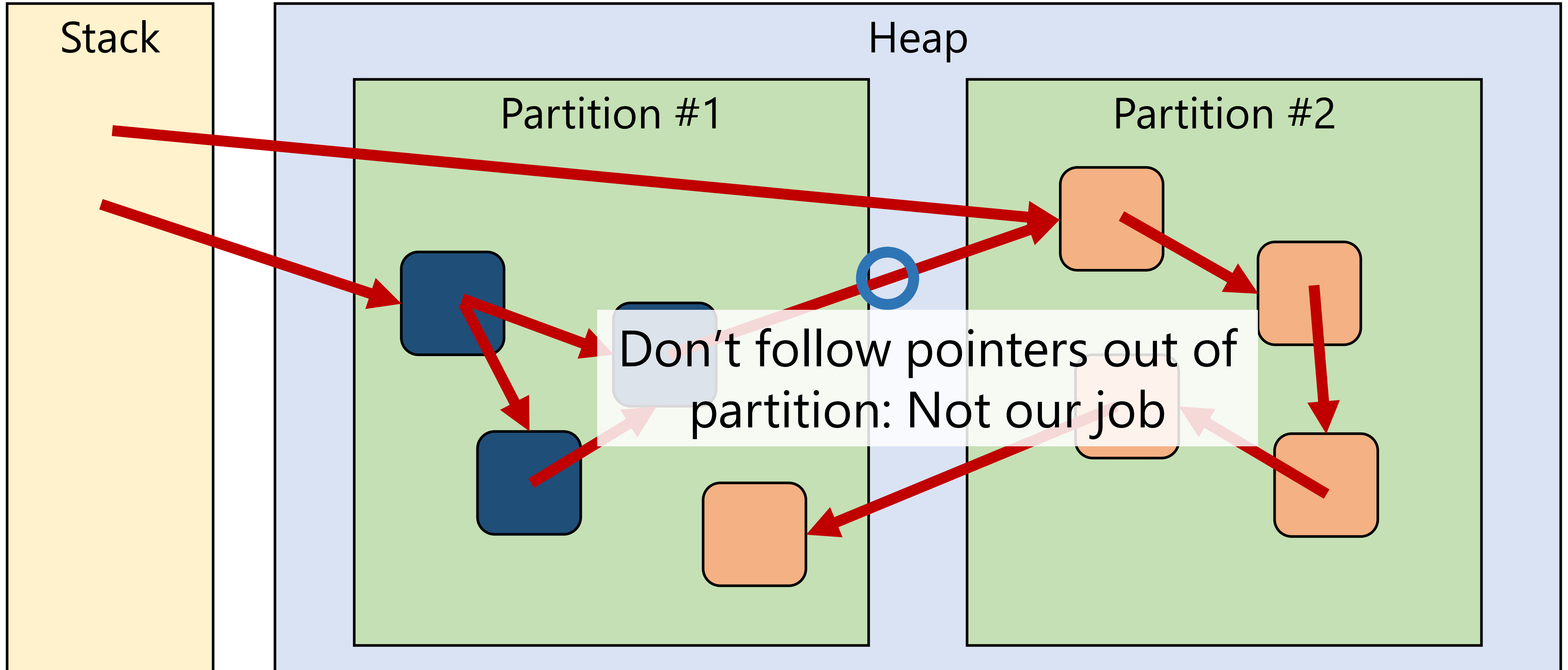
Review

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

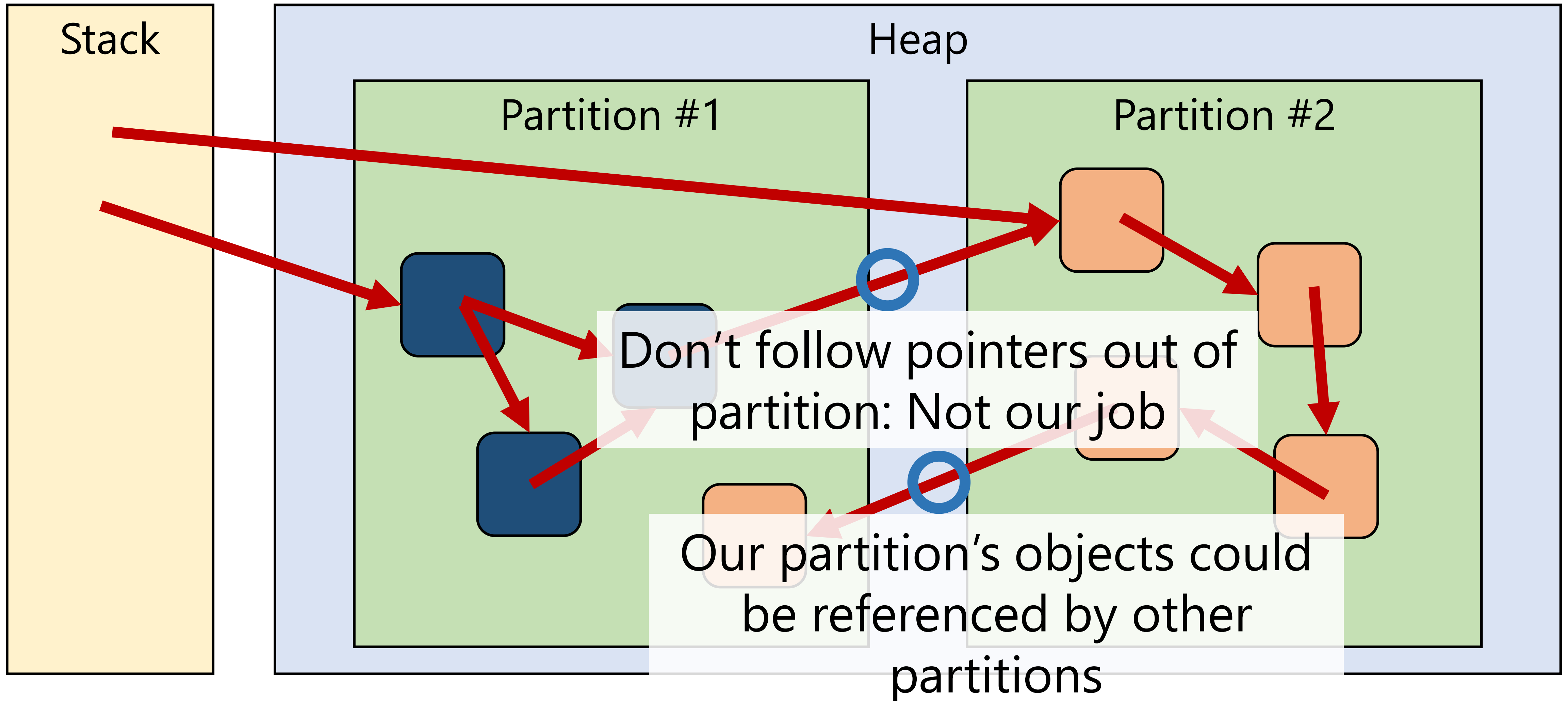
Single-partition scanning



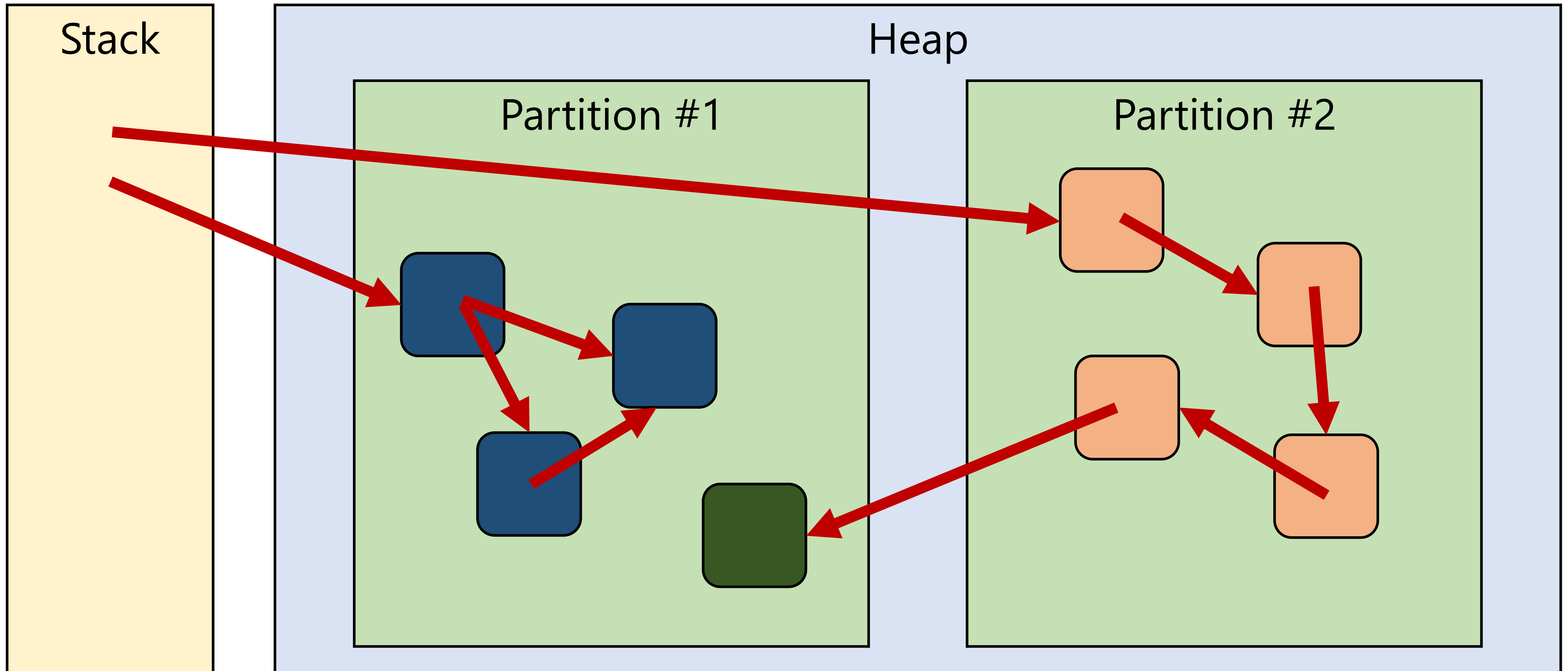
Single-partition scanning



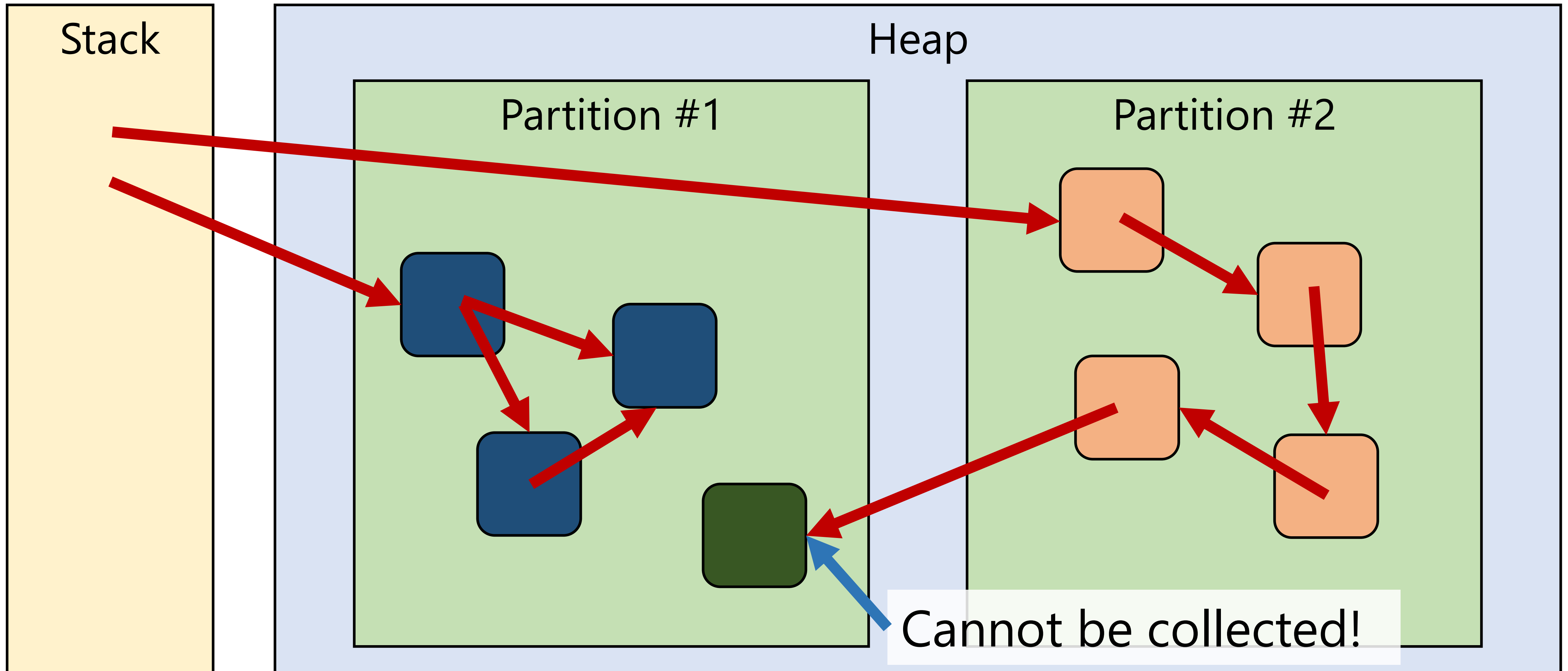
Single-partition scanning



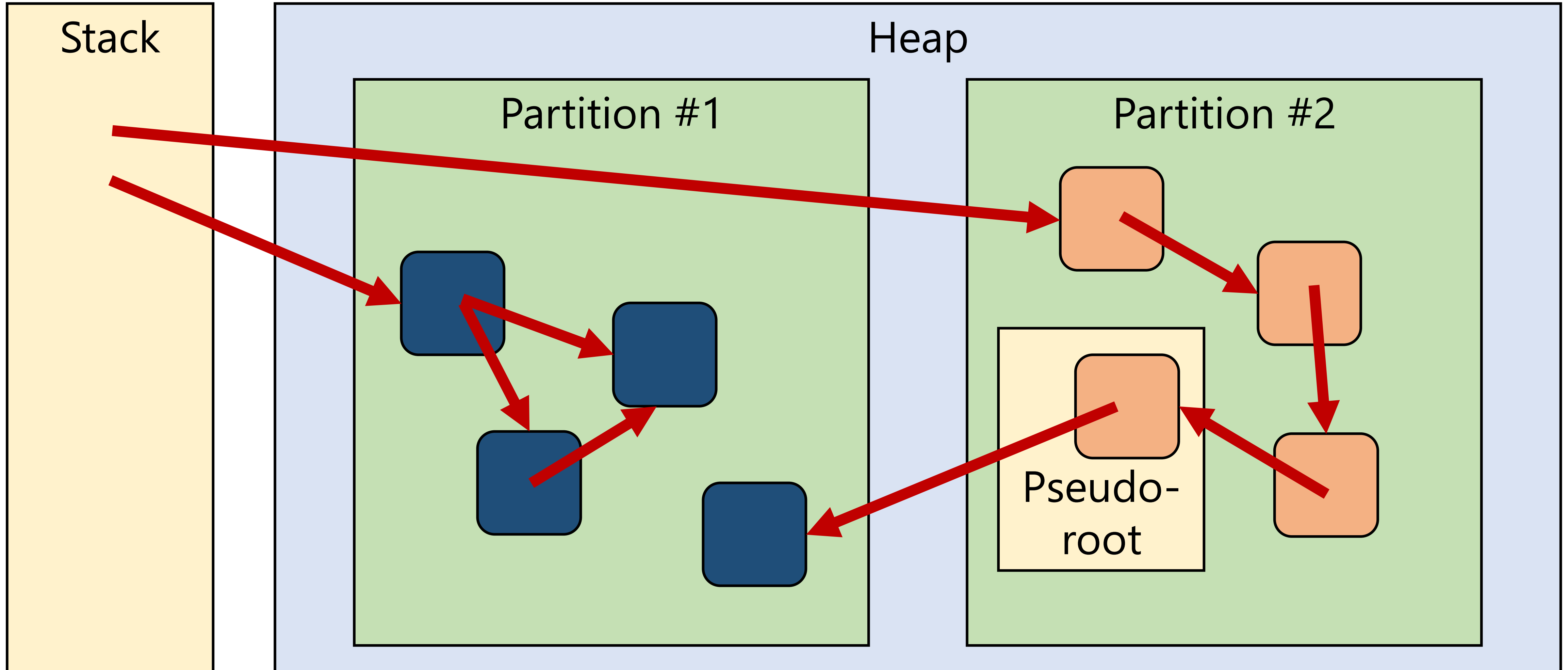
Single-partition scanning



Single-partition scanning



Alternate approach



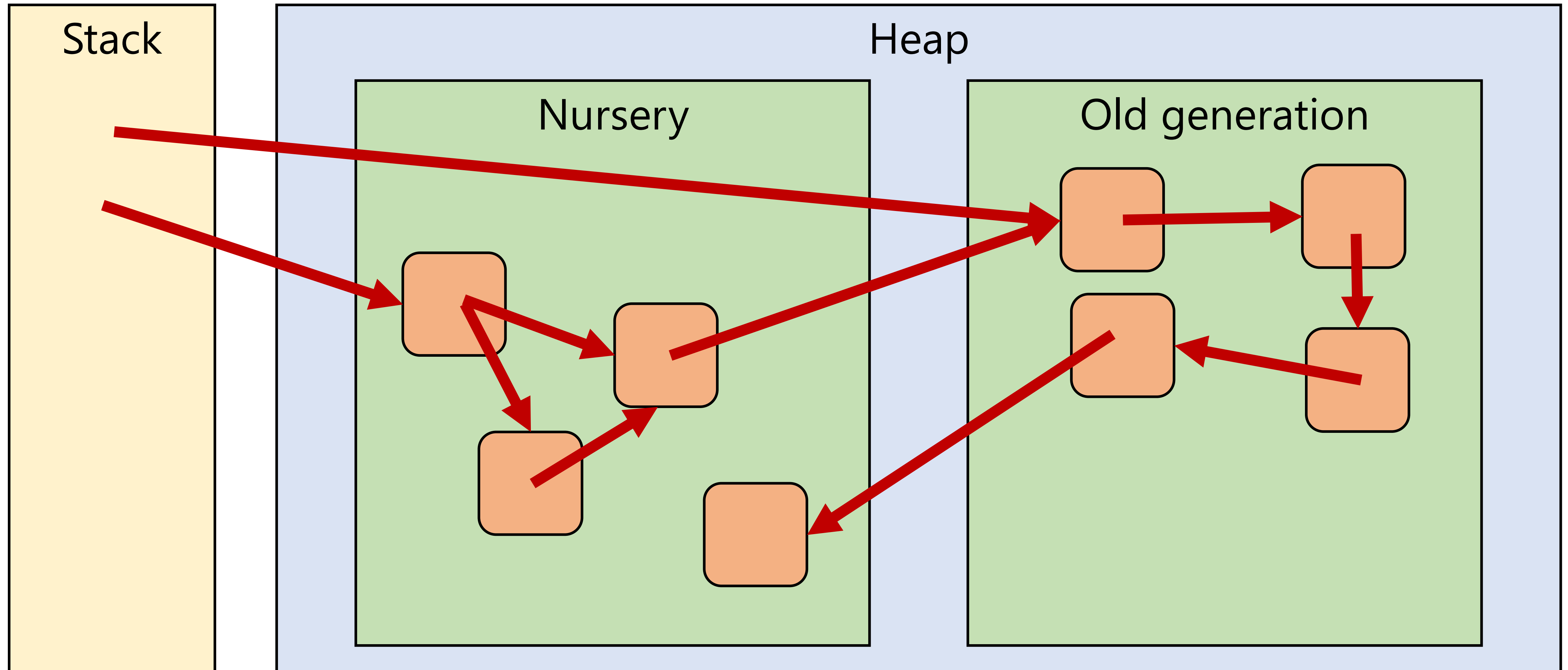
The big idea

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

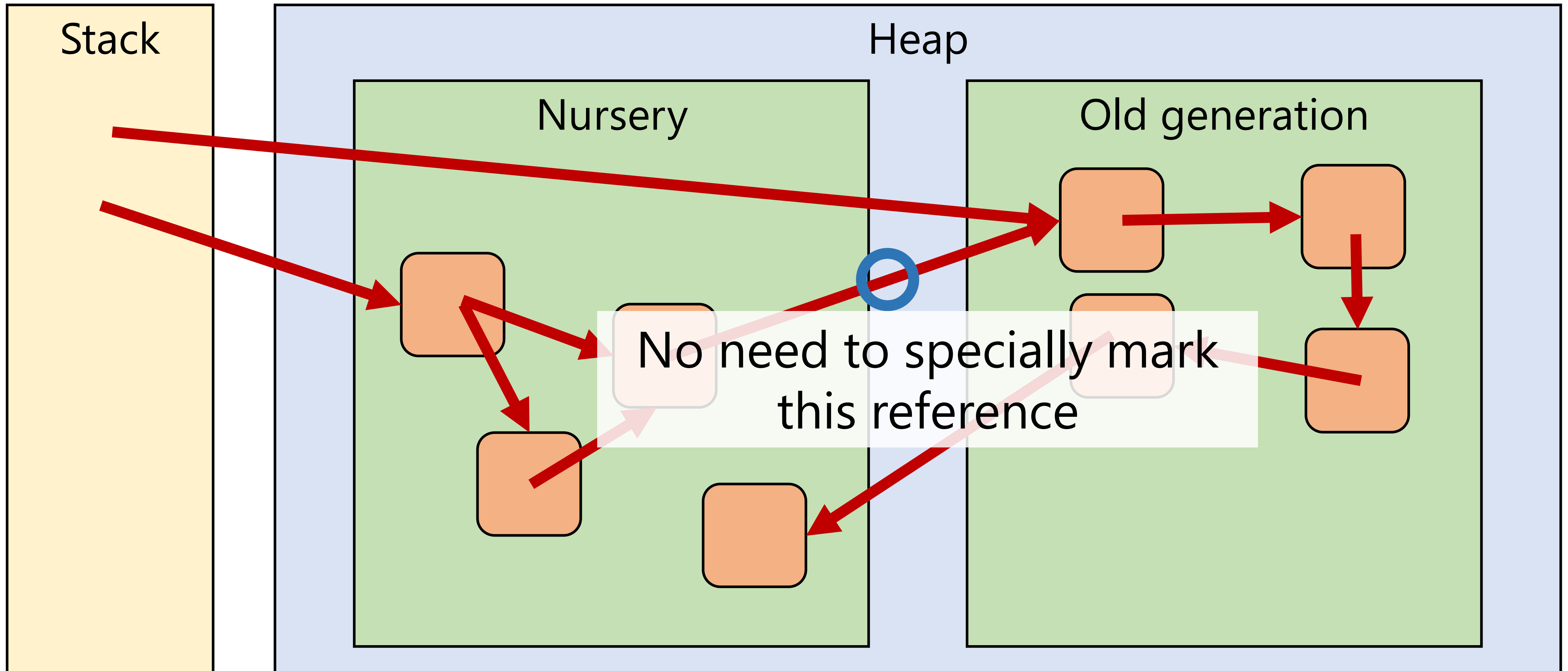
Inter-partition references

- 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

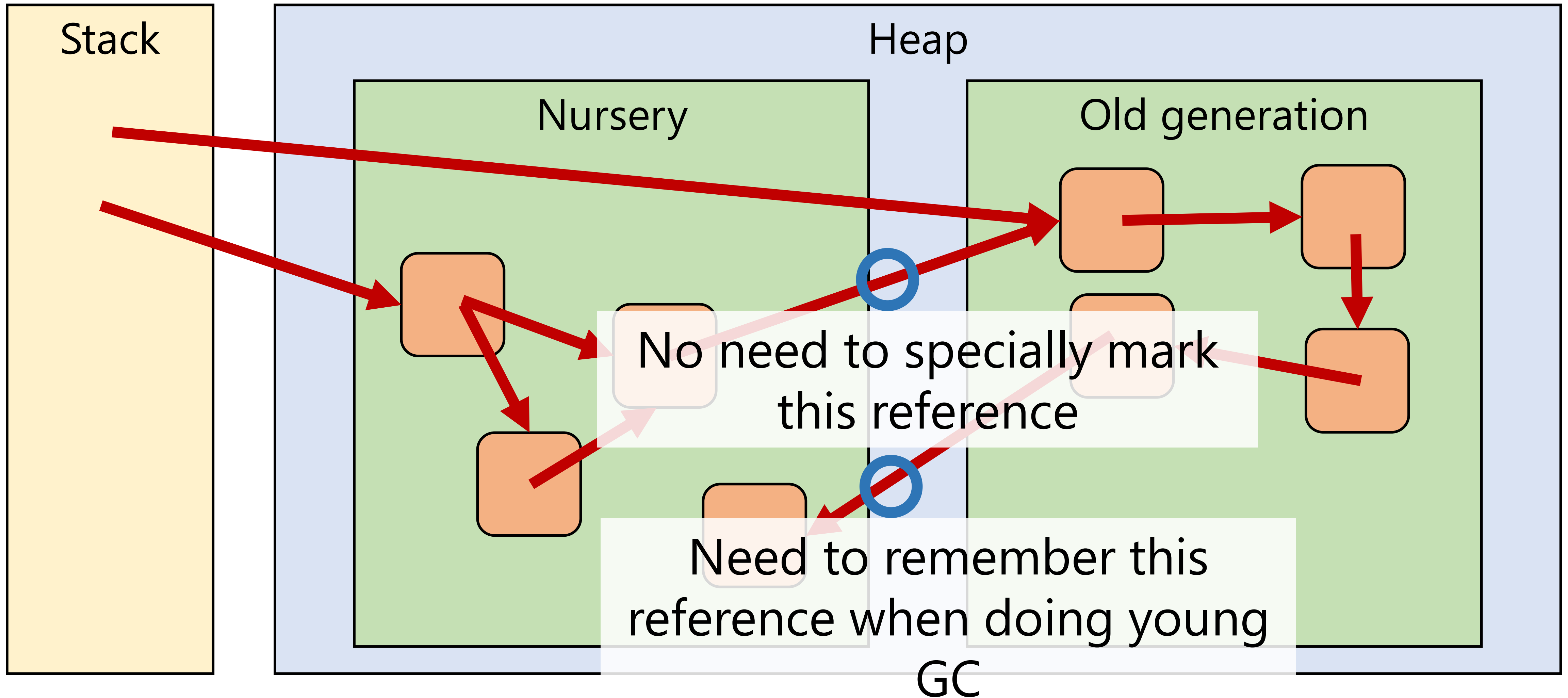
Generational GC



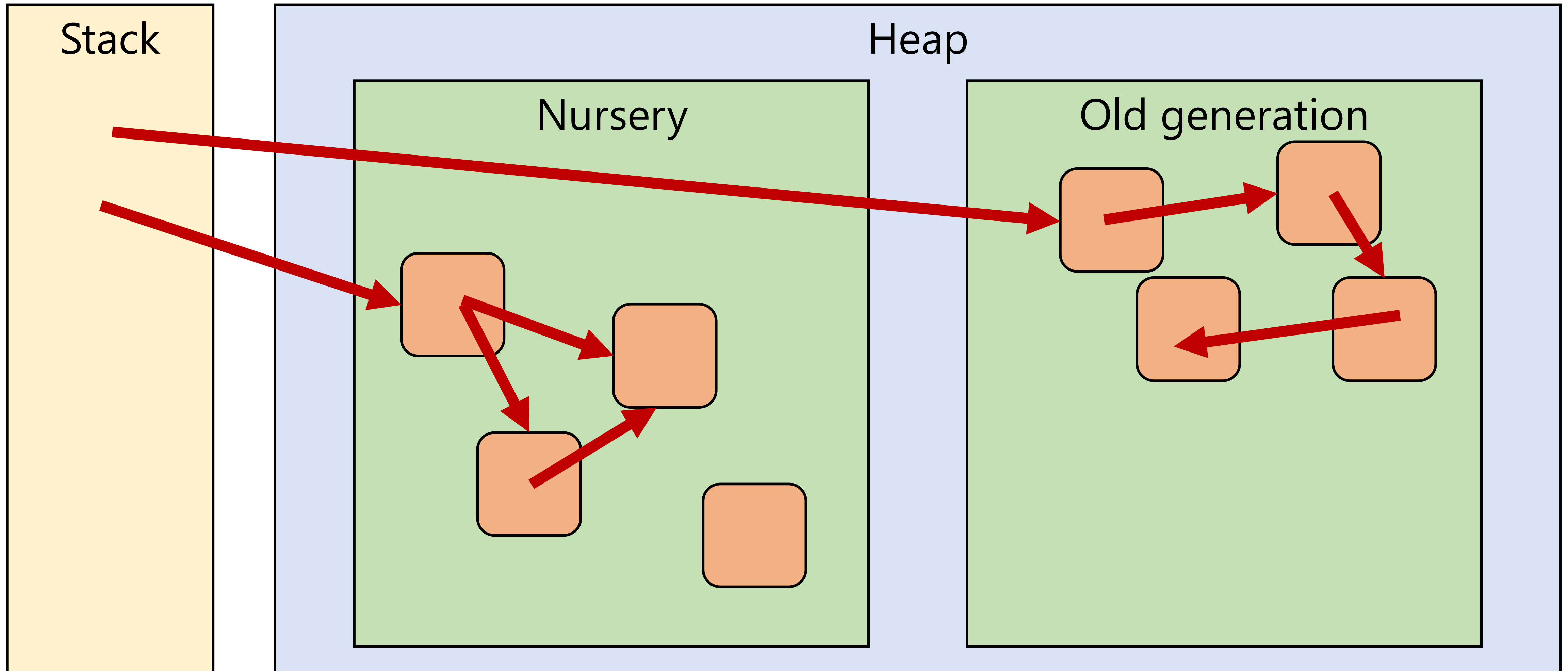
Generational GC



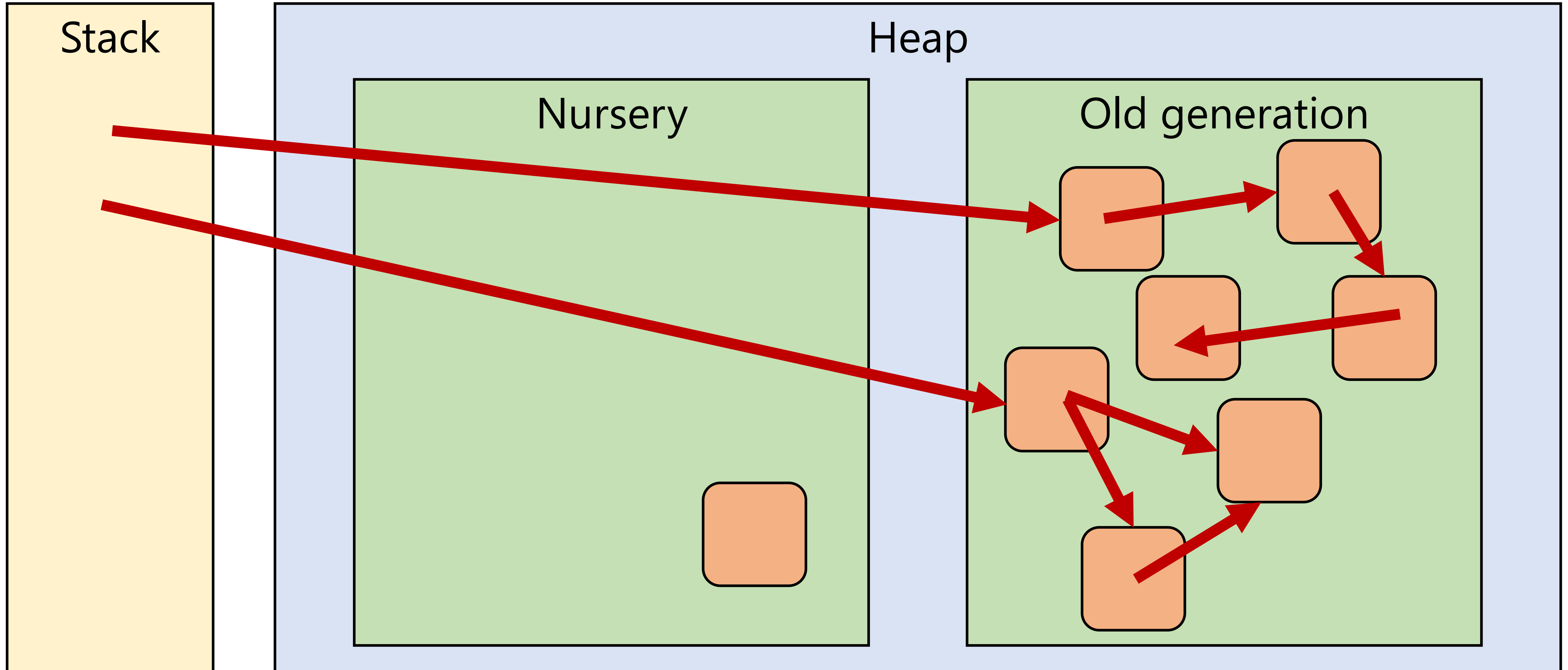
Generational GC



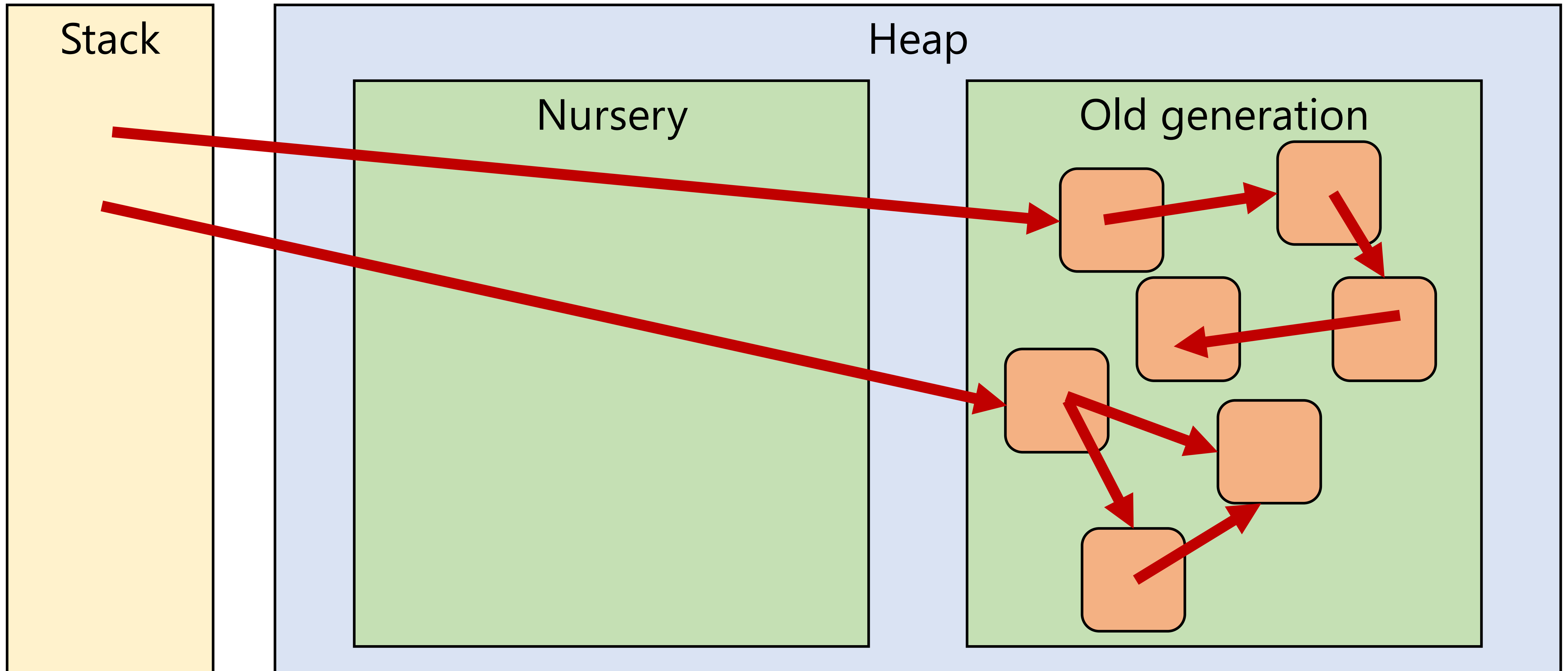
Promotion



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

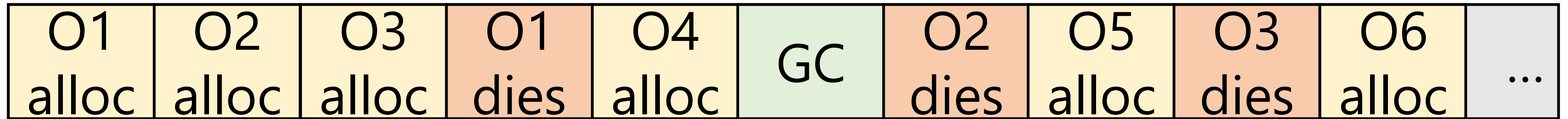
Solution: No worries!

- 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!

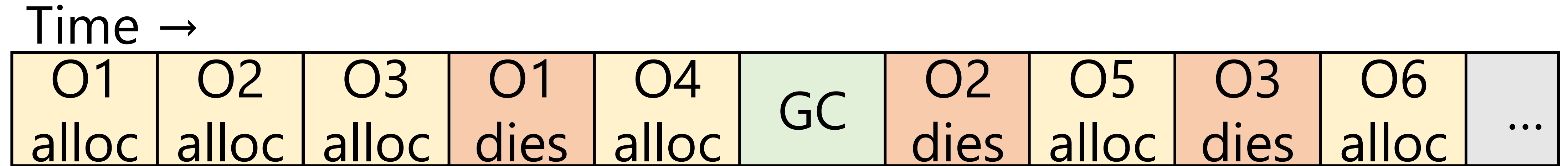
En-masse advantages

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

GC timeline



GC timeline



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

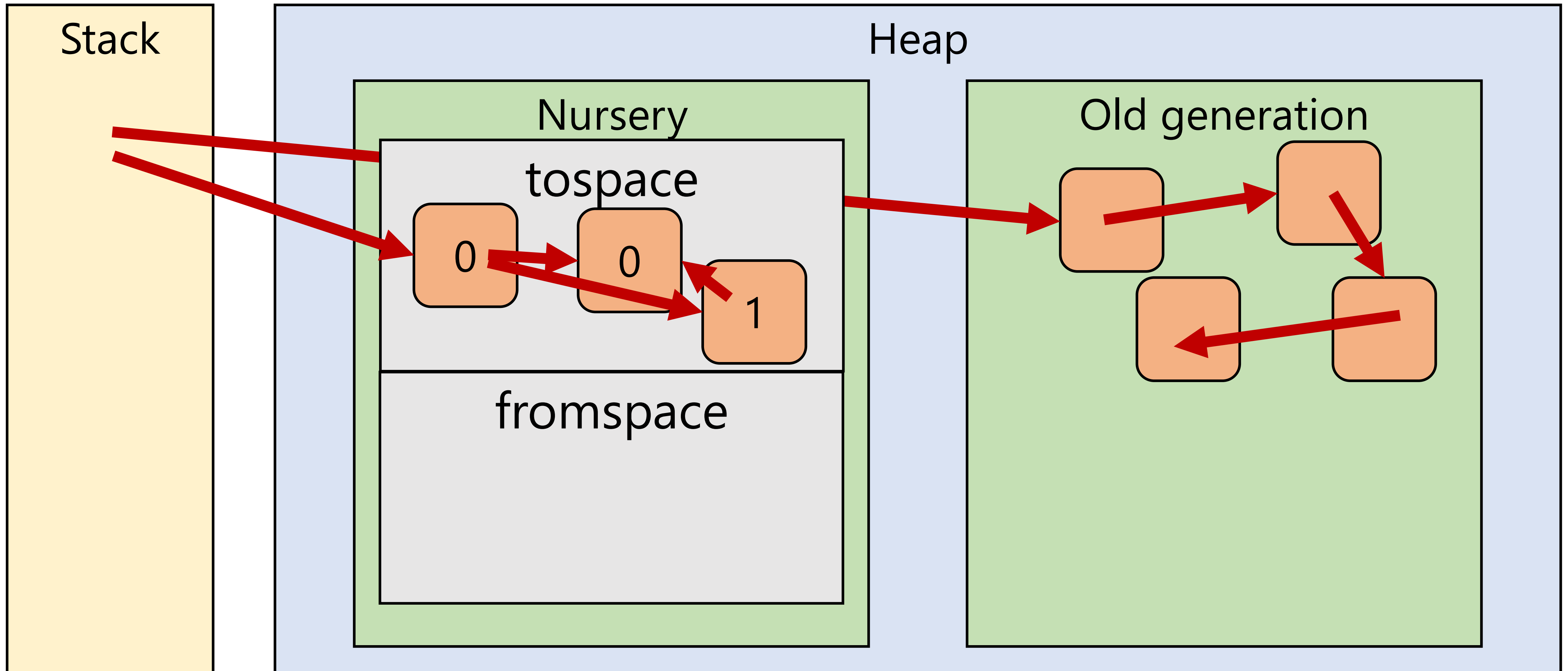
Determining age

- 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

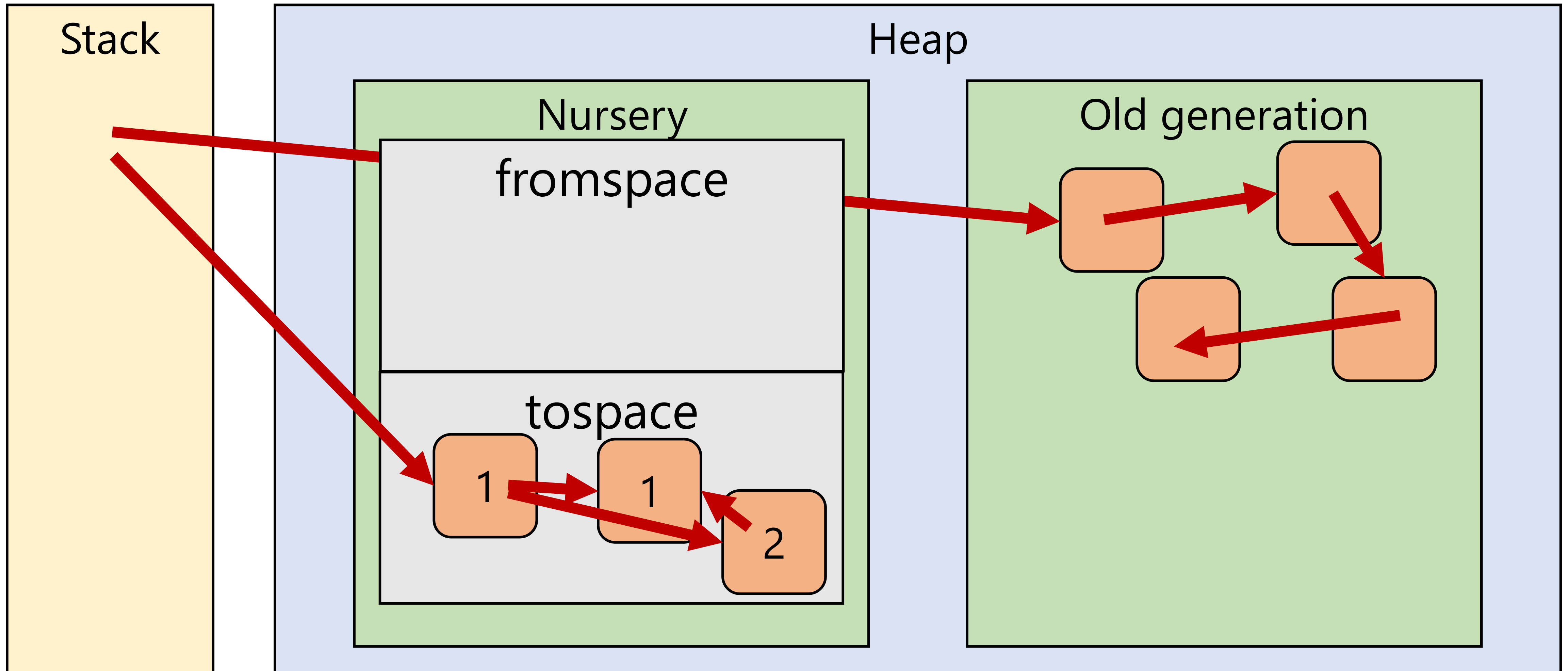
Determining age

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

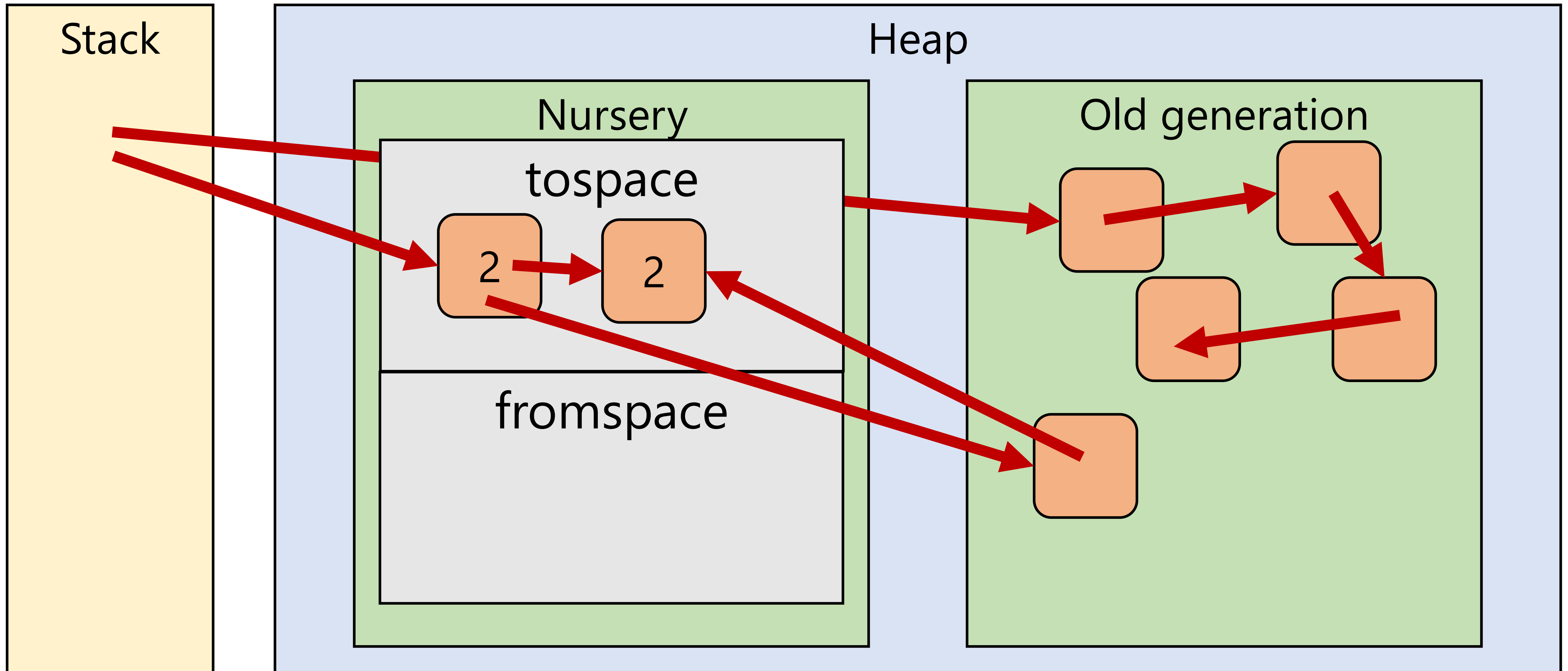
Collection counter



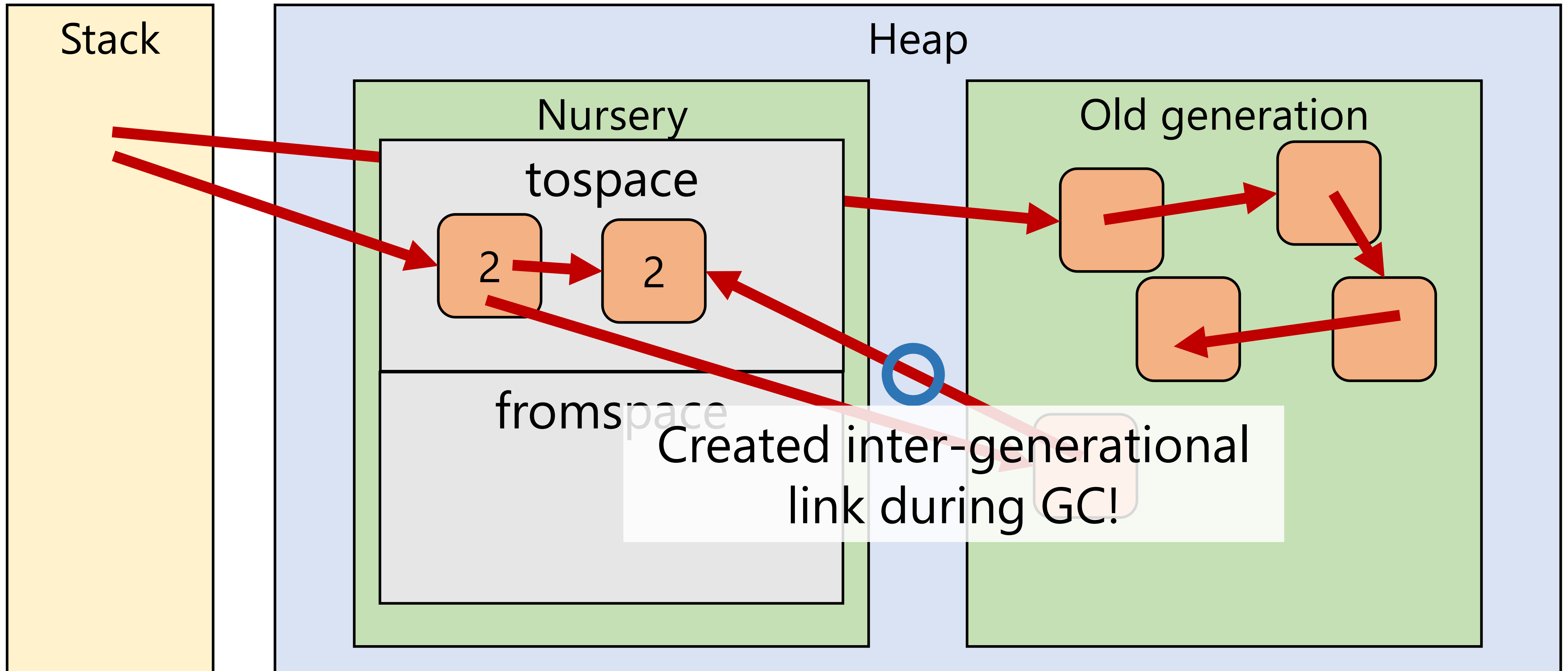
Collection counter



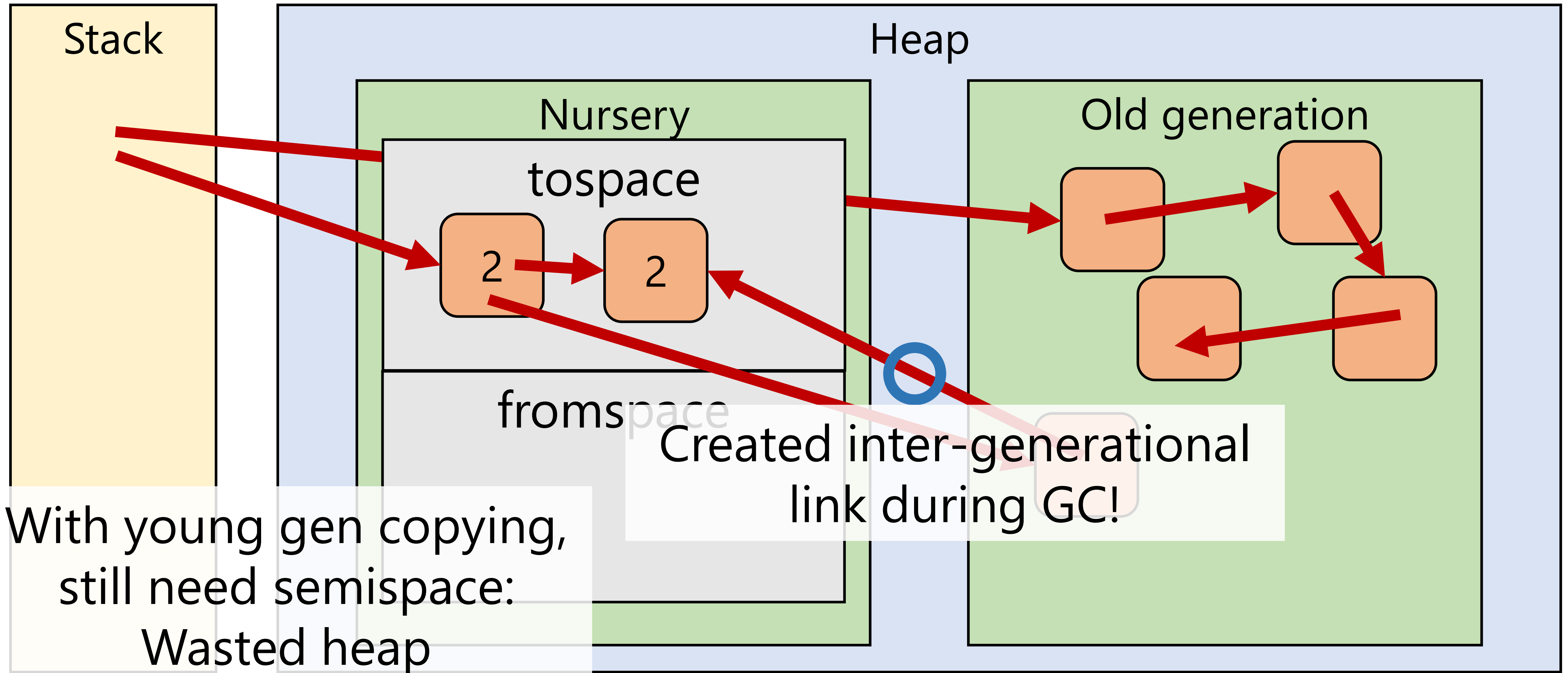
Collection counter



Collection counter



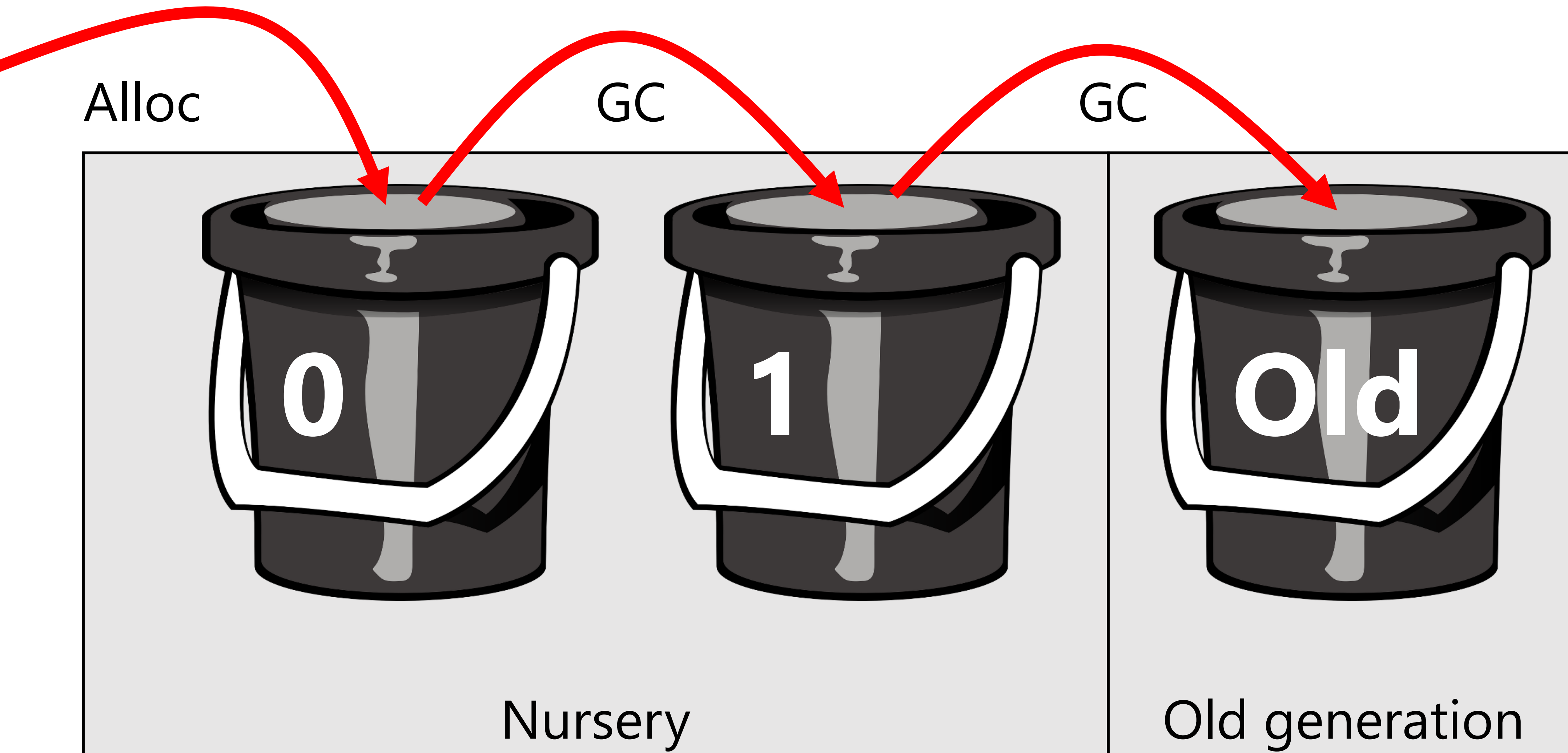
Collection counter



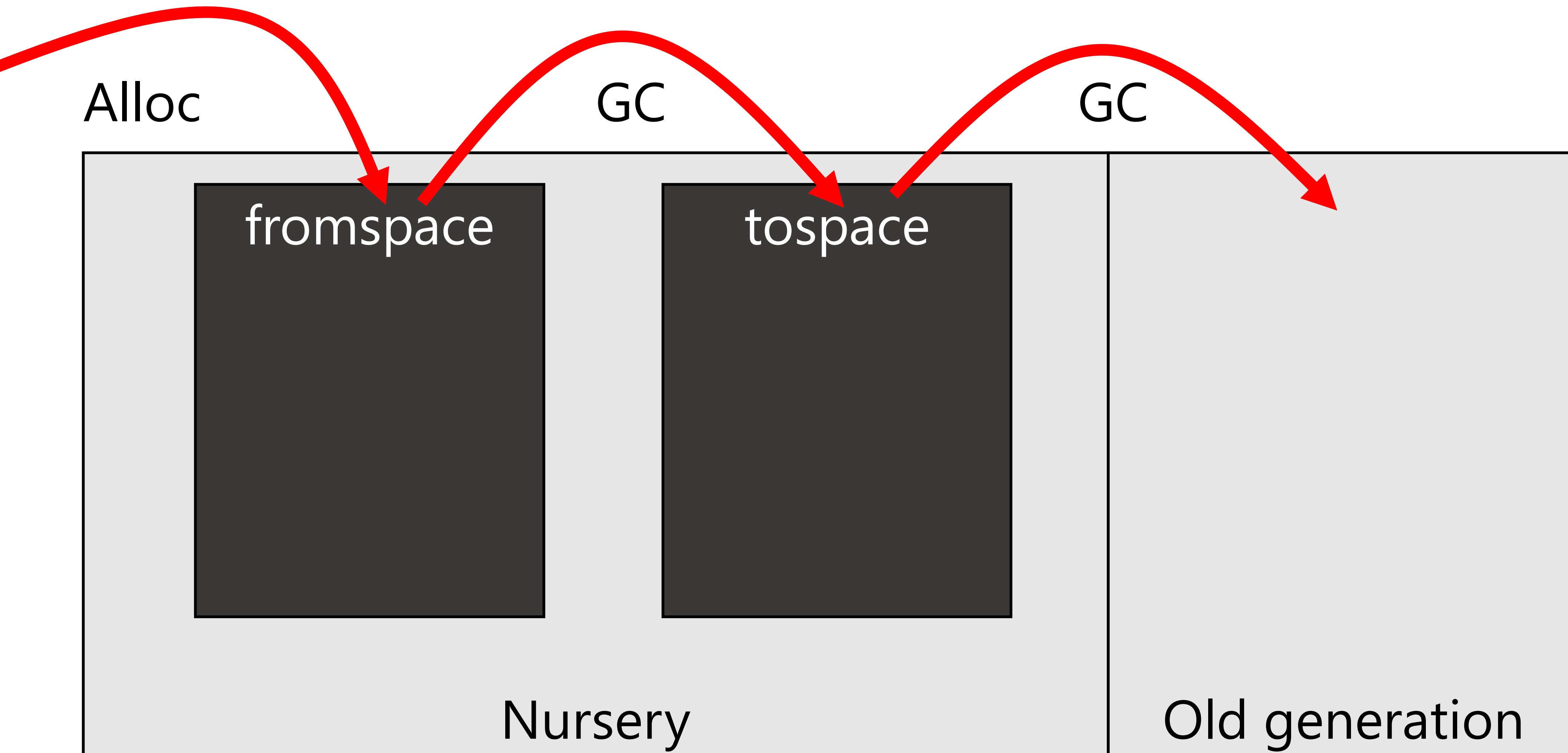
Collection counter

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

The bucket brigade



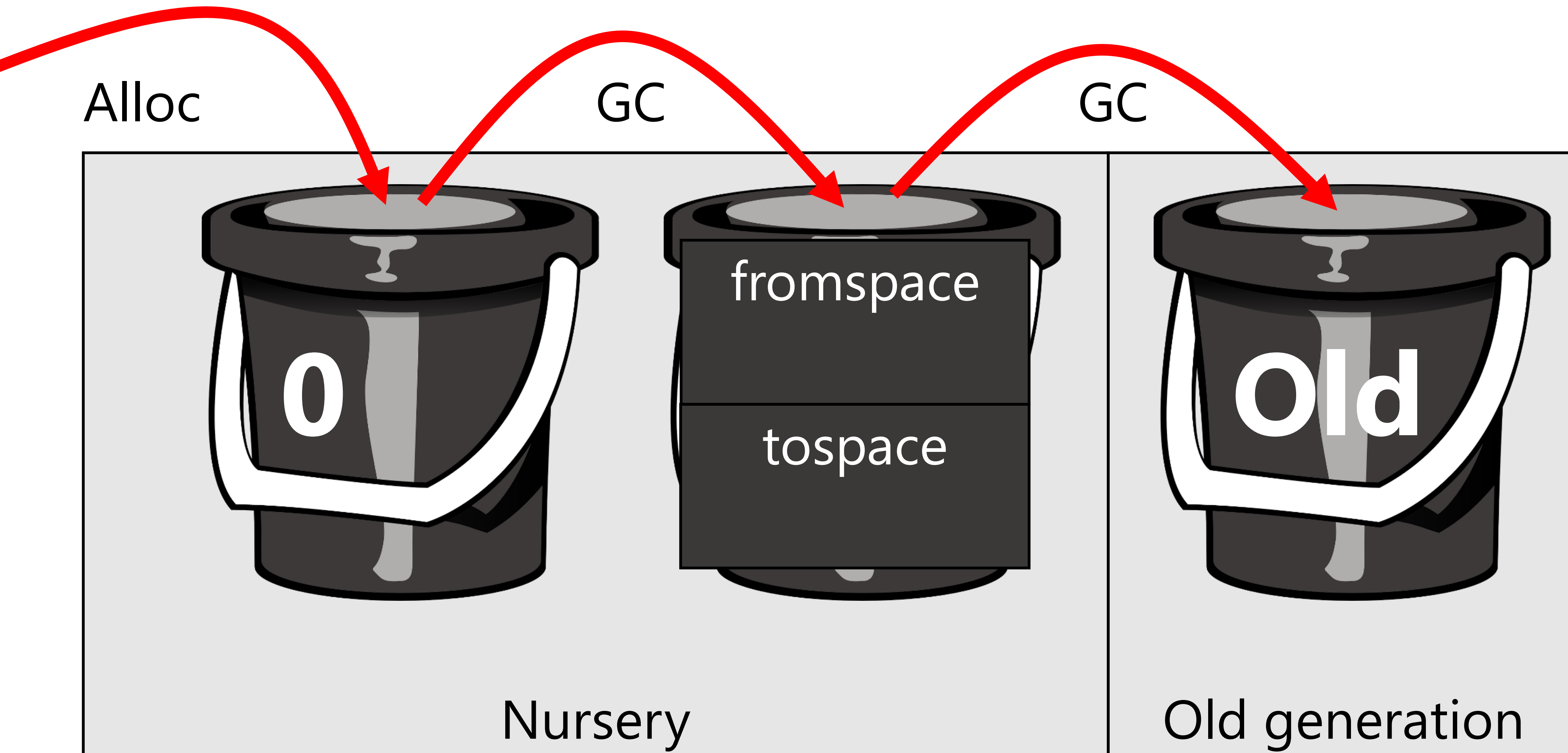
The broken bucket brigade



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
```

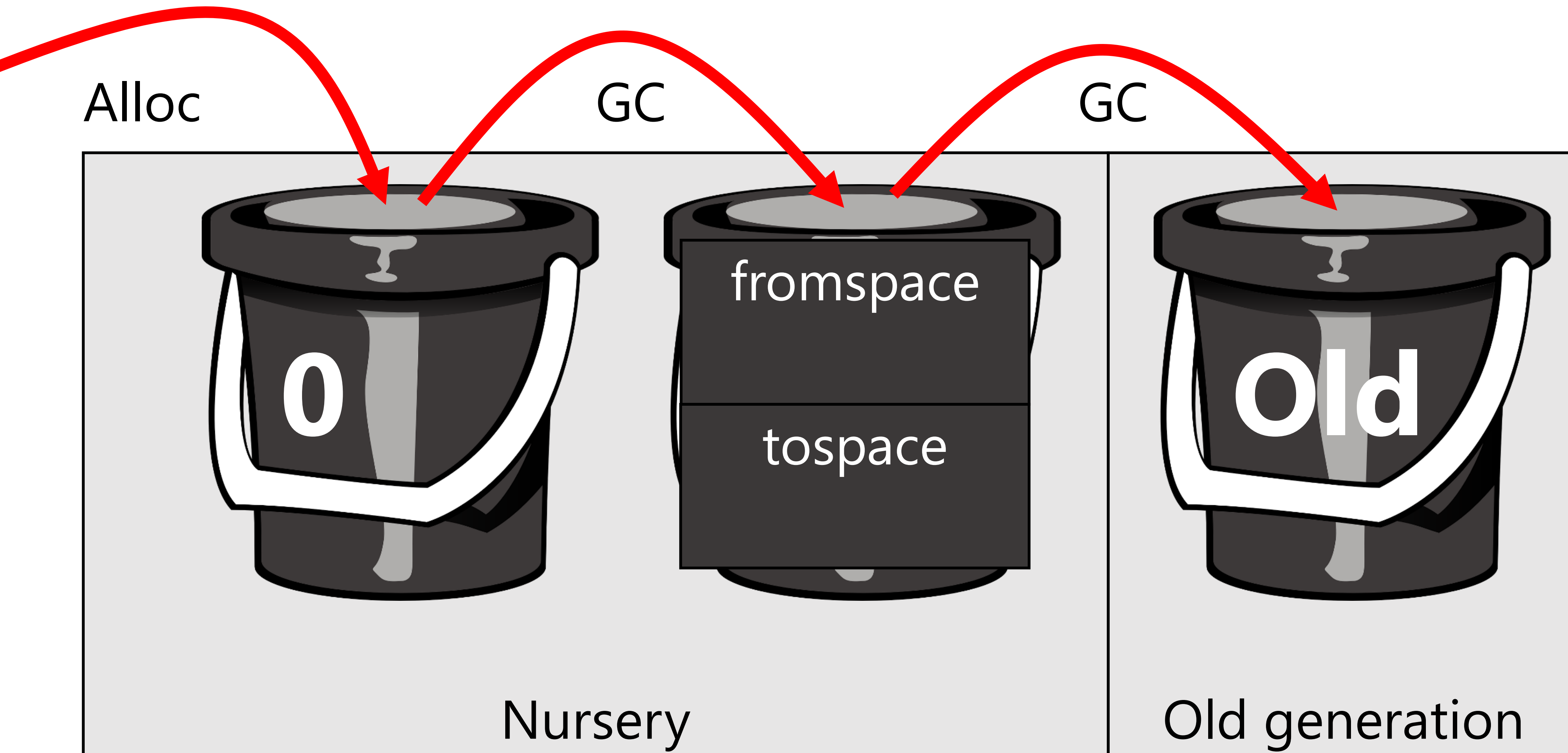
Overhead

- 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)

Cards (part deux)

Cards

- Partitioned (e.g. generational) GC must remember certain inter-partition refs
- Remembering each object too expensive
- Divide generation into cards

Pools with cards

```
struct Pool {
   struct Pool *next;
   struct FreeObject *freeObjects;
   void *freeSpace;
   char remember[CARDS_PER_POOL];
   unsigned short firstObjs
      [CARDS_PER_POOL];
};
```

Allocation with cards

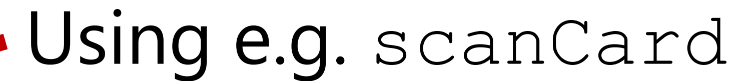
- Remember: firstObjs there to make cards parsable
- Need to worry about firstObjs
- Matters during:
  - Bump-pointer allocation
  - Free-list allocation splitting
  - Coalescence

Parsing cards

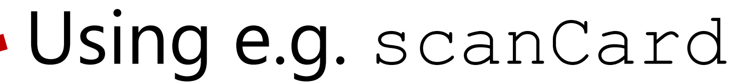
```
scanCard(pool, cardNo):
  loc := pool +
         cardNo*CARD_SIZE +
         pool->firstObj[cardNo]
  while cardOf(loc) == cardNo:
    obj := loc
    (scan this object)
    loc += obj->header.size
```

```
youngCollect():
  (add roots to worklist)
  (add remembered set to worklist)
  fromspace, tospace := tospace, fromspace
  while loc := worklist.pop()
    obj := *loc
    if obj is young and !obj->header.forward:
      if obj->header.collectionCount < 1:
        (move obj to newObj in tospace)
        obj->header.collectionCount++
        obj->header.forward = newObj
      else:
        newObj := (try old gen allocator)
        if newObj == NULL:
          collectFull()
          (retry young collection)
        obj->header.forward = newObj
      (scan newObj)
    if obj->header.forward:
      *loc := obj->header.forward
```

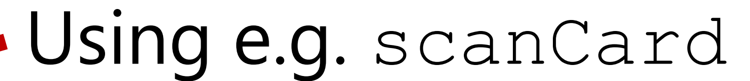
```
youngCollect():
  (add roots to worklist)
  (add remembered set to worklist)
  fromspace, tospace := tospace, fromspace
  while loc := worklist.pop()
    obj := *loc
    if obj is young and !obj->header.forward:
      if obj->header.collectionCount < 1:
        (move obj to newObj in tospace)
        obj->header.collectionCount++
        obj->header.forward = newObj
      else:
        newObj := (try old gen allocator)
        if newObj == NULL:
          collectFull()
          (retry young collection)
        obj->header.forward = newObj
      (scan newObj)
    if obj->header.forward:
      *loc := obj->header.forward
```

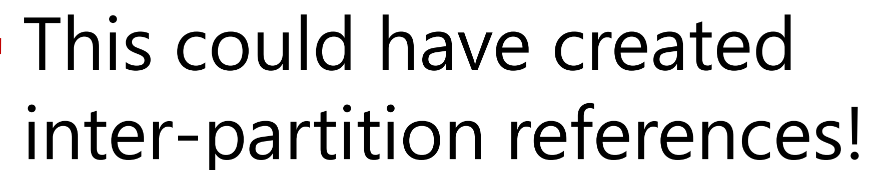


```
youngCollect():
  (add roots to worklist)
  (add remembered set to worklist)
  fromspace, tospace := tospace, fromspace
  while loc := worklist.pop()
    obj := *loc
    if obj is young and !obj->header.forward:
      if obj->header.collectionCount < 1:
        (move obj to newObj in tospace)
        obj->header.collectionCount++
        obj->header.forward = newObj
      else:
        newObj := (try old gen allocator)
        if newObj == NULL:
          collectFull()
          (retry young collection)
        obj->header.forward = newObj
      (scan newObj)
    if obj->header.forward:
      *loc := obj->header.forward
```



```
youngCollect():
  (add roots to worklist)
  (add remembered set to worklist)
  fromspace, tospace := tospace, fromspace
  while loc := worklist.pop()
    obj := *loc
    if obj is young and !obj->header.forward:
      if obj->header.collectionCount < 1:
        (move obj to newObj in tospace)
        obj->header.collectionCount++
        obj->header.forward = newObj
      else:
        newObj := (try old gen allocator)
        if newObj == NULL:
          collectFull()
          (retry young collection)
        obj->header.forward = newObj
      (scan newObj)
    if obj->header.forward:
      *loc := obj->header.forward
```





Runtime interface

Review

- Unreachable objects dead
- Sweep, copying, compacting, ref counting
- Combine with partitioning, generational
- Most objects die young, so partitioning by age universal
- Also useful to partition large objects, threads, executable “objects”, etc

Runtime interface

- Compiler, memory manager and (possibly) programmer must agree
- Compiler motivated by language, so
- memory manager often motivated by language too
- GGGGC's interface is simple by design

Runtime interface

- Important components:
  - Allocation
  - Where references are in roots, objects
  - When collection can occur
  - Barriers on writing to (reading from?) objects

Allocation

Allocation

C `malloc`
"Just give me some bytes."

Memory manager needs no type info.

Object returned full of garbage.

Allocation

C `malloc`
"Just give me some bytes."

Memory manager needs no type info.

Object returned full of garbage.

Haskell allocation
"I promise not to touch!"

Memory manager knows all!

Object returned fully initialized, immutable.

Allocation

C `malloc`
"Just give me some bytes."

Memory manager needs no type info.

Object returned full of garbage.

Java `new`
"Good enough is good enough."

Memory manager needs refs.

Object returned full of zeroes.

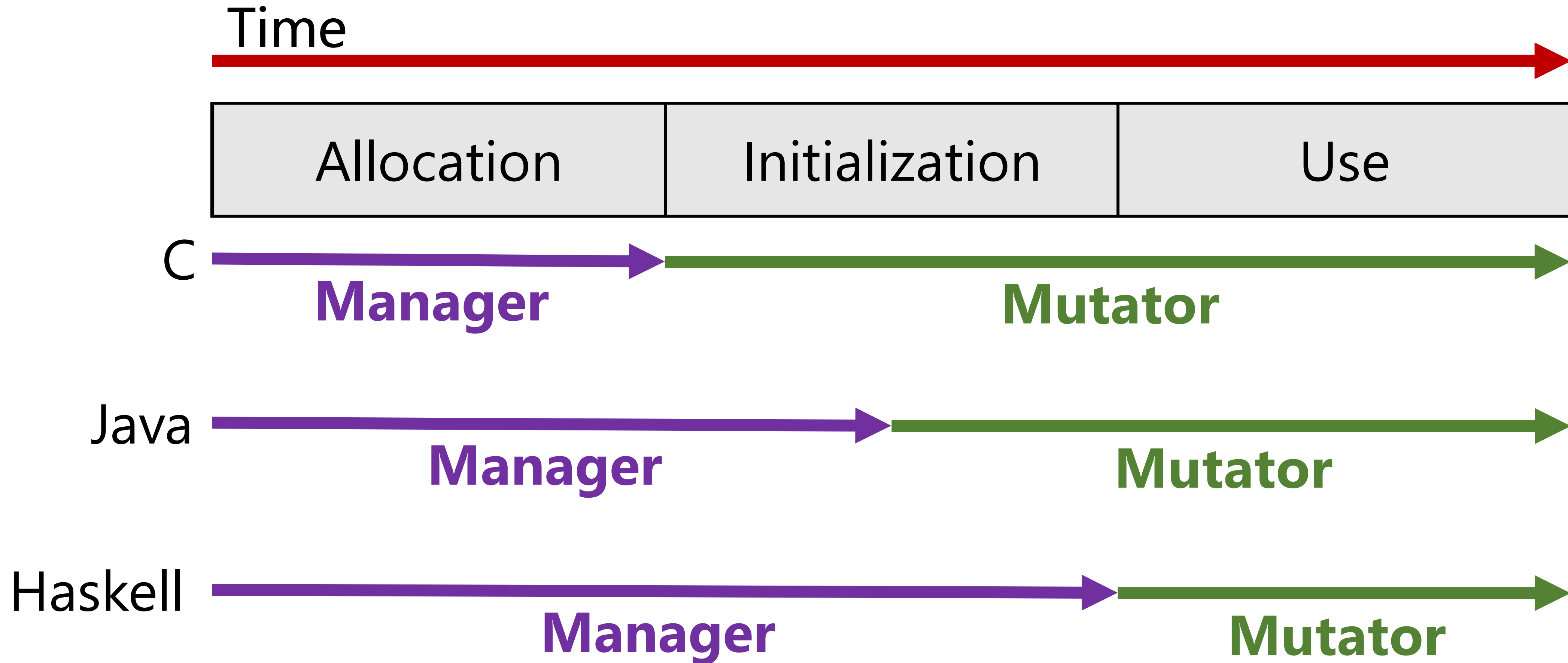
Haskell allocation
"I promise not to touch!"

Memory manager knows all!

Object returned fully initialized, immutable.

Allocation and initialization

Allocation and initialization



Constructor problem

- GC depends on “correct” objects
- “Correct” for us means: References refer to valid objects or **`NULL`**
- If an object scanned mid-initialization, it may not be correct!
- Options: Prevent GC mid-initialization, or initialize ourselves

Mid-initialization GC

- *If* initialization cannot allocate objects, we could prevent collection
- Does not generalize
- Initialization often not guaranteed correct anyway (just user code)
- Generally, manager must initialize

Zeroing

- Simple initialization: All bits zero
- Mundane value for virtually all types
- Simple initialization for manager
- Consequence: Objects initialized twice!
  - (Once by manager, once by program)

When to zero

When to zero

During allocation

Makes allocation slower.

Inefficient to spread work over many allocations.

When to zero

During allocation

Makes allocation slower.

Inefficient to spread work over many allocations.

During collection

Makes collection (much!) slower.

Impossible with free-list.

Efficient.

When to zero

During allocation

Makes allocation slower.

Inefficient to spread work over many allocations.

Ahead of allocation

Zero “chunks” beyond current object during allocation.

Choose chunk size wisely (cache line or page) for efficiency.

During collection

Makes collection (much!) slower.

Impossible with free-list.

Efficient.

Runtime interface

- Important components:
  - ~~Allocation~~
  - Where references are in roots, objects
  - When collection can occur
  - Barriers on writing to (reading from?) objects

How to identify references

How to identify references

Conservative

"If it looks like it might be a reference, it's a reference."

How to identify references

Conservative

"If it looks like it might be a reference, it's a reference."

Precise

Compiler must know location of all references and tell the GC.

Usual for statically-typed languages.

How to identify references

Conservative

"If it looks like it might be a reference, it's a reference."

Tagged

References stored differently from data. Known only at runtime.

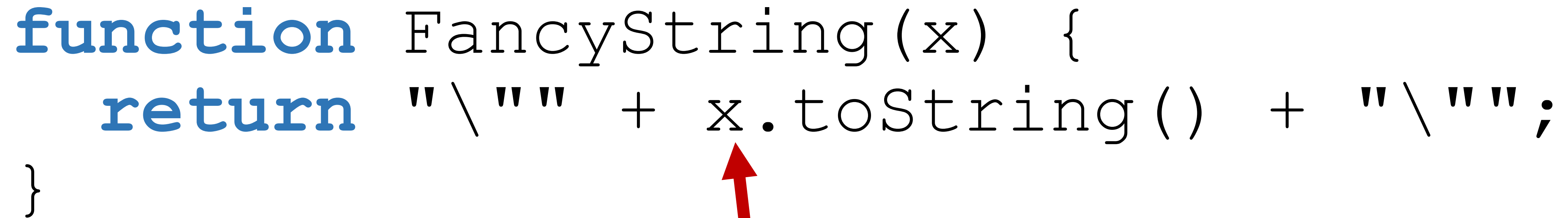
Usual for dynamically-typed languages.

Precise

Compiler must know location of all references and tell the GC.

Usual for statically-typed languages.

Tagged references



This code works whether x is a reference or not

(That’s JavaScript!)

Refresher: Bit-sneakiness

- There are three[1] wasted bits in our header

0000000000000000000000000000000001011111 | 010000111010001111010 | 000

Pool address | Object address | Always 0!

- *All* pointers have this extra space!

[1] On 32-bit systems, two

Tagged references

- Use extra bits as type
- e.g.: References end in 000,
  Integers end in 1,
  Strings end in 010,
  …
- Compiler and GC must agree
- Compiler must untag values to use them

Tagged references

```
trace(obj):
  for loc in obj to obj + (size)
    if (*loc & 0b111) == 0:
      …
      trace(*loc)
      …
```

Precise object references

- You've seen plenty of this!
- Bitmaps is pretty much how it's done
- In some (usually functional) language, object type includes 'trace' function:
  - Compiler creates trace function per type
  - GC calls trace function referred to in descriptor

Stack references

- Pointer stacks (ala GGGGC) is not the most common choice
- Options:
  - Secondary reference stack
  - Parsable stack
  - Heap-allocated stack
  - Object-shaped stack

Secondary reference stack

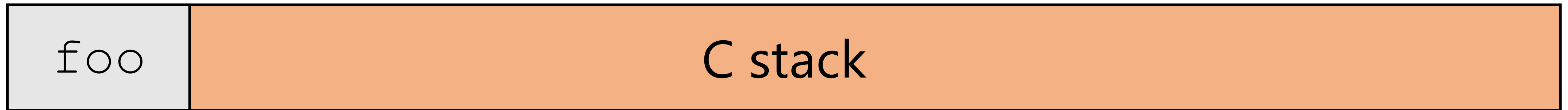
- Normal stack has no references, second stack has only references

C stack

Ref stack

Secondary reference stack

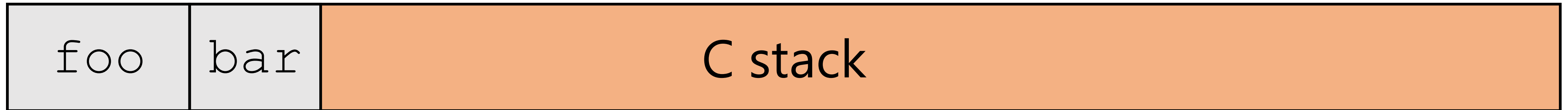
- Normal stack has no references, second stack has only references



```
foo()
```

Secondary reference stack

- Normal stack has no references, second stack has only references



`foo() bar()`

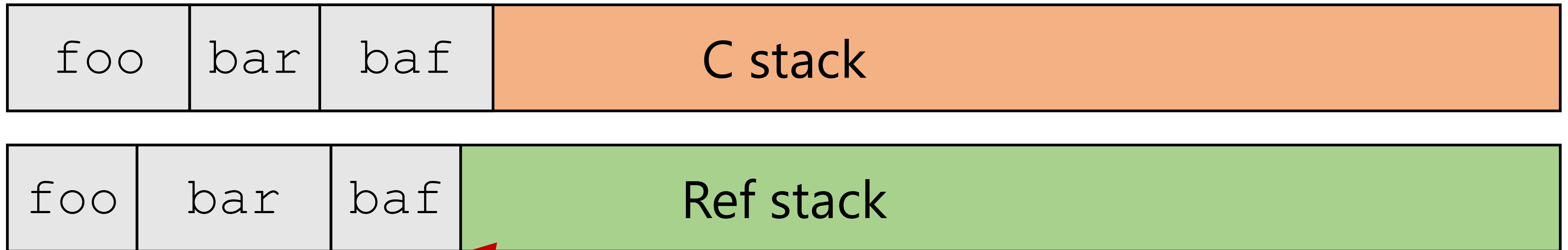
Secondary reference stack

- Normal stack has no references, second stack has only references

`foo() bar() baf()`

Secondary reference stack

- Normal stack has no references, second stack has only references



Reference stack caveats

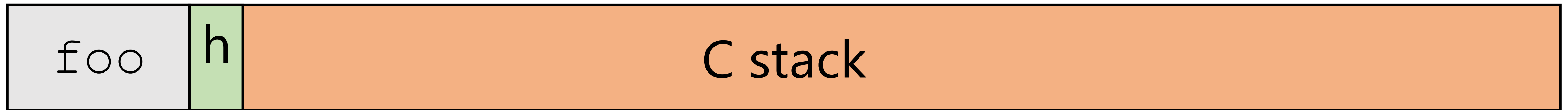
- Usually needs reference stack register
- CPU registers are rare!
- Complicates other parts of compiler, e.g. exception handling

Parsable stack

- Like parsable heap: Make sure GC can find info in stack

Parsable stack

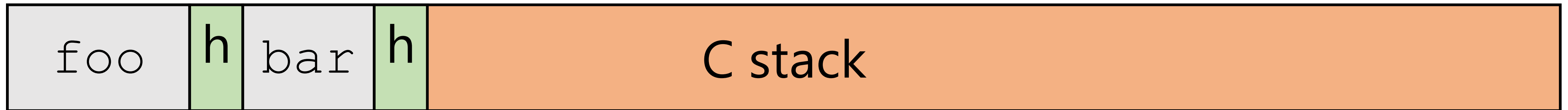
- Like parsable heap: Make sure GC can find info in stack



`foo()`

Parsable stack

- Like parsable heap: Make sure GC can find info in stack



`foo() bar()`

Parsable stack

- Like parsable heap: Make sure GC can find info in stack

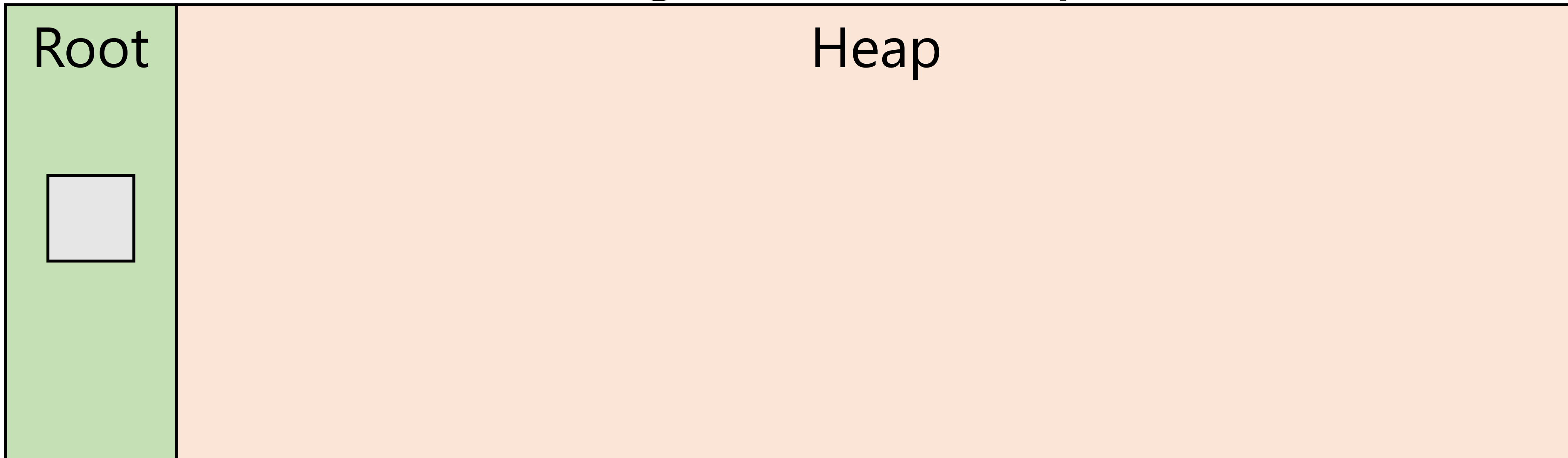
`foo() bar() baf()`

Parsable stack caveats

- Does not play nicely with others (C, C++)

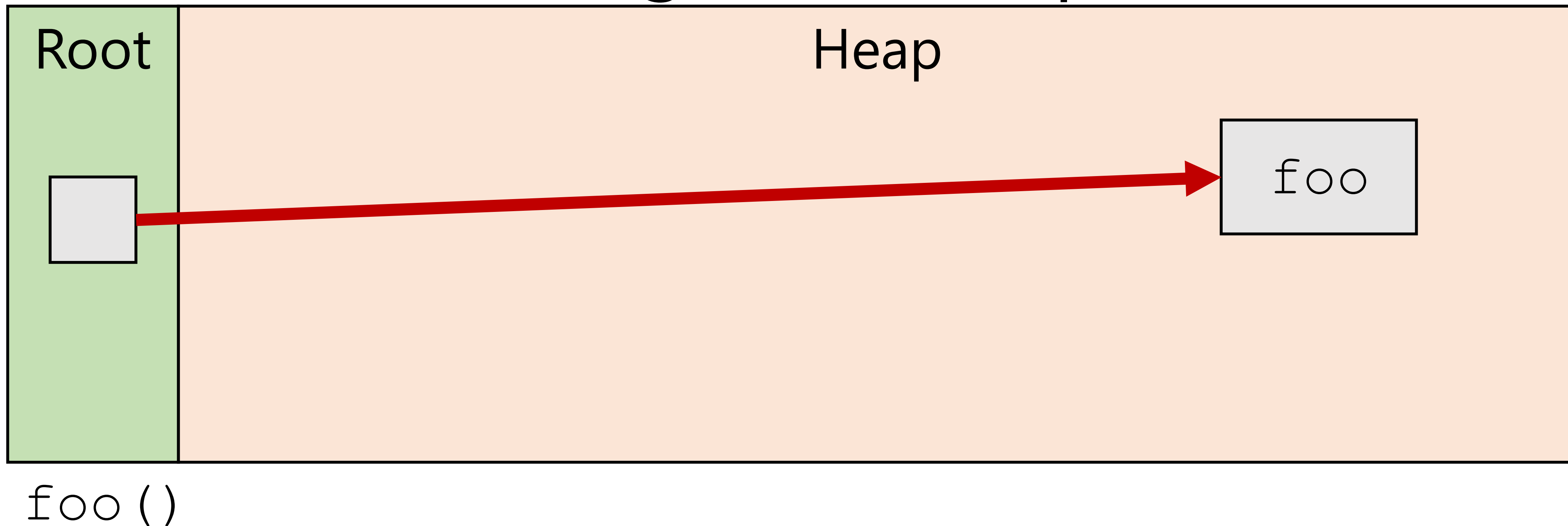
Heap-allocated stack

- Every function allocates a stack “object”, chains those together into pseudo-stack



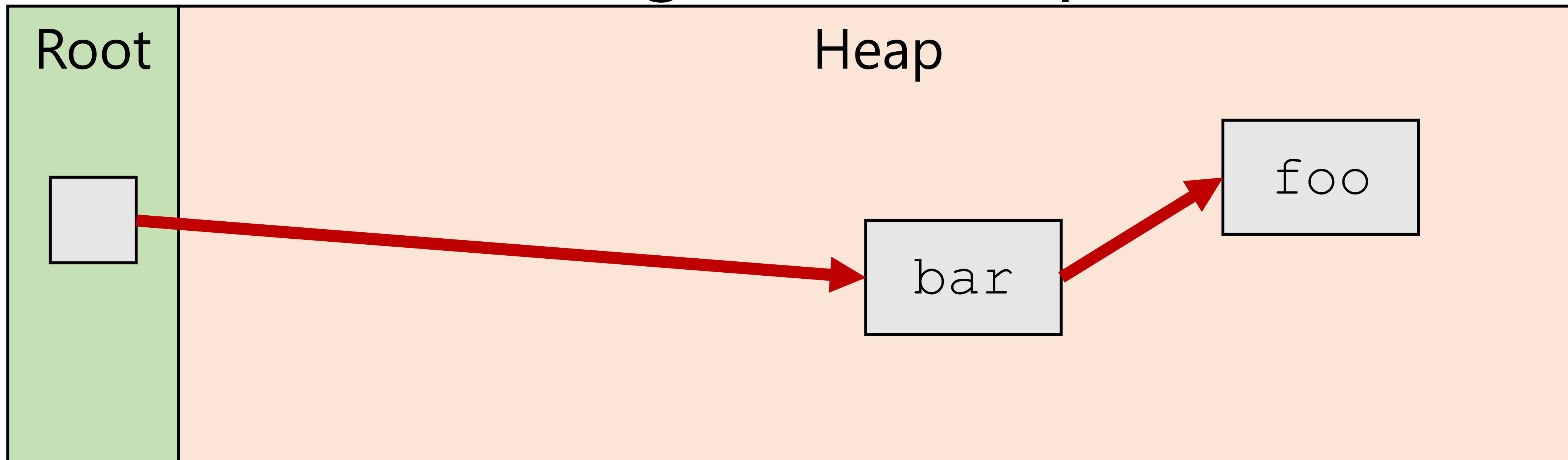
Heap-allocated stack

- Every function allocates a stack "object", chains those together into pseudo-stack



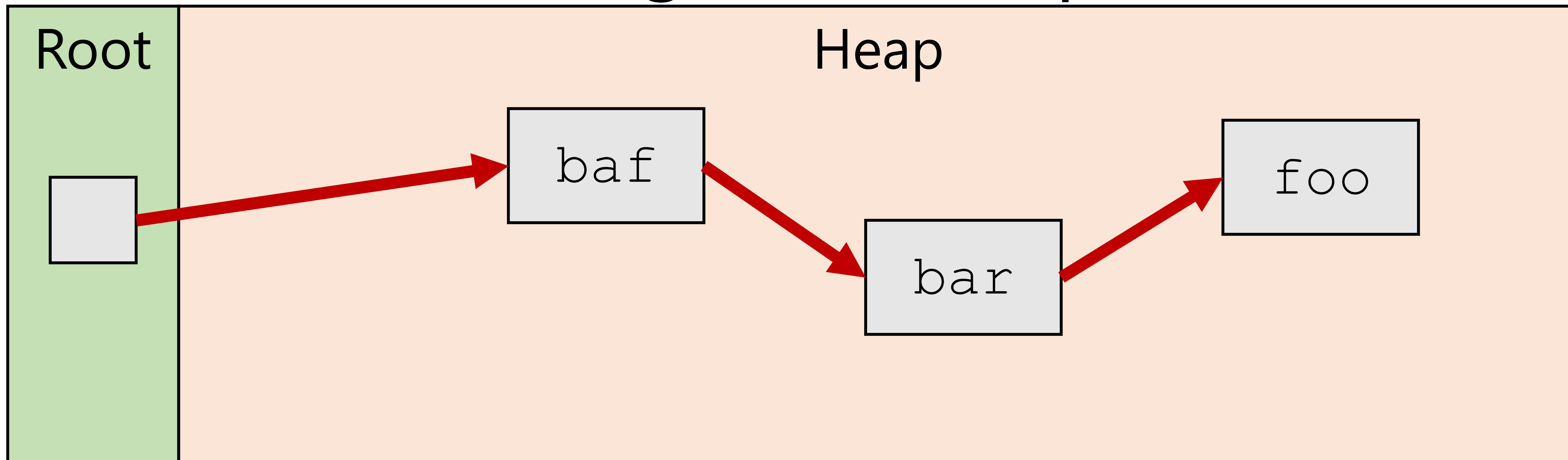
Heap-allocated stack

- Every function allocates a stack “object”, chains those together into pseudo-stack



Heap-allocated stack

- Every function allocates a stack "object", chains those together into pseudo-stack



Heap-allocated stack caveats

- Stack frames outlive function execution
  - Easy closures!
- Need descriptors for every possible stack frame
- Restrictive to compiler

Object-shaped stack

- Exactly like heap-allocated stack, but allocate stack frame objects on stack
- Caveat: GC must expect objects in stack
- Stack is now a partition

What is a reference?

- Thusfar: Reference is pointer to beginning of object
- Possibilities to consider:
  - Interior pointers
  - Indirect pointers

Interior pointer

```c
struct List {
   struct List *next;
   int val;
};

void breakThings(struct List *l) {
   int *nasty = &l->val;
   l = l->next;
   // yield to GC
   printf("%d\n", *nasty);
}
```

Interior pointers

- Usual solution: Not allowed
- When allowed:
  1. Get chunk (e.g. card) associated with interior pointer
  2. Parse chunk to find containing object
  3. If moving, preserve offset

Object tables

- Alternative contract between compiler and memory manager
- Instead of program using (e.g.)
  ```
  struct List *
  ```
  always use (e.g.)
  ```
  struct List **
  ```

Object tables

- Accessing object is double-dereference
- User pointer refers to object table entry,
- object table entry refers to object.
- Object table forms its own partition
- Object table entries don’t move

Object table thoughts

- Slows down object access
- Allows *atomic object replacement*
  - i.e., one object may replace another
- Makes moving objects simple (only one ref to update)
- Makes mark-and-compact require only one sweep!

Runtime interface

- Important components:
  - ~~Allocation~~
  - ~~Where references are in roots, objects~~
  - When collection can occur
  - Barriers on writing to (reading from?) objects

Any-time collection

- Threads may be doing anything at any time
- Can we stop mutator at any time to collect?
- Think about stack references: Mutator must never deviate from description
- Very limiting to optimizations

Yieldpoint collection

- Compiler adds “yieldpoints” to allow collection
- Between yieldpoints, stack in inconsistent state
- To stop, wait for all threads to reach yeildpoint

When to yield

- Mandatory: During allocation, at function calls
- Optional: At tight loops or other long-running operations
- Threads must yield frequently to allow GC to occur promptly

How to yield

- Polling:

```
void yield() {
  if (stopTheWorld)
    collect();
}
```

- Patching: Overwrite the yield function with a version that stops the thread

Runtime interface

- Important components:
  - ~~Allocation~~
  - ~~Where references are in roots, objects~~
  - ~~When collection can occur~~
  - Barriers on writing to (reading from?) objects

Write barriers

- Either reference counting or inter-partition remembering
- For inter-partition remembering:
  - Guard (check that ref must be remembered)
  - Remember (e.g. write bit for card)

Write barrier guard

```
write(obj, loc, val):
  if genOf(obj) == old and
     genOf(val) == young:
    …
```

- genOf isn’t cheap (bit math to get pool, read gen from pool header)

Arranging for better barriers

- Sometimes can ask for pools at certain locations in memory
- Approximate generation check with location check
- Imperfect guard means more remembered set writing, but less time in write barrier

Sequential heap guard

```
write(obj, loc, val):
   if obj > val:
      …
```

- Write barrier can trigger improperly, but
- the check is (usually) one CPU operation
- Young gen must have (pointless) remembered set

Unguarded write barrier

```
write(obj, loc, val):
  poolOf(obj)->(remember obj)
  *loc := val
```

- All writes more expensive,
- remembered set less precise,
- but write time is predictable.

Runtime interface

- Summary:
  - Allocation
  - Where references are in roots, objects
  - When collection can occur
  - Barriers on writing to (reading from?) objects

Presentations

- Next week we start presentations
- [If any presentation doesn’t happen, I will blather in the interim]
- I will have some final words on the last day
- *Make sure to talk to me about your final projects!*